Scientists' Guide to the LSST Applications Software

David Wittman, Perry Gee, Steve Bickerton, and a cast of thousands

June 7, 2011

Purpose of this Document

The Large Synpotic Survey Telescope (LSST) Data Management (DM) team have developed a large set of software modules for astronomical image reduction and analysis. The code is open-source and designed to be generally useful for all sorts of imaging surveys. Development has reached the point where it may be useful for scientists inside or outside the LSST project to use the software for general work. This guide is intended for scientists who have not written any of the code and who are unfamiliar with the tools upon which the LSST software stack is built. It contains pointers to resources for those who wish to drill down very deeply, but aims to be reasonably self-contained for newbies.

In LSST world, we write number-crunching code in C++ and string together the number-crunching steps with Python. This allows us to take advantage of the speed of C++ when necessary, and the ease of Python otherwise. You may be familiar with SciPy/NumPy, which give you the ease of Python[1] while enabling the speed of C++ "under the hood." You can think of the LSST software as providing astronomy-specific functionality "under the hood" in a similar way, but you will not be quite as insulated from the C++ details as you would be with SciPy/NumPy. This document assumes that you will be writing Python rather than C++, but you may need to refer to the C++ documentation to see what features are available. We will guide you through that process.

[1] If you are not familiar with Python, you should become familiar with it, even if you have no plans to work with LSST software. It's extremely useful. See XXX for a good tutorial.

Acknowledgments

The LSST DM Applications team has worked very hard to produce these tools. In alphabetical order, they are: Tim Axelrod, Andy Becker, Steve Bickerton, Martin Dubcovsky, Simon Krughoff, Dustin Lang, Robert Lupton, Fergal Mulally, Russell Owen, Nicole Silvestri, and I've probably missed many more. Our fearless leader is Robert Lupton. K.-T. Lim from the LSST DM Middleware team has also made enormous contributions.

Contents

Chapter 1

Hello, world

This section gives you a very basic idea of how to get started with simple tasks. To run the following examples, you will first need to:

- Install the LSST software stack on your machine. Appendix A describes how to do this.
- Make sure that you have defined the LSST_HOME environment variable.
- Source $LSST_HOME/loadLSST.csh or $LSST_HOME/loadLSST.sh from your shell
- At your shell prompt, type `setup afw`. Appendix A explains more about the purpose of the `setup` command.
- Optionally, configure Python to remember your command history and provide word completion; see Appendix C.
- Type `python` to start the python interpreter, and copy/paste the examples in.
- If you are unfamiliar with object-oriented programming lingo, read or consult Appendix B as necessary.

1.1 Reading and displaying a FITS image

The main package we'll be using is the `afw` (applications framework) package, so we start by importing some afw subpackages.

```
import lsst.afw.display.ds9 as ds9
import lsst.afw.image as afwImg

im1 = afwImg.ImageF('image1.fits')
ds9.mtv(im1, frame=0)
```

Use the name of any handy FITS image in place of image1.fits. `ImageF` creates an image of floats, `ImageD` creates an image of doubles. If you do not already have ds9 running, Python will start it for you at this point.

1.2 Image arithmetic

To save memory and time, images are operated on in-place. That is, you can subtract images of identical size like this:

```
import lsst.afw.display.ds9 as ds9
import lsst.afw.image as afwImg

im1 = afwImg.ImageF(‘image1.fits’)
im2 = afwImg.ImageF(’image2.fits’)

im1 -= im2
```

Be careful: this modifies im1! If you were looking to do something like `im3=im1-im2`, where im3 is a new image, you must explicitly force the creation of the new image and *then* do the in-place arithmetic:

```
import lsst.afw.display.ds9 as ds9
import lsst.afw.image as afwImg

im1 = afwImg.ImageF(‘image1.fits’)
im2 = afwImg.ImageF(’image2.fits’)

im3 = afwImg.ImageF(im1,True) # create deep copy
im3 =- im2
```

Again, this is because when working with large amounts of data we want to minimize the amount of pixel copying. The system is designed to avoid new copies unless explicitly asked. The penultimate line creates a new copy of im1, called im3. Setting the second argument to `True` is necessary to make a *deep copy*, meaning that the resulting image is a completely new image, independent of the parent image and occupying its own area of memory. If the second argument is `False` or is omitted, a shallow copy is generated, meaning that the copy *still refers to the pixel values stored in the parent image*. Thus, if the pixels of a shallow copy are modified, the corresponding pixels in the parent image will be modified as well. *Astronomers who are new to the concept of deep and shallow copies will want to pay particular attention to this point.* Shallow copies save memory and time (for example, making postage stamps from a large image can be very fast and efficient) but you must handle them properly (don’t modify the postage stamps if you will need to refer to the original pixel values).

Similar operators `+=`, `*=`, and `/=` are defined, which do the obvious things. The object on the right-hand side can also be a scalar rather than an image.

1.3 Cutting, copying, pasting, and writing images

To cut out a sub-image, we must first create a `BBox` or bounding box object, which itself must be defined by two `PointI`[1] objects which specify the corners. We then call `ImageF` in a new way:

```
import lsst.afw.image as afwImg
```

[1] `PointI` defines a point where the coordinates are integers, hence the “I”.

```
im1 = afwImg.ImageF('image1.fits')

llc = afwImg.PointI(290,250)
urc = afwImg.PointI(300,260)
bbox = afwImg.BBox(llc,urc)
im4 = afwImg.ImageF(im1,bbox,False)
```

`llc` and `urc` define the lower left and upper right corners of the bounding box, respectively. This form of the `ImageF` constructor takes an image as its first argument, a bounding box as its second argument, and thirdly the boolean indicating the type of copy (`False` indicating that a deep copy should not be made). Shallow copies are excellent for efficiently extracting postage stamps, as long as you remember the implications of modifying their pixel values.

You can set pixel values using the `<<=` operator. For example, if you wanted to set all the pixels inside the bounding box to a scalar value, say 99:

```
im4 <<= 99
```

This also works if the right-hand side is an image of the correct size.

You can also easily mosaic images together. The following example makes a new mosaic image in memory and displays it. This would be useful for making a picture gallery of interesting galaxies, for example.

```
import lsst.afw.image as afwImg
import lsst.afw.display.utils as utils

im1 = afwImg.ImageF('image1.fits')
im2 = afwImg.ImageF('image2.fits')
im3 = afwImg.ImageF('image3.fits')
im4 = afwImg.ImageF('image4.fits')

images = [im1, im2, im3, im4]
labels = ['Label 1', 'Label 2', 'Label 3', 'Label 4']

m = utils.Mosaic()
m.setMode('square')
m.setGutter(0)

mosaic = m.makeMosaic(images)
ds9.mtv(mosaic, frame=0)
m.drawLabels(labels, frame=0)
```

In the `setMode` command, the default is "square" which will make the mosaic image as square as possible. Other allowed values are "y" and "x" which make the mosaic one image wide and one image high, respectively. The `setGutter` command sets the number of pixels between each image in the mosaic.

To save your mosaic image as a new FITS file:

```
afwImg.ImageF.writeFits(mosaic,'filename.fits')
```

1.4 Image statistics

`math.makeStatistics` computes image statistics and returns an object which you can then query for the statistics you want. To save computation time, you specify beforehand which statistics to compute, as follows:

```
import lsst.afw.image as afwImg
import lsst.afw.math as math

im1 = afwImg.ImageF('image1.fits')

flags = math.MEAN | math.STDEV | math.ERRORS
stats = math.makeStatistics(im1,flags)

print stats.getResult(math.STDEV) # prints stdev AND its uncertainty
print stats.getResult(math.MEAN) # ditto for mean
```

Possible statistics to calculate:

- ERRORS include errors of requested quantities.
- NPOINT number of sample points
- MEAN estimate sample mean
- STDEV estimate sample standard deviation
- VARIANCE estimate sample variance
- MEDIAN estimate sample median
- IQRANGE estimate sample inter-quartile range
- MEANCLIP estimate sample N-sigma clipped mean
- STDEVCLIP estimate sample N-sigma clipped stdev
- VARIANCECLIP estimate sample N-sigma clipped variance
- MIN estimate sample minimum
- MAX estimate sample maximum
- SUM find sum of pixels in the image
- MEANSQUARE find mean value of square of pixel values
- NOTHING We don't want anything.

This list is provided here to give you an idea of the capabilities in the system, but capabilities change over time. It's a good idea to check the C++ documentation (as described below) for the most up-to-date list.

If you want detailed control over how the statistics are computed, for example changing the clipping from the default 3σ and 3 iterations, you must create a StatisticsControl object with five arguments: the number of sigma to clip at, the number of iterations for clipping, a specification of which bitplanes of the mask to use (0 means use all of them; mask bitplanes will be explained in the next section, and in this case we have no mask yet anyway), a boolean indicating whether to avoid using pixels which have NaN (not-a-number) values, and a boolean indicating whether to weight pixels by their inverse variance. In the example below, we use all the default values except to ask for 5σ rather than 3σ clipping.

```
import lsst.afw.image as afwImg
import lsst.afw.math as math

im1 = afwImg.ImageF('image1.fits')

flags = math.MEAN | math.STDEV | math.ERRORS
stats = math.makeStatistics(im1,flags)

ctrl = math.StatisticsControl(5.0,3,0,True,False)
stats = math.makeStatistics(im1,flags,ctrl)
print stats.getResult(math.STDEV)
```

In this case, we get a larger STDEV because we clipped less harshly than the default.

1.5 Masked images

So far this may seem rather mundane, but the framework has some features that will make it much more powerful with almost no extra work on your part. For example, we often want to propagate uncertainties through these arithmetic operations, and apply masks which ensure that the operations are not done on bad areas of the image. The applications framework defines a `maskedImage` which contains an image, a variance image, and a mask image. Once you have constructed `maskedImage`s, masking and uncertainty propagation is done *automatically* when you invoke these arithmetic operations.

Before the examples, a few remarks about masks are in order:

- The mask part of MaskedImage has pixels of type unsigned, so is created with the lsst.afw.image.MaskU constructor. However, you may not need to specifically create the mask, because there are other constructors, such as lsst.afw.image.MaskedImageF, which will create both the data image (float in this case) and the mask in one call.
- The mask is quite flexible, far more than just a binary one or zero indicating a good or bad pixel. It has 16 bitplanes, and you can assign any meaning you want (eg, saturated, bad pixel, etc) to any of the planes. In this manual, we will stick to the default set of planes:

  ```
  Plane 0 -> BAD
  Plane 1 -> SAT
  Plane 2 -> INTRP
  ```

```
Plane 3 -> CR
Plane 4 -> EDGE
Plane 5 -> DETECTED
Plane 6 -> DETECTED_NEGATIVE
```

In fact, we do not have to worry about which plane is assigned which meaning because we can always use getters and setters to manipulate data by meaning rather than by plane number.

Let's create an artificial masked image and then show how it works. First, we'll demonstrate that variances automatically propagate correctly when we do math operations on a MaskedImage (remember, having a mask is not the only key feature of a MaskedImage; the variance image is also useful and powerful).

```
import lsst.afw.image as afwImg

im1 = afwImg.MaskedImageF(100,100) # new 100x100 masked image
                                   # with zero pixel vals in data,
                                   # variance, and mask images

# set the variance to something nonzero
vari = im1.getVariance()
vari.set(1.0)  # this sets ALL pixels of the variance image

# the variance should increase by 4 if we multiply
# the data by 2
im1 *= 2
vari.get(0,0) # we just look at the 0,0 pixel because they're
              # all the same at this point
```

The result is a variance of 4, as it should be. Let's try operations involving two images:

```
import lsst.afw.image as afwImg

im1 = afwImg.MaskedImageF(100,100)
im2 = afwImg.MaskedImageF(100,100)

# set the variances to something nonzero
var1 = im1.getVariance()
var1.set(1.0)
var2 = im2.getVariance()
var2.set(2.0)

# the variances should add when we add or multiply the images
im1 += im2
var1.get(0,0)
```

The variance is now 3; the variances automatically added when I added two images. Note that *multiplying* these images would result in zero variance because propagation of errors through multiplication involves multiplying by the pixel values themselves, which are zero here. To convince yourself that variances are propagated through multiplication correctly, you might want to set the pixels to nonzero values.

Now let's simulate a saturated pixel and show how the mask autmatically excludes it from the statistics:

```
import lsst.afw.image as afwImg
import lsst.afw.math as math

# set up our usual image with zero pixvals and unit variance
im1 = afwImg.MaskedImageF(100,100)
var1 = im1.getVariance()
var1.set(1.0)

# set one saturated pixel
img = im1.getImage()
img.set(0,0,65000.0) # this sets just the 0,0 pixel

# get statistics BEFORE masking
stats = math.makeStatistics(im1,math.MEAN|math.NPOINT|math.STDEV)
stats.getResult(math.MEAN)
stats.getResult(math.STDEV)
stats.getResult(math.NPOINT)

# over 10000 pixels, the mean is 6.5 and the stdev is 650,
# because of the one saturated pixel (the rest are zero)

# if you want to visualize the image we created
import lsst.afw.display.ds9 as ds9
ds9.mtv(im1)

# now mask the saturated pixel
mask = im1.getMask()
satbit = mask.getPlaneBitMask('SAT')
mask.set(0,0,satbit) # again, setting just the 0,0 pixel

# and try the statistics again
stats = math.makeStatistics(im1,math.MEAN|math.NPOINT|math.STDEV)
stats.getResult(math.MEAN)
stats.getResult(math.STDEV)
stats.getResult(math.NPOINT)

# now, over 9999 pixels, the mean and variance are zero
```

So once you have masked your pixels and defined your variances, the framework makes math operations on images extremely useful because the masking and variance propagation happen automatically. In real life, you will not be manually setting mask bits and variances. We will cover more realistic examples in later chapters, but this should be enough to demonstrate the usefulness of the framework.

Note: because you are running interactively, stats.getResult(math.WHATEVER) prints the result. In a real script, you would store the result for later use with something like `mean = stats.getResult(math.MEAN)`.

1.6 Exposures

As you have probably already noticed, calling an object an "image" in the LSST environment is not as simple as it sounds - there are several classes which in everyday English could all be called images, but which behave differently in the LSST software. The image on the following page charts their relationships.

To begin, an "image" is simply a 2D array of pixels. A MaskedImage includes an image, a variance image, and a mask, as already explained. An Exposure is what you will most likely actually be working with. It aggregates a MaskedImage and a few other useful things:

- Metadata, or information about the data. You can think of this as a sort of "FITS header", but it is far more flexible because it is really a Python dictionary.
- An optional point-spread function (PSF). We'll see later how to generate and use this.
- An optional World Coordinate System (WCS) which maps pixel locations to (usually) RA and DEC. We'll see later how to generate and use this.

You can construct an Exposure from a FITS file with[2]:

```
import lsst.afw.image as afwImg
im1 = afwImg.ExposureF('foo.fits')
```

but foo.fits will be expected to be a multi-extension FITS file, with extensions representing the data, mask, and image. Alternatively, `im1 = afwImg.ExposureF('foo')` will try to read three separate FITS files named foo_img.fits, foo_msk.fits, and foo_var.fits.

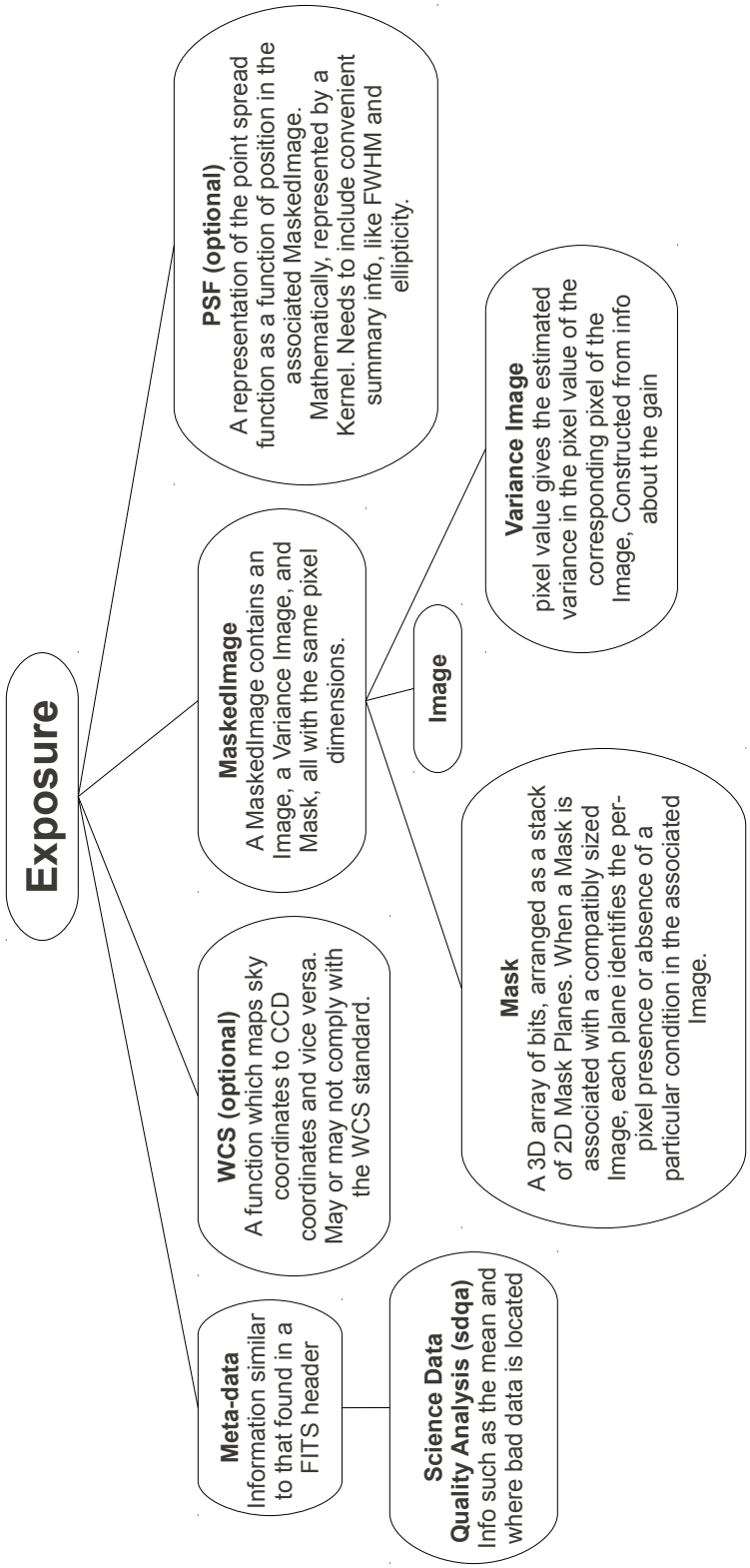
If you are just getting started and have some images you want to work with, you probably do not have variance and mask images lying around like that. You would probably like to construct the variance image using the pixel values and the gain, and you probably have a bad pixel mask image with just zeros and ones rather than multiple bitplanes. So let's write a Python function to import that into LSST land.

```
import lsst.afw.image as afwImg

# doing this example right requires more time than I have right now!

gain = 3 # need to get gain from header
im = afwImg.ImageF('foo.fits')
var = im1/gain
mask = afwImg.MaskU('bpm.fits') # need to make sure have right bitplane
exp = afwImage.ExposureF(im,mask,var) # this doesn't grab any metadata!
```

*

This concludes our set of "Hello, world" examples. To move beyond these very simple tasks, we must first get an overview of what's available in the LSST packages.

[2]Note the templating: it is ExposureF in Python even though in C++ Exposure is templated over the types of image, mask, *and* variance: `Exposure< ImageT, MaskT, VarianceT >`. In other words, if you read the C++ documentation you might have expected to use ExposureFUF. To simplify life in Python, only ExposureF and ExposureD (meaning float and double images, with unsigned short masks, and float or double variance images respectively) are defined.


Exposure
PSF (optional)
A representation of the point spread function as a function of position in the associated MaskedImage. Mathematically, represented by a Kernel. Needs to include convenient summary info, like FWHM and ellipticity.
MaskedImage
A MaskedImage contains an Image, a Variance Image, and Mask, all with the same pixel dimensions.
Image
Variance Image
pixel value gives the estimated variance in the pixel value of the corresponding pixel of the Image, Constructed from info about the gain
WCS (optional)
A function which maps sky coordinates to CCD coordinates and vice versa. May or may not comply with the WCS standard.
Mask
A 3D array of bits, arranged as a stack of 2D Mask Planes. When a Mask is associated with a compatibly sized Image, each plane identifies the per-pixel presence or absence of a particular condition in the associated Image.
Meta-data
Information similar to that found in a FITS header
Science Data Quality Analysis (sdqa)
Info such as the mean and where bad data is located

Chapter 2

Overview of LSST Packages

Having demonstrated how easy it is to do powerful things with this software, let's take a brief look at each of the available packages and what they can do. Later in this book, each package will star in its own more detailed chapter.

2.1 Online documentation

It will be useful to browse the online documentation while reading through this chapter. As the C++ programmers write their code, they build in a certain amount of documentation through a system called Doxygen. Doxygen generates a web page for each package and class and makes it relatively easy to click through. For example, if a method takes an argument of a type you don't recognize, you should be able to click on it and get the definition of that type.

The most convenient version of the Doxygen docs is available online at `http://www.astro.princeton.edu/~rhl/LSST/Doxygen/htmlDir/index.html`. While the code is constantly changing, this website is not constantly updated; however, it should be sufficient for most needs. If necessary, there is a way to generate the docs from the exact version of the code that exists on your computer, but it won't be as nicely crosslinked as this version; you will have to search each package separately, for example.

This section will describe how to navigate the online documentation, but first we must discuss the concept of *namespace*. In older languages such as Fortran and C there is only one namespace; each distinct function must have a distinct name. This could become a problem if you were using multiple third-party libraries and different libraries happened to use the same name for different things, resulting in a *name collision*. In Python and C++, name collisions are avoided by having multiple namespaces. For example, in Python the statement `import lsst.afw.image as afwImg` means that there is an lsst namespace, within that an afw namespace, and within that an image namespace; and to avoid the hassle of constantly typing lsst.afw.image, we will define afwImg as a nickname. So we can refer to the constructor `lsst.afw.image.ImageF` as `afwimage.ImageF` without worrying that there might also be an `ImageF` defined in `some.other.package`; that `ImageF` would have to be referred to as `some.other.package.ImageF`. See `http://docs.python.org/tutorial/classes.html#python-scopes-and-namespaces` for a more detailed description.

Navigating the maze: The tabs

Go ahead and click on `http://www.astro.princeton.edu/~rhl/LSST/Doxygen/htmlDir/index.html` if you haven't already. The *Main Page* is titled *lsst::afw; the LSST Application Framework* and is indeed afw-centric. While afw is the obvious starting place for newbies, keep in mind that there is a *lot* of functionality in other namespaces, and it's *all* documented here, despite the title. The major difference is that afw has a fair amount of readable documentation here which goes above and beyond the standard "here is the definition of class foo and here are all of its constructors." The main page links directly to a handful of useful writeups, such as *How to display images*.

The *Namespaces* and *Classes* tabs will be your workhorses. Clicking on the Namespace tab yields an alphabetical list of namespaces. One quirk: some namespaces within afw are listed first, because afw is first alphabetically; but others are listed under lsst::afw. (Note: the '::' separator is C++'s equivalent of Python's '.' separator for namespaces.) The reason for this is that some namespaces were defined in Python and some in C++; Doxygen makes the former appear as afw::xxxx while the latter appear as lsst::afw::xxxx. Although of no real significance, it can be confusing: clicking on afw::image yields a very boring page containing only testUtils and utils, while clicking on lsst::afw::image yields the richly detailed page you really wanted.

Classes are the abstract definitions of what we previously referred to as "objects." For example, lsst::afw::image::ImageF is a class. When we use the constructor lsst::afw::image::ImageF(filename), it returns an object of that type. Internally, classes contain data members (for example, the size of the image), but these are generally not accessible externally. All external manipulations are supposed to be done using the methods listed. Note that if you go to the Classes tab, you will *not* find lsst::afw::image::ImageF. Rather, you will find `lsst::afw::image::Image< PixelT >`. This is because Image is a *templated* class. The bit in angle brackets is a stand-in for F, D, or I depending on whether the image is float, double, or int. Templates allow C++ programmers to write the code once, and have the compiler "expand the template" for them, rather than write essentially the same code over again for each new type.

Go ahead and click on the `lsst::afw::image::Image< PixelT >` you find listed in the Namespaces tab. As promised in the previous chapter, you will find all the gory detail of all the different constructors and operators that have been defined. (Note: rhs and lhs stand for right-hand side and left-hand side respectively.) You will often see the same operator defined multiple times. For example, '-=' is defined once for a scalar rhs and once for an image rhs. This is what allows you, the Python programmer, to put either type on the rhs and have it just work automatically.

A further word on navigation: at the top of the page you see the clickable text `lsst::afw::image::Image`. Note that *each item separated by :: is separately clickable.* So if you want to see what else is in the `lsst::afw::image` namespace, click on the first `image`; to see what else is in the `lsst::afw` namespace, click on `afw`, etc. This is often a useful way to get where you want, if you find the semi-alphabetical listing of namespaces in the Namespaces tab too confusing.

For completeness, we also describe the less oft-used tabs. The *Related Pages* tab contains a fairly long list of pages which are meant to be tutorials or introductions in the style of those linked to from the Main Page. But many of these pages are blank, and some of the non-blank ones have syntax errors or are out of date. These may still be useful as a rough guide to how the developers intended the code to be used, but do not expect them to work verbatim. Ignore the *Modules* tab; "module" is not a well-defined concept as is namespace or class, and here it contains an odd collection of things. The *Files* and *Directories* tabs list all source code files and directories; while straightforward, these are not immensely useful to the beginner. But note that everything is clickable, so if you must hunt through source code files and directories, doing it here may be more convenient than doing it at your Unix prompt. The *Examples* tab will be useful mainly for C++ programmers.

Ready? Fasten your seat belts for a quick tour through LSST land!

2.2 lsst::afw

The application framework defines basic behavior of images, background models, source detections, and the like, as well as supporting actors such as bounding boxes. At the highest level (`http://www.astro.princeton.edu/~rhl/LSST/Doxygen/htmlDir/namespacelsst_1_1afw.html`[1]) we see only eight items, the namespaces `cameraGeom, coord, detection, display, formatters, geom, image`, and `math`. Rest assured, there is a lot within! The really useful namespaces are `detection, display`, and `image`. We'll start with `image` because it's so fundamental.

2.2.1 lsst::afw::image

Defines basic behavior of images.

2.2.2 lsst::afw::detection

Tools for detecting sources in images.

2.2.3 lsst::afw::display

Tools for displaying images and (presumably) associated helpers like bounding boxes.

2.3 lsst::meas

2.4 lsst::ip

2.4.1 lsst::ip::isr

2.4.2 lsst::ip::diffim

[1] Not to be confused with `http://www.astro.princeton.edu/~rhl/LSST/Doxygen/htmlDir/namespaceafw.html`, as mentioned above.

Chapter 3

Extended examples

3.1 Using the ISR to remove instrumental signature from your data

3.2 Writing Sextractor in 100 lines of python

In this example, we will demonstrate how to subtract the background from an Exposure, then locate the sources, and finally perform some calculations on them. We'll be creating a few policies, which specify the allowed values for certain parameters of the different measurements we'll be using.

First, we have the usual importing of packages, and then the importing of the exposure with which we will be working.

```
import lsst.afw.image as afwImg
import lsst.afw.display.ds9 as ds9
import lsst.meas.utils.sourceDetection as sDet
import lsst.pex.policy as pexPol
import lsst.afw.detection as afwDet
import lsst.meas.algorithms as measAlg
import math

exposure = afwImg.ExposureF('myImage.fits')
mi = afwImg.ExposureF.getMaskedImage(exposure)
wcs = afwImg.ExposureF.getWcs(exposure)
```

Then we specify the parameters for the background measurement.

```
bgPol = pexPol.Policy()
bgPol.add('binsize',30)
bgPol.add('algorithm','NATURAL_SPLINE')
```

`binsize` is the number of pixels used in the binning when separating the background from the sources. Smaller numbers give more precision, but cost more time, while larger numbers are faster but less precise. `algorithm` is the type of algorithm used when detecting the background.

First up is the background subtraction, and displaying the resulting image.

```
background,subtractedImg = sDet.estimateBackground(exposure,bgPol,True)
mi = afwImg.ExposureF.getMaskedImage(subIm)
img = afwImg.MaskedImageF.getImage(mi)

ds9.mtv(img)
```

Don't close ds9 here because we'll need it later.

Next, we specify the parameters for the *detection* of sources.

```
detPol = pexPol.Policy()
detPol.add('minPixels',10) ## The source must have at least this many pixels
detPol.add('nGrow',1)
detPol.add('thresholdValue',30)
detPol.add('thresholdType','value')
detPol.add('thresholdPolarity','both')
```

The value you specify for `thresholdValue` will be the result of some trial and error. This is the minimum value that the pixels in a source might have - it can be determined by displaying the subtracted image in ds9 and placing your cursor at the edge of something you know is a source, then reading the value from the "Value" field. Remember to use the subtracted image and not the original to determine this number! The value of `minPixels` is the minimum number of adjacent pixels that must have a value of at least `thresholdValue`.

Now we do the detecting.

```
FWHM = 2  ## stands for full-width half-max of the Gaussian which models the PSF
psf = measAlg.createPSF("DoubleGaussian", 15, 15, FWHM/(2*math.sqrt(2*math.log(2))))

dsPos,dsNeg = sDet.detectSources(subIm,psf,detPol)  ## Returns the positive and negative detections, but
objects = dsPos.getFootprints()
print len(objects) ## Prints how many sources were found in your image
```

Next, we specify the values for the *measurement* of sources, and finally perform some measurements. [this portion still needs work]

```
sourcePol = pexPol.Policy()
sourcePol.add('source','astrom')
sourcePol.add('astrometry','SDSS')
sourcePol.add('shape','SDSS')
sourcePol.add('photometry','NAIVE')
sourcePol.add('centroidAlgorithm','SDSS')
sourcePol.add('shapeAlgorithm','SDSS')
sourcePol.add('photometryAlgorithm','NAIVE')
sourcePol.add('apRadius',3.0)

measureSources = measAlg.makeMeasureSources(exposure,sourcePol,psf)
sourceList = afwDet.SourceSet()

for i in range(len(objects)):
    source = afwDet.Source()
```

```
    measureSources.apply(source,objects[i])
    sourceList.append(source)
    source.setId(i)

    print source

    ## Locate the center of each source and display a red +
    xc, yc = source.getXAstrom() - mi.getX0(), source.getYAstrom() - mi.getY0()
    ds9.dot("+", xc, yc, size = 5, ctype = ds9.RED)
```

Chapter 4

Working with LSST Pipeline Code

In this chapter, we will use some of the techniques we learned in the previous chapter to work in detail through the steps required to build a catalog from a real astronomical image. Along the way, we will cover many of the important classes and techniques used by the LSST pipelines, including the Image Characterization and Single Frame Measurement pipelines.

4.1 Starting Points

The current LSST DC3 Pipelines are built around images and telescope metadata specifically designed to simulate LSST. Since our purpose is to demonstrate how the LSST framework can be used to reduce images from any astronomical source, we will start instead with an image taken from the NOAO 4m telescope at KPNO. The image has already been bias subtracted and flat-fielded, but has not been background subtracted.

Fits Files:

In this example, we will extract a single 2048 x 4096 pixel CCD image from an eight CCD mosaic image, and prepare it for processing by the LSST pipeline. Our input image is called obj074.fits. It is an ordinary 9 segment fits file: the first segment is used to store the overall image header, and the remaining 8 segments contain a header and image data for each of the 8 CCDs in the KPNO mosaic.

We also have metadata from the telescope, including bad pixel masks and amplifier gain information for each chip.

```
obj074.fits               8 CCD mosaic image
bpm.4.fits                bad pixel mask from amplifier 4
amplifier 4 gain          2.9
saturation level          28000
```

Package Setup and Imports:

In order to run the Python code in this chapter, you will need to install and setup the datarel package, which contains runStage, the procedure used to run the various image processing steps. If you have the LSST software stack installed, datarel and its dependencies can be fetched using

```
%lsstpkg install datarel
```

The list of Python package imports and aliases which you will need to run the code in this section include:

```
importpdb
import pickle
fromlsst.datarel import runStage
importlsst.afw.detection as afwDetect
importlsst.afw.image as afwImg
importlsst.meas.pipeline as measPipe
importlsst.afw.geom as afwGeom
importlsst.afw.math as afwMath
import lsst.afw.display.ds9 as ds9
importlsst.daf.persistence as dafPersist
importlsst.pex.policy as pexPol
importlsst.daf.base as dafBase
```

4.2 Creating an LSST exposure

As discussed in the previous chapter, an afw.lsst.image.ExposureF object can be used to house many different pieces of information about an astronomical observation. In this section, we will assemble an exposure from our test file and known metadata supplied by the telescope, including instrumental gain and bad pixel masks.

The data and bad pixel mask images are first read into ImageF objects from the fits files in the current directory. Note that only CCD 4 and the corresponding bpm file are read. The saturation level and gain for that chip are supplied in the header of the observation, and are reproduced in the code shown below:

```
#   Read the data and bad pixel images
img = afwImg.ImageF("obj074.fits", 4)
gain = 2.9
saturation = 28000
var = afwImg.ImageF(img, True)
var /= gain

#   Create a mask object to contain bad pixel and saturation info
mi = afwImg.MaskU(img.BBox(afwImg.PARENT), 0)

#  Set the saturated bit in the Mask for all pixels > the saturation
satbit = mask.getPlaneBitMask('SAT')
```

```
fori in range(img.getWidth()):
    for j in range(img.getHeight()):
        ifimg.get(i,j) > saturation:
            mask.set(i,j,satbit)

#  Set the bad bit in the Mask for all bad pixels in the bpm
bpm = afwImg.ImageF("bpm_4.fits")
badbit = mask.getPlaneBitMask('BAD')
fori in range(bpm.getWidth()):
    for j in range(bpm.getHeight()):
        if not bpm.get(i,j) == 0:
            mask.set(i,j,badbit)

#  Finally, create the masked image and exposure, save to a fits file
mi = afwImg.makeMaskedImage(img, mask, var)
exposure = afwImg.makeExposure(mi)
exposure.writeFits("exposure.fits")
```

The code should be pretty self explanatory by now, as it is a real world example of what we did in chapter 3. The image, variance, and mask images are assembled from the information contained in the input fits files, an ExposureF object is constructed, and for future use, the object is written to disk as a fits file under the name "exposure.fits".

4.3 Pipelines, Stages, Policies, and Clipboards

Before we go any further, we need to introduce some basic LSST pipeline terminology. The LSST software stack contains code to do a wide variety of highly organized tasks on input data. These tasks are divided into "pipelines", each pipeline applying a sequence of operations on the data to accomplish a specific task.

In our example, we will be examining code from the Image Characterization and Single Frame Measurement pipelines. These two pipelines must be performed in sequence on an image: the first to characterize overall properties of the image, such as the size and variation of the Psf, background and noise; the second to create a detection catalog and measure the positions and magnitudes of the sources contained in the image.

In LSST stack, pipelines are comprised of "stages", and stages are controlled by "policies". For example, the Image Characterization pipeline is made up of this sequence of stages:

```
A stage to remove background and detect bright sources in an image.
A stage to measure bright source fluxes and shapes.
A stage to characterize the Psf from stellar sources..
A stage to model the aperture correction
A stage to provide accurate WCS information
```

Our tutorial code will run these stages in the same way that the Image Characterization pipeline does, doing a first pass to identify stars and characterize the PSF and Aperture Correction. After the image characterization has been done, a simplified form of the Single Frame Measurement pipeline is run to create astrometric and photometric catalogs.

A Simple Example of Running a Stage:

To give the reader the flavor of how you go about calling a Stage, here is an example of a call to the SourceMeasurementStage:

```
exposure = afwExposureF("exposure.fits")
clip = {'inputExposure': exposure}
clip = runStage(measPipe.SourceDetectionStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: inputExposure
    }
    outputKeys: {
        positiveDetection: positiveFootprintSet
        psf: simplePsf
    }
    detectionPolicy: {
        thresholdType: stdev
        thresholdValue: 5.0
        minPixels:  20
    }
    """, clip)
```

In this example, an exposure is first read from disk and a reference to it is placed in a dictionary, called by LSST convention the "**clipboard**". The "inputKeys" specification tells the stage that the input object can be found on the clipboard under the key "inputExposure". The Stage will process the exposure and return a new clipboard which contains the references to the requested output objects. Our output is a set of detected sources, called the "positiveFootprintSet". We also get back a Psf object which was constructed during the source detection process. On return from the runStage call, you should be able to fetch the output FootprintSet as follows:

```
footprintSet = clip['positiveFootprintSet']
```

The object returned on the clipboard is an object of type lsst.afw.detection. FootprintSetF. It is the Python representation of the C++ templatedFootprintSetF class. We will talk more about this class later.

Policies:

When you call a stage, you must supply an lsst.pex.policy.Policy object or a string which can be used to create such a policy (called a policyString). The Policy object is used to pass the Stage configuration information, including its runtime parameter values and the names of its input and output clipboard objects.

The examples in the previous chapter explicitly constructed a Policy object and used the add method to create the individual policy settings , whereas in our Stage examples, all of the policy settings will be specified in a single policyString. Either construction is allowed, so use the one you are most comfortable with.

If you want to see all of the Policy options which can be supplied to this stage, they are defined in the file:

meas_pipeline/*version*/policy/SourceDetectionStageDictionary.paf

where "version" is the particular version number of meas_pipeline which you currently have installed. The paf file tells you what input parameters are available for you to set, which ones are required or optional, what ranges of values are allowed, and what the default values are. The "inputKeys" and "outputKeys" allow you to control the clipboard keys used to send objects to the Stage and to receive objects back on the clipboard.

Files with extension .paf are used in LSST code to specify the Policy structure and defaults for all kinds of code, not just Stages. One extremely useful feature of paf files is that they are allowed to include other paf files, so they can be used to create a hierarchy. For example, the SourceDetectionStageDictionary.paf contains the item:

```
#parameters
backgroundPolicy: {
    type: "policy"
    dictionary: @@meas_utils:policy/BackgroundDictionary.paf
    description: "Parameters needed to estimate background"
    minOccurs: 0
    maxOccurs: 1
}
```

The backgroundPolicy is a Policy which tells LSST how to create a background image from your original image. The dictionary for this policy is found in a separate file:

meas_utils/*version*/policy/BackgroundDictionary.paf

In this way LSST is able to reuse policy definitions throughout the software stack.

4.4 Image Characterization

Now that we have covered the basics of Policies and Stages, let's try it out with the Image Characterization Pipeline. The code we are about to cover is actually a slightly modified version of the Image Simulation pipelines used to test LSST code on simulated LSST images. Changes have been made to accommodate our own test images, and to allow us to insert code between the stages for pedagogical purposes.

The role of the Image Characterization pipeline is to do an initial examination of our exposure, detect stellar objects in the field, and use those objects to characterize overall features of the image. The stage will create models of the background and PSF over the image, correct the astrometric solution, and build a model for aperture correction. At the end of this pipeline, we will save the results of the various stages to disk so that we can use them to do real photometric measurements on our image in section 4.5.

The Image Characterization Pipeline comprises five stages:

```
measPipe.SourceDetectionStage
measPipe.SourceMeasurementStage
measPipe.PsfDeterminationStage
measPipe.ApertureCorrectionStage
MeasPipe.WcsDeterminationStage
```

The detection and source measurement are done here in a first pass to allow us to find the stars on which the last three stages depend. We will cover these detection and measurement stages only briefly, with a more detailed discussion in the next section.

To start the pipeline off, first read the exposure we saved to disk in the last section into an exposure object and create a clipboard whose only entry is the exposure:

```
exposure = afwImg.ExposureF("exposure.fits")
clip = {'visitExposure': exposure}
```

4.4.1 Source Detection Stage

Now call the detection stage with policies set as follows:

```
clip = runStage(measPipe.SourceDetectionStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: visitExposure
    }
    outputKeys: {
        positiveDetection: positiveFootprintSet
        psf: simplePsf
    }
    psfPolicy: {
        height: 15
        width: 15
        parameter: 2. 10.
    }
    backgroundPolicy: {
        algorithm: NATURAL_SPLINE
        statisticsproperty: "MEANCLIP"
```

```
        binsize: 255
        numiter: 10
        numsigmaclip:2.5
    }
    detectionPolicy: {
        thresholdType: stdev
        thresholdValue: 5.0
        minPixels:  20
    }
    """, clip)
```

The policyString in this example is the same as our early one, with the input and output clipboard keys defined by "inputKeys" and "outputKeys". The background, detection, and psf policies are all included from paf files defined in meas_utils. You may see all of the policy controls available for this stage by examining the paf files in meas_pipeline/*version*/policy.

Our stage is called to produce a set of "positive" Footprints which are at least 5 standard deviations above threshold and at least 20 pixels in size. A positive Footprint is a set of contiguous pixels, all of which are above threshold.

Since the detection stage does a psf convolution as part of the detection process, a Psf object must be supplied or created during the detection stage. In our example, we specify a simple 15 x 15 convolution kernel represented with a double Gaussian, and request that it be returned on the clipboard using the key "simplePsf". The input value "2. 10." is the Policy file syntax to indicate two doubles as input values: in this case, it indicates the sigma1 and sigma2 components of the double gaussianpsf, in pixels. This psf only needs to be approximate: it is used in the initial stages of the pipeline, prior to the construction of a more accurate Psf model.

The detection stage must also perform a background subtraction prior to detecting which pixels are above threshold. In this example, the background image is produced using a grid of cells which are **binsize** pixels on edge. The background in each cell of the grid is estimated using a 2.5 sigma clip against the mean with up to 10 iterations. This course background image is then converted into a fine background image using a natural spline.

On return from the SourceDetectionStage, you can examine the output by first looking at the items in the clipboard:

```
 printclip.keys()
['backgroundSubtractedExposure', 'visitExposure', 'background',
 'simplePsf','positiveFootprintSet']
```

Note that even though we did not request a background subtracted image, we received one on the clipboard. That is because the SourceDetectionStage automatically provides the background and background subtracted image on the clipboard unless the backgroundPolicy is set to "NONE". If you want to look at the background object to see what the background algorithm has done with the supplied backgroundPolicy, you can use the following code on return from the SourceDetectionStage:

```
background = clip.['background']
ds9.mtv(background.getImageF())
```

You can also write the image to disk as a fits file:

```
background.getImageF().writeFits("background.fits")
```

If you request a convolution, he Psf which was used will also be returned on the clipboard. A convolution is requested either by providing an input object called "**psf**", or by setting a **"psfPolicy**". The created Psf object is returned using the 'simplePsf' key as we specified in the **outputKeys** section of the policy. To view the Psf:

```
simplepsf = clip['simplePsf']
ds9.mtv(simplepsf.computeImage())
```

4.4.2 Source Measurement Stage

We can now use the products of the SourceDetectionStage to measure our sources. We simply take the clipboard returned by the stage and identify the objects for the next stage that are to be used as inputs.

```
clip = runStage(measPipe.SourceMeasurementStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: backgroundSubtractedExposure
        psf: simplePsf
        positiveDetection: positiveFootprintSet
    }
    outputKeys: {
        sources: sourceSet
    }
    """, clip)
```

The output of the SourceMeasurementStage will be a SourceSet object, which we can fetch on return from the stage in the usual way:

```
sourceset = clip['sourceSet']
print "We found %d sources."%(len(sourceset))
```

However, we do not need to do anything with the SourceSet, except to know that each source has a MeasurementShape associated with it, which will be used by the PsfDetermination Stage.

4.4.3 Psf Determination Stage

The Psf Determination Stage creates a more precise measurement of the spatially varying Psf than our previous simplePsf. To do this, stellar objects are selected from the SourceSet and binned in a 2D grid, called a spatialCellSet.

```
clip = runStage(measPipe.PsfDeterminationStage,
     """#<?cfgpaf policy?>
     inputKeys: {
         exposure: backgroundSubtractedExposure
         sourceSet: sourceSet
     }
     outputKeys: {
         psf: measuredPsf
         cellSet: cellSet
         sourceSet: psfSourceSet
         sdqa: sdqa
     }
     """, clip)
 psf = clip['measuredPsf']
```

If we fetch the CellSet returned by the stage, we find a regular grid of 256x256 pixel cells. Each cell in the set contains a list of candidate stars, which we can look at on a ds9 image using the following code, with the sources in each grid cell color coded.

```
ds9.mtv(exposure.getMaskedImage().getImage())
cellSet = clip['cellSet']
cellindex = 0
colors=(ds9.RED,ds9.GREEN,ds9.YELLOW,ds9.CYAN,ds9.BLUE)
for cell in cellSet.getCellList():
    cellindex = cellindex + 1
    color = colors(cellindex % len(colors))
    forcand in cell.begin(True): # ignore bad candidates
        x, y = cand.getXCenter(),cand.getYCenter()
        ds9.dot("O",x,y,size=5,ctype=color)
```

The Psf representation itself is returned in the "psf" key. It may represent the Psf in a variety of ways, depending on the Psf class implementation. While it is outside the scope of this document to discuss the Psf model, you may view an image of it at the origin of your pixel coordinate system as follows:

```
ds9.mtv(clip['psf'].computeImage(afwGeom.makePointD(0.0,0.0)))
```

4.4.4 Compute Aperture Correction Stage

An Aperture Correction object can be created: this object is used to correct source photometry. The use of this object will be explained in more detail in section 4.5.3.

The Aperture Correction object is a spatially varying correction, and like a Psf object, uses a spatially binned CellSet to create its model. If you provide the Stage with a SourceSet, it will create a CellSet from it on its own. But since a CellSet is already available from our previous stage, we can use it as an input.

```
clip = runStage(measPipe.ApertureCorrectionStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: backgroundSubtractedExposure
        cellSet: cellSet
    }
    outputKeys: {
        apCorr: apCorr
        sdqa: sdqaApCorr
    }
    """, clip)
```

The ApertureCorrection object is returned on the clipboard in the specified key, and since we will need to use this object later to correct aperture photometry on each object depending on the spatially varying Psf,, we will fetch this object off of the clipboard and save it to disk as a Python pickle:

```
import pickle
apcorr = clip['apCorr']
fout = open('apcorr.pickle', 'w')
pickle.dump(apcorr, fout)
fout.close()
```

4.4.5 Wcs Determination Stage

The WcsDeterminationStage is needed if Wcs information is not already attached to the image, or if the accuracy of the Wcs information is suspect. Wcs information is usually recorded in the image header when an observation is made, but may later be calibrated more exactly using a standard astometric catalog, such as the USNOB-1 catalog or the SDSS catalog.

The WcsDeterminationStage uses **astrometry.net** code to create a TAN-SIP representation of our image, given the catalog of sources we have already created. A version of astrometry.net is contained in the LSST package astrometry_net. If you have not used this package before, please see the section entitled "Setting up the Astrometry.Net Indexes" at the end of this section.

In the case of our KPNO images, there is a good reason why be can't keep the Wcs information in the original fits file. The original image header from KPNO had a TAN-TNX projection which can't be read by the LSST software stack. Rather than attempt to preserve this information from the header, we will reconstruct the distortion correction using the WcsDeterminationStage.

Since the Wcs is not usable with the current LSST pipeline, our strategy will be to create a rough Wcs using the information in the fits header, feed that Wcs to the WcsDeterminationStage, and let the stage fine time the information. Our header has:

```
CTYPE1  = 'RA---TNX'           / Coordinate type
CTYPE2  = 'DEC--TNX'           / Coordinate type
CRVAL1  =      139.04792062781 / Coordinate reference value
CRVAL2  =      30.039519922129 / Coordinate reference value
CRPIX1  =    -2182.11820383008 / Coordinate reference pixel
CRPIX2  =     4132.22507160628 / Coordinate reference pixel
CD1_1   =  -6.9484856570845E-7 / Coordinate matrix
CD2_1   =  -7.1254354314285E-5 / Coordinate matrix
CD1_2   =  -7.1679894102440E-5 / Coordinate matrix
CD2_2   =  -3.5512019342818E-7 / Coordinate matrix
```

We can use the CD matrix and center points in the header as follows, neglecting the TNX corrections:

```
crval = geom.makePointD(139.0479,30.0395)
crpix = geom.makePointD(-2182.1, 4132.2)
cd11 = 0.0
cd21 = -7.13e-5
cd12 = -7.18e-5
cd22 =  0.0

wcs = afwImg.createWcs(crval,crpix,cd11,cd12,cd21,cd22)
clip['backgroundSubtractedExposure'].setWcs(wcs)
```

Assuming the indexes are properly set up, you can run the astrometry.net code to determine an accurate Wcs for your image.

```
clip = runStage(measPipe.WcsDeterminationStage,
    """#<?cfgpaf policy?>
    inputExposureKey: backgroundSubtractedExposure
    inputSourceSetKey: sourceSet
    outputWcsKey: measuredWcs
    outputMatchListKey: matchList
    numBrightStars: 150
    defaultFilterName: mag
    """, clip)
```

The stage takes as its inputs a SourceSet and Exposure and outputs a Wcs object and MatchList. The numBrightStars and defaultFilterName are parameters for the solver. 'numBrightStars' is the number of stars the solver should attempt to use, extracted from the SourceSet. 'defaultFilterName' is the name of the column from the Astrometry.Net comparison catalog which should be used to compare with the input image. The column is named "mag" in our example.

If the WcsDeterminationStage is able to find the pattern of sources in our catalog within the tables provided by lsst.meas.astrom, it will return an object on the clipboard of type afwImg.Wcs, which can be fetched using the "measuredWcs" key:

```
wcs = clip['measuredWcs']
```

The Wcs information is also appended to the input exposure, and can be fetched with the getWcs() method.

The stage also returns a **matchList**, a mapping between members our Source Set and the stars listed in the astrometry.net indexes. If you like, you can compare the matched objects as well as list the distance between them in arcseconds using the matchList.

In the next section, we will use the Wcs object to convert the Pixel Coordinates in our images to RA and DEC. The Wcs class has many useful routines available for transforming between coordinate systems, the simplest of which can be used to do pixel to-Wcs and the reverse transformations, such as in the code below:

```
importlsst.afw.Coord as Coord
printwcs.pixelToSky(0,0).toFk5().getDecStr()
30:02:07.92
printwcs.pixelToSky(0,0).toFk5().getRaStr(Coord.HOURS)
09:17:33.33
```

Saving the Exposure with WCS Information:

While the Wcs information can be saved on its own, it is also appended to the Exposure object which was sent to WcsDeterminationStage. We can save both the exposure and its associated Wcs information as a fits file:

```
calexp = clip['backgroundSubtractedExposure']
calexp.writeFits('calexp.fits')
```

A dump of the fits header might appear as follows:

```
EQUINOX = 2000.0000000000 // Equinox of coordinates
RADESYS = "FK5" // Coordinate system for equinox
CRPIX1 = 881.04156212955 // WCS Coordinate reference pixel
CRPIX2 = 2074.0829942426 // WCS Coordinate reference pixel
CD1_1 = -3.7860955487717e-07 // WCS Coordinate scale matrix
```

```
CD1_2 = -7.2190393375173e-05 // WCS Coordinate scale matrix
CD2_1 = -7.2496486443239e-05 // WCS Coordinate scale matrix
CD2_2 = 1.1228178931884e-07 // WCS Coordinate scale matrix
CRVAL1 = 139.21727125798 // WCS Ref value (RA in decimal degrees)
CRVAL2 = 29.972397382831 // WCS Ref value (DEC in decimal degrees)
CUNIT1 = "deg"
CUNIT2 = "deg"
A_ORDER = 4
A_0_2 = -3.6365798894617e-07

B_ORDER = 4
B_0_2 = 3.5629731526012e-06

AP_ORDER = 5
AP_0_0 = 0.066716813851651

BP_ORDER = 5
BP_0_0 = 0.083261391798342

CTYPE1 = "RA---TAN-SIP" // WCS Coordinate type
CTYPE2 = "DEC--TAN-SIP" // WCS Coordinate type
```

Setting up the Astrometry.Net indexes

The WcsDeterminationStage uses code in lsst.meas.astrom.net, which in turn relies on a set of fits tables in the package astrometry_net_data. These fits tables contain information about patterns of sources on the sky which can be used to find a "solution" for our image. The program either starts with no information at all about your image (blindSolve), or with the exisitingWcs. It will not only locate your image RA,Dec and CD matrix, but also calculates a TAN-SIP projection with astometric corrections.

But before the WcsDeterminationStage can work correctly, you must provide it with a set of Astrometry.Net "indexes" which are suitable for your images. There are a wide variety of ways to fetch or make images for your particular data set. The simplest way to get indexes is to go to the Astrometry.Net website and request access to the indexes made from the USNOB1 catalog. You will not need all of the indexes: only the ones which are required for the size of your images.

Since the astrometry.net code looks for distinctive patterns of objects which might be contained in your exposure, it is important that the patterns in your indexes be of the same "scale" as your images. Too large a scale and the patterns won't be present in your field of view; too small a scale and the search will be inefficient. To read more about the indexes and which ones your will need for a given set of images, see the documentation:

LSST/Linux64/external/astrometry_net/0.30/doc/README

For our 2K x 4K image which is about 9 x 18 arcminutes, we will probably need the indexes 203-205. These files should be downloaded and placed in the root of the astrometry_net_data package directory. Use "eups list" to find the location of this directory if you are not sure. You must also alter the file **metadata.paf** to contain the names of all the files you have downloaded. Your paf file might look like this:

```
#<?cfgpaf dictionary ?>
#
# Specify the coordinate system upon which these indices are based.
#

equinox: 2000.0
raDecSys: "FK5"

#Index files to load. Paths are give relative to $ASTROMETRY_NET_DATA_DIR
indexFile : index-203.fits
indexFile : index-204.fits
indexFile : index-205.fits
```

4.5 Source Measurement

We will now use the Image Characterization outputs produced in the last section to create a catalog of sources for our image. The last section was a preliminary pass through the image during which we did background subtraction, developed a Psf Model over the image, and created a Wcs object. We also created an ApertureCorrection object which we will use to perform aperture correction on the photometric measurements we do in this section.

First read in the results from the last section and put them on the clipboard:

```
exp = afwImg.ExposureF("calexp1.fits")

fin = open('apcorr.pickle','r')
apCorr = pickle.load(fin)
fin.close()

additionalData = dafBase.PropertySet()
pol = policy.Policy()
storageType = "Boost"
loc = dafPersist.LogicalLocation("psf.boost")
persistence = dafPersist.Persistence.getPersistence(pol)
storageList2 = dafPersist.StorageList()
storage2 = persistence.getRetrieveStorage("BoostStorage", loc)
storageList2.append(storage2)
psfptr = persistence.unsafeRetrieve("Psf", storageList2, additionalData)
psf = afwDetect.Psf.swigConvert(psfptr)

clip = {
```

```
    'scienceExposure': exp,
    'psf': psf,
    'apCorr': apCorr
}
```

4.5.1 Running Source Detection

We can now run the Source Detection Stage on our exposure again, but this time with a realistic Psf model as a convolution filter, and with a much lower detection threshold than with our call to this stage in section 4.4 In this example, we measure objects which are at least 3 sigma, and with a minimum area of 6 contiguous pixels. Since background subtraction has already been performed on this image, it was not requested.

```
clip = runStage(measPipe.SourceDetectionStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: scienceExposure
        psf: psf
    }
    outputKeys: {
        positiveDetection: positiveFootprintSet
    }
    backgroundPolicy: {
        algorithm: NONE
    }
    detectionPolicy: {
        thresholdType: stdev
        thresholdValue: 3.0
        minPixels:      6
    }
    """, clip)
```

The results are returned on the clipboard using the "positiveFootprintSet" key. Source detection allows either positive or negative detections: in this case, we are only interested in the positive ones, or detections above threshold.

What is a Footprint?

The SourceDetectionStage returns a FootprintSet, a collection of Footprint objects. Each footprint is a set of contiguous pixels above threshold, similar to a Sextractor "segment". The following code shows you how to access the Footprints in a FootprintSet. Each Footprint has a "span" for each row that it occupies, and each span has a beginning and ending column position:

```
fpset = clip['positiveFootprintSet']
```

```
footprints = fpset.getFootprints()
iflen(footprints) > 0:
    footprint = footprints[0]
    spans = footprint.getSpans()
    for j in range(len(spans)):
        print spans[j],spans[j].getX0(),spans[j].getX1()
```

Viewing the segmentation map:

You may find it instructive to view the segmentation map as an image, using the following code. You can also save the image to a fits file using the writeFits method, as in our earlier examples.

```
ds9.mtv(fpset.insertIntoImage(True)
```

The call to insertIntoImage generates a new image (with the same dimensions and origin as the image we detected on) and writes the footprint's id at every pixel it covers.

The effect is to produce an image with each footprint displayed as a uniquely numbered region.

4.5.2 Doing the Source Measurement:

Finding a PositiveFootprintSet is the first step towards building a catalog for our image. The next stage will examine each of the footprints and measure the PsfFlux for each object. The output of the measurement stage is a "SourceSet". There is one Source in the SourceSet for each Footprint in the FootprintSet, but whereas Footprints are just pixel sets, Sources have measured centroids, shapes, and fluxes.

In our call below to the SourceMeasurementStage, the psf is one of the input keys, as the default measurement algorithm needs the Psf to create Psf Fluxes.

```
clip = runStage(measPipe.SourceMeasurementStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        exposure: scienceExposure
        psf: psf
        positiveDetection: positiveFootprintSet
    }
    outputKeys: {
        sources: sourceSet
```

```
    }
    """, clip)
```

While the Footprint gives an isophotal region and a bounding box, the Source object gives more exact information about the source centroid and shape:

```
ss = clip.getItem("sourceSet")
 fps = clip.getItem("PositiveFootprintSet")
fp = fps[100]
printfp.getBBox()
(1129, 89) -- (1132, 93)
source = ss[100]
printsource.getXAstrom(),source.getYAstrom()
1130.42326491 90.7022056676
printsource.getIxx(),source.getIxy(),source.getIyy()
0.677250247005 0.341197346808 3.79150529619
```

NOTE: The Source class is capable of holding many different pieces of information about the position, size and shape of a given detection. When initially constructed, a Source will only have a subset of all the information it might possibly contain. Later stages take the SourceSet as an input and fill in information about the Sources. Prior to a value being set, the Source methods "getter" for that value will return 0.0.

For example, prior to calling the ComputeSourceSkyCoordsStage, the Source methods for fetching Wcs coordinate return default 0.0 values:

```
printsource.getRa(), source.getDec()
    0.0   0.0evinc
```

4.6 Applying Aperture Correction:

We can now apply that aperture correction to the fluxes.

```
printsource.getPsfFlux(),source.getPsfFluxErr()
1688.14037746 341.339111328

clip = runStage(measPipe.ApertureCorrectionApplyStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        apCorr: apCorr
        sourceSet: sourceSet
    }
    """, clip)
```

The aperture correction can be seen to modify the psfFlux on items in the SourceSet. The code below prints the PsfFlux and error of our Source before and after the aperture correction is done. The PsfFlux and error have been reduced by roughly 6

```
printsource.getPsfFlux(),source.getPsfFluxErr()
1595.27814881 329.345733643
```

You can learn more about what the apCorr object does by calling the object at the position of our source:

```
apc = clip.getItem("apCorr")
printapc.computeAt(x,y)
(0.94499140599311726, 0.041684436096061615)
```

The aperture correction model return both a correction coefficient and an error. The PsfFluxErr has been correctly recalculated by propagating errors from both the PsfFlux measurement and the aperture correction.

4.6.1 Adding Sky Coordinates:

The Wcs object we found in the last section and added to the exposure object can now be used to add FK5 coordinates to the catalog. This pipeline stage can be called explicitly with a Wcs object, but if a Wcs object is attached to the Exposure, it will be used by default. For that reason, the "wcs" keyword is optional in this example:

```
clip = runStage(measPipe.ComputeSourceSkyCoordsStage,
    """#<?cfgpaf policy?>
    inputKeys: {
        sources: sourceSet
        exposure: scienceExposure
        wcs: measuredWcs
    }
    """, clip)
```

Prior to running this stage, the source objects have no Wcs coordinates, and the getRa() and getDec() methods return 0.0. On return from this stage, you can get values from

```
printsource.getRa(),source.getDec()
2.43265225585 0.52279899529
```

These methods return the position in RADIANS: afwCoord.radToDeg can we used to conver to degrees.

4.6.2 Writing a catalog to disk

j

In the LSST pipeline, objects are typically saved to disk using one of the object serialization techniques. This our simplified example the results of our calls will be saved to a text file. You would probably do this in real pipeline only if you needed to create an export file for non-LSST tools.

The last two stages have not done anything to our clipboard keys. We still have the same objects as were present at the end of the SourceMeasurementStage. However, the previous two stages have added information to the source set which we will now access to create a finished catalog.

```
printclip.keys()
['apCorr', 'sourceSet', 'sourceSet_persistable', 'psf', 'scienceExposure', 'backgroundSubtractedExposure',
 'convolvedImage', 'positiveFootprintSet']


ss = clip.getItem("sourceSet")
fpset = clip.getItem("positiveFootprintSet")
fps = fpset.getFootprints()
iflen(fps) != len(ss):
        raiseStandardException("Number of footprints and number in source set do not agree")

fout = open("measure1.cat", "w")
fout.write("# ttype1 = ra\n")
fout.write("# ttype2 = dec\n")
fout.write("# ttype3 = id\n")
fout.write("# ttype4 = x\n")
fout.write("# ttype5 = y\n")
fout.write("# ttype6 = psfflux\n")
fout.write("# ttype7 = psffluxerr\n")
fout.write("# ttype8 = npix\n")
fout.write("# ttype9 = x0\n")
fout.write("# ttype10 = y0\n")
fout.write("# ttype11 = x1\n")
fout.write("# ttype12 = y1\n")

fori in range(len(fps)):
        fp = fps[i]
        source = ss[i]
        bb = fp.getBBox()
        ra = source.getRa()*Coord.radToDeg
        dec = source.getDec()*Coord.radToDeg
        id = source.getId()
        x = source.getXAstrom()+1
        y = source.getYAstrom()+1
        psfflux = source.getPsfFlux()
        psffluxerr = source.getPsfFluxErr()
```

```
        fout.write("%.6f %.6f %d %.2f %.2f %.4f %.4f %d %d %d %d %d\n"
 %(ra,dec,id,x,y,psfflux,psffluxerr,fp.getNpix(), bb.getX0(),
   bb.getY0(), bb.getX1(), bb.getY1()))
fout.close()
```

Chapter 5

How do I...

We've seen that when browsing the online documentation, it's quite easy to look at a C++ definition and see how to use it in Python and what it's useful for. It's much harder to go the other way: given a goal, what are the classes that enable that goal? This chapter provides answers for common tasks. If you encounter a task you think should be listed here, just send it to us and we'd be happy to include it.

5.1 python

5.1.1 Importing an LSST package in python

on the surface, importing an LSST package is just like importing any other python package:

```
import lsst.afw.image
import lsst.meas.algorithms
```

However, if you look a little harder you'll see that there's something odd going on; `lsst/afw/image` and `lsst/meas/algorithms` are not subdirectories of the same parent (they refer to `$AFW_DIR/python/lsst/afw/image` and `$MEAS_ALGORITHMS_DIR/python/lsst/meas/algorithms` respectively); that's not something that python's `import` command understands. What's more, there is no file `$AFW_DIR/python/lsst/__init__.py`.

The trick is that there *is* a file `$BASE_DIR/python/lsst/__init__.py`, and it installs a custom package importer into `sys.meta_path` (this requires python 2.5; the details are in PEP 302 if you care)[1]. This importer is not restricted to loading a sub-module from a subdirectory of the directory where its parent module was loaded from. Thus, if your sys.path contains multiple directories that contain an `lsst` module, then a submodule `lsst.foo` can appear below any of those directories.

[1] There's similar code in `sconsUtils` too, as it too has an lsst directory that needs to be imported

Chapter 6

Documentation

6.1 Documentation Overview

LSST software is documented in two fundamental ways:

The LSST Developer's Manual This is a single document, written in LaTeX, and is intended to be a 'manual' in the traditional sense of the word. It's available online in HTML format, and also as a PDF document. You're currently reading the Developer's Manual! The HTML version of the current manual can be found online at **TODO: add location of html manual**, and the PDF version is availabe at: **TODO: add location of pdf manual**.

The Doxygen source code documentation This is a source code reference, containing summaries of the various classes and functions, and their associated parameters (ie. arguments, return values, etc.). It is an HTML (online) reference which allows a developer to look up the application programming interface (API) for a given class or function, and to link quickly to other related classes and parameters. Doxygen is machine generated directly from source code, with additional information (variable definitions, brief summaries, etc) taken from specific comments included by the programmer. Because of the way it is used ('browsing' source code), it is available only in HTML. The current LSST Doxygen documentation can be found at: `http://dev.lsstcorp.org/doxygen/trunk/`. For more information about Doxygen software, see `http://www.doxygen.org`.

Each of these forms of documentation is available for individual products, and for the entire LSST project. Here, we'll describe the details of the LSST documentation system.

6.2 The Main Documentation Product, `devenv/doc`

The main documentation product lives in `devenv/doc`, and much of the documentation contained in the Developer's Manual can be found here. However, the parts of the manual dealing with individual products are stored with the relevant products, and are pulled into the manual when it is compiled.

Conversely, there is very little Doxygen documentation stored in `devenv/doc`. When built, the product collects Doxygen configuration files from all available (`setup`) products and creates a single comprehensive Doxygen reference.

The documentation product differs in structure from other standard LSST products, containing only `doxygen/` and `latex/` directories.

6.2.1 The Main LaTeX Documentation

The highlights of the `latex/` directory include:

`lsstManual.tex` This is the central LaTeX document that gets built. However, it contains only `\input` statements for the various chapters.

`lsstPackages.tex` This is a machine-generated file which `\input` s the LaTeX documentation for the individual LSST products. Its creation is handled in the `latex/SConscript` file, where the absolution paths to the currently setup product documentation are prepended to the `\input` filename.

***everything-else*`.tex`** The remaining `*.tex` files contain the chapters which are imported into `lsstManual.tex`.

`figures/` This subdirectory contains any figures used in the manual.

`htmlDir/` This subdirectory contains an HTML version of the manual, generated with `latex2html`. The directory and its contents will not exist before `scons` is run.

If you'd like to edit existing documentation, the rest of this section is unlikely to be of interest (or use) to you. All you have to do is find the appropriate `.tex` file and make your changes. Then run `scons` to build the new manual.

If you'd like to add a new chapter, simply put your chapter in an appropriately named `.tex` file and add an `\input` statement to `lsstManual.tex` to import it.

If you'd like to understand how LaTeX from the individual products gets imported into the top-level manual, read on.

The importing of the individual product documentation requires a bit of explanation. The system is designed such that the final document which gets built (by scons) will pull in the individual product documentation from any products which were `setup` (using the `eups` '`setup`' command).

As alluded to in the file explanations above, these `.tex` files for each product are imported in the `lsstPackages.tex` file. During the build, the `latex/SConscript` code determines which products are `setup` (and *where*) and generates the `lsstPackages.tex` file with an `\input` statement for each. However, if you look in the `lsstPackages.tex` file, that's not what you'll see.

It's entirely possible that each imported `.tex` file will also contain `\input` statements for files in its own directory tree. To deal with this, *all individual products must only use our own* `\lsstinput` *command!* The `\lsstinput` command is a thin wrapper for the regular LaTeX `\input` command. For each imported product, the `lsstPackages.tex` file contains two lines: the first renews the `\lsstinput` command to prepend the absolution path for the product in question, the second then `\lsstinput`'s the product.

When `scons` is run, the LaTeX is compiled and `lsstManual.pdf` is created in the root directory of `devenv/doc`. At the same time, an HTML version of the manual will be created in `latex/htmlDir`.

6.2.2 The Main Doxygen Documentation

Doxygen builds its documentation by digesting the source code for a project. The construction of the LSST doxygen is controlled by a `doxgyen.conf` file which identifies the locations (directories) of the source code to be scanned. The `devenv/doc doxygen/` directory contains no `doxgyen.conf` file, but rather a python script `writeDoxyConfig.py` which is executed when `scons` is run. As you might suspect, `writeDoxyConfig.py` scans all available `doxgyen.conf` files (ie. those `setup` via `eups`) and generates a master `doxgyen.conf`. This machine-generated file is then used to build the top-level Doxygen. This happens when `scons` is run and is handled in the `doxygen/SConscript` file.

When `scons` is run, the Doxygen output will be created in `doxygen/htmlDir`.

6.3 Documenting an Individual Product

The documentation for each individual product is within the product's `doc/` directory. There, you'll find a `doxgyen.conf` file (which you should have no need to edit), and a `latex/` subdirectory.

6.3.1 Standard LSST Doxygen Coding Practices

The LSST Coding practices are described in detail on the LSST Trac wiki. To avoid duplication (and inevitable inconsistency), we simply refer you to: `http://dev.lsstcorp.org/trac/wiki/DocumentationStandards`.

When `scons` is run, the Doxygen HTML will be written to `doc/htmlDir/`.

6.3.2 Contributing to the Manual Entry for a Product

The `doc/latex/` directory should contain at least two `.tex` files: one named after the product itself (eg. `meas_astrom.tex`, `afw.tex`), and one named `package.tex`. **Documentation should go in the `package.tex` file**. It is the `package.tex` file which will be imported into the top-level manual when `devenv/doc` is built. The `.tex` file bearing the name of the product is just a thin wrapper which `\input`'s the `package.tex` file to build a local PDF copy of the manual for the product.

If you'd like to put your section/subsection in a separate `.tex` file, feel free to do so. However, **in order for the documentation to be properly imported into the top-level manual, you must use the `\lsstinput` command rather than the normal LaTeX `\input` command!**

Not all products are documented in the LaTeX manual, so you may find that there's no `doc/latex/` directory in a some products. If you'd like to add one, please do!

Here's a checklist of things to do to add latex documentation to a product:

1. Create the `doc/latex/` directory.
2. Create `doc/latex/`*`myproduct`*`.tex` (adapt it from a product which already has documentation).
3. Create `doc/latex/package.tex` with your contribution to the manual.
4. Create `doc/latex/SConscript` to build it (again, adapt it from the `SConscript` file in a product which already has documentation).
5. Edit `SConstruct` to call the new SConscript.

Appendix A

Installing the LSST Software Stack

A.1 How do I install all this stuff?

The place to start is `http://lsstdev.ncsa.uiuc.edu/trac/wiki/Installing`. From here you can jump to *Installing on Linux* or *Installing on Macintosh*. Don't worry that the list of officially supported platforms is short and probably doesn't include yours; in practice, the stack should run on most Linux or Mac systems with only minor tweaks, and most developers actually work on "unsupported" platforms! The "supported" moniker indicates that when push comes to shove, the highest priority for the project is that the code runs on the big Linux clusters that will be used for production. But if you need help on an "unsupported" platform, ask around and you will most likely get help.

The code does tend not to work on the absolute latest versions of things (for example MacOS 10.6 and Linux versions which use gcc 4.4, although there is an easy workaround for the latter). Machines with middle-aged OS's like MacOS 10.5 and Linux versions with gcc 4.3 are going to work out of the box.

This author installed on Ubuntu 10.04, so we will go through the process step-by-step for that OS. Clicking on *Installing on Linux*, we get to a page which (at the bottom) lists the OS packages which must be present before the stack can be installed; for Debian/Ubuntu, we see g++, subversion, etc. Copy and paste the apt-get command for installing all these packages. It may vary slightly for other versions of Ubuntu/Debian, as package names sometimes change.

With that done, rejoin the OS-independent part of the install instructions at `http://dev.lsstcorp.org/GettingStarted.html#lsst_home`. They ask you to define a few environment variables, choose a directory to install into (user space is probably best), grab the install script from the given URL, and run it. That's it. It should take at least 60 minutes to download and compile everything.

A warning: the install script may say "Installation complete" at the end even if previous steps have failed! Be sure to skim the output of the install script to see if everything went ok. In the case of Ubuntu 10.04, afw fails to compile; this turns out to be a known issue with a known, easy fix[1] described at `http://dev.lsstcorp.org/`

[1] So why doesn't the project automatically apply the fix? Because it only occurs with gcc 4.4, which hasn't been officially adopted yet. By the time it is, a new version of the third-party library with the relevant bugfix will also be adopted.

`trac/ticket/1124` (note: you may need a login to see this part of the dev.lsstcorp.org site). After applying the fix, the install script runs to completion.

You may wish to scroll down the install instructions page for a brief explanation of the subdirectory structure that's been installed. This explanation is most relevant for power users who will be running multiple versions and platforms.

To use the software you must perform two configuration steps:

1. Tell Python where to find the LSST packages by running `source /your/install/path/loadLSST.csh` (or loadLSST.sh for bash users). You can test this by starting Python in interactive mode (just type `python`) and then telling Python `import lsst`. If that doesn't fail with an error (sucess will be silent), then loadLSST.csh has done its job.
2. For each package you wish to use, specify which version of that package you wish to use by running `setup [packagename]` on the Unix command line, for example `setup afw`. Setup is designed to help developers switch different versions of, eg, afw in and out for testing. Most readers of this document will not wish to do that, but will still need to run setup without specifying any particular version; this automatically pulls in the latest version. You can test this step by again entering Python in interactive mode and typing `import lsst.afw`. Note that the package names you specify to setup are less hierarchical than the corresponding Python or C++ namespaces. For example, you would `setup meas_astrom` and then `import lsst.meas.astrom`. You do *not* `setup meas::astrom` or `setup meas.astrom`.

A.2 Routinely updating your LSST Software Stack

The pseudo-package LSSTPipe contains everything you need to run the pipeline; you can re-install at any time to get the latest versions of required pipeline packages, and declare them as current (the default version to be used): `lsstpkg install --current LSSTPipe`

When installation is complete, you must run `setup` on each newly installed package: `setup afw` Appending `--current` to the end declares your freshly installed version of `afw` as current: `setup afw --current`

If you wish to setup a different version which is already installed but not declared current: `setup afw 3.5.2`

NOTE: If there is no revision declared current at all, `setup` will fail and print a message that it doesn't have a current version of that package.

You will most likely find it helpful to understand what the digits in the version numbers mean:

```
3.0.1
    |
```

Indicates minor revisions, such as bug fixes, which can happen quickly and often.

```
3.0.1
  |
```

If this number changes, everything that depends on the package must be upgraded as well.

A.3 Update after a critical bug fix

Sometimes, after a critical bug fix, you might need this latest version before it is available on the distribution server, which means that re-installing LSSTPipe will not deliver the most recent revision. Here are some notes on what to do in these circumstances.

- To view all revisions of a package: `lsstpkg list afw`
- To list package versions installed on your local machine; this command lists all tags, such as `Current` or `Setup`, associated with that particular package: `eups list afw`
- To install a new package, or a new revision of a package, say afw 3.6.0: `lsstpkg install afw 3.6.0`
  - To also declare this revision as current: `lsstpkg install --current afw 3.6.0`

Now you are ready to try the hello world examples in Chapter 1!

Appendix B

Quick Summary of Object-Oriented Programming Lingo

If you are unfamiliar with object-oriented programming (OOP), this is an *extremely* brief summary, just enough to define some terms so that you can start reading this manual fruitfully. You are advised to consult other references to gain a deeper understanding.

An *object* or a *class* is a software construct that contains both data and algorithms (called *methods* for manipulating that data. (A key insight of OOP is that keeping data and algorithms together makes software more robust and reusable.) For example, the *image* class contains data members such as number of pixels in each dimension, and memory for containing the actual pixel values. When the name of the class is invoked on the right-hand side (for example, `im1 = afwImg.ImageF('image1.fits')`), it serves as a *constructor*, which allocates and initializes the data members (or *instantiates* the object) and returns a reference to the object.

The image class also contains methods such as multiplication by a scalar and multiplication by another image. Obviously, the internal workings of " multiplication by a scalar" and "multiplication by another image" are quite different. But we can present a simple interface to people using our class, so that `im1 *= 2` invokes " multiplication by a scalar" and `im1 *= im2` invokes "multiplication by another image." This is known as *operator overloading*.

Inheritance is another big idea of OOP. We can define *subclasses* which inherit the properties of their parent classes, but tweak certain details. As an example, if operation `foo` can be done blindingly fast on square images but not on general rectangular images, we would write the slow version of `foo` for the `image` class, and then define a `squareimage` subclass which inherits all the methods of an `image` but overloads `foo` with its own blindingly fast algorithm. This is great for code reuse, because now if we add a new feature to (or fix a bug in) the `image` class, it *automatically* also gets incorporated into `squareimage` via inheritance. We don't need to touch the `squareimage` subclass.

Note that classes often have a variety of constructors. For example, `im1 = afwImg.ImageF('image1.fits')` calls a constructor which creates an in-memory `image` from a FITS file, but `a = afwImg.ImageF(10,10)` creates an in-memory `image` knowing only the desired dimensions (initializing the pixel values to zero). By looking at the types of the arguments starting with the first one, you will be able to deduce which method is actually called

in any given case. The error message you get when you use a constructor incorrectly is not very specific. You might get a list of all possible constructors, which is not very helpful when there are dozens and you just want to know where your one little mistake is. This is rather different from say, C, where the compiler knows what type the nth argument should be, and tells you quite specifically when you give it the wrong type.

Appendix C

Using Python Interactively

Python is able to remember your history from one session to another, and to use TAB to help you with your typing by completing variable names and object methods.

One way to get these features is to use `iPython` (`http://ipython.scipy.org/moin/`). If you want to avoid installing another package, you can home-brew your environment.

- Create a file `$HOME/.pythonstartup`
- Set the environment variable `PYTHONSTARTUP` to point there, e.g. `export PYTHONSTARTUP=$HOME/.pythonstartup` (or the csh equivalent with `setenv`)
- Put the following lines in `$HOME/.pythonstartup`:

```
import re, os

# The place to store your command history between sessions
histfile=os.environ["HOME"] + "/.python-history"

try:
    # Try to set up command history completion/saving/reloading
    import readline, atexit, rlcompleter

    isEditline = False
    try:
        if re.search(r"^/System/Library", readline.__file__): # editline; the BSD rewrite of readlin
            isEditline = True
    except AttributeError:
        pass

    if isEditline:
        readline.parse_and_bind("bind ^I rl_complete")
```

```
        readline.parse_and_bind("bind ^R em-inc-search-prev")
        readline.parse_and_bind("bind ^U ed-move-to-beg em-next-word")
    else:                               # readline
        readline.parse_and_bind('tab: complete')

    readline.set_history_length(-1)     # don't truncate history list

    try:
        readline.read_history_file(histfile)
    except IOError:
        pass  # It doesn't exist yet.

    def savehist(nsave=1000):
        try:
            readline.set_history_length(nsave)
            readline.write_history_file(histfile)
        except Exception, msg:
            print 'Unable to save Python command history:', msg

    atexit.register(savehist)
    del atexit
except ImportError:
    pass
```

That may look complicated, but it should work on os/x as well as linux boxes. And you should see the version that I *really* use...

The source to this appendix is at `http://dev.lsstcorp.org/trac/browser/DMS/devenv/doc/trunk/latex/interactivePython.tex`, from where you should be able to cut-and-paste the suggested contents of `$HOME/.pythonstartup`.

Appendix D

Overview of Third-Party Tools

The LSST C++ programmers themselves build upon numerous open-source libraries and tools. This appendix gives a brief overview of those tools. You do not have to read it thoroughly to make sense of the rest of this document, but it may be useful to have some reference point for the various terms that may appear when you install and run the software. LSST has blessed certain versions of these things as the official LSST versions, which are installed in one big gulp by the procedure described in Appendix A. Thus you do not have to follow the URLs in this appendix; they are for your information only.

D.1 SCons

Think of SCons (`http://www.scons.org`) as the new "make." (The name comes from "software construction.") Why don't we simply use make? Because SCons is more powerful; an SCons script *is* a Python script, so it has the full power of Python when necessary, but it automates the routine jobs. That said, SCons will take some getting used to if you are writing and building C++ code. Those of you sticking to Python will not need it.

D.2 EUPS

EUPS is the Extended Unix Product System. It's a way to manage all of the versions of all of the libraries and tools you may have on your system. For example, if you already have NumPy installed but it's not the official LSST version, you need to set up various paths so that when you run LSST software, it uses the expected version. You do *not* have to delete your personal version, which you may need for other projects. (LSST software may in fact run with your pre-installed version, but that is not guaranteed, and you will not receive much help from the project if you encounter problems this way.) EUPS is designed to make this easy even in the presence of all the various LSST and third-party packages with all their versions and running on many different flavors of OS. EUPS grew out of a similar tool written by astronomers for the Sloan Digital Sky Survey, and is now maintained by LSST. Its home page appears to be `http://dev.lsstcorp.org/trac/wiki/Eups`.

D.3 svn

As a user rather than an LSST developer, you may never have to use svn, but you will likely hear references to it. SVN (short for "subversion", `http://subversion.tigris.org`) is a modern software repository system. When developers check in new versions of code, svn automatically saves the old version, records who checked in the new version, can generate diffs when requested, etc. And it's not limited to code; the writers of the LSST Science Book used svn to coordinate the contributions of hundreds of authors. There is a distinction between svn the software on one hand, and any particular svn repository on the other. LSST DM has one repository, the LSST Science Book has another, and thousands of other software projects have their own svn repositories. An example of the kind of thing EUPS should do is set up an environment variable that tells svn where the LSST DM repository is, rather than force you to type it on the svn command line.

Most large software projects make a public release by extracting a particular snapshot of their svn repository and making it downloadable in, say, one big gzipped tar file. Users who download it thus do not need an svn client; access to the bleeding-edge version of the code would actually be harmful for many people. LSST software is not quite this publicly-accessible, but XXX

D.4 Other third-party packages

The LSST software stack makes use of many other third-party packages, for example Boost, a set of general-purpose C++ libraries; cfitsio; swig (a system for making C++ and Python, among other languages, work together); Python and various Python modules; etc. You probably don't need to worry about any of these in any detail, but you will likely see their names when you drill down a little deeper than this document.

Appendix E

Tips on Debugging

E.1 Turning on debugging output (often image display)

Many (but not all) of the LSST production scripts can be configured using the `lsstDebug` package. `lsstDebug` has a very negative view of the world; `lsstDebug.Info(xyz).abc` returns `False` for any value of`xyz` and `abc`. By convention, packages call it as

```
import lsstDebug
display = lsstDebug.Info(__name__).display
```

at the top of their main procedure (the choice of `display` is common but not required); for example in `python/lsst/meas/utils/sourceDetection.py`. If you do nothing else, this sets `display` to `False` as `lsstDebug.Info(lsst.meas.utils.sourceDetection).display` returns `False`.

More interestingly, I can `import` (or `reload`) a private file that looks something like:

```
import lsstDebug

def DebugInfo(name):
    di = lsstDebug.getInfo(name)        # N.b. lsstDebug.Info(name) would call us recursively
    if name in (
        "lsst.meas.utils.sourceDetection",
        "lsst.meas.astrom.determineWcs",
        ):
        di.display = 1

    return di

lsstDebug.Info = DebugInfo
```

I.e. change `lsstDebug.Info("lsst.meas.utils.sourceDetection").display` to return `1`. Suddenly, `lsst/meas/utils/sourceDetection.py` starts displaying fascinating images without my having to modify the file and reload it.