pixcorrect development

Eric H. Neilsen, Jr.

February 12, 2015

1 Initial state

1.1 `imsupport`

The `imsupport` product contains C code for loading, saving, and otherwise working with DES images. Particularly significant contents of this product are:

mask bit definitions are defined in C pre-processor directives, duplicated in `include/imutils.h` and `include/imsupport.h`. There are separates sets of bits for use in the initial BPM and the mask HDUs of DES images.

`desimage` struct `include/imreadsubs.h` contains the `desimage` C struct, which includes elements for metadata corresponding to some FITS keywords, and pointers to arrays containing science data, weight, and mask pixel data.

1.2 `imcorrect`

The `imcorrect` C code does the actual pixel-level correction on DES images. For each operation, there appears to be a central bit of code that actually does the calculation on each pixels. Sometimes these are combined in the same loop over pixels, sometimes a calculation is contained in its own loop.

1.3 `fitsio`

`fitsio` is a python wrapper around the `CFITSIO` library, and can be used to manipulate headers, and load and save pixel data as `numpy` arrays.

1.4 `ctypes`

`ctypes` is a base python method of calling C libraries. C manitulates memory at a lower level than is typical for python program, and `ctypes` includes tools for generating and interpreting arbitrary C structures, mapping them to python objects.

Using `ctypes` with `numpy` can be slightly tricky. Internally, `numpy` can store data in a variety of ways (row-major or column major ordering of array elemets, big-endian or little-endian, and the assorted combinations thereof). When `numpy` is used intirely within python, the way in which the data is stored is invisible to the user: the `numpy` code keeps track and does whatever is appropriate.

When one needs to pass a pointer to arrays of pixels to C code, however, one needs to be sure that `numpy` is storing the pixels in the order, type, and endian-ness expected by the C library. `numpy` includes tools for supporting this.

2 Progress

2.1 `despyfits`

`despyfits` contains code for manipulating DES images from python libraries, roughly analogous to `imsupport`.

`despyfits` includes a header file, with lines cut-and-pasted from `imsupport`'s `include/imsupport.h`, that define bits in pixel masks. There is also a trivial C library that imports these and assigns the values to C extern variables, and a python module that uses `ctypes` to provide access to these values from python. So, for example, the `BAD-PIX_SATURATE` bit can be printed from python thus:

```
Python 2.7.6 (default, Nov 10 2014, 12:26:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from despyfits import maskbits
>>>
>>> print maskbits.BADPIX_SATURATE
2
>>>
```

To make sure there is only one cannonical definition of mask bits, `imsupport`'s `imutils.h` and `imsupport.h` now need to be modified to include the file from `despyfits` rather than their own definitions. Who owns `imsupport`?

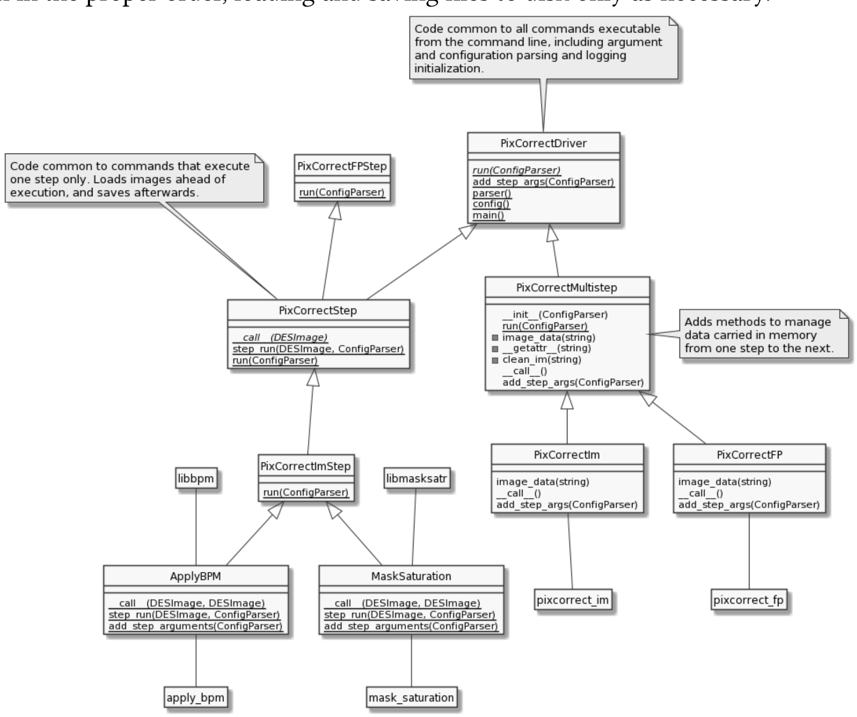
`despyfits` includes a `DESImage` class for holding data associated with a DES image, roughly analogous to the `desimage` C struct.

Because the code that does the calculations in `imcorrect` makes heavy use of `imsupport`'s `desimage` struct, and we would like to cut-and-paste this code into C libraries to be called by python through `ctypes`, `DESImage` includes a method that returns a `ctypes.Structure` object that supplies a subset of the elements defined in `imsupport`'s `desimage` struct, and a C header file that declares a C structure that corresponds to the structure generated by `DESImage`. When the `DESImage` class is used to create this `ctypes.Structure`, it automatically ensures that the numpy arrays are stored in the right endianness and memory layout for use in C code. If data in numpy arrays are passed to C code by some other mechanism, this needs to be done in the new code, because numpy does not guarantee the memory layout or endianness by default.

2.2 pixcorrect

The development of the python utility that replicates `imcorrect` is in progress.

It will include C libraries that replicate operations performed by `imcorrect`, with C code copied directly from `imcorrect`. There will be python modules that correspond to each operation, each of which can be loaded independenly in python or executed independently from a shell. In addition, there is a high-level driver that loads each, and executes each in the proper order, loading and saving files to disk only as necessary.



3 Installing and running pixcorrect

3.1 Running pixcorrect on des21.fnal.gov

3.1.1 *Log into the DES cluster at Fermilab*

Log into des21.fnal.gov, and prepare the shell you are using to use eups and the DESDM product stack.

It is possible that your .bashrc or .profile prepares your login environt that uses mainline UPS rather than the eups fork, in which case you need to start a new bash what doesn't have any of this setup. This will do that:

```
env - PATH=/usr/bin:/bin TERM=$TERM "$(command -v bash)" --noprofile --norc
```

Then, prepare the eups environment needed:

```
source /cvmfs/descfs.fnal.gov/eeups/SL6/eups/desdm_eups_setup.sh
setup Y2Nstack 1.0.5+0
```

3.1.2 *Make a devel directory and go there*

The /usrdevel/dp0 directory on the DES cluster at fermilab is intended for for software development. It isn't large, and so isn't suitable for holding images or other data, but it is good for checking out subversion productes.

Make yourself a directory there, if you don't already have one:

```
mkdir /usrdevel/dp0/$(whoami)
```

3.1.3 *Check needed products out of subversion*

There are two products you need: the br_pixcorrect_0.1.1 branch of despyfits, and the trunk of pixcorrect. This will get you checked out versions:

```
DESDMSVN=https://dessvn.cosmology.illinois.edu/svn/desdm/devel
cd /usrdevel/dp0/$(whoami)
svn co ${DESDMSVN}/despyfits/branches/br_pixcorrect_0.1.1 despyfits-pc
svn co ${DESDMSVN}/pixcorrect/trunk pixcorrect-trunk
```

3.1.4 *Build and eups setup despyfits*

These products need to be built using the python setup.py script, which takes care of compling the C components and moves executable python scipts into standard places.

This can be done like this, assuming the directories are as made by the above instructions:

```
cd /usrdevel/dp0/$(whoami)/despyfits-pc
python setup.py build
```

Once built, you can use eups setup to set environent variables (PATH, PYTHONPATH, etc.) as needed:

```
setup despyfits \
  -r /usrdevel/dp0/$(whoami)/despyfits-pc \
  -m /usrdevel/dp0/$(whoami)/despyfits-pc/ups/despyfits-devel.table
setup fitsio 0.9.6+0
```

3.1.5 *Build and eups setup pixcorrect*

Similar operation need to be performed for pixcorrect.

First, build it:

```
cd /usrdevel/dp0/$(whoami)/pixcorrect-trunk
python setup.py build
```

The use eups setup to set the environment:

```
setup pixcorrect \
  -r /usrdevel/dp0/$(whoami)/pixcorrect-trunk \
  -m /usrdevel/dp0/$(whoami)/pixcorrect-trunk/ups/pixcorrect-devel.table
```

3.1.6 Running tests

Prepare the product with the test data:

```
setup pixcorrectTestData
```

Actually run the tests:

```
cd $PIXCORRECT_DIR/test
python test.py
```

3.1.7 Running overscan_correct

```
setup pixcorrectTestData
overscan_correct -i $PIXCORRECTTESTDATA_DIR/DECam_00394250.fits.fz \
    -o my_output.fits -n 10 -v
```

3.1.8 Running apply_bpm

```
setup pixcorrectTestData
apply_bpm -i $PIXCORRECTTESTDATA_DIR/scix.fits \
    --bpm $PIXCORRECTTESTDATA_DIR/bpm.fits \
    -o test_post_bpm.fits
```

3.1.9 Running mask_saturation

```
setup pixcorrectTestData
mask_saturation -i $PIXCORRECTTESTDATA_DIR/scix.fits \
    -o test_mask_saturation.fits
```

4 Instructions for adding steps from imcorrect to pixcorrect

4.1 Find the code in imcorrect that does the pixel level calculation

4.2 Verify and update desimage if neede

4.2.1 Check desimage

Check that all elements of imsupport's desimage structure used in this code are present in despyfits's desimage (in include/desimage.h). Hopefully, all elements needed by the imcorrect code to be moved will already by present, in which case you can skip to the next step.

4.2.2 Supplement include/desimage.h in despyfits.

If there are needed elements missing from include/desimage.h in despyfits, copy them from the declaration of the desimage struct in include/imreadsubs.h in imsupport.

4.2.3 Supplement DESImageCStruct class in python/despyfits/DESImage.py

Add the missing elements to the DESImageCStruct class in python/despyfits/DESImage.py. Be sure to both add the new helement to the _fields_ property and the code to set the values in the class's __init__ method.

The _fields_ property must list the same elements in the same order with types corresponding to the declarations in the C header. The mapping between ctypes types and basic C types can be found here.

If pointers are needed, use ctypes.POINTER. If an array of values is needed, the clearest way to handle it is probably by declaring a new python class. For example, the ampsecan field in the desimage structure is an array of four integers, so the the DESImage python module declares the class FourInts thus:

```
FourInts = ctypes.c_int * 4
```

and the ampsecan element of the _fields_ property of the DESImageCStruct class is declared to be of that type.

4.2.4 recompile despyfits using the setup.py

The C libraries used in despyfits depend on the desimage.h header, and so must be recompiled to incorporate any changes. Because this is a python package, use python's setuptools to do this:

```
cd $DESPYFITS_DIR
python setup.py build
```

4.2.5 check the updated depyfits into svn, and svn update pixcorrect

The desimage.h header file appears in the pixcorrect product by way on an svn external directory; to be seen by the pixcorrect C code, it must be checked into svn from despyfits, and the svn update=d into your checked out =pixcorrect product.

4.3 Wrap C code from imcorrect.c

Copy the code in imcorrect.c to a C file in pixcorrect's src directory, calling it lib${STEPNAME}.c. Make sure the loop over pixels is included in the library; see src/libbpm.c or src/libmasksatr.c for examples.

Be careful allocating and freeing memory within the C library. Python's memory managemen and garbage collection knows nothing about memory allocated or freed within the library, so if you allocate code and do not free it, it will be a memory leak, and if you free memory allocated by python (and seen by the C library through a passed pointer), it will cause python to segfault.

4.4 Add the new C library to setup.py in pixcorrect

The setup.py builds the libraries in the pixcorrect product. Two changes are need for this. First, an object of class SharedLibrary needs to be created with instructions for building the new library. For example, the creation of the SharedLibrary object for libbpm looks like this:

```
libbpm = SharedLibrary(
    'bpm',
    sources = ['src/libbpm.c'],
    include_dirs = ['include'],
    extra_compile_args = ['-O3','-g','-Wall','-shared','-fPIC'])
```

Other libraries are likely to look very similar.

Then, this new object needs to be added to the list of shared libraries that need to be built to build the product, specified in the shlibs parameter in the setup call.

4.5 Compile pixcorrect

Run setup.py in the root of pixcorrect:

```
cd ${PIXCORRECT_DIR}
python setup.py build
```

4.6 Create a python module for the new step

The module should supply an API for calling the step programatically, and also code for running it stand-alone. mask_saturation.py is an example of an operation that uses only the data desimage structure itself for paramaters and returns, and apply_bpm is a slightly more complex example, where a second desimage is used.

4.6.1 *Example 1: `mask_saturation`*

Following the `mask_saturation` example, it begins by declaring a module constant that defines the section of a configuration file from which the stand-alone executable might read its input:

```
config_section = 'mask_saturation'
```

(This same string will also be used as the switch to determine whether this step is performed in the driver that drives multiple steps.)

Next comes the code that loads the C library:

```
masksatr_lib = load_shlib('libmasksatr')
mask_saturation_c = masksatr_lib.mask_saturation
mask_saturation_c.restype = ctypes.c_int
mask_saturation_c.argtypes = [DESImageCStruct, ctypes.POINTER(ctypes.c_int)]
```

The first line loads the library. The second maps the `mask_saturation` function in that library to the python callable object referenced by `mask_saturation_c` (in other words, `mask_saturation_c` acts like a python function).

The assignments to `mask_saturation_c.restype` and and `mask_saturation_c.argtypes` set the return type and argument types for the C function, respectively.

The value of `DESImageCStruct` corresponds to a `desimage` argument in the C function, and the `ctypes.POINTER(ctypes.c_int)` to a pointer to an `int`.

Following this management of the C library, the `MaskSaturation` class is declared. It is a subclass of `PixCorrectImStep`, from which it gets most of its functionality; only code specific to this step is here.

At the start of the class definition are the properties `description` and `step_name`, which determine the help string to be provided on the command line in response to an `--help` argument and the switch to be used to turn on and off execution of this function (mapped here to the module's `config_section` constant).

Then, a `__call__` method is defined. `__call__` is a special python method which is called when an instance if an object is called as if it were a function, an in this case it defines the main functionality of the step.

The definition begins with defining the `num_saturated` variable to a `ctypes.c_int()`, which gets python to allocate the appropraite memory so it can pass a pointer to it in the next line.

The next line actually calls the function in the C library. The first argument triggers the creation of a `DESImageCStruct` object from the `DESImage` object, and the second the pointer to the variable in which the number of flagged pixels will be returned.

After definition of the `MaskSaturation` class, this line:

```
mask_saturation = MaskSaturation()
```

creates an instance of the class and assigns it to `mask_saturation`, meaning that `mask_saturation` can now be treated like a function (in which case `MaskSaturation.__call__` will be called).

4.6.2 *Example 2: `apply_bpm`*

The `apply_bpm` step is similar, but has the additional complication that a second image, the bad pixel mask, needs to be loaded and passed to the call.

The first thing that needs to happend is that a new command line argument needs to be made to let the user supply the file name for the BPM. This is done by the `add_step_args` method. For additional information on adding arguments to a python argument parser, see the python `argparse` documentation.

We also need to add the code to load the BPM image, which is done in the `step_run` method. Note that the second argument of the `config.get` call needs to match the second argument of the `parse.add_argument` call in `add_step_args`, which is also the option that would be used if the file name were to be loaded from a configuration file.

Note that `MaskSaturation` class also has `step_run` and `add_step_args` method, but these are inhereted from the `PixCorrectImStep` parent class, and do not do anything special.

4.7 Create an executable file for the step

Once the step's class has been defined as above, this is trivial. In the case of apply_bpm, for example, a python script, ${PIXCORRECT_DIR}/bin/apply_bpm, that looks like this:

```
#!/usr/bin/env python

from pixcorrect.apply_bpm import apply_bpm

if __name__ == '__main__':
    apply_bpm.main()
```

does the trick.

Note that we could accomplish the same thing by just running apply_bpm.py directly, and so we could have gotten away with just putting a symlink to apply_bpm.py in ${PIXCORRECT_DIR}/bin, but I personally prefer to keep them as real, distinct files, for reasons too obscure to go in to here.

4.8 Add the step to pixcorrect_im.py

The code that runs all steps in sequence can be found in ${PIXCORRECT_DIR}/python/pixcorrect/pixcorrect_im.py.

Start adding your new step by adding an import of the callable instance of your object defined above, for example in the case of fix_cols this is:

```
from pixcorrect.fix_cols import fix_cols
```

Next, if your new step requires arguments not already supplied by other steps, add them to the PixCorrectIm.add_step_args method. In the case of fix_cols, the bpm argument was already present, but the --fix_cols switch to set whether to perform that step was not. So, these lines were needed to support fix_cols:

```
parser.add_argument('--bpm', nargs=1,
                    default=None,
                    help='bad pixel mask filename')
parser.add_argument('--fix_cols', action='store_true',
                    help='fix bad columns')
```

Then, figure out where your new step should be called in the PixCorrectIm.__call__ method, and add it. For example, in the case of fixcol, the new code added was:

```
if do_step('fix_cols'):
    fix_cols(self.sci, self.bpm)
```

Note that the argument to do_step is the same as the option name in the add_argument statement above.

Other methods of the PixCorrectIm class (which do not need to be modified for new steps) make sure the images passed to fix_cols, self.sci and self.bpm, get loaded as needed. When you reference a method of PixCorrectIm that is not already defined, the class looks for a command line argument (or parameter in the config file) with the same name, and if one is present, it interprets it as a file name for a DESImage and loads it. Once it is loaded, future references to the same member just return the same loaded object, avoiding the need to load the image multiple times from disk.

The clean_im method releases the memory for images loaded in this way. For example, this line:

```
self.clean_im('bpm')
```

tell python it can free them memory into which the BPM was loaded. If self.bpm is referenced after this, it will need to be reread from disk. (Note that python does not guarantee that it will free the memory as soon as it is allowed to.)

4.9 Add a doctest or unittest

4.9.1 Test infrastructure in pixcorrect

Python has two built-in mechanisms for automated testing, doctest and unittest. Which is more useful depends on what tests you are performing. Personally, if the test code itself does not require a lot of debugging, I find doctests easier, but if the testing code itself is complicated and likely to have bugs, I find the unittest approach easier.

The pixcorrect product has a driver for running tests of both types.

Each doctest is in a text file in ${PIXCORRECT_DIR}/test/doctests and each unittest in is a python file in ${PIXCORRECT_DIR}/test/unittests.

The configuration file ${PIXCORRECT_DIR}/test/test.config has options corresponding to each unittest in its unittest section, and each doctest in its docs section. Set the values of these options to true or false depending on whether you want to run those tests, and run the desired tests thus:

```
setup pixcorrectTestData
cd $PIXCORRECT_DIR/test
python test.py
```

Note that you can use a different configuration file if you wish, and run it thus:

```
setup pixcorrectTestData
cd $PIXCORRECT_DIR/test
python test.py my_favourite_tests.config
```

4.9.2 doctests

At present, the doctests in pixcorrect mostly just test that the various modules run without crashing.

doctests are particularly easy to write in python. When a doctest test document is run, the test driver looks through the text document for anything that looks like example code, and tries to run it and check that it gets the same results as in the example.

So, when I first start testing a piece of python code interactively, once I get an example that works, I just cut and paste the output on the terminal into a text document, put a little explanatory text around it, and make that the initial doctest.

Note that doctest can be used to test examples embedded in python code itself, but I won't describe that here.

4.9.3 unittest

unittest more closely resembles the unit testing approach. So far, I have put code that tests whether the steps accurately reproduce the results from imcorrect in this framework.