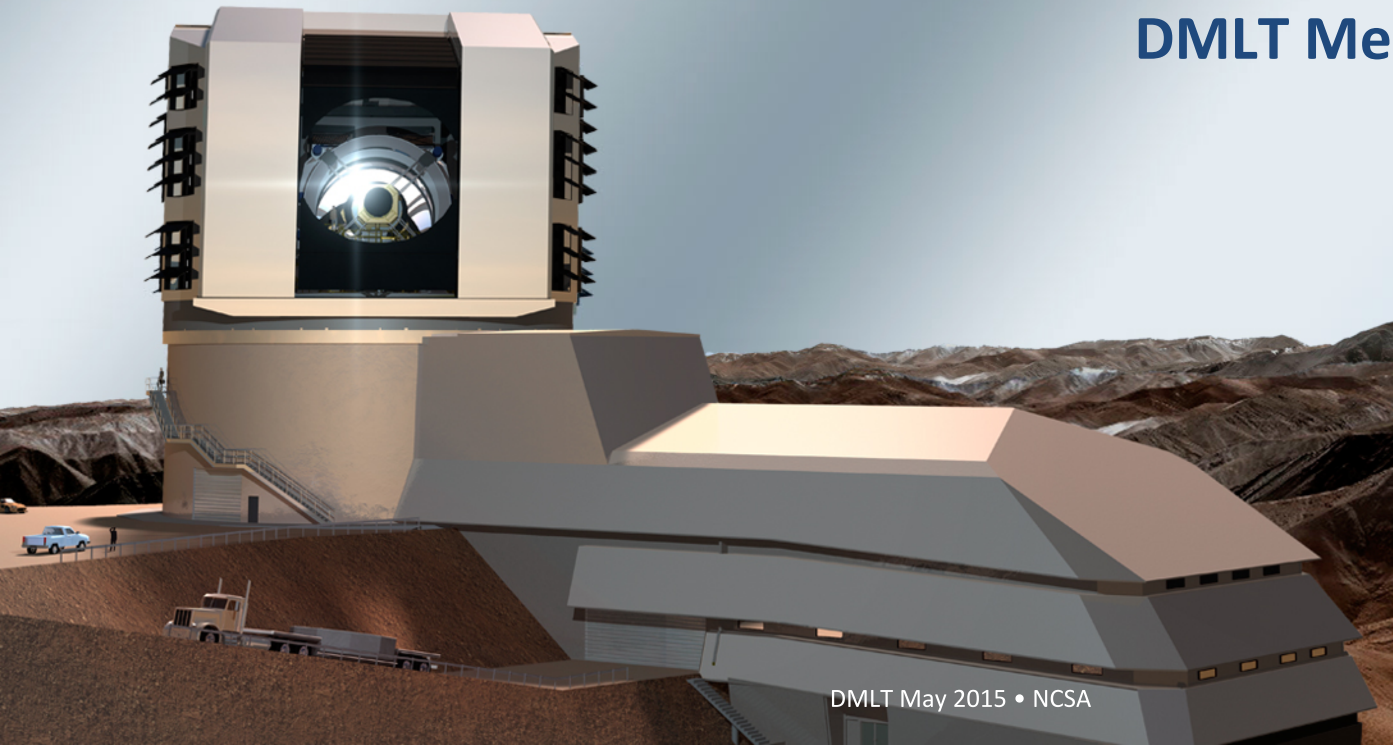
Python C++ interaction in the DM Software Stack

Tim Jenness

DMLT Meeting @ NCSA
May 20, 2015



Non-Pythonic Interface

- Attributes don't look like Python attributes.
- Naming conventions don't match PEP-8.
- Constructors are scary mess of C++ signatures, not all of which are relevant, and mostly lacking in documentation, using *args.
- Similarly, most methods are documented simply by showing C ++ API but lack the doxygen C++ docs.
- Exceptions usually end up coming from C++ land. This is particularly bad in constructors but also genuinely surprising to a new user.
- Objects tend to stringify (or repr()) in opaque ways referring to swig. repr() output can't be used to construct an object in many cases.

Interoperability

- Plays poorly with other python astronomy software
  - Astropy coordinates and quantity might be useful
  - PANDAS vs AFW table

- Build system surprisingly complicated for python programmer (no “pip install lsst_apps”)

Investigations

Three schemes investigated:

- Write “shim” classes to hide the SWIG interfaces from the user.
- Rewrite high level classes in Python
- Experiment with numba and cython in high performance code.

Shims

For each C++ SWIG class, add new python class that mediates calls to C++

```
e = geom.Extent(3,5)
print(e.x, e.y)
exp = image.Exposure(e, dtype=np.float64)
print(exp.bbox, bbox.area)

mi = exp.masked_image
var = mi.variance
var[0:2,3:5] = np.array([[1,2],[3,4]])
```

See https://github.com/lsst-dm/python-experiments

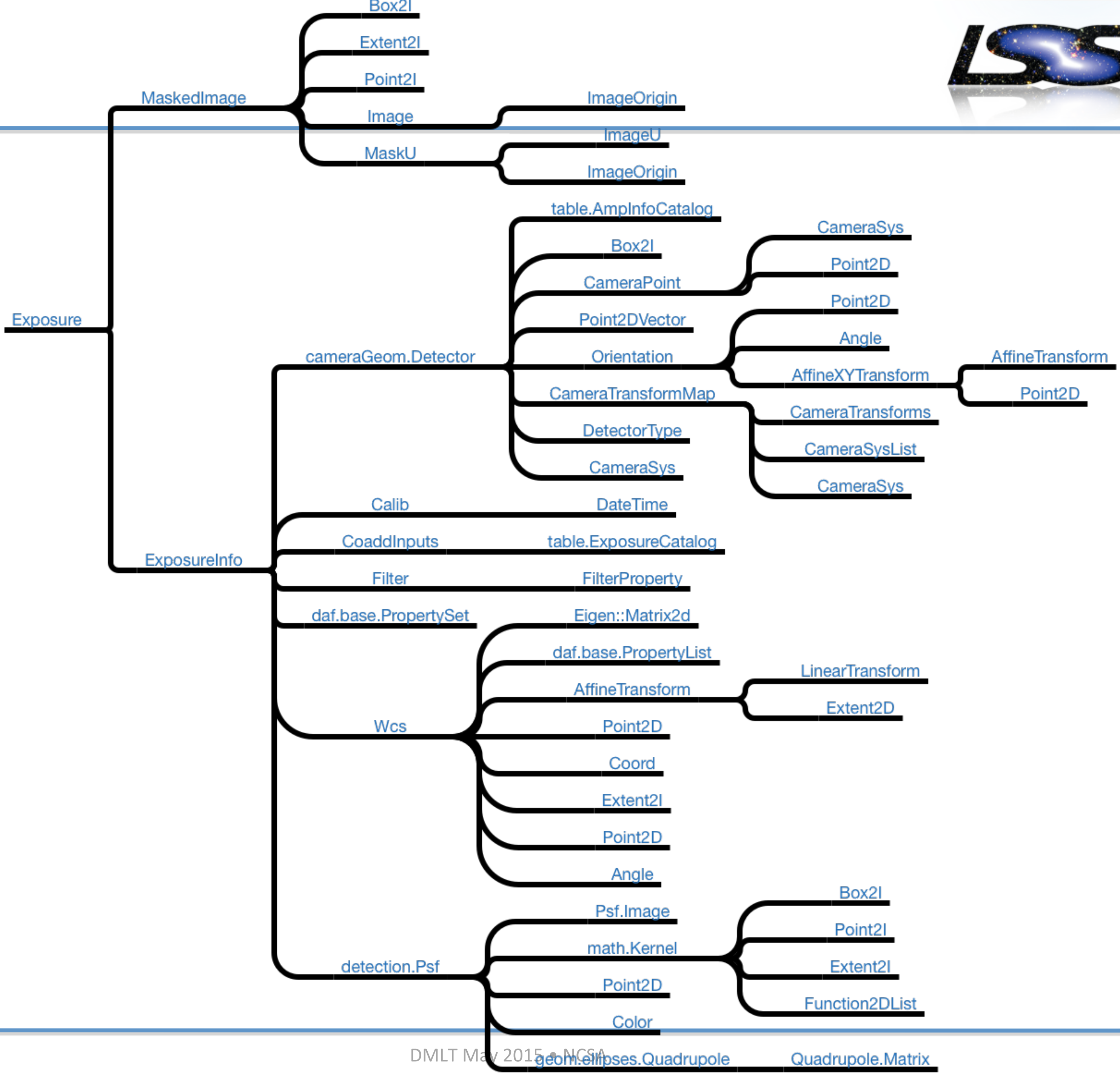
Shims: thoughts

- Straightforward to add new classes
- C++ still leaks through at times (especially with exceptions)
- Every time C++ API is modified the Python needs to be changed to match
- No scheme for dealing with documentation duplication (can it be extracted from C++ doxygen source?)
- Possible performance hit with all the extra function calls.
- Could include mediation layer for translating AFW coordinates (and others) and tables to Astropy and PANDAS so that public Python interface looks more normal. Again, lots of overhead.

Pushing the C++ down

The approach here is to take the large aggregate classes and rewrite them in Python. When calling a C++ routine replace a single object argument with explicit attributes of the object.

- Exposure was chosen as a good example.
- Exposure contains a MaskedImage and an ExposureInfo. ExposureInfo contains important items such as WCS and Filter and CoaddInputs.
- A new demo class written that contains a MaskedImage and all the contents of ExposureInfo
- Quite a lot of infrastructure needed to test the new interface from Warping or Measurement layer so not yet tested.


Exposure
MaskedImage
Box2I
Extent2I
Point2I
Image
ImageOrigin
MaskU
ImageU
ImageOrigin
ExposureInfo
cameraGeom.Detector
table.AmpInfoCatalog
Box2I
CameraPoint
CameraSys
Point2D
Point2DVector
Orientation
Point2D
Angle
AffineXYTransform
AffineTransform
Point2D
CameraTransformMap
CameraTransforms
CameraSysList
CameraSys
DetectorType
CameraSys
Calib
DateTime
CoaddInputs
table.ExposureCatalog
Filter
FilterProperty
daf.base.PropertySet
Wcs
Eigen::Matrix2d
daf.base.PropertyList
AffineTransform
LinearTransform
Extent2D
Point2D
Coord
Extent2I
Point2D
Angle
detection.Psf
Psf.Image
math.Kernel
Box2I
Point2I
Extent2I
Function2DList
Point2D
Color
geom.ellipses.Quadrupole
Quadrupole.Matrix

Downward Pressure

- This all should work in theory
- But how to avoid “Fortran in C++”: very large argument lists passed through to the C++ layer.
- Fundamental classes like Extent, Point and Box may be better off existing in both worlds.

Python Performance enhancements

- What if the default position was always to use Python unless we needed to go faster for a specific bit of number crunching?
- Is there a middle ground between slow python and C++?
- Cython and numba are interesting approaches
  - Cython is a very mature system that allows python-like code with type annotations to be compiled to C. Widely used in the Python community. Allows trivial access to C code.
  - Numba is a Jit compiler using llvm. Relatively immature (upgrade often). No type annotations. Just use @jit decorator. Magic.

Convolution

- AFW convolve.py has a handy "pure python" convolver to compare against the AFW C++ compiler version.
- Numba and Cython implementations were written (with the tight loop factored out and the numpy reduce calls explicitly turned in to loops)
- C++ 3000 times faster than Python
- C++ 15 times faster than Cython
- C++ 2 times faster than numba
- Numba was impressive given the lack of work involved (noting that allocating numpy buffers can't be done)
- Needs to be tested with a spatially variable kernel (a lot of kernel infrastructure needs to be ported for a good test).

```python
@jit(nopython=True)
def runRefConvolveJitLoop(ignore_zero_pix, retRowRef, retColRef, numRows, numCols, kWidth, kHeight,
                          kImArr, image, variance, mask, retImage, retVariance, retMask):
    retRow = retRowRef
    for inRowBeg in range(numRows):
        inRowEnd = inRowBeg + kHeight
        retCol = retColRef
        for inColBeg in range(numCols):
            inColEnd = inColBeg + kWidth
            subImage = image[inColBeg:inColEnd, inRowBeg:inRowEnd]
            subVariance = variance[inColBeg:inColEnd, inRowBeg:inRowEnd]
            subMask = mask[inColBeg:inColEnd, inRowBeg:inRowEnd]
            sum = 0.0
            varsum = 0.0
            outmask = 0
            for i in range(kWidth):
                for j in range(kHeight):
                    sum += subImage[i, j] * kImArr[i, j]
                    varsum += subVariance[i, j] * kImArr[i, j] * kImArr[i, j]
                    if ignore_zero_pix:
                        if kImArr[i, j] != 0:
                            outmask |= subMask[i, j]
                    else:
                        outmask |= subMask[i,j]
            retImage[retCol, retRow] = sum
            retVariance[retCol, retRow] = varsum
            retMask[retCol, retRow] = outmask
            retCol += 1
        retRow += 1
    return
```

Thoughts on numba/cython

- Numba does feel like magic: some thought is required to write your python code in a manner amenable to the Jit compiler (things go slower if you don't do it right).
  - No type declarations!
  - Multi-CPU + GPU support for numba costs serious money
  - Numba has a reputation for being a pain to build if you are not working with Anaconda.
- Cython is a "safe pair of hands" and it's fairly obvious what is going on and how to call out to hand-crafted C.
  - Cython can generate functions callable from cython without Python object overhead but also from Python.
  - Now uses MemoryViews so not reliant on numpy.
- Where are we going to be relying on threads?

What are we?

- Are we a Python project with some C/C++ in the hot spots?
- Are we a C++ project with some Python glue?
- Is it desirable to move the C++ boundary deeper into the stack?
- Do we know what really needs to be in C++?
- Can we stop the “C++ first” default approach where everything is written in C++ “just in case”.
- Are we building a data reduction system that will be embraced by the community or one that will scare people off with the difficult learning curve? Astropy does not have the best algorithms in many cases but it’s good enough for a lot of people.
- How brave are we?