Coding Standards Exception Request (originally Third Party Software Proposal) - ndarray

Jim Bosch (of UC Davis) has written a C++ template-based N-dimensional array library called ndarray. While the original proposal was for ndarray to exist as a third-party package, following the TCT's previous discussion and a better understanding of the options, we have changed to propose integrating it directly into the LSST code base. We intend to limit the use of this library to multifit for DC3b, but we feel that in the future it may be useful to integrate it more closely with the current afw::image API. While we are of the opinion that the most logical ultimate place for ndarray is in afw, and that it will be of broader use, we accept that in the future ndarray may not be needed at all if the afw::image API is modified in the future to support our use cases, so simply putting ndarray in meas/multifit is a workable solution for DC3b.

What separates this code from regular LSST development is that it is impossible to make ndarray conform to certain LSST coding standards, and (we argue) undesirable to make it conform to others. As a result, we are seeking an explicit exception for these standards for the ndarray package.

Conflicts with LSST Coding Standards

- ndarray is a header-only library that makes extensive use of template metaprogramming, which makes it impossible to provide explicit instantiations of all template classes. (Standard 4-4b)

- Certain names are not camel-case verbs because they are designed to make classes conform to certain STL concepts ("size()" is present as well as "getSize()") or to mirror STL/boost interfaces ("make_index" mirrors "make_pair", etc). (Standard 3-4)

- The library contains many classes with very similar APIs and implementations which nevertheless cannot be completely merged. I have used preprocessor macros to put the shared code in a single place and make it easier to maintain. I have taken care to ensure that doxygen is aware of this usage so the documentation it produces is accurate and provides the reference information the code itself lacks (because of the macros), but this may require running doxygen with slightly non-standard options. (No specific coding standard; however, preprocessor macros are widely acknowledged as something to be avoided when possible, and my code uses them extensively - so I expected someone would object to them unless I explained their use here).

Motivation

Multifit has a strong need for a unified interface for 1, 2, and 3 dimensional arrays which either:

- share data (e.g. Numpy, `afw::Image`); *or*
- include both strong owning and non-owning variants, in which the owning version is convertible to the non-owning version (boost::gil::image and boost::gil::view).

Existing LSST code seems to prefer the former, and since ndarray supports both, I'll assume that usage. Right now, LSST has:

- Two main one-dimensional array classes in use (`std::vector<double>`, `Eigen::VectorXd`). Neither of these has shared ownership.

- Two main two-dimensional array classes in use (`afw::Image`, `Eigen::MatrixXd`). The former has shared ownership, but only the latter support optimized linear algebra operations.
- No three-dimensional array classes. In our case, 3D arrays are almost always stacks of images, and while `std::vector<afw::image::Image>` is a possible workaround sometimes, it is no more the appropriate replacement for a contiguous strided 3D array than a vector-of-vectors is a suitable replacement for an image.

The situation is somewhat worse, however, because Eigen has different compile-time types for dealing with arrays that own their data, blocks of arrays that own their data, and arrays that reference external data (`Matrix`, `Block`, and `Map`, respectively). This makes it necessary to use templates to support any operation on Eigen-based arrays that doesn't care about how they were allocated, which in turn makes it impossible to write such an operation as a virtual member function. Meanwhile, `std::vector` has no support for views or shared data, while `afw::Image` only allows sharing between images, making it impossible to construct, for instance, an `Eigen::Map` that references data in an `afw::Image`, or to construct an `afw::Image` (or `std::vector`) view into a row of an `Eigen::MatrixXd`.

Clearly none of these types should go away; they all have their specific uses. And not all of the above cases are necessary. However, algorithm code that operates on a simple 1-, 2-, or 3-dimensional strided array concept should be built around that bare concept, not on the details of how the memory that comprises that array was allocated or how its lifetime is managed, and all objects that can support that concept should somehow be adaptable to it.

`ndarray` is not the only solution to this problem, and right now it is not even a complete solution; modifications would be required to `afw::Image` to expose an `ndarray` view of an `afw::Image`, and full shared-owner interoperability with `std::vector<double>` or `Eigen` is impossible. However, it is a partial solution that already exists, and considering the DC3b timescale we don't see any other alternative for multifit, where the need for shared multidimensional arrays is perhaps most acute.

For now, we can limit `ndarray` usage to the `multifit` package, and simply copy `afw::Image` objects into 2D `ndarray` objects on the boundary between multifit and other code. In the future, we hope that either:

- `ndarray` will be used more widely throughout the project, and integrated closely with `afw::Image` so that both `ndarray` and `afw::Image` can share references to the same memory (this will require changes to the internals of `afw::Image`, but crucially it need not change the ownership semantics or other external behavior of `afw::Image`); *or*
- `ndarray` will be replaced in multifit by a custom set of 1- and 3-dimensional array classes that provide the `ndarray` functionality needed by multifit and are similarly integrated with `afw::Image`, along with additional updates to `afw::Image` to allow it to fulfill the 2d `ndarray` role.

We intend to use `Eigen` mostly via temporary `Map` objects which will be constructed from `ndarray`-owned data; this will allow us to benefit from optimized `Eigen` operations while avoiding constructing `ndarray` objects (or wanting to construct `afw::Image` objects) that reference memory that is not reference-counted.

Additional Materials

?Doxygen Documentation