How the Large Synoptic Survey Telescope (LSST) is using Python

Robert Lupton (rhl@astro.princeton.edu) – *Princeton University,* USA

The Large Synoptic Survey Telescope (LSST) is a project to build an 8.4m telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014.

The scientific goals of the project range from characterizing the population of largish asteroids which are in orbits that could hit the Earth to understanding the nature of the dark energy that is causing the Universe's expansion to accelerate.

The application codes, which handle the images coming from the telescope and generate catalogs of astronomical sources, are being implemented in C++, exported to python using swig. The pipeline processing framework allows these python modules to be connected together to process data in a parallel environment.

The Large Synoptic Survey Telescope (LSST) is a project to build an 8.4m telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014.

The scientific goals of the project range from characterizing the population of largish asteroids which are in orbits that could hit the Earth to understanding the nature of the dark energy that is causing the Universe's expansion to accelerate.

The application codes, which handle the images coming from the telescope and generate catalogs of astronomical sources, are being implemented in C++, exported to python using swig. The pipeline processing framework allows these python modules to be connected together to process data in a parallel environment.

Introduction

Over the last twenty-five years Astronomy has been revolutionized by the introduction of computers and CCD detectors; the former have allowed us to employ telescope designs that permit us to build telescopes with primary mirrors of diameter 8-10m, as well as handle the enormous volumes generated by the latter; for eample, the most ambitious imaging project to date, the Sloan Digital Sky Survey (SDSS [SDSS]), has generated about 15Tby of imaging data.

There are a number of projects being planned or built to carry out surveys of the sky, but the most ambitious is the Large Synoptic Survey Telescope (LSST). This is a project to build a large telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014. The telescope is a novel 3-mirror design with an 8.4m diameter primary mirror and will be equipped with a 3.2Gpixel camera at prime focus. The resulting field of view will have a diameter of 3.5 degrees --- 7 times the full moon's diameter (and thus imaging an area 50 times the size of the moon with every exposure). In routine operations we expect to take an image of the sky every 15s, generating a data rate of over 800 Mby/s. In order to handle these data we will be building a complex software system running on a large cluster. The LSST project is committed to an "Open-Data, Open-Source" policy which means that all data flowing from the camera will be immediately publically available, as will all of the processing software.

The large area imaged by the LSST telescope will allow us to image the entire sky every 3 (clear!) nights. This survey will be carried out through a set of 6 filters (ultra-violet, green, red, very-near-infrared, nearish-infrared, near-infrared; 320nm -- 1050nm) allowing us to characterize the spectral properties of the several billion sources that we expect to detect --- approximately equal numbers of stars and galaxies. This unprecedented time coverage (at the faint levels reached by such a large telescope, even in as short an exposure as 15s) will allow us to detect objects that move as well as those that vary their brightness. Taking the set of images at a given point, taken over the 10-year lifetime of the project, will enable us to study the extremely faint Universe over half the sky in great detail. It is perhaps worth pointing out that the Hubble Space Telescope, while able to reach very faint levels, has a tiny field of view, so it is entirely impractical to dream of using it[2] to carry out such survey projects.

The LSST's scientific goals range from studies of the Earth's immediate neighbourhood to the furthest reaches of the visible Universe.

The sensitivity to objects that move will allow us to measure the orbits of most[3] asteroids in orbits that could hit the Earth.[4] If it's any consolation, only objects larger than c. 1km are expected to cause global catastrophes, while the main threat from smaller objects is Tsunamis (the reindeer-killing object that hit Tunguska in 1908 is thought to have been c. 100m in diameter). More distant moving objects are interesting too; LSST should be able to characterise moving objects in our Galaxy at distances of thousands of light years.

The LSST's frequent measurement of the brightness of enormous numbers of sources opens the possibility of discovering new classes of astronomical objects; a spectacular example would be detecting the flash predicted to occur when two super-massive black holes merge in

[2] Or its planned successor, the James Webb Space Telescope

[3] 90% of objects larger than 140m

[4] There are other projects, such as Pan-STARRS on Haleakala on Maui, that are expected to identify many of these objects before LSST commences operations

a distant galaxy. Such mergers are expected to be a normal part of the evolution of galaxies, and should be detectable with space-based gravitational wave detectors such as LISA. The detection of an optical counterpart would dramatically increase how much we learn from such an event.

Finally, the LSST will provide extremely powerful ways of studying the acceleration of the Universe's expansion, which is generally interpreted as a manifestation of a "dark energy" that currently comprises 70% of the energy-density of the Universe. Two techniques that will be employed are studying distant type Ia supernovae (which can be used to measure the expansion history of the Universe) and measuring the distortions imposed on distant galaxies by the curvature of space caused by intervening matter.

The requirements that this system must meet are rather daunting. The accuracy specified for measurement of astronomical quantities such as brightnesses and positions of sources significantly exceeds the current state of the art, and must be achieved on a scale far too large for significant human intervention.

LSST's Computing Needs

Analyzing the data coming from LSST will require three classes of software: The applications, the middleware, and the databases. The applications codes process the incoming pixels, resulting in catalogues of detected sources' properties; this is the part of the processing that requires a understanding of both astronomical and computing algorithms. The middleware is responsible for marshalling the applications layer over a (large) cluster; it's responsible for tasks such as disk i/o, load balancing, fault tolerance, and provenance. Finally the astronomical catalogues, along with all relevant metadata, must be stored into databases and made available to the astronomical and general public (including schools --- outreach to students between Kindergarten and 12th grade is an important part of the LSST).

The entire system is of course reliant on a build system, and here we decided to eschew the gnu toolchain (automake, autoconf, gnumake, libtool) in favour of scons, a python-based build system that replaces make, much of the configure part of the auto* tools, and the joys of building shared libraries with libtool. We felt that scons support for multi-platform builds was sufficient, especially in this modern age of ANSI/ISO C++ and Posix 1003.1 compatibility. Additionally, we are using a pure python tool eups to manage our dependencies --- we strongly felt that we needed to support having multiple sets of dependent libraries simultaneously deployed on our machines (and indeed for a developer to be able to use one set in one shell, and a different set in another).

The Application Layer

The application codes are being implemented in C++, exported to python using swig. This is a different approach to that employed by the PyRAF group at the Space Telescope Science Institute [PyRAF] which defines all of its classes in python, making extensive use of numpy and scipy. For example, if your primary need is to be able to read images from disk, manipulate them (e.g. add them together or warp them) and then either pass them to an external program or write them back to disk, a numpy-based approach is a very good fit. In a similar way, if you wish to read a catalogue of objects into a python array, then the objects in the array --- corresponding to the entries in the catalogue --- are naturally created in python.

However, for the LSST, we rejected this solution as we felt that the natural place to create many objects was in C++. For example, given an image of the sky the first stage of processing (after correcting for the instrumental response) is detecting all of the objects present in the image. This isn't a particularly hard problem, but it is not one that's well matched to python as it involves searching through all the pixels in the image to determine connected sets --- and iteration through all the elements of an array is not an array-based extension such as numpy's strong point. On the otherhand, it's very natural in a language such as C or C++; as you detect each source you add its data structure to a list. There are many technologies available for linking python and C/C++ (ctypes, boost::python, swig, pyrex, cython, ...) with various strengths and weaknesses. We chose SWIG because of its non-invasive nature (when it works it simply works --- you pass it a .h file and out pops the python interface), it's level of support, and its high-level semantics --- a C++ std::list<...> becomes a python list; a C++ std::map<string, ...> becomes a python dictionary.

Where we are using Python

As described, our fundamental operations are implemented in terms of C++ classes, and in C++. Where does python fit in? The first place is in writing tests; we have used unittest to write the majority of our (admittedly still small) test suite. Because swig can be used to wrap low-level as well as high-level interfaces, we are able to write tests that would usually be coded directly in C++[6]. A downside of this is that the developer has to be sure that problems revealed are in the C++ code not the interface layer --- but in the long run we need to test both[7].

The next major application of python is as a high-level debugger[8] For example, an C++ object detector returns an std::list of detections; but are they correct? It's easy to write a little piece of python to plot the objects in an image display programme to see if they

[6] Using e.g. boost::test or CppUnit

[7] The classic problems are due to reference counting. E.g. if operator+= is given its usual C++ meaning of returning its argument, swig will generally get the reference counts wrong.

make sense; then to plot the ones that only satisfy a certain condition, and so on. This is making use of python's strengths as a fast-prototyping language, where we can keep the same list of objects while writing visualisation code --- if we were doing this in C++, we'd have to rerun the object detector as well as the display code at each iteration. A more long-lasting aspect of the same style of coding is the quality assurance that a project such as LSST is crucially dependent on. We shall have far too much data to dream of looking at more than a tiny fraction by eye, so extensive suites of analysis programmes will be run looking for anomolies, and such programmes are also naturally written in python.

Finally, we have pushed the C++ interfaces down to quite a low level (e.g. detect objects; measure positions; merge detections from multiple detection algorithms). The modularity desired by the middleware is higher --- more at the level of returning all objects from an image, with properly measured positions. The solution is to write the modules themselves in python, making calls to a sequence of C++ primitives.

Conclusions

We have adopted python as the interactive layer in a large image processing system, and are happy with the results. The majority of the pixel-level code is written in C++ for efficiency and type-safetly, while the pieces are glued together in python. We use python both as a development language, and as the implementation language to assemble scientific modules into complete functioning pipelines capable of processing the torrent of data expected from the LSST telescope.

Appendix

Integration with numpy

There are of course very good reasons for wanting our array classes to map seamlessly onto numpy's array classes --- having been through the numeric/numarray/numpy wars, we have no wish to start yet another schism. There are two issues here: How well our image classes map to (or at least play with) numpy's; and the extent to which our C++ function calls and methods return numpy arrays rather than pure python lists.

Let us deal with the former first. We have a templated Image class which looks much like any other; it may be described in terms of a strided array[9]. This is similar to numpy's 2-D array classes, but not identical. In the past (prior to swig 1.3.27) it was possible to create python classes that inherited from both numpy's ndarray and LSST's Image but this solution was fragile, and we understood the question of exactly who owned memory and when it could be safely deleted was only hazily. Another approach would be to make the LSST image classes inherit from ndarray --- but there we have problems with the C --- C++ barrier. It seems likely that a solution can be implemented, but it may not be clean.

The second question, that of numpy record arrays versus python lists, seems to be purely a matter of policy, and writing the proper swig typemaps. However, it does raise the question of how much one wants numpy's arrays to dominate the data structures of what is basically a python program.

References

[SDSS] http://www.sdss.org
[PyRAF] http://www.stsci.edu/resources/software_hardware/pyraf

[8] A similar approach was taken by the SDSS, but using TCL7.4 bound to C

[9] The internals are in fact currently implemented in terms of NASA's VisionWorkbench (http://ti.arc.nasa.gov/projects/visionworkbench)