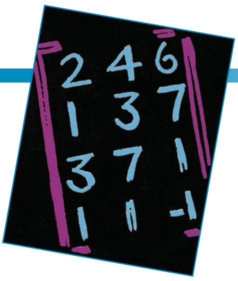
Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu



PYTHON AND XML FOR AGILE SCIENTIFIC COMPUTING

By Michele Vallisneri and Stanislav Babak

To set up a mock data challenge, the authors needed to string together a lot of existing and new code. Here, they describe how Python and XML came to the rescue.

Gravitational waves (GWs) are self-sustaining perturbations of space–time geometry, predicted by Albert Einstein's theory of general relativity. They propagate at the speed of light and have the effect of alternately stretching and squeezing the distance between freely falling masses (as measured by light traveling between them). GWs are emitted by accelerating mass, much like electromagnetic radiation is generated by accelerating charge, but the coupling of GWs with matter is exceedingly weak, so only the waves emitted by very rapidly moving and heavy bodies (typically astrophysical objects of mass comparable to our Sun's) are detectable in practice. The existence of GWs was confirmed by observations of binary pulsar B1913+16, discovered in 1974 by Russell Alan Hulse and Joseph Hooton Taylor Jr. (a feat that won them the 1993 Nobel Prize): the orbit of this binary is shrinking at the rate predicted by the loss of orbital energy to GWs. Researchers have yet to achieve direct detection, but ground-based interferometric GW detectors such as the Laser Interferometer Space Antenna (LIGO) and Virgo are expected to reach the necessary sensitivity within the next decade. In addition, the Laser Interferometer Space Antenna (LISA), a space-borne GW observatory planned jointly by NASA and the European Space Agency, will assuredly detect a wide variety of GW sources throughout the universe, thanks mostly to the quietness of the space environment and to LISA's very long 5-million-km baseline.[1]

Because all GW detectors are by their nature omnidirectional and because GWs are generated by the bulk motion of matter rather than by surface processes, LISA won't produce images; rather, it will produce *time series*, not unlike sound (but in the 0.1-mHz to 0.1-Hz frequency range), consisting of the superposition of the gravitational waveforms from all sources within range, typically thousands of them. We can tease out the signals from individual sources by zeroing in on their specific, complex "fingerprints," much as we can still conduct a conversation with a friend at a noisy party. Still, the analysis of LISA data will be a delicate job—and a very important one because we won't have detections without successful data analysis. In this sense, data analysis is integral to the LISA measurement concept.

As part of LISA's data analysis development activities, its project and science teams have decided to sponsor a series of mock data challenges. Such exercises are widely used in developing and validating scientific experiments characterized by complex data flows, large collaborations, extensive use of information technology, and difficult or unproven data analysis techniques. They involve the generation of mock data sets that impersonate the future output of measurements at varying levels of realism. Scientists then use these data sets to exercise selected parts of the data collection and analysis chain—for example, a mock data challenge for a particle collider might help verify that the data storage system was fast enough to record all triggers or to validate the statistical analysis of event backgrounds.

In the mock LISA data challenges (MLDCs; http://astrogravs.nasa.gov/docs/mldc), the challengers distribute several simulated LISA data sets to challenge participants; the data include GW signals from sources of undisclosed parameters as well as realistic instrument noise. Participants must analyze the data and report their estimates for the GW source parameters. These challenges are meant to be blind tests but not really contests: their greatest benefit comes from the quantitative comparison of results, analysis methods, and algorithm implementations.[2]

The MLDC Workflow

At the beginning of 2006, the LISA science team formed a taskforce whose mission was to prepare and administer the first challenge within a timeframe of a few months. It became immediately apparent that a significant amount of work would be required to produce and assemble the computational pipeline that would generate the MLDC data sets. As Figure 1's orange "task" boxes show, the pipeline involves producing random sets of physical parameters for the GW sources included in the challenge data set; generating the gravitational waveforms for these systems; running the waveforms through faithful models of LISA that simulate its signal response and noises; and merging all the noises and responses to individual sources into the data set distributed to the participants. Figure 2 shows an example of such a data set, with all constituents broken out as separate spectra. The only software components that were already available for this pipeline (the purple boxes in Figure 1) were small stand-alone C programs to produce gravitational waveforms (which we eventually rewrote from scratch in C++) and two larger applications to simulate the LISA response:

- Synthetic LISA (www.vallis.org/syntheticlisa) is a C++ library steered from Python through SWIG wrappers (Simplified Wrapper and Interface Generator; www.swig.org).[3]
- The LISA Simulator (www.physics.montana.edu/LISA) is a file-driven C application.

We felt that only an expressive scripting language such as Python[4] (which has been compared to pseudocode for its readability and terseness) would let us accomplish this task rapidly enough, yet still give us confidence in the code's correctness and let less-experienced programmers (many in our taskforce) participate fruitfully. Indeed, we relied on Python to write the missing pipeline components, including interoperable wrappers for the existing code (the green boxes in Figure 1) along with a master script to glue everything together and run the tasks in sequence. Python made it easy to

- write transparent steering scripts that exploit Python's easy access to the shell and file system—the master script even includes a bare-bones queuing system (based on the Python `subprocess` module) that enables the concurrent processing of tasks on parallel (or multicore) computers;
- wrap the GW-generating C/C++ components using SWIG, which can parse header files and automatically generate Python wrappers for functions, variables, and classes;
- wrap the file-based LISA Simulator—which takes input and writes output to a set of rigidly named files in its own compilation directory—by dynamically creating working directories filled with symbolic links to executables and shared files, and then copying and renaming input and output files before and after running the simulator

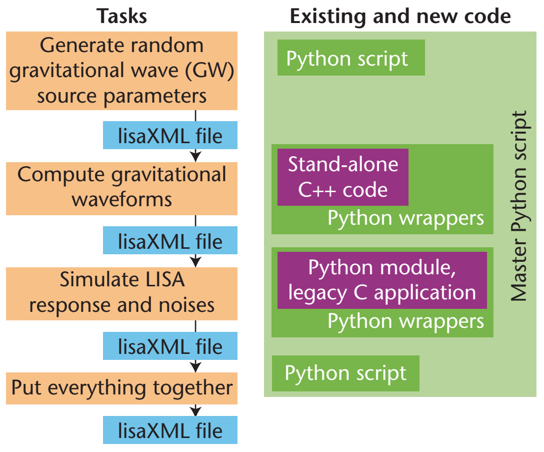


Figure 1. The mock LISA data challenge (MLDC) data-set-generation pipeline. We use the lisaXML format throughout to exchange parameters and data between all tasks in the pipeline (orange boxes). Purple boxes represent software components that were already available when work began on the pipeline; green boxes represent ad hoc Python components and wrappers.

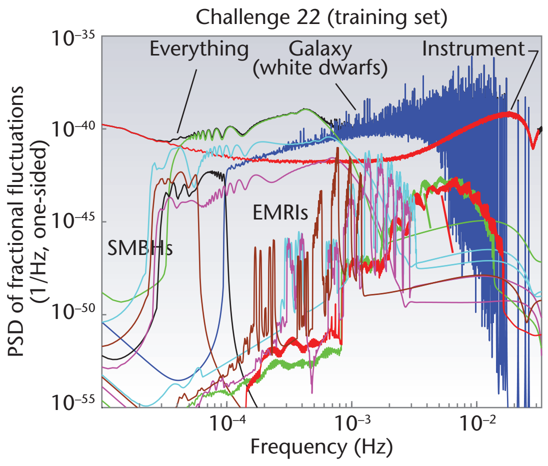


Figure 2. Power spectral density (PSD) plot for a typical mock LISA data challenge (MLDC) data set, broken apart into gravitational wave (GW) source and noise components. Contestants receive only the time series corresponding to the black "everything" curve and must determine the source parameters, labeled here as "SMBHs" for the inspiraling binaries of supermassive black holes; "EMRIs" for the extreme mass ratio inspirals of small bodies into central galactic black holes; and "galaxy" for several millions of lighter white-dwarf binaries in the Milky Way.

Läufer's Lounge

Free Java Test Coverage Tools

In "A Hike through Post-EJB J2EE Web Application, Part III" (*Computing in Science & Eng.*, vol. 9, no. 1, pp. 82–95), we discussed unit testing for Java applications and test coverage as a way to determine how well our tests do their jobs. Back then, we used Coverlipse (http://coverlipse.sourceforge.net), an Eclipse plug-in for code coverage visualization that, unfortunately, supports only up to version 3.2 of Eclipse. Indeed, it didn't work for me on Eclipse 3.3 (Europa), so I had to look for an alternative for my courses and other work.

A student of mine recommended EMMA (http://emma.sourceforge.net), a Java code coverage tool under active development and getting quite a bit of attention. Not only does EMMA support various types of coverage (class, method, line, basic block), it also generates high-level stats that let you drill down into detail as needed (project, package, class, and method level). My student is a fan of command-line tools, whereas I was hoping for an Eclipse plug-in, so luckily for me, there's EclEmma (www.eclemma.org), which packages EMMA as an Eclipse plug-in that works with Eclipse 3.3. It integrates nicely with Eclipse by adding a "Run > Coverage As" menu analogous to the "Run > Run As" menu. It shows its results in the new "Coverage" view as a tree you can drill into, and it indicates coverage by source code highlighting (see Figure A).

Open Source Virtualization

I do most of my work between my Apple MacBook Pro running OS X and my Linux desktop server running Ubuntu. Yet, once in a while, I have to run a Windows application. There are several ways to achieve this goal, and the Wikipedia article on virtual machines (http://en.wikipedia.org/wiki/Virtual_machine) is an excellent starting point, listing tons of virtualization software options, including compatibility layers, emulators, and hardware virtualization.

VirtualBox (www.virtualbox.org) follows the latter approach and caught my attention for several reasons:

- It's available for all three major operating systems running on x86 and AMD64 hardware, including Windows, Linux, and OS X.
- It seems to be the only solution freely available as open source software (binaries are also available for personal use and evaluation).
- It's a high-quality, well-supported professional product under active development.

Installation was a snap on both Ubuntu and OS X, and as I typed up this sidebar using Google Docs in Firefox running on a Windows XP guest machine inside a VirtualBox running on my MacBook, the user experience was close to native except for some printing problems (which could be Windows-related) and a little quirk with the `fn` key. The Mac version is

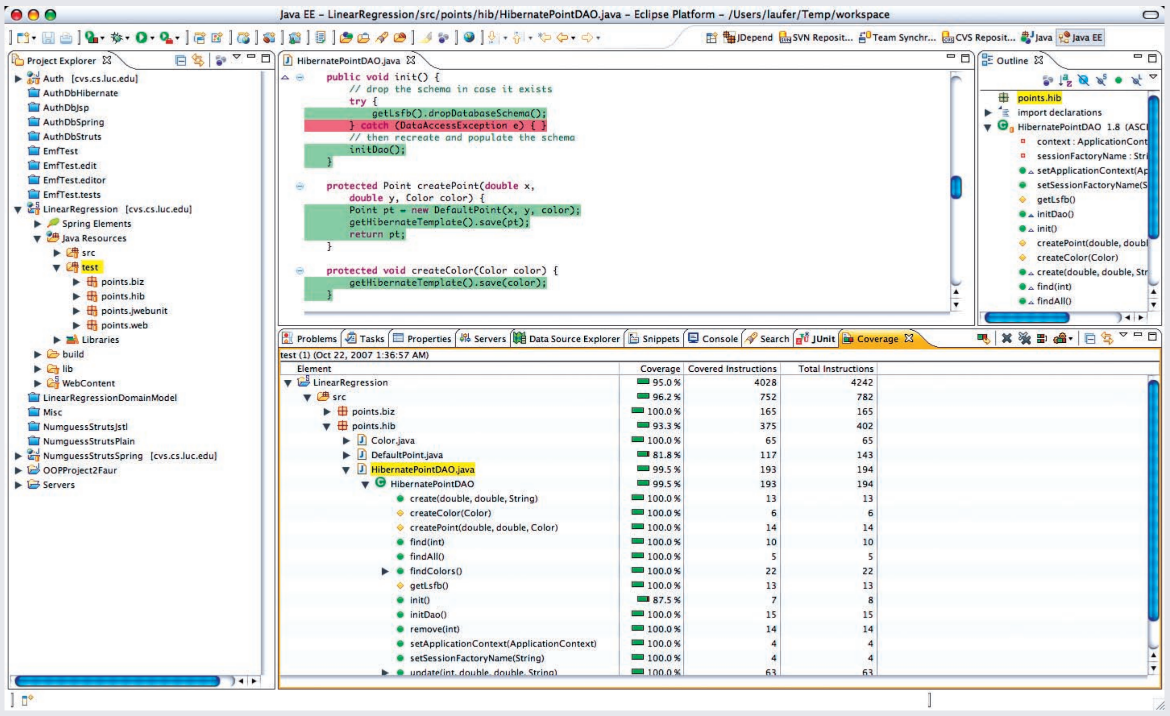
(also letting us run multiple instances of the simulator concurrently);
- write I/O routines for the MLDC data files that would be especially convenient to use from the scripts and Python-wrapped components; and
- provide a master compilation and installation script for the pipeline and all required Python packages. This was paramount because we knew that less experienced contributors would be stymied (in different, creative ways) by having to compile and install packages such as the GNU Scientific Library. This script facilitated interesting synergies with the master pipeline script, such as tagging output data sets with the Subversion revision of the pipeline code (an elementary example of provenance tracking).

In the rest of this article, we offer a narrative of our experiences with the taskforce, focusing particularly on the implementation details we consider most innovative or enabling. We hope this proves useful to readers considering similar endeavors. The entire MLDC code base is available on Google Code (http://lisatools.googlecode.com).

The lisaXML Format

Early on, we made the crucial simplifying decision that all tasks in the pipeline would communicate via a single data format, which we would also use to distribute the data sets to challenge participants and to store challenge *keys* (the undisclosed source parameters) in a secure location. Our format needed to describe both the rather structured data (each source's physical parameters) and the more homogeneous but bulky arrays (the time series of the LISA measurements, at a combined 134 Mbytes for two years of data sampled every 15 seconds).

Text-based formats seemed a reassuring choice (the resulting files would be easier for humans to parse and edit directly) and less dependent on the availability of the right I/O libraries on the users' platforms. XML,[5,6] in particular, seemed promising because the data files would be essentially self-describing, I/O libraries would be plentiful, and the files could even be displayed nicely with standard Web browsers, which can render XML in HTML with the help of application-specific stylesheets. (As an example, try looking at the challenge keys at http://astrogravs.nasa.gov/docs/mldc/round2/datasets.html and then compare


Java EE – LinearRegression/src/points/hib/HibernatePointDAO.java – Eclipse Platform – /Users/laufer/Temp/workspace

```
public void init() {
    // drop the schema in case it exists
    try {
        getLsfb().dropDatabaseSchema();
    } catch (DataAccessException e) { }
    // then recreate and populate the schema
    initDao();
}

protected Point createPoint(double x,
    double y, Color color) {
    Point pt = new DefaultPoint(x, y, color);
    getHibernateTemplate().save(pt);
    return pt;
}

protected void createColor(Color color) {
    getHibernateTemplate().save(color);
}
```

test (1) (Oct 22, 2007 1:36:57 AM)

Element	Coverage	Covered Instructions	Total Instructions
LinearRegression	95.0 %	4028	4242
src	96.2 %	752	782
points.biz	100.0 %	165	165
points.hib	93.3 %	375	402
Color.java	100.0 %	65	65
DefaultPoint.java	81.8 %	117	143
HibernatePointDAO.java	99.5 %	193	194
HibernatePointDAO	99.5 %	193	194
create(double, double, String)	100.0 %	13	13
createColor(Color)	100.0 %	6	6
createPoint(double, double, Color)	100.0 %	14	14
find(int)	100.0 %	10	10
findAll()	100.0 %	5	5
findColors()	100.0 %	22	22
getLsfb()	100.0 %	13	13
init()	87.5 %	7	8
initDao()	100.0 %	15	15
remove(int)	100.0 %	14	14
setApplicationContext(ApplicationContext)	100.0 %	4	4
setSessionFactoryName(String)	100.0 %	4	4
update(int, double, double, String)	100.0 %	63	63

Figure A. Eclipse with EclEmma views. The hierarchical view and Java source code highlighting provide detailed test coverage information.

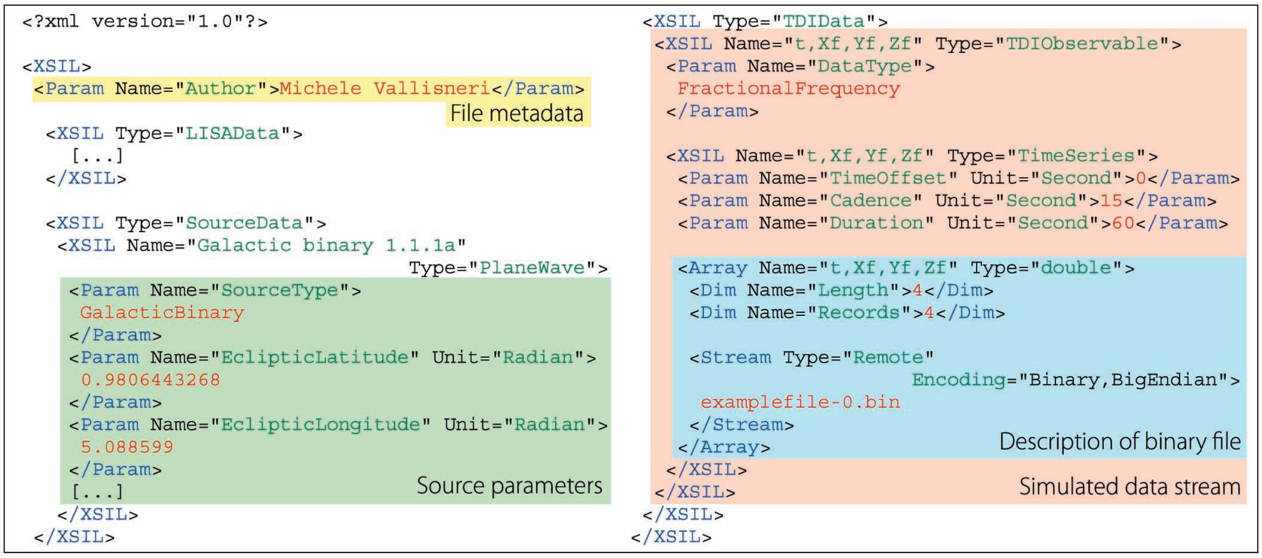
currently one minor release behind the other versions, and I expect the VirtualBox team to fix these issues soon.

Certain features are availably only in the closed source edition targeted at enterprise customers—most notably, access to USB devices such as external drives or Webcams on host machines.

with the XML version by using the "View page source" option in your browser.) The choice of XML eliminated from consideration several standard binary formats for scientific data, such as the Hierarchical Data Format (HDF; http://hdf.ncsa.uiuc.edu), Network Common Data Form (netCDF; www.unidata.ucar.edu/software/netcdf), and Flexible Image Transport System (FITS; http://fits.gsfc.nasa.gov), all of which have XML representations that are meant primarily as interfaces for special parsing tasks or for translating to other formats. On the other hand, given the length of our time series, we longed for the performance that reading and writing binary files can achieve. We ended up choosing the Extensible Scientific Interchange Language (XSIL; www.cacr.caltech.edu/SDA/xsil), dubbed a "flexible, hierarchical, extensible transport [XML] language for scientific data objects" by its creators at Caltech's Center for Advanced Computing Research and used in projects such as LISA's cousin LIGO. Although the enthusiasm for developing scientific XML languages (XSIL included) seems to have peaked around 2001, XSIL still stands out for its simple yet expressive structure based on just eight XML elements and for its ability to link to external binary files containing time series or tabular data: the main XML file describes the binary data's layout and structure, which can be stored locally or accessed remotely through URLs.

The alternative approach of *embedding* the binary data in the XML is hardly practical: we can't just insert a stream of "bits" in an XML document because all the bits therein must represent legal characters in the document's encoding; even if this were somehow arranged, the binary data might contain special sequences, such as `</`, which could confuse the XML parsing. The workaround is to encode the binary data with a limited set of ASCII characters (such as hexadecimal numbers or the XSIL-supported Base64 encoding), but doing this eats away at the very performance gain sought by "going binary" in the first place. (The emerging binary XML standards discussed in the "Binary XML" sidebar should eventually support some form of binary data embedding, but they aren't quite there yet.)

Depending on its use in the pipeline or to distribute data sets, a lisaXML file can include several sections, such as a `SourceData` section to describe gravitational wave

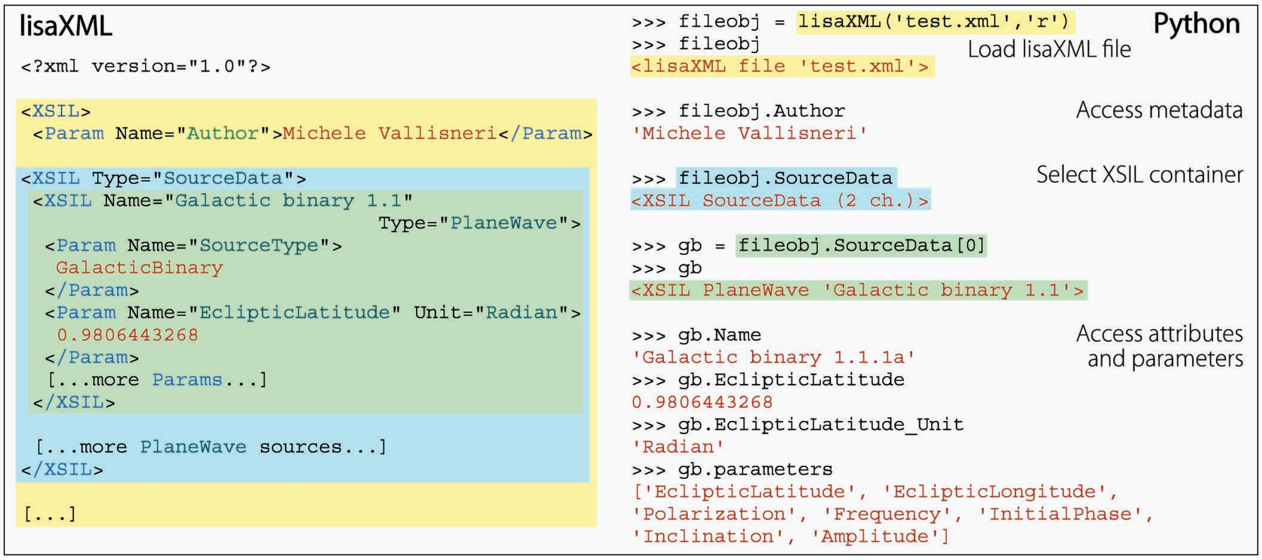


```
<?xml version="1.0"?>

<XSIL>
 <Param Name="Author">Michele Vallisneri</Param>

  <XSIL Type="LISAData">
    [...]
  </XSIL>

  <XSIL Type="SourceData">
   <XSIL Name="Galactic binary 1.1.1a"
                              Type="PlaneWave">
    <Param Name="SourceType">
     GalacticBinary
    </Param>
    <Param Name="EclipticLatitude" Unit="Radian">
     0.9806443268
    </Param>
    <Param Name="EclipticLongitude" Unit="Radian">
     5.088599
    </Param>
    [...]
   </XSIL>
  </XSIL>
```

```
<XSIL Type="TDIData">
 <XSIL Name="t,Xf,Yf,Zf" Type="TDIObservable">
  <Param Name="DataType">
   FractionalFrequency
  </Param>

  <XSIL Name="t,Xf,Yf,Zf" Type="TimeSeries">
   <Param Name="TimeOffset" Unit="Second">0</Param>
   <Param Name="Cadence" Unit="Second">15</Param>
   <Param Name="Duration" Unit="Second">60</Param>

   <Array Name="t,Xf,Yf,Zf" Type="double">
    <Dim Name="Length">4</Dim>
    <Dim Name="Records">4</Dim>

    <Stream Type="Remote"
                   Encoding="Binary,BigEndian">
     examplefile-0.bin
    </Stream>
   </Array>
  </XSIL>
 </XSIL>
</XSIL>
</XSIL>
```

File metadata
Source parameters
Description of binary file
Simulated data stream

Figure 3. The lisaXML format. The Extensible Scientific Interchange Language (XSIL) format can represent complex hierarchical data structures using a few simple XML elements.



lisaXML
Python
Load lisaXML file
Access metadata
Select XSIL container
Access attributes and parameters

```
<?xml version="1.0"?>

<XSIL>
 <Param Name="Author">Michele Vallisneri</Param>

<XSIL Type="SourceData">
 <XSIL Name="Galactic binary 1.1"
                          Type="PlaneWave">
  <Param Name="SourceType">
   GalacticBinary
  </Param>
  <Param Name="EclipticLatitude" Unit="Radian">
   0.9806443268
  </Param>
  [...more Params...]
 </XSIL>

 [...more PlaneWave sources...]
</XSIL>

[...]
```

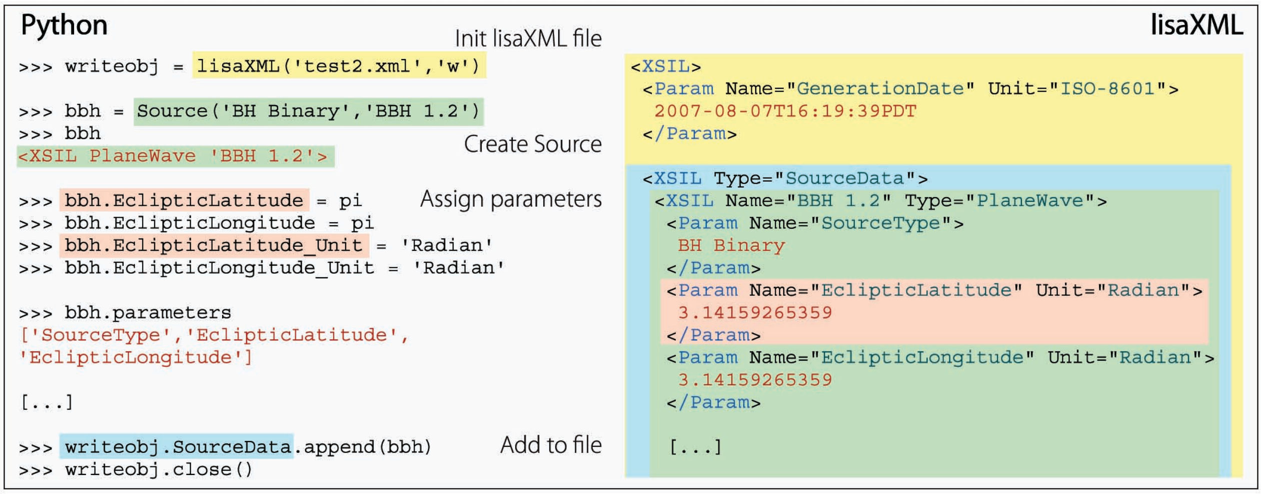
```
>>> fileobj = lisaXML('test.xml','r')
>>> fileobj
<lisaXML file 'test.xml'>

>>> fileobj.Author
'Michele Vallisneri'

>>> fileobj.SourceData
<XSIL SourceData (2 ch.)>

>>> gb = fileobj.SourceData[0]
>>> gb
<XSIL PlaneWave 'Galactic binary 1.1'>

>>> gb.Name
'Galactic binary 1.1.1a'
>>> gb.EclipticLatitude
0.9806443268
>>> gb.EclipticLatitude_Unit
'Radian'
>>> gb.parameters
['EclipticLatitude', 'EclipticLongitude',
'Polarization', 'Frequency', 'InitialPhase',
'Inclination', 'Amplitude']
```

Figure 4. Reading a lisaXML file with the Python `lisaxml` interface. The yellow, light blue, and green highlighting matches the lisaXML sections and containers with the corresponding Python objects.

sources or a `TDIData` section with time series of simulated LISA readouts (see Figure 3). The XSIL format can represent complex hierarchical data structures using a few simple XML elements, such as `<XSIL Name="..." Type="...">`, a generic container used in lisaXML to delimit the file sections and enclose composite objects; `<Param Name="..." Unit="...">`, to represent physical quantities and other atomic data; `<Array>`, `<Dim>`, and `<Stream>`, to represent homogeneous matricial data; `<Table>` and `<Column Name="..." Unit="...">`, for heterogeneous tabular data; and a few others.

Reading and Writing lisaXML in Python

Because we chose Python as the preferred language for scripting and gluing together the MLDC workflow, we needed a good I/O library for lisaXML (the XSIL creators provide a very accomplished parsing library for Java but not for Python). Several good Python modules are available



```
>>> writeobj = lisaXML('test2.xml','w')

>>> bbh = Source('BH Binary','BBH 1.2')
>>> bbh
<XSIL PlaneWave 'BBH 1.2'>

>>> bbh.EclipticLatitude = pi
>>> bbh.EclipticLongitude = pi
>>> bbh.EclipticLatitude_Unit = 'Radian'
>>> bbh.EclipticLongitude_Unit = 'Radian'

>>> bbh.parameters
['SourceType','EclipticLatitude',
'EclipticLongitude']

[...]

>>> writeobj.SourceData.append(bbh)
>>> writeobj.close()
```

```
<XSIL>
 <Param Name="GenerationDate" Unit="ISO-8601">
  2007-08-07T16:19:39PDT
 </Param>

 <XSIL Type="SourceData">
  <XSIL Name="BBH 1.2" Type="PlaneWave">
   <Param Name="SourceType">
    BH Binary
   </Param>
   <Param Name="EclipticLatitude" Unit="Radian">
    3.14159265359
   </Param>
   <Param Name="EclipticLongitude" Unit="Radian">
    3.14159265359
   </Param>

   [...]
```

Figure 5. Writing a gravitational wave (GW) source object to a lisaXML file. The highlighted Python objects in this example map into the file's sections and containers.

for parsing XML, but it wasn't realistic to ask the average taskforce developer to work directly with one of the two standard APIs (SAX, the Simple API for XML, is event-based and therefore requires writing unintuitive handler functions, and DOM, the Document Object Model, is tree-based and so comprehensive that it can easily become unwieldy). Instead we chose a more simple-minded approach (and one married more directly to lisaXML's structure and needs) that was closer to the idea of XML data binding and to the BetterXML/NaturalXML project's aims.[5]

The idea was to match the lisaXML and Python semantics. Look at Figure 3 for the structure of lisaXML files, which consist of trees of nested `<XSIL>` containers; our interface represents them as instances of an `XSILobject` class that extends the Python list object, so it can contain its `<XSIL>` children very naturally (they can be accessed as `xsilobj[0]`, `xsilobj[1]`, and so on). The `Name` and `Type` of `<XSIL>` elements are assigned naturally to the attributes `xsilobj.Name` and `xsilobj.Type`; in fact, Python attributes are so convenient that we used them in several additional ways:

- to provide access to `<XSIL>` children by `Name` (for example, `rootxsil.SourceData`, where `rootxsil` is the outermost `XSIL` element of the lisaXML file);
- to store any `<Table>` or `<Array>` contained in an `<XSIL>` element; and
- to provide access to the `<Param>` defined for an `<XSIL>` element (for example, `xsilobj.EclipticLatitude` for a GW source's sky position; `xsilobj.parameters` keeps the list of `<Param>` types that have been defined in order to distinguish them from the `Name`'d `XSIL` children).

The resulting lisaXML I/O interface is very natural to use, following Python's principle of minimum surprise; Figures 4 and 5 give examples of reading and writing lisaXML files. The `lisaxml` library supports creating and writing lisaXML trees, as well as reading and rewriting them after modification.

If you need some technical details, the lisaXML arrays and tables stored in external binary files are read and written directly to and from Numerical Python (`numpy`) arrays,[7] swapping bytes if necessary to read data written on platforms with different *endianness* (the ordering of the bytes that comprise a multibyte data type, such as a floating-point number). The attribute-mapping "magic" is achieved via Python's introspection capabilities, such as `__dict__`, `__getattr__`, and `__setattr__`. The `lisaxml` library is built on top of the very fast C/Python module `pyRXP`, which reads an entire XML file in one go and returns a simple tree structure of Python "tuples" of the form

```
(tagname,attrdict,children|content,[more]),
```

where `tagname` is a string (such as `'Param'` for the first `Param` element in the green box of Figure 3); `content` is a list of more tuples describing children elements or text data (just `['GalacticBinary']` for our `Param` element); `attrdict` is a Python dictionary (`{'Name': 'SourceType'}`); and the fourth element is sometimes used for more advanced parsing functions. An equally swift but more elegant XML interface is the `ElementTree` module, which is in the Python 2.5 standard library.

All in all, `lisaxml` takes up roughly 1,000 lines of Python, which are devoted mostly to the management of binary files; the code that translates back and forth between XML and Python objects is very terse, again thanks to Python's expressiveness and introspection (try that in C!).

Binary XML

The emerging binary XML standards seek to reduce the size and improve the reading and writing performance of XML documents, while preserving the language's flexibility by encoding the entire infoset (the abstract class of data structures that can be expressed with XML, as opposed to their explicit realization in the XML syntax). In addition, binary formats should ease the inclusion of typed data such as images, sound, and the large numerical arrays useful in many scientific applications, which are represented awkwardly in standard text-oriented XML.[1]

In response to these aspirations, the World Wide Web Consortium (W3C) chartered the XML Binary Characterization (XBC) working group to scope out the use cases and desired properties of binary XML implementations; among the properties the working group is discussing is support for embedding binary data and standard data types. Working from the XBC recommendations, the W3C Efficient XML Interchange (EXI) working group has since released a format specification based on the pre-existing Efficient XML, which distills the XML syntax into an event grammar that can be mapped to variable bit-length tokens, with the shortest sequences reserved for the most common occurrences. Work is ongoing on actual implementations.

Sun Microsystems' competing Fast Infoset format (recently adopted as an ISO specification) is based on similar principles, but is more mature, with open source and commercial libraries already available in several languages (though not Python). Other binary XML formats focus on specific applications (multimedia players or mobile phones) or can be described more accurately as compressing textual XML, rather than directly encoding the XML infoset.

The overall impression is that the situation is still very fluid, and it might be too early to embrace binary XML in mainstream scientific applications. It would be good to revisit the question in a few years, with the hindsight of the initial experiments that are now beginning to appear in the literature. In the meantime, we can adopt more simple-minded methods to reduce storage size (such as gzip-compressing textual XML) to improve the reading and writing of large data sets (such as the hybrid approach discussed in the main text of this article).

Reference

1. J. Kangasharju and S. Tarkoma, "Benefits of Alternate XML Serialization Formats in Scientific Computing," *Proc. Workshop on Service-Oriented Computing Performance: Aspects, Issues and Approaches,* ACM Press, 2007, pp. 23–30; http://portal.acm.org/citation.cfm?id=1272457.

The greatest advantage of using an introspective language such as Python rather than a compiled language for this application is that, although there's some regularity in the lisaXML files' structure (which we've used to some extent in designing the `lisaxml` interface), the structure is still very fluid: for instance, different lisaXML files can have different `<Param>` elements defined in different containers, users might want to include annotations from their own analysis pipelines, and so on. In Python, we can change data object structures on the fly to accommodate such differences, whereas doing so would be difficult in a language such as C without adding layers of indirection (and therefore complicating the interface).

The true power of having such a simple, intuitive interface became clear when it was time to ask taskforce members to develop GW-generating plug-ins for the MLDC architecture. Figure 6 shows an example. The C++ program `BBHChallenge1.cc` generates gravitational waveforms for a system of two inspiraling black holes. We use it by first instantiating a `BBHChallenge1` object and then calling its `ComputeWaveform` method. We plugged this code into the MLDC pipeline by using SWIG to generate a Python wrapper: except for a little SWIG boilerplate, all that we required was a new Python class (`BlackHoleBinary`) to extend `lisaxml Source`. Doing so automatically ensures that by the time the pipeline calls the `waveforms` method, the lisaXML interface has read source parameters from lisaXML and assigned them to the instance attributes of the same name, ready to be used in the `BBHChallenge1` constructor. Indeed, such wrappers are simple enough that we can easily adapt them from a template for any additional GW source class; they're also transparent and easy to debug.

Collaborative computational projects carry many risk factors (getting things running fast and keeping them running for an extended time; merging contributions from multiple developers and managing diverse groups of users on different platforms; dealing with even moderately complicated workflows) that can result in unnecessary delays, duplicated effort, steep learning curves, difficult-to-maintain code bases, and even (alas!) incorrect results. We learned that we can fight such complexity with simplicity, and we found strong allies in powerful, flexible, and expressive tools such as Python and XML. At its best, scientific programming becomes the transparent and elegant embodiment of scientific notions, equations, and processes into code, and many *CiSE* readers are familiar with the satisfaction and enjoyment that come from occasionally reaching such highs. We think that Python and XML can help in this quest.

```
%module BBH
[...]
%include "BBHChallenge1.hh"

%pythoncode %{
import lisaxml, numpy

class BlackHoleBinary(lisaxml.Source):
  def waveforms(self,samples,deltat,inittime):

    bbh = BBHChallenge1(self.Mass1,self.Mass2,[...])

    hp = numpy.empty(samples,'d')
    hc = numpy.empty(samples,'d')

    bbh.ComputeWaveform(hp,hc,deltat,inittime)

    return hp,hc
%}
```

This is a SWIG interface, which will create a Python module with wrappers for all functions and classes defined in the C++ header. It can also add Python code

All MLDC GW source objects extend lisaxml.Source to get access to lisaXML-declared parameters, and define waveforms(...) to provide access to the C/C++ waveform-generation capabilities

We initialize an instance of the C++ source object as we would in C++, using the parameters put in place by the Python lisaxml.Source "factory"

We allocate two empty *numpy* arrays...

...and fill them with appropriate C++ call (the SWIG numpy.i *typemap* makes sure that the numpy objects appear as simple C arrays to C++)

Figure 6. Wrapping gravitational wave (GW)-generating code for use in the mock LISA data challenge (MLDC) pipeline. The text on the right in this example explains what's happening on the left.

Acknowledgments

Michele Vallisneri's work was supported by the LISA Mission Science Office and by the Human Resources Development Fund at the Jet Propulsion Laboratory, California Institute of Technology, where it was performed under contract with NASA.

References

1. J. Baker et al., eds. "LISA: Probing the Universe with Gravitational Waves," LISA Mission Science Office whitepaper, 2007; www.lisa-science.org/resources/talks-articles/science/lisa_science_case.pdf.
2. K.A. Arnaud et al., "An Overview of the Second Round of the Mock LISA Data Challenges," *Classical and Quantum Gravity*, vol. 24, no. 19, 2007, pp. S551–S564.
3. T.L. Cottom, "Using SWIG to Bind C++ to Python," *Computing in Science & Eng.*, vol. 5, no. 2, 2003, pp. 88–96.
4. *Computing in Science & Eng.*, special issue on Python, vol. 9, no. 3, 2007.
5. G.K. Thiruvathukal, "XML and Computational Science," *Computing in Science & Eng.*, vol. 6, no. 1, 2004, pp. 74–80.
6. G.K. Thiruvathukal and K. Laufer, "Natural XML for Data Binding, Processing, and Persistence," *Computing in Science & Eng.*, vol. 6, no. 2, 2004, pp. 86–92.
7. T.E. Oliphant, "Python for Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 10–20.

Michele Vallisneri is a research scientist at NASA's Jet Propulsion Laboratory in Pasadena, California; he is also co-chair of the Mock LISA Data Challenge taskforce. His research interests include gravitational waves and computational physics. Contact him at michele.vallisneri@jpl.nasa.gov; via www.vallis.org.

Stanislav Babak is a relativist in the Astrophysical Relativity research team of the Max-Planck-Institut für Gravitationsphysik in Golm, Germany. His research interests include general relativity and data analysis for ground-based and space-based gravitational wave detectors. Contact him at stba@aei.mpg.de.