# Natural programming in Python for scientific applications: the case of the Mock LISA Data Challenges

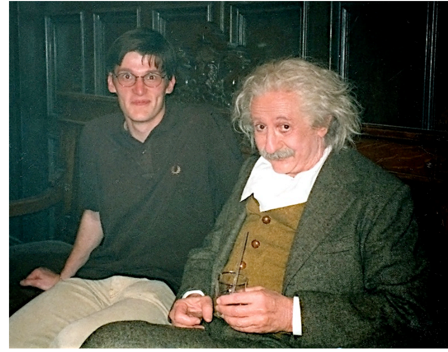
Michele Vallisneri
Jet Propulsion Laboratory

Copyright 2010 California Institute of Technology

# I should first establish my credentials to talk about:

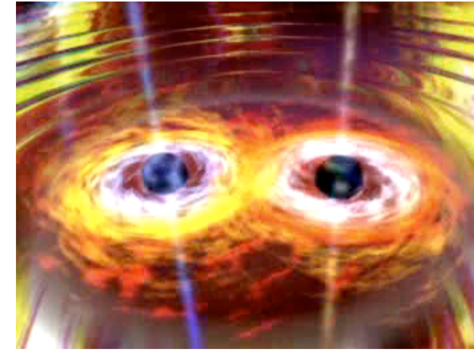
Gravitational-wave astronomy

Einstein's general relativity

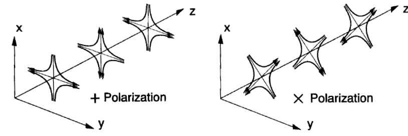


Gravitational waves are propagating fluctuations of spacetime curvature, emitted by massive bodies in rapidly accelerated motion...

Such as black holes...



...that form in-spiraling binaries.

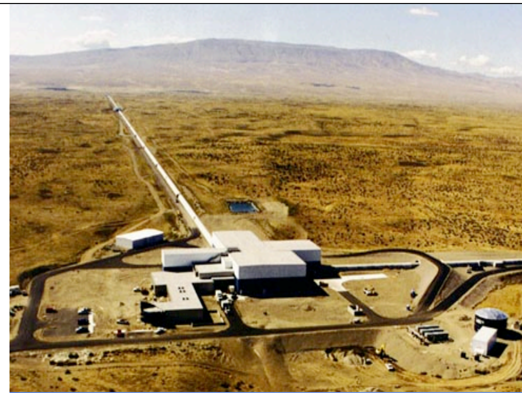



...and detected as transverse oscillations in the distance between test masses.

---

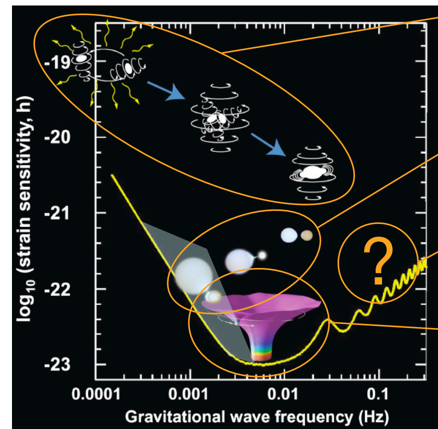
Gravitational waves...

- have typical strength $10^{-21}$
- interact weakly with matter
- are emitted by bulk motions
- are phase coherent
- Detectors are omnidirectional and do not form images



## NRC: “LISA [is] an entirely new way of observing the universe, with immense potential to enlarge our understanding of both physics and astronomy in unforeseen ways”

- LISA science received the highest ranking in the NRC Beyond Einstein review
- LISA will detect and characterize many thousands of individual GW sources, as well as the diffuse background from millions more



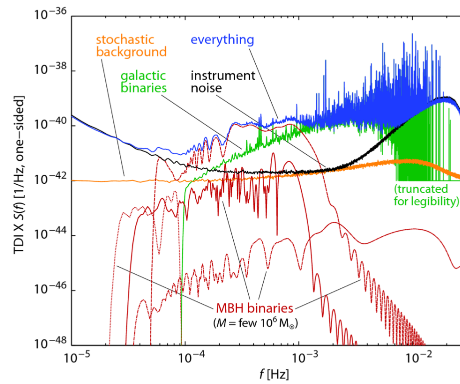

MBH mergers
- study the coevolution of galaxies and MBHs
- measure accurate distances of high-*z* objects
- test GR in the nonlinear regime

Galactic binaries
- study the astrophysics of binary stellar evolution, including the common envelope phase

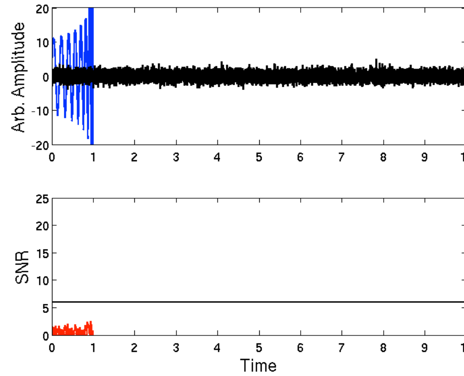
bursts and stochastic backgrounds
- look for new physics from the early Universe and string theory

extreme–mass-ratio inspirals (EMRIs)
- study MBHs and their environment in the dense nuclei of galaxies
- map BH spacetimes and test cosmic censorship

GW signals are (mostly) submerged in detector noise:
source detection and parameter estimation are never trivial, and always fun.
(See MV, Class. Quant. Grav. 26, 094024, for a nonpractitioner's intro.)




Most LISA sources are long-lived, so LISA data analysis poses the cocktail-party problem of separating and reconstructing 1000s simultaneous signals




The most precise weapon in GW data analysis is matched filtering—computing the cross-correlation of detector data with "all possible" theoretical waveform shapes

## When GW signals are not sufficiently separated (e.g., Galactic binaries at nearby frequencies), they must be fit simultaneously

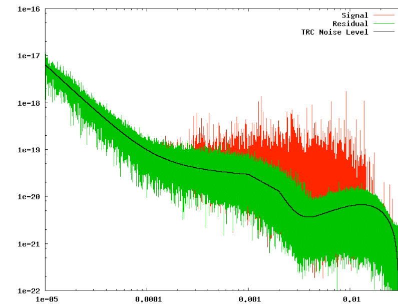
**Iterative strategy**

1. Find the strongest source, and add it to the catalog;
2. Simultaneously re-fit parameters for all the sources in catalog;
3. Subtract the catalog sources from detector data;
4. Repeat

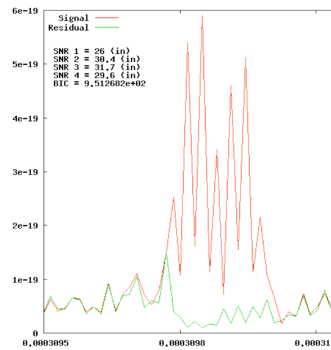
**Global strategy**

Sample the global multi-source parameter space à la Monte Carlo.
(Many jumping tricks may be required for good mixing;
it is however sufficient to fit together blocks of tens of Galactic binaries.)

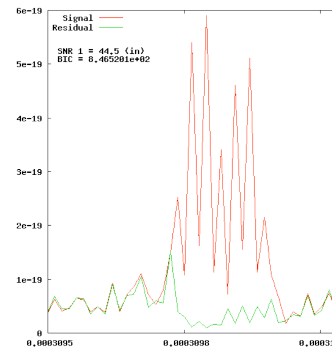
Cornish and Crowder's block-fitting Markov-Chain Monte Carlo (2007) successfully recovered 20,000 Galactic binaries from a simulated population of 30 million



# However, in some cases we must accept ambiguous results!




Template blending: multiple templates match a single true source




Source blending: multiple sources matched by a single template

[Crowder 2007]

And also:

Quasi-degeneracies: e.g., MBH binaries at antipodal sky positions

Sorting problem: in a multi-modal, multi-source posterior probability distribution, which source is which in each mode?

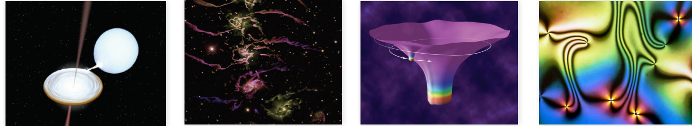
To develop our tools and methods, we started the Mock LISA Data Challenges:

- issue simulated datasets with synthetic noise and GWs of undisclosed parameters
- participating groups return parameter estimates and compare methods

---

Phase I (MLDC 1, 1B): establish common standards for the LISA orbits, noises, response.
2006–2007
Test recovery and parameter estimation of simple sources:

- Galactic binaries (verification, isolated, moderately interfering)
- isolated, nonspinning SMBH binaries



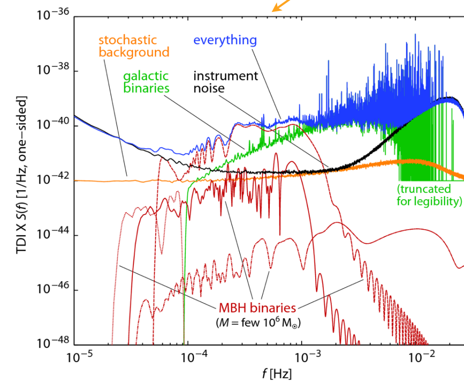
Phase II (MLDC 2, 3): test recovery and parameter estimation of astrophysical sources:
2007–2008

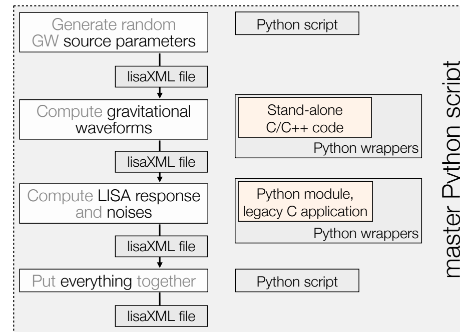
- 30 million interfering Galactic binaries
- nonspinning/spinning SMBH binaries (on top of Galaxy confusion)
- extreme–mass-ratio inspirals (EMRIs)
- cosmic-string bursts; stochastic backgrounds

Phase III (MLDC 4, 5): face global-fit problem; analyze real-world data with non-ideal noise
2009–

So far: 70 participants from 25 institutions, 30+ papers

# So how did we make this in 3 months? With Python.




- Reading and writing a custom data format for challenge solutions, intermediate files, final datasets for distribution
- Wrapping existing C and C++ codes for generating waveforms and noise, and simulating the LISA response
- Scripting high-level science functions
- Steering (and installing!) the whole pipeline

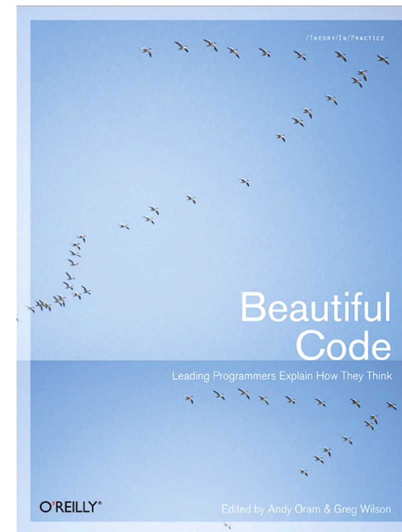
# Beautiful code and natural programming

For a scientist, code is beautiful when

- It expresses mathematical constructs clearly and economically
- It represents the flow of information transparently
- It enables simplicity and beauty as defenses against complexity
- It "feels like home"

Python can help because

- It provides high-level math objects and libraries (e.g., numpy)
- It is object-oriented, but casually so
- It is expressive (Python/C = 6) and introspective
- It just feels right (but see the Zen of Python)




Mathematical constructs—side-by-side comparisons, code should be transparently parallel; Fortran was formula translator
Flow of information—easy inspection by collaborators and by our future selves
Simplicity and beauty—symmetry is a guiding principle for physicists... easy to see when a crystal is broken
Feels like home—naturalness

## Indeed, the Zen of Python could have been written by a physicist

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough
to break the rules.
Although practicality beats purity.
...

Physicists try to build their own little experiment, isolated and protected from the world...

## For our data format, we adopted CACR's XSIL dialect of XML

### Why XML?

- We figured a text-based format[1] would be reassuring to neophytes (easy to parse, less dependent on I/O libraries)
- Thus, we eliminated standard binary[2] formats for scientific data (HDF, FITS)
- XML offered ample I/O libraries, self-describing data, and nice formatting in web browsers with XSLT[3] and Javascript

[1] For our long time series, binary-file performance was very desirable. Linked raw files + XML headers offer the best of both worlds. "Binary XML" formats (Fast Infoset, BiM, even Protocol Buffers) really have different use cases.

[2] They all have XML implementations, but those are mostly useful in transcoding...

[3] The nastiest language I've ever met. Turing-complete, but boy you have to sweat it!

### What XML?

- XSIL (Extensible Scientific Interchange Language), is a "flexible, hierarchical, extensible transport language for scientific data objects"
- It is based on eight simple XML elements (XSIL, Param, Array, Table, Stream, ...)
- It is used (not well) in LIGO, and in a few other CACR projects.
- It allows linking (even remote) of "raw" floating-point data streams.

- If I had to do it again: probably the Virtual Observatory's VOTable.

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

  <XSIL Type="SourceData">
    <XSIL Name="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>
      <Param Name="Polarization"      Unit="Radian">
        3.703239
      </Param>
      [...]
    </XSIL>
  [...]
  </XSIL>

  <XSIL Type="TDIData">
    <XSIL Name="t,Xf,Yf,Zf" Type="TimeSeries">
      <Param Name="TimeOffset" Type="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">
          examplefile-0.bin
        </Stream>
      </Array>
    </XSIL>
  </XSIL>
</XSIL>
```

GW sources

source parameters

time series

linked binary array

XSIL source

**Mock LISA Data Challenge XML File Format, v. 1.0**

**File Info**

| **Authors** | mergeXML.py | |
|---|---|---|
| **GenerationDate** | 2010-02-25T15:17:37CET | ISO-8601 |

```
challenge4.0 (frequency), source seed = 5860216, noise seed = 5860216, LISAtools
SVN revision 1149
lisaXML 1.0 [M. Vallisneri, June 2006]
```

**Source data [+/-]**

**SMBH-1 (PlaneWave)**

| SourceType | FastSpinBlackHoleBinary | |
|---|---|---|
| EclipticLatitude | 1.27472112256 | Radian |
| EclipticLongitude | 3.31085042806 | Radian |
| PolarAngleOfSpin1 | 2.99040169508 | Radian |
| PolarAngleOfSpin2 | 0.133340540849 | Radian |
| AzimuthalAngleOfSpin1 | 2.48731570862 | Radian |
| AzimuthalAngleOfSpin2 | 4.80316790452 | Radian |
| Spin1 | 0.691835240154 | MassSquared |
| Spin2 | 0.373503188309 | MassSquared |
| Mass1 | 3368177.53564 | SolarMass |
| Mass2 | 379447.00194 | SolarMass |
| CoalescenceTime | 7393873.60158 | Second |
| PhaseAtCoalescence | 0.872784027283 | Radian |
| InitialPolarAngleL | 2.20157082198 | Radian |
| InitialAzimuthalAngleL | 1.76096605001 | Radian |
| Distance | 4889885610.17 | Parsec |
| Polarization | 0 | Radian |

**TDI data [+/-]**

**challenge4.0 (TDIObservable)**

| DataType | FractionalFrequency | |
|---|---|---|

**TimeSeries: t,Xf,Yf,Zf**

| Cadence | 15.0 | Second |
|---|---|---|
| TimeOffset | 0.0 | Second |
| Duration | 62914560.0 | Second |
| **Array Stream: t,Xf,Yf,Zf** | Filename | challenge4.0-training-frequency-0.bin |
| | Encoding | Binary,LittleEndian |
| | Type | |
| | Unit | |
| | Length | 4194304 |
| | Records | 4 |

Firefox rendering (not really the same file...)

# We then needed a natural interface for XML from Python

- We need to read, write, edit lisaXML files. Expressive data binding is crucial to natural scripts
- So we wrote our own intuitive IO interface to mirror the semantics of Python and lisaXML: <Param>s → attributes; <Array>s → numpy arrays; <Table>s → iterators
- (DOM was too complicated... SAX too clumsy... we chose RXP—quick, simple, and a little dirty

```
<?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']
```

# Armed with the lisaXML interface, we used SWIG to wrap our legacy C/C++ code

- SWIG connects programs and libraries written in C/C++ with many high-level languages.
- It requires little boilerplate (but allows wrappers to do smart things) and has especially strong Python integration, including numpy.
- This code is simple enough that it can be copied and adapted by non-Python savvy contributors

```
%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 module... it will create Python wrappers for all classes declared in the C++ header BBHChallenge1.hh

All source classes inherit from lisaxml.Source, which gives them access to lisaXML parameters

The only new method we need!

Initialize a source instance as we would in C++, using a natural syntax for parameters

Then call the C++ code that generates waveforms (using a little typemap magic to pass numpy arrays)

## Last, we wrote Python scripts for high-level science functions, to tie the pipeline together, and for a global installer script

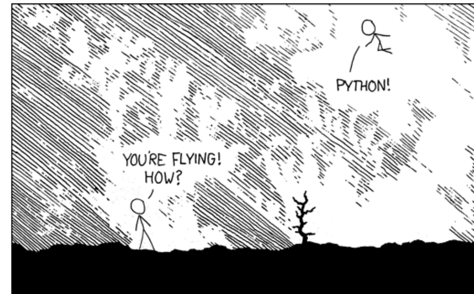
- To: write high-level science scripts: e.g., to choose GW-source parameters randomly
- Do: enjoy Python expressiveness and intuitive lisaXML interface:
  bbh.Longitude = random.uniform(0,2*pi)

- To: wrap legacy applications that read and write from fixed filenames
- Do: "fool" the applications by symlinking them into temporary directories, renaming files

- To: tie command-line applications (e.g., to make source, LISA response) into pipeline
- Do: write a master script using Python's OS, file-system and regex capabilities.
  Even throw in basic queueing/multiCPU functionality with Python's subprocess

- To: implement a master installer (all libraries and codes) for helpless remote users
- Do: let Python (not make!) do the wgets, run setup.py and configure/make/make install

## So it's almost like that...



- It's important to have fun!

But:

- Choose your packages wisely
- Beware of kludges and hacks (fix them before they bite you)
- Try to avoid most idioms (e.g., sincx = x and sin(x)/x or 1)
- Document as you write (yeah, right)

And:

- Remember that scientists are not native speakers of computer languages
- Software-development theory does not really apply to scientific programming

Packages: numpy good, matplot lib bad (unsteady API)
Worst kind of hack: fixing somebody else's package at runtime
Idioms: things that a non-native speaker cannot figure out logically

Non-native speakers: eventually we'll make an embarassing mistakes
Software development for scientific programming—just do your best, pick your examples