Keeping the Chandra Satellite Cool with Python

Tom Aldcroft
CXC Operations Science Support
Smithsonian Astrophysical Observatory

Boston Python Meetup
May 23, 2012

github.com/taldcroft
Python is the right tool for serious science and engineering analysis

- Ease of development and strong interactive analysis environment
- NumPy (fast numerical support)
- IPython (interactive development)
- Matplotlib (plotting, GUI support)
- SciPy (algorithms)
- Easy integration with C and Fortran
- Good options for parallelization
- PyTables (HDF5 for big-ish data)
- Sherpa (fitting and modeling)
- (and much much more...)
- Free!

PSR B1509-58
go to "Chandra Hand of God"
Chandra X-ray Observatory launched by NASA in July 1999

- X-rays reveal energetic processes from black holes, supernova explosions, massive galaxy clusters, pulsars, and more
- Factor of 8 better angular resolution than any other X-ray observatory
- Operating superbly since launch with many ground-breaking discoveries

Fun Fact #1
Chandra is the largest satellite ever launched by the Space Shuttle. It was first proposed to NASA in 1976 by Harvey Tannanbaum and (now) Nobel Laureate Riccardo Giacconi.
Ionizing particle radiation environment is worse than expected and is degrading the silverized teflon insulation.

Ever since launch spacecraft temperatures have been rising steadily.

No repair possible since Chandra is in a high-Earth elliptical orbit.

Fun Fact #2
This beautiful artists rendition depicts Chandra pointing almost at the Sun. This makes ops folks queasy because if that really happened the sensitive optics and instruments would quickly fry.
Different parts of the spacecraft get too warm or cold depending on the relative Sun angle and internal power draw.

Observing schedule is carefully planned to prevent overheating or freezing.

*This process requires accurate predictive models of spacecraft temperatures*
Thermal modeling strategy

**Problem:**
Making detailed time-dependent thermal models by traditional finite element modeling is not feasible (too difficult and too slow).

- Passive cooling to space (2.7K)
- Detailed FEM calculation of radiative and conductive heat transfer
- Thousands of nodes!
Thermal modeling strategy

**Solution:**

Hybrid approach - Simple physical model with empirical data-driven training of *many (up to 100)* model parameters

- Passive cooling to space (2.7K)
- Detailed FEM calculation of radiative and conductive heat transfer
- Thousands of nodes!

**Focal plane for primary X-ray detector**

**X-rays**

**Solar heating**

**Cold radiator**

- Thousands of nodes!
Thermal modeling strategy

Solution:
Hybrid approach - Simple physical model with empirical data-driven training of many (up to 100) model parameters

\[ \frac{dT_0}{dt} = U (T_1 - T_0) \]  # Coupled masses
\[ \frac{dT_0}{dt} = P \]  # External heating

\[ \text{Heat} = k(T_{\text{set}} - T_{FP}) \quad \text{if} \ T_{FP} < T_{\text{set}} \]

A handful of nodes!
Thermal modeling strategy

Solution:
Hybrid approach - Simple physical model with empirical data-driven training of many (up to 100) model parameters

Requirements:
- Fast access to large amounts of training data (years of thermal values)
- Flexible way to generate and evaluate thermal models
- Fast model evaluation (predict one year in < 1 second)
- Fitting infrastructure to robustly handle many parameters
Developing models requires fast access to years of spacecraft data

- Chandra records science and engineering data onboard at 32 kbit/s
- Delivered as a giant packed structure with > 6000 distinct data items (”MSIDs”)
- “Official” telemetry access tools spec'd to 1990's hardware
- These tools can take hours to retrieve a year of data (10^7 elements/hour...)

HDF5 with PyTables is awesome!

- Tables with 10^{10} elements are fine, fast random access and easy appends
- Automatic compression even as new data are appended
- PyTables is simple to use with good documentation and examples

For Chandra telemetry:

- *Use one huge table per MSID covering the entire duration of the mission*
- Entire telemetry archive is less than 300 Gb
- Data retrieval speed ~10^7 elements/sec from NetApp disk (*fast enough*)
- Fast telemetry access has driven migration from Matlab to Python at OCC
Create a new HDF5 table and append data

```python
import tables

def make_h5_file(dat, filename, colname, n_rows):
    """Make a new HDF5 table to hold numpy array data \``\dat``\n    ""
    filters = tables.Filters(complevel=5, complib='zlib')
    h5 = tables.openFile(filename, mode='w', filters=filters)

    h5shape = (0,) + dat.shape[1:]
    h5type = tables.Atom.from_dtype(dat.dtype)
    h5.createEArray(h5.root, 'data', h5type, h5shape, title=colname,
                    expectedrows=n_rows)

    h5.close()

def append_h5_file(dat, filename):
    """Append data to an existing HDF5 table """
    h5 = tables.openFile(filename, mode='a')
    h5.root.data.append(dat)
    h5.close()
```
PyTables and HDF5

Read values from HDF5 file

```python
22 def read_h5_file(h5_slice, filename):
23     """Read a slice of an HDF5 file"""
24     h5 = tables.openFile(filename)
25     dat = h5.root.data[h5_slice]
26     h5.close()
27     return dat
```
Xija – thermal modeling framework

Thermal models can be expressed as a linear 1\textsuperscript{st} order set of coupled ODEs

\[
\frac{d\vec{Y}}{dt} = \vec{A}(t, \vec{Y}, \vec{p}) \cdot \vec{Y} + \vec{B}(t, \vec{Y}, \vec{p})
\]

- Model free parameters
- Vector of temperatures
- Coupling matrix
- External heat sources

Xija thermal modeling framework provides:

- Class-based model components for easy extensibility
- Simple way of defining thermal models in Python
- Fast C code to integrate the corresponding ODEs
- Connection to the Sherpa fitting package for determining model parameters
- GUI fit tool to manage complex data inputs and parameters
Xija – model components

- SolarHeat (HeatPower)
- PropHeater (HeatPower)
- Coupling (ModelComponent)
- Camera body
- Focal plane
- Heater
- Earth heating
- Sun
- Attitude
- Ephemeris
- EarthHeat (HeatPower)
- Passive cooling to space (2.7K)
- SIM - X
- Node (ModelComponent)
- HeatSink (ModelComponent)
import xija

# Create Xija model named "dpa" (Digital Processor Assembly)
mdl = xija.ThermalModel('dpa')

# Add node for DPA temperature 1DPAMZT in telemetry
dpamzt = mdl.add(xija.Node, '1dpamzt')

simz = mdl.add(xija.SimZ)  # Science Instrument Module posn
pitch = mdl.add(xija.Pitch)  # Sun angle w/r/t spacecraft
eclipse = mdl.add(xija.Eclipse)  # Eclipse status

# Solar heating of the DPA electronics box
mdl.add(xija.DpaSolarHeat, dpamzt, pitch_comp=pitch,
        simz_comp=simz, eclipse_comp=eclipse)

# External heat sink (cooling to rest of spacecraft and space)
mdl.add(xija.HeatSinkRef, dpamzt)

# Self-heating of DPA electronics due to internal power draw
mdl.add(xija.AcisDpaStatePower, dpamzt)

# Nearby thermostatically controlled proportional heater
mdl.add(xija.PropHeater, dpamzt)
Doing ODE integration in pure Python is too slow... Not a problem.

Many options exist for calling C code from Python / NumPy
- NumPy C-API (slightly intimidating but good examples exist)
- Cython (wrap C code or write fast C-like Python)
- ctypes standard library module
- .. and many more. See e.g. http://www.scipy.org/PerformancePython

Xija uses a ctypes interface because it is just so simple.
Xija – a bit of speed

```c
int calc_model(int n_times, int n_preds, int n_tmals, double dt,
               double **mvals, int **tmal_ints, double **tmal_floats)
{
    // C code
}
```

```python
@property
def core(self):
    """Lazy-load the "core" ctypes shared object library that does the low-level model calculation via the C "calc_model" routine. Only load once by setting/returning a class attribute.
    """
    if not hasattr(XijaModel, '_core'):
        loader_path = os.path.abspath(os.path.dirname(__file__))
        _core = np.ctypeslib.load_library('core', loader_path)
        _core.calc_model.restype = ctypes.c_int
        _core.calc_model.argtypes = [
            ctypes.c_int, ctypes.c_int, ctypes.c_int,
            ctypes.c_double,
            ctypes.POINTER(ctypes.POINTER(ctypes.c_double)),
            ctypes.POINTER(ctypes.POINTER(ctypes.c_int)),
            ctypes.POINTER(ctypes.POINTER(ctypes.c_double))
        ]
        XijaModel._core = _core
    return XijaModel._core
```
Xija – a bit of speed

```python
model.py
450 mvals = convert_type_star_star(self.mvals, ctypes.c_double)
451 tmal_ints = convert_type_star_star(self.tmal_ints, ctypes.c_int)
452 tmal_floats = convert_type_star_star(self.tmal_floats, ctypes.c_double)
453
454 self.core.calc_model(self.n_times, self.n_prets, len(self.tmal_ints),
455 dt, mvals, tmal_ints, tmal_floats)
```

```python
setup.py
1 core_ext = Extension('xija.core', ['xija/core.c'])
2 setup(name='xija',
3     version=version,
4     description='Thermal modeling framework for Chandra',
5     author='Tom Aldcroft',
6     author_email='aldcroft@head.cfa.harvard.edu',
7     url='https://github.com/sot/xija',
8     license='BSD',
9     platforms=['any'],
10    ext_modules=[core_ext],
11   packages=['xija', 'xija.component'],
12   package_data={'xija': ['libcore.so']},
13 )
```
Fitting model parameters

These thermal models contain up to 100 free parameters (conductances, solar heating vs. pitch, long-term and annual variations)

CIAO Sherpa modeling and fitting package to the rescue:

- Powerful model language
  - Complex models as a Python expression
  - Parameter linking, freezing
- Good optimization methods for many-parameter fits (e.g. Monte-carlo / Levenberg-Marquardt hybrid to avoid local minima)
- Good documentation and support
- 15 years of professional development and use in astronomy

Good reasons to worry about fitting so many parameters, but IT WORKS.

http://cxc.harvard.edu/contrib/sherpa
http://cxc.harvard.edu/sherpa/
Fitting model parameters – GUI tool

Fit, Save, and Add component plots

Specify and control model parameters

Visualize model inputs and predictions with embedded matplotlib plot windows
Fitting model parameters – video demo
Embedding Matplotlib plot in a GUI

```python
import gtk
import numpy as np

from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(1, 1, 1)
# Do something (make a plot)

canvas = FigureCanvasGTKAgg(fig) # a gtk.DrawingArea
win.add(canvas)

win.show_all()
gtk.main()
```

See http://matplotlib.sourceforge.net/examples/user_interfaces/
Putting it all together

Four certified (NASA Level-4 review board) thermal models are being used in Chandra operations:

- Formal command load review process verifies that the schedule of planned observations will not thermally damage spacecraft hardware.

Developing higher-fidelity thermal models for Chandra was a difficult task that was made possible by the Python ecosystem.
This is the 2\textsuperscript{nd} year my colleagues and I have run the Python for Astronomers Workshop series [1] at the Harvard/Smithsonian Center for Astrophysics:

- Six hands-on workshops with an emphasis on using Python to solve real world problems encountered in research.
- At the most recent workshop 60\% of participants were women.

Please visit AstroPy [2] if you are interested in developing astronomy tools!

The reason we bother
BACKUP SLIDES
Beer

Chandra X-ray Observatory
Operations Control Center

Microsoft NERD Center
Parallelization

- Computation is easily parallelized by splitting into independent time segments
- Simple code extension with mpi4py with MPICH2
- Master-worker architecture:
  - Master controls the fit process and initialization
  - Workers read in thermal data, calculate model and $\chi^2$ fit statistic over time segment
Parallelization with MPI

In the master program replace data initialization, model and fit statistic calculation functions with new functions:

```python
comm = MPI.COMM_SELF.Spawn(sys.executable,
                         args=['fit_worker.py'],
                         maxprocs=n_workers)

def init_workers(metadata):
    """Init workers using values in metadata dict""
    msg = {'cmd': 'init', 'metadata': metadata}
    comm.bcast(msg, root=MPI.ROOT)

def calc_model(pars):
    """Calculate the model for given pars""
    comm.bcast(msg={'cmd': 'calc_model', 'pars': pars},
                root=MPI.ROOT)

def calc_stat():
    """Calculate chi^2 diff between model and data""
    msg = {'cmd': 'calc_statistic'}
    comm.bcast(msg, root=MPI.ROOT)
    fit_stat = numpy.array(0.0, 'd')
    comm.Reduce(transfer_list, [fit_stat, MPI.DOUBLE],
                op=MPI.SUM, root=MPI.ROOT)
    return fit_stat
```
The main logic of the master fit program is nearly unchanged except for the addition of code to dynamically spawn workers:

```python
init_workers({'start': date_start, 'stop': date_stop})

# Sherpa commands to register and configure a function
# as a user model
load_user_model(calc_model, 'mpimod')
add_user_pars('mpimod', parnames)
set_model(mpimod)

# Configure the fit statistic
load_user_stat('mpistat', calc_stat)
set_stat(mpistat)

# Do the fit
fit()
```
The fit worker code just waits around to get instructions:

```python
comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

while True:
    msg = comm.bcast(None, root=0)
    if msg['cmd'] == 'stop':
        break
    elif msg['cmd'] == 'init':
        x, y = get_data(msg['metadata'], rank, size)
    elif msg['cmd'] == 'calc_model':
        model = calc_model(msg['pars'], x, y)
    elif msg['cmd'] == 'calc_statistic':
        fit_stat = numpy.sum((y - model)**2)
        comm.Reduce([fit_stat, MPI.DOUBLE], None, op=MPI.SUM, root=0)

comm.Disconnect()
```
Parallelization speedup

- The speedup obtained is useful
- Parallel fraction = 0.94
- Ultimate speedup = 16