Advanced Sherpa

Tom Aldcroft
CXC Operations Science Support

CIAO Workshop
Feb. 3, 2010
Sherpa

**Generalized fitting package with a powerful model language to fit 1D and 2D data**

**Basic Sherpa**
- Interactive (command line) usage
- Scripting using command syntax
- Data access with `show_*` and `print`

```python
sherpa> load_pha('acis_pha3.fits')
sherpa> set_source(xsphabs.abs1 * powlaw1d.p1)
sherpa> subtract()
sherpa> fit()
sherpa> show_fit()
```

Optimization Method: LevMar
- `name` = `levmar`
- `ftol` = `1.19209289551e-07`
- `xtol` = `1.19209289551e-07`
- `gtol` = `1.19209289551e-07`
- `maxfev` = `None`
- `epsfcn` = `1.19209289551e-07`
- `factor` = `100.0`
- `verbose` = `0`

Statistic: Chi2Gehrels
- `Fit:Dataset` = `1`
- `Method` = `levmar`
- `Statistic` = `chi2gehrels`
- `Initial fit statistic` = `6.83386e+10`
- `Final fit statistic` = `37.9079` at function evaluation `22`
- `Data points` = `44`
- `Degrees of freedom` = `42`
- `Probability [Q-value]` = `0.651155`
- `Reduced statistic` = `0.902569`
- `Change in statistic` = `6.83386e+10`

- `pl.gamma` = `2.15852`
- `pl.ampl` = `0.00022484`

This works very well *most* of the time, but ...
Doing more with Sherpa

Python Inside: Sherpa user interface and high-level functions are Python

• Sherpa provides an interface to let users:
  • Access the internal objects used within Sherpa
  • Easily define user model and user statistic functions
  • Use Sherpa as an imported library in your python program
• Paradigm change – CIAO/Sherpa is not the environment, it is a powerful library tool in your Python analysis environment.
• Move beyond short scripts to full-blown programs¹.
• Real life examples:
  • Fit models to hundreds of Chandra L3 sources (including faint ones), put results in a database, and import to a google web app to browse the results.
  • Fit data where the errors are dominated by quantization (i.e. data are integerized)
  • Generate complex thermal models requiring parallel fitting using > 30 CPUs

¹See http://www.astropython.org for more about Python and astronomy
Topics

Take another swig of coffee and get ready for some code

- Getting data into Sherpa
- Digging into Sherpa: getting at the objects underneath
- Creating user models and user statistics functions
  - Using functions
  - Using classes (you too can write an OOP)
- Parallelization with MPI

Sit back and relax

- Deproject: a Sherpa extension module
- Keeping Chandra cool: a Sherpa success story
Getting data into Sherpa

- Sherpa has many ways of loading data and other things:

```
sherpa-1? load_<TAB>
load_arf   load_bkg_arf
load_arrays load_bkg_rmf
load_ascii  load_colormap
load_ascii_transform load_conv
load_bkg    load_data
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
load_arrays           load_bkg_rmf          load_filter           load_pha              load_state            load_user_model
```

- One of my favorites doesn't appear in any Sherpa thread\(^1\): `load_arrays()`
- This provides a generic way to load *memory arrays* as Sherpa datasets
- Example:
  - ASCII file in a format not understood by Sherpa. Instead use `asciitable`\(^2\)

```
sherpa> load_data('csc.rdb[cols ra, dec]')
IOErr: opening file has failed with ERROR - Failed to open 'csc.rdb[cols ra, dec]'.

sherpa> import asciitable
sherpa> dat = asciitable.read('csc.rdb', Reader=asciitable.RdbReader)
sherpa> load_arrays(1, dat['ra'], dat['dec'], Data1D)
```

- Didn't we just replace one line with three? But now we own the data!

```
sherpa> dat = asciitable.read('csc.rdb', Reader=asciitable.RdbReader)
sherpa> ra = dat['ra']
sherpa> dec = dat['dec']
Sherpa> dist = calc_dist(ra, dec, ra.mean(), dec.mean())
sherpa> load_arrays(1, dist, dat['mag'], Data1D)
```

- Works for 2-D and PHA data as well.

\(^1\) From the google search “sherpa load_arrays”
\(^2\) [http://cxc.harvard.edu/contrib/asciitable](http://cxc.harvard.edu/contrib/asciitable)
Digging into Sherpa: getting the good bits

- Sherpa also has many ways of showing the current analysis state:

```
sherpa> show_<TAB>
show_all         show_bkg_model   show_conf        show_data        show_fit         show_method      show_proj        show_source
show_bkg         show_bkg_source  show_covar       show_filter      show_kernel      show_model       show_psf         show_stat
```

```
sherpa> load_pha('acis_pha3.fits')
sherpa> set_source(xsphabs.abs1 * powlaw1d.p1)
sherpa> subtract()
sherpa> fit()
sherpa> show_fit()
```

- Great for interactive analysis but what about using the results?
- OLD school
  - Run as a script and pipe output to a file
  - Write a separate script (perl?) to parse and store in a new table
- Writing code to reliably parse all these tidbits is a very fun and interesting way to spend your day. NOT.
- Nice shiny way
  - Run as a python script
  - Directly access results and store in desired format .. or use python twitter API to immediately tweet the results.
Digging into Sherpa: getting the good bits

- Sherpa lets you `get_*` what you need:
Most everything you get_*() will be a python object and that's the prize
- Internally Sherpa uses hierarchical objects for most things
- You can find and examine internal object attributes by <TAB> digging

With some care you can manipulate the internal object attributes

```python
sherpa> load_pha(1, 'acis_pha3.fits')
sherpa> dataset = get_data(1)
sherpa> dataset
<DataPHA data set instance 'acis_pha3.fits'>
sherpa> dataset.<TAB>
Display all 159 possibilities? (y or n) n
sherpa> dataset.get_<TAB>
...
scherpa> counts = dataset.counts
sherpa> b = numpy.where(dataset.counts > 3)
scherpa> b
(array([ 14,  16,  30,  45,  97, 118]),)
scherpa> c = dataset.channel[b]
scherpa> e = dataset._channel_to_energy(c)
array([ 0.2117,  0.2409,  0.4453,  0.6643,  1.4235, 1.73])
scherpa> dat = asciitable.read('csc.rdb', Reader=asciitable.RdbReader)
scherpa> load_arrays(1, dat['ra'], dat['dec'], Data1D)
scherpa> dataset = get_data()
scherpa> dataset.y = dataset.x**2
sherpa> dataset.staterror = dataset.y / 20
```
Digging into Sherpa: source and fit results

- Now something more useful: examine source model parameters and fit results

```python
sherpa> source = get_source()
sherpa> source.parts
<XSphabs model instance 'xsphabs.abs1'>,
<PowLaw1D model instance 'powlaw1d.p1'>)

sherpa> for par in source.pars:
    print par.fullname, par.val, par.min, par.max, par.frozen

abs1.nH 0.112891604641 0.0 100000.0 False
p1.gamma 3.0762703235 -10.0 10.0 False
p1.ref 1.0 -3.40282346639e+38 3.40282346639e+38 True
p1.ampl 0.000112120110443 0.0 3.40282346639e+38 False

sherpa> fit = get_fit_results()
sherpa> print [x for x in dir(fit) if not x.startswith('_')]  
['covarerr', 'datasets', 'dof', 'dstatval', 'extra_output', 'format', 'istatval', 'message', 'methodname', 'modelvals', 'nfev', 'numpoints', 'parnames', 'parvals', 'qval', 'rstat', 'statname', 'statval', 'succeeded']
```
Digging into Sherpa: source and fit results

- Don't just examine. **Organize** and tabulate!

```python
import sqlite3
conn = sqlite3.connect('csc_fits.db')
c = conn.cursor()
c.execute('''create table fit_pars
    (source_name text, par_name text, par_val real)''')
for parname, parval in zip(fit.parnames, fit.parvals):
    c.execute("insert into fit_pars values (?, ?, ?)",
            (source.name, parname, parval))
conn.commit()
c.close()
```

ccosmos% sqlite3 csc_fits.db
SQLite version 3.3.6
Enter ".help" for instructions
sqlite> select * from fit_pars;
(xsphabs.abs1 * powlaw1d.pl)  | abs1.nH | 0.112891604640818
(xsphabs.abs1 * powlaw1d.pl)  | pl.gamma| 3.07627032349591
(xsphabs.abs1 * powlaw1d.pl)  | pl.ampl | 0.00011212011044343
User models with python functions

• Adding a user model defined with a python function is **shockingly simple!**

```python
def myline(pars, x):
    return pars[0] * x + pars[1]

load_user_model(myline, "myl")
add_user_pars("myl", ["m","b"])
set_source(myl)

myl.m=30
myl.b=20
```

Sure, but any real model has to be written in C or fortran, right? Not necessarily.

• Numerical processing with **numpy** is in C so any vectorized calculations are fast.

• The **scipy** library provides a large selection of optimized numerical algorithms using well known fortran and C numerical libraries.

• Prototype the user model in Python. If it's too slow then profile the code and convert the hot spots to C or C++.

But what about my existing C / fortran model code? Google "sherpa user models".
User models with Python classes

- Frequently a user model function requires associated metadata (atomic data, table file names, non-fitted parameters, etc)
- This is a typical problem in fitting (remember fortran COMMON blocks?)
- Python provides a very clean solution: classes

```python
import numpy
import pyfits

class FITS_TableModel:
    """Simplest possible FITS table model. Table has two columns:
    kT : temperature
    spectrum : corresponding spectrum in an N-element array
    In this model the spectrum nearest in temperature is returned.
    The energy bins of the fitted spectrum is ignored here.
    """
    def __init__(self, filename):
        hdus = pyfits.open(filename)
        self.kT = hdus[1].data.field('kT')
        self.spectra = hdus[1].data.field('spectrum')
        hdus.close()

    def __call__(self, pars, x):
        kT = pars[0]
        i = numpy.searchsorted(self.kT, [kT])[0]
        if (kT - self.kT[i-1]) < (self.kT[i] - kT):
            i -= 1
        return self.spectra[i]

user_model_func = FITS_TableModel('plasma_spectra.fits')
load_user_model(user_model_func, "myspec")
add_user_pars("myspec", ["kT"])
set_source(myspec)
```

Ever wonder what's the deal with "object oriented programming"? Here it is. The object stores metadata.

Object initialization with "filename" via `__init__`. Read the FITS data and store within the object.

Here's the magic: the object created by the class can be called directly as a function and it will run the special `__call__` method.
Parallelization

- For some problems with large datasets or computationally intensive models it may be possible to improve fit performance by using multiple processors.
- Processors can be on the same machine or in a networked cluster.
- Sherpa already takes advantage of multiple cores in projection and conf.
- Improving fit performance is tricky for convolved models but easy for models that can be split in data space\(^1\).

\(^1\)Splitting in data space is just one of many possible strategies
Parallelization with MPI

- Can do parallel processing using C and Python implementations of the widely used Message Passing Interface standard.

```python
class CalcModel(object):
    def __init__(self, x, y):
        msg = {'cmd': 'init', 'x': x, 'y': y}
        comm.bcast(msg, root=MPI.ROOT)

    def __call__(self, pars, x):
        comm.bcast(msg={'cmd': 'calc_model', 'par': par}, root=MPI.ROOT)
        return numpy.ones_like(x)  # Dummy value of correct length

def calc_staterror(data):
    return numpy.ones_like(data)

class CalcStat(object):
    def __call__(self, data, model, staterror=None, syserror=None, weight=None):
        msg = {'cmd': 'calc_statistic'}
        comm.bcast(msg, root=MPI.ROOT)
        fit_stat = numpy.array(0.0, 'd')
        comm.Reduce(None, [fit_stat, MPI.DOUBLE], op=MPI.SUM, root=MPI.ROOT)
        return fit_stat.tolist(), numpy.ones_like(data)

comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['fit_worker.py'],
                           maxprocs=nproc)
load_arrays(1, x, y)
load_user_model(CalcModel(x, y), 'mpimod')
add_user_pars('mpimod', parnames)
set_model(1, mpimod)
load_user_stat('mpistat', CalcStat(), calc_staterror)
set_stat(mpistat)
fit(1)
```
Parallelization with MPI

The fit_worker code just waits around to get instructions.

def calc_model(pars, x):
    # calculate the model values
    return model

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':
        i = numpy.int32(numpy.linspace(0.0, len(msg['x']), size+1))
        i0 = i[rank]
        i1 = i[rank+1]
        data_x = msg['x'][i0:i1]
        data_y = msg['y'][i0:i1]
    elif msg['cmd'] == 'calc_model':
        model = calc_model(msg['pars'], data_x)
    elif msg['cmd'] == 'calc_statistic':
        fit_stat = numpy.sum((data_y - model)**2)
        comm.Reduce([fit_stat, MPI.DOUBLE], None, op=MPI.SUM, root=0)

comm.Disconnect()
Deproject: a Sherpa extension module

Deproject is a CIAO Sherpa extension package to facilitate deprojection of two-dimensional annular X-ray spectra to recover the three-dimensional source properties.

- The deproject module creates a framework for manipulation of a stack of related input datasets and their models.
- Most of the functions resemble ordinary Sherpa commands (e.g. set_par, set_source, ignore) but operate on a stack of spectra.
Keeping Chandra cool: a Sherpa success story
Keeping Chandra cool: a Sherpa success story
Keeping Chandra cool: a Sherpa success story

**SOT PSMC model**

- Key inputs to model are pitch angle, SIM-Z position and ACIS power.
- Total of 13 model coefficients.
Keeping Chandra cool: a Sherpa success story

Calibration: Sherpa

Predictions for mission planning