UI6A. User Environments and Scripting Languages: Embedding vs Extending

Date: 16 Aug 2002 (draft v0)
Contributors: J. Chiang (GSFC-UMBC)

Purpose

Analyzing data interactively generally presupposes the existence of a user environment. Assuming that we will want to automate certain complex or repetitive analysis tasks, that environment will need a scripting or macro capability. A central implementation issue is whether to embed a scripting language into our own home-brewed user-environments or to provide our basic analysis tasks as extensions of existing scripting languages. This memo summarizes the arguments for and against each approach and serves as a brief manifesto in favor of scripting in general.

Embedding

A relevant example of an environment with an embedded scripting language is XSPEC, the X-ray spectral fitting program that uses Tcl. A user runs the program by typing `xspec` at the shell command line and thence enters the XSPEC environment. A number of commands are available that implement specific analysis tasks. A typical sequence might look something like the following[1]:

```
XSPEC> data phafile           ! specify the data file to be analyzed
XSPEC> model wabs*(pow + gauss) ! define a spectral model consisting
                              ! of three components: intervening
                              ! absorption, a power-law continuum,
                              ! and a Gaussian emission line
XSPEC> fit                    ! fit the model to the data
XSPEC> plot ldata chi         ! plot the data, the fit, and the
                              ! contribution to chi^2
XSPEC> save model my_model    ! save the model and fitted parameter
                              ! values to a file
XSPEC> newpar 7 2e-5          ! set the value of parameter 7 by hand
XSPEC> eqw 3                  ! compute the equivalent width of
                              ! component 3
XSPEC> error 6                ! compute the errors for parameter 6
```

Note that each command performs a single, well-defined task. Now, suppose that we have a series of `.pha` files — `file1, file2, ..., file20` — that contain data for observations of a single source over twenty different epochs; and suppose that we wish to characterize the spectral variation of the source over time. We could enter,

[1]For brevity, the screen output of each command and certain input steps, such as the entering of starting values for the various model parameters, have been omitted.

by hand, something like the above sequences of commands for each `.pha` file; or we could execute the following Tcl script:

```
tcsh> cat fit_many_files.tcl
for {set i 1} {$i < 21} {incr i} {
   data file$i
   @my_model
   fit
   save model my_model$i
}
```

Here the XSPEC command `@my_model` loads the saved model and parameters. To run this script from XSPEC one types

```
XSPEC> source fit_many_files.tcl
```

and twenty model files — `my_model1.xcm,..., my_model20.xcm` — are created that contain the best-fit parameters for each epoch. These data can then be processed off-line or from within XSPEC to extract additional information, such as equivalent widths or uncertainties. Tcl scripting gives one the flexibility to customize and perform these tasks rather easily.

In XSPEC, Tcl is fully interpreted at the command line, so that one can still do some repetitive tasks without writing a special script. This loop will compute the fitted line equivalent widths for each of the twenty epochs:

```
XSPEC> for {set i 1} {$i < 21} {incr i} {@my_model$i; eqw 3}
```

One could also write Tcl procedures to augment the existing list of native XSPEC commands. A fairly straight-forward example is the script `http://lheawww.gsfc.nasa.gov/~jchiang/SSC/makeerrors.tcl` which computes uncertainties for a number of model parameters automatically. The embedding of Tcl in XSPEC thus provides the two principal benefits of scripting: repetitive and time-consuming sequences of tasks can be automated; the basic set of analysis tasks can be augmented by writing procedures.

There are some downsides to the embedding approach. First, a specific choice of scripting language is imposed on the user by the software developers. Although Tcl has some syntax features that are desirable from the implementation perspective, those same features are not very intuitive and make it difficult to use. The developers could choose a more user-friendly language such as Python, but that would likely make the implementation more difficult, since in order to incorporate the scripting language along with the analysis commands, a lot of programming work has to be done. Specifically, the software must parse and separate the scripting language commands from the analysis commands. The former must then be fed to the proper scripting language API routines for execution while the latter must trigger the corresponding function calls within the C/C++ code itself.

It is worth noting that if the user interface and basic analysis commands are properly designed, the user should be able to analyze her data without using the scripting language. The scripting capability doesn't benefit her in this case, but she can still use the software to do her analysis.

Extending

In the extending approach, one encodes the analysis tasks as *modules* that can be loaded by an already existing scripting or macro language. The coding could be done in the scripting language itself, or for faster execution time, in C/C++. In this case, the scripting language's own interactive command line interpreter (assuming it has one) becomes the user environment. At first glance, this approach may seem no less difficult to implement than embedding. If we use C/C++, our code still has to go through the API to communicate with the scripting language; C constructs such as structures or even simple arrays usually do not have directly compatible representations in the host scripting language — special accessor functions would have to be written; and there are certain technical issues (such as keeping track of reference counts) that are specific to each language and its API. Therefore, we may have to pick just one or two languages to extend and thereby still enforce a scripting language choice on the user. Finally, because each scripting language has its own syntactic quirks, the user would probably still need to have some knowledge of at least one of the supported languages.

Almost all of these issues would be serious considerations were it not for SWIG (Simplified Wrapper and Interface Generator, `http://www.swig.org/`). SWIG is a development tool that generates the wrapper code required to make C/C++ functions available as methods in a number of different languages, including Perl, Python, Tcl, Ruby, and Java. One can take the same C/C++ functions and compile extensions for each of the supported languages almost effortlessly. All of the API calls are automatically coded for each language and accessor functions are created to manipulate the contents of the C/C++ data structures. Furthermore, if the scripting language is used simply as "glue", as in the XSPEC/Tcl examples above, the accessor functions are largely unnecessary since SWIG provides pointers to C/C++ arrays and structures that can be passed to the extension modules just as in normal C/C++ function calls.

To see how well SWIG works, I wrote a simple data analysis package, `line_fit`, that fits a straight line to (x,y) data. It has two commands, one to read the data from a file and another to do the linear regression. The source code, compilation, and sample sessions for Perl, Python, and Tcl are given in the appendix. After sorting out how it all works in one language, it was trivial to implement and use in the other two.

Summary

If you haven't already guessed, I'm a big fan of extending, especially after trying a simple embedding exercise in Python (see `http://www.python.org/doc/current/ext/high-level-embedding.html`) and having a peek at the XSPEC source code. Here's my summary of the reasons for and against either embedding or extending:

Embedding

- For:
  - The user doesn't have to learn the scripting language to analyze the data. (Although she does have to learn our command syntax, but...)
  - We can tailor the command syntax of our tasks to suit the needs and desires of the users.
- Against:
  - We impose a scripting language choice on the user.
  - It's a substantial amount of programming effort to implement. (Check out the XSPEC source code if you don't believe me.)

Extending

- For:
  - It's much easier to implement. SWIG allows for trivial support of the big three scripting languages — Perl, Tcl, and Python — plus several more.
  - The user gets to use her favorite scripting language(s).
  - Extension writing encourages desirable design features, such as task compartmentalization, and using SWIG, quick prototyping and development.
  - We can direct our efforts at the analysis algorithms, rather than implementing the user interface.
- Against:
  - The user must learn at least one supported language; plus, Perl doesn't have an interactive command line interpreter. However, the Tcl environment is very Xspec-like, and a GUI could protect the casual user from ever directly interacting with a scripting language.
  - This is somewhat uncharted territory. This model for implementing an analysis environment as an extension of a traditional scripting language hasn't really been done before in high energy astronomy to my knowledge. However, CIAO's Sherpa and CHiPS programs are effectively implemented as extensions of S-Lang; and adding cfitsio as a class in ROOT extends that language. Furthermore, tools have been written in IDL for analyzing EGRET data and HST data — those examples amount to extensions of IDL. There is a difference, however, between traditional scripting languages such as Perl and Tcl and languages such as S-Lang, ROOT, and IDL.

Appendix: The `line_fit` Extension Module

With some minor tweaking of the makefiles, I was able to compile and run `line_fit` on a couple different linux boxes, all running different versions of the various scripting languages.

Source Code

```
tcsh> cat line_fit.c
#include <stdio.h>
#include <math.h>
#include "my_globals.h"

int fit() {
     int i;
     float Sx=0., Sy=0., Sxx=0., Sxy=0., det;

     for (i=0; i<npts; i++) {
          Sx += xx[i];
          Sy += yy[i];
          Sxx += xx[i]*xx[i];
          Sxy += xx[i]*yy[i];
     }
     det = abs(npts*Sxx - Sx*Sx);
     if (det != 0.) {
          intercept = (Sy*Sxx - Sxy*Sx)/det;
          slope = (Sxy*npts - Sy*Sx)/det;
          chi2 = 0.;
          for (i=0; i<npts; i++) {
               chi2 += pow((yy[i] - (slope*xx[i] + intercept)),2);
          }
          return 0;
     } else {
          fprintf(stderr, "fit error: det = 0\n");
          return 1;
     }
}

int read_data(char file[]) {
     float x, y;
     int i=0;

     FILE *fp;
     if ((fp = fopen(file, "r")) == NULL) {
          printf("read_data: can't open %s", *file);
          return 1;
     }
     while (fscanf(fp, "%e  %e", &x, &y) != EOF && i < NMAX) {
          xx[i] = x;
          yy[i] = y;
          my_data[i].x = x;
          my_data[i].y = y;
/*          printf("%e  %e\n", x, y);*/
          i++;
     }
     npts = i;
     fclose(fp);
     return 0;
}

float get_fltarr_val(float array[], int i) {
     return array[i];
}

void print_struct_array(struct point data[]) {
     int i;
     for (i=0; i<npts; i++) {
          printf("%6.2f   %6.2f\n", data[i].x, data[i].y);
     }
}

tcsh> cat my_globals.h
#define NMAX 100

float slope, intercept, chi2;
int npts;
float xx[NMAX], yy[NMAX];

struct point {
      float x;
      float y;
};

struct point my_data[NMAX];
```

Here's the "interface" file required by SWIG to generate the wrapper code:

```
tcsh> cat line_fit.i
%module line_fit
%{
```

```
#include "my_globals.h"
%}

extern float intercept, slope, chi2;
extern int npts;
extern float xx[NMAX], yy[NMAX];
extern struct point my_data[NMAX];

extern int fit();
extern int read_data(char file[]);
extern float get_fltarr_val(float array[], int i);
extern void print_struct_array(struct point data[]);
```

Compilation

For Python,

```
tcsh> make
gcc -c line_fit.c -I/usr/include/python1.5
swig -python line_fit.i
gcc -c line_fit_wrap.c -I/usr/include/python1.5
ld -shared line_fit.o line_fit_wrap.o -o line_fitmodule.so
```

for Tcl,

```
tcsh> make
gcc -c line_fit.c -I/usr/local/include
swig -tcl line_fit.i
gcc -c line_fit_wrap.c -I/usr/local/include
ld -shared line_fit.o line_fit_wrap.o -o line_fit.so
```

and for Perl,

```
tcsh> make
gcc -c line_fit.c -I/usr/lib/perl5/5.6.0/i386-linux/CORE
swig -perl5 line_fit.i
gcc -c line_fit_wrap.c -I/usr/lib/perl5/5.6.0/i386-linux/CORE
ld -shared line_fit.o line_fit_wrap.o -o line_fit.so
```

That's it! The only customization that's required in going from one language to another is specifying the library and include paths and the SWIG flag.

Sample Sessions

Here's a sample session in Python (with comments indicated by `!`):

```
tcsh> python
Python 1.5.2 (#1, Mar  3 2001, 01:35:43)  [GCC 2.96 20000731
(Red Hat Linux 7.1 2 on linux-i386
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> dir()                       ! dir() tells you what you have in the
                                ! global namespace
['__builtins__', '__doc__', '__name__']
>>> from line_fit import *      ! load the module
>>> dir()
['__builtins__', '__doc__', '__name__', 'cvar', 'fit', 'get_fltarr_val',
 'print_struct_array', 'read_data']    ! now you have the line_fit variables
                                       ! and methods
>>> cvar                               ! these are the globally defined
                                       ! C variables
Global variables { my_data, yy, xx, npts, chi2, slope, intercept }

>>> read_data                   ! entering without () tells you it's
<built-in function read_data>   ! a function
>>> read_data('lf_1.dat')
0                               ! returns a 0 to show everything's ok
>>> print cvar.slope, cvar.intercept  ! we haven't fit yet, so these are 0.0
0.0 0.0
>>> status = fit()
>>> print cvar.slope, cvar.intercept, cvar.chi2
2.30121946335 0.505137145519 0.420483201742
>>> cvar.my_data                ! this is a pointer to a C structure
'_a00f0240_p_point'
>>> print_struct_array(cvar.my_data)  ! it can be passed as an argument
  1.00     2.82                       ! to the C function as usual
  2.00     5.09
  3.00     7.59
  4.00     9.77
  5.00    11.87
  6.00    13.98
  7.00    16.62
  8.00    19.27
  9.00    20.91
 10.00    23.69
>>> get_fltarr_val(cvar.yy, 1)    ! values can be passed as well
5.08518981934
>>>
tcsh> cat lf_1.dat                ! this is the data file we just fit
1  2.82137
2  5.08519
3  7.59315
```

```
4  9.77053
5  11.8671
6  13.9833
7  16.6191
8  19.2732
9  20.9109
10  23.6946
```

A Tcl session illustrates some of its syntax differences with Python:

```
tcsh> tclsh
% load ./line_fit.so line_fit
% read_data lf_1.dat
0
% fit
0
% puts $slope
2.30121946335
% puts $intercept
0.505137145519
% puts $chi2
0.420483201742
% puts $xx
_40d90140_p_float
% puts $my_data
_20d60140_p_point
% print_struct_array $my_data
  1.00     2.82
  2.00     5.09
  3.00     7.59
  4.00     9.77
  5.00    11.87
  6.00    13.98
  7.00    16.62
  8.00    19.27
  9.00    20.91
 10.00    23.69
% get_fltarr_val $yy 3
9.77052974701
% exit
```

As far as I know, Perl doesn't have an interactive command line interpreter. Here's a Perl script that does the same analysis:

```
tcsh> cat line_fit_example.pl
#!/usr/bin/perl
```

```
use line_fit;

line_fit::read_data('lf_1.dat');
line_fit::fit;

printf "%6.2f ", $line_fit::slope;
printf "%6.2f ", $line_fit::intercept;
printf "%6.2f\n", $line_fit::chi2;

print $line_fit::xx,"\n";
print $line_fit::my_data,"\n";

line_fit::print_struct_array($line_fit::my_data);
```

Executing this script from the shell:

```
tcsh> line_fit_example.pl
  2.30   0.51   0.42
_p_float=SCALAR(0x812f9b4)
_p_point=SCALAR(0x812fa20)
  1.00     2.82
  2.00     5.09
  3.00     7.59
  4.00     9.77
  5.00    11.87
  6.00    13.98
  7.00    16.62
  8.00    19.27
  9.00    20.91
 10.00    23.69
```

The `line_fit` source code and makefile are available as a tarball, `http://lheawww.gsfc.nasa.gov/~jchiang/SSC/line_fit.tar.gz`.