Chandra X-Ray Center

Towards a CIAO 3 Interactive Analysis and Rapid Prototyping Architecture Definition

SDS/DS Meeting for Summary and Discussion of Issues & Options

(Re: white paper, `http://space.mit.edu/CXC/docs/CIAO_3_Arch.ps.gz`)

25 Jan 2002

David Huenemoerder

Slang in CIAO

There are many slang utilities available in CIAO, and a Slang-like parser waiting at the sherpa and chips prompts. Help and examples are on-line, via `ahelp -k slang`, whose primary nodes are:

```
slang -- The S-Lang programing language (slang)
slang-chips -- Using S-Lang and Chips (slang)
slang-ciao -- Using S-Lang from CIAO applications (ChIPS and Sherpa) (slang)
slang-math -- Mathematical operations in S-Lang. (slang)
slang-overview -- Overview of the S-Lang programing language (slang)
slang-sherpa -- Using S-Lang and Sherpa (slang)
slang-tips -- S-Lang tips and example functions (slang)
slang-usermodel -- Creating Sherpa Usermodels with S-Lang (slang)
slang-variables -- Variables in S-Lang (slang)
```

Slang in ISIS

Slang is the basis for the ISIS analysis system. There is a brief introduction to Slang in the manual:

> ***3.4 Language Syntax***
>
> *The ISIS command language syntax is the syntax of Slang, its C-like embedded scripting language. Although ISIS can be used without becoming an expert with Slang, it will be helpful to understand something of how the scripting language works and, in particular, how to use its array-manipulation syntax. . . .*

With only a couple exceptions (“executive” commands), *all* ISIS interaction is done in Slang (but much underlying code is C).

What/How/Why Slang? (a *very* brief review)

Mike Noble's recent presentation of a Slang application to add grating orders went into detail on many of the advantages of Slang as a programming language. I only give a very brief summary here:

What is Slang?

Slang is an *embedded scripting language*. Applications can be written in Slang which never expose Slang syntax. However, Slang comes with an interpreter which can be exposed to the user. This is what we have done in ISIS and CIAO.

How do we use Slang?

As an applications developer: Slang has a subroutine library - you can do what you want to write applications or glue them together (mutt, jed, lynx, slrn, isis, varmm, chips, ...). What we do for data analysis is to provide functions to be parsed and evaluated by an interactive Slang interpreter. The functions themselves can come from arbitrary code (C, C++, Fortran, or Slang, for example).

Specific scientific examples are provision of an FFT function in Slang by calling a C application, or of spectral models by wrapping XSPEC libraries into Slang functions.

As a Slang programmer: Given a Slang interpreter (slsh, ISIS, CIAO), you can write programs, simple as averaging two arrays (see the `compute_average` example in `ahelp slangg-ciao`), to the complex, such as Sherpa or ISIS user models.

Given a Slang interpreter, programming can be done *interactively*. (no edit, compile, link, run cycle).

As an end user: You can run applications written in Slang, presented through an interactive Slang interpreter, without programming. You just need to know the basic syntax of functions, arguments, variables, and arrays. You can use it as a high-level calculator, since it does math well, and supports array arithmetic.

Why use interpreted Slang?

Extensible, interactive, customizable, rapid prototypes, memory safe, small, fast, portable, ...

Slang "Issues"

Syntax: A. Critic in Cambridge, MA, writes,

```
() = load_data( "d1.fits" );
```

This is the epitome of the syntax many of us really hate.

Addressing the technical aspects of this statement:

1. Interactive use, no "()" required:
   ```
   load_data( "d1.fits" );
   ```
2. If you care about the return value:
   ```
   result = load_data( "d1.fits" );
   ```
3. In a program: the return value must be popped from the stack:
   ```
   () = load_data( "d1.fits" ); % throw result away
   result = load_data( "d1.fits" ); % or keep it
   ```
   (A user at a higher level might be blissfully ignorant of this.)
4. A more enlightening example - those dangling "()" could delimit multiple return values:
   ```
   (fft_real, fft_imag) = fft1d(re_z, im_z, FORWARD_FFT);
   ```
   (`fft_real, fft_imag` are arrays.)
5. Or by conscious design, we could decide not to use multiple return values:
   ```
   fft_z = fft1d_alt(re_z, im_z, FORWARD_FFT);
   ```
   (`fft_z` is a structure with two tags, `fft_z.real, fft_z.imag`.)

Syntax: Anonymous writes:

I don't like that ";" at the end of each line.

1. The ";" is a slang statement terminator. It is required in slang statements by the slang parser. If a statement doesn't end in ";", it isn't a valid slang statement (ciao inserts it for you before evaluation). The implication is that the interpreter is not a *line-interpreter*, but a statement interpreter (carriage returns are not syntactical elements!).
2. The ";" actually provides functionality in interactive use, and is more than a "preference" (in the sense that font size and color are preferences):
   ```
   result = load_data (
   ''/data/da/testing/acis_process_events/in_1103/acisf01103_000N001_evt1.fits'' );
   ```
3. In general, multi-line statements can be useful:

```
sl>  define silly_tmp_function ( x )
     {
     variable sx, mx, lx;
      sx = sum(x);
      lx = length(x);
      mx = sx / lx;
      () = printf("The sum is %g, the mean is %g (for %d points)\n",
        sx, mx, lx);
     }
 sl> silly_tmp_function([1.,2.,3.]);
 The sum is 6, the mean is 2 (for 3 points)

 sl> silly_tmp_function( get_data_flux(26).value );
     The sum is 0.0299897, the mean is 3.66085e-06 (for 8192 points)

 sl> silly_tmp_function( sin([0:2*PI:0.01]) );
 The sum is -0.00108535, the mean is -1.72551e-06 (for 629 points)
```

4. Requiring the interactive parser to do the same thing as Slang source file evaluation means you only need one version of your code (cf., IDL's interactive blocks vs compiled blocks.) This makes prototyping, development, and cut-and-paste uniform and painless.

A Slang Analysis Environment

The "White Paper"[1] goes into detail on the ISIS architecture as a model for Slang-based functional data analysis enviroment. Here is a synopsis:

Primary goals:

1. Define high-level functions for common operations (fit, load, delete, model).
2. Provide access to data (spectra, images, events, as well as responses, models, fit parameters).
3. Allow customization (user-models, user responses, user-programmed control, user data structures).

NOTE well: since these are *functions*, they may take arguments and may return values! They may be used in programs! They may even be dynamically constructed by programs!!

Classes of functions

Load — Load a file from disk into an internal table.

Delete — Remove a data structure from the internal table.

Get — Read from an internal table into a variable.

Put — Put data from a variable into an internal table.

Set — Set miscellaneous attributes of the tables.

Define — Define a functional form as an alternative to built-in or file-based entities.

List — List information in tables on the terminal.

Plot — Plot, overplot, residual plot from the internal tables or from variables.

Miscellaneous — ad hoc functions.

[1] `http://space.mit.edu/CXC/docs/CIAO_3_Arch.ps.gz`

Classes of Information

(an incomplete list...)

Spectral Data: Load, read, write, define, associate counts arrays and responses

Atomic Data: The atomic data functions support specification of a database, retrieving atomic data (energy levels, wavelengths, transition probabilities, spectral designations) into arrays, given search criteria such as element, ion, and wavelength, definition of line groups, and display of the quantities.

Plasma Emissivity: These functions deal primarily with the emission for a given abundance and ionization mix, as well as for temperature and density components and their weights.

Fitting: Define, fit models to the data, apply responses, statistic, optimization method. Provide functional forms for data and responses. Read/write model parameters. Compute confidences.

Plot: Display data, models, responses, captions, labels, to screen or file. Generic plots of slang variables, or interact with the plot device.

Utility: generic functions for manipulating data, managing the environment, or performing mathematical calculations.

Fill in the blank... other stuff, like images, psfs, timing, regions.

Design Issues

1. Should plots be formatted by function calls?

   or by setting fields in a global structure?

2. What global structures should be defined? Should functions read them implicitly or accept them as arguments? E.g., as per a plot structure:

3. Should data be managed by the user? or by the application?

4. Which functions return error codes? Only those which which are likely to fail (loading data)? What is done on error? Exception handling?

5. Which functions return structures? Should functions be allowed to return multiple values?

6. What other data types should we provide? (linked lists; relational databases)

7. Which common functions should be intrinsics? (e.g. how much should be left to the user?)

8. What's the best way to organize libraries of scripts so that we don't have the problem of "where's the entry point?"

9. What slang coding conventions should be enforced? recommended? (usage, help, return values, error handling)

10. Should modal behavior be supported, or excluded in lieu of explicit command names or arguments?

11. Should arguments be positional, w/ null values allowed and parsed?

12. When should we use parameter i/o on slang functions?

13. How are journaling and logging to be done?

14. What save/restore, do/undo capabilities are desired?

15. When/how is header manipulation to be done? A functional interface definition is needed.

16. How are large files to be handled? For some operations, buffering is possible (via `CFITSIO`).

17. What debugging tools are needed? Slang has a traceback function; scripts are easier to debug via print statements, and are not prone to memory corruption. A debugging tool is probably not required, as for compiled code.

18. What parser modes are permitted? What are precedence rules to resolve ambiguous arguments (if any)?

Examples

Part of an ISIS continuum-fitting tutorial

From

`http://space.mit.edu/CXC/ISIS/contrib/Contin_fit/Contin_tutorial.sl`

This is interactive mode (variables are not declared; stack items are ignored).

...

```
%
%   Demonstrate continuum fitting in isis, using APED continuum data
%   and user-defined model.
%
%   Also demostrate definition of thermal plasma model for use in
%   defining bright lines.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Define the model; single component to start.
% Later, we will try adding another component to see if it matters.

evalfile("aped_contin.sl");              % load the model functions.

fit_fun("aped_contin(1)");               % model is thermal plasma continuum

list_par;   % to show the model parameters

% set some likely defaults:

                 freeze
%       n  value /thaw  min   max
set_par(1, 0.004, 0,    0,    0.1);
set_par(2, 2.3e7, 0,    1.e4, 5.e8);


% Group data coarsely, since this is a continuum fit, and we would
%       like good statistics.
```

```
meg_group_factor = 8;   heg_group_factor = 2*meg_group_factor;

% redefine order indices to map to the loaded histograms:

mm = 3; mp = 4;    hm = 1;   hp = 2;

group_data(mm, meg_group_factor); group_data(mp, meg_group_factor);
group_data(hm, heg_group_factor); group_data(hp, heg_group_factor);


%%%%%%%%%%%%%%%%%%%%%
%
% First, use only line-free regions.  Function applies ignore region table.

line_free_regs([hm,hp,mm,mp]);

% For first pass, single high-T component, only notice the
% short-wavelength region, and fit both HEG and MEG:

ignore([hm,hp,mm,mp], 10, 25);

list_data ;                             % check
yrange(0); xrange(1,8);
plot_data_counts(mm,1);  oplot_data_counts(mp,2);
oplot_data_counts(hm,3);  oplot_data_counts(hp,4);


% Do a fit, list the params, compute confidence:
fit_counts; list_par;

%  Parameters[Variable] = 2[2]
%             Data bins = 342
%   Best fit chi-square = 50.35
%    Reduced chi-square = 0.1481
% aped_contin(1)
%  idx  param              tie-to  freeze         value          min          max
%   1  aped_contin[1].norm    0      0        0.02229772            0          0.1
%   2  aped_contin[1].T       0      0      1.835275e+07        10000        5e+08
```

```
oplot_model_counts(mm,5);  oplot_model_counts(mp,6);
oplot_model_counts(hm,7);  oplot_model_counts(hp,8);

% evaluate 90% confidence intervals:
%
(clo_norm, chi_norm) = conf(1);   ()=printf("Conf norm = %f , %f\n", clo_norm, chi_norm)
(clo_T, chi_T) = conf(2);         ()=printf("Conf T = %e , %e\n", clo_T, chi_T);

% Conf norm = 0.020989 , 0.023607
% Conf T = 1.713500e+07 , 1.957111e+07

%  Hardcopy:
%
pid = dup_plot("Contin_fit_0.gif/gif",p1);  resize(6*2.54, 0.6);

plot_data_counts(mm,1);  oplot_data_counts(mp,2);
oplot_data_counts(hm,3);  oplot_data_counts(hp,4);
oplot_model_counts(mm,5);  oplot_model_counts(mp,6);
oplot_model_counts(hm,7);  oplot_model_counts(hp,8);

close_plot(pid); window(p1);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%  If satisfied, freeze this, and add a new component, and
%%%%%%%%%%  fit long-wavelength region where low-T has more affect.

freeze([1,2]);
fit_fun("aped_contin(1) + aped_contin(2)");   % insert another T component

set_par(3, 0.004, 0,    0,    0.1);           % set initial params
set_par(4, 2.3e6, 0,    1.e4, 5.e8);
list_par;

ignore([hm,hp]);                    % no good region for HEG - ignore it.
notice([mm,mp]);                    % notice the MEG, then apply regions
line_free_regs([mm,mp]);            % line free,
ignore([mm,mp], 1, 8);              % ignore short waves.
```

```
yrange(0); xrange(8,25);
plot_data_counts(mm,1);    oplot_data_counts(mp,2);

fit_counts; list_par;

%  Parameters[Variable] = 4[2]
%             Data bins = 328
%   Best fit chi-square = 21.81
%    Reduced chi-square = 0.0669
% aped_contin(1) + aped_contin(2)
%  idx  param            tie-to  freeze         value         min          max
%   1  aped_contin[1].norm   0      1        0.02229772          0          0.1
%   2  aped_contin[1].T      0      1      1.835275e+07      10000        5e+08
%   3  aped_contin[2].norm   0      0       0.002478395          0          0.1
%   4  aped_contin[2].T      0      0      3.381622e+08      10000        5e+08
```

Line-free region function

This is a simple but instructive function. It shows how one can easily customize simple functions to automate some tedious repetitive tasks. (also from `http://space.mit.edu/CXC/ISIS/contrib/Contin_fit/Contin_tutorial.sl`)

```
%
% Purpose: Defines line-free regions for coronal spectrum.
%          Based on AR Lac ad hoc model, which has broad temperature
%          distribution.   Ignores all in histogram list, i, then
%          notices the regions.

%
% Syntax: ()=line_free_regs(i);
%
%  i is the index of the data to notice, and can be an array.

define line_free_regs(i)
{
  variable wav_min = [
    1.9,      3.25,     4.05,     5.25,     6.25,
    6.8,      10.85,    11.6,     16.4,     17.15,
    17.7,     19.05,    22.2
    ];
  variable wav_max = [
    2.9,      3.65,     4.7,      6.0,      6.6,
    7.05,     10.95,    11.7,     16.7,     17.55,
    18.6,     21.5,     24.5
    ];
  variable j;
  ignore(i);                   % first, ignore all histograms, i.
                              % now notice each region in histograms, i.
  for ( j = 0; j < length( wav_min ); j++ )
    notice(i, wav_min[j], wav_max[j] );
  return 0;
}
```

Model Name as a Variable Example

This is a fragment of an ad hoc, customized emission line fitting script. The wavelength region min and max and line-positions are passed as parameters. The model is generated by concatenating as many gaussians as needed in each line group.

```
  ...

 xnotice(idx,wmin,wmax);       % narrow down for fitting
 nlines = length(wav_ids);     % determine how many components there are.
 fit_fun_str="fast_aped_contin(1)";  % establish the continuum
                                     % component for the line group.
 for (i=1; i<=nlines; i++)           % Loop over lines, build model string.
 {
   fit_fun_str += sprintf(" + gauss(%d)",i);
 }
 fit_fun(fit_fun_str);              % set the fit function
 if (usepar == 0)                  % are we guessing params?
 {
   for (i=0; i<nlines; i++)
   {
     set_par(1, 1, 1, 0.8, 1.2);             % aped contin; relatively frozen.
     set_par(3*i+2, 5.e-4, 0, 0.0, 5.e-2);   % gauss norm; wild guess
     set_par(3*i+3, wav_ids[i], 0, wav_ids[i]-0.25, wav_ids[i]+0.25); %gauss pos
     set_par(3*i+4, 1.0e-3, 1, 1.e-5, 0.1);  % gauss width - frozen to tiny.
   }
 }
 if (usepar == 2)                  % Or are we loading  from a previous fit?
 {
   load_par(of_root+".par");
 }
 fit_counts;                       % do the fit.

% have a look..............
 multiplot(mparr); erase;
 plot_data_counts(idx[0],2);
 oplot_model_counts(idx[0],3);
 plot_group( brightest(10,where(wl(wmin,wmax))) , orange );   % mark lines
 for (i=1; i<n_idx; i++)            % overplot individual histograms being fit.
 {
```

```
    plot_data_counts(idx[i],2);
    oplot_model_counts(idx[i],3);
  }
  list_par;                        % list params of fit; also save to file.
  save_par(of_root+".par");
  id;                              % write confidence limits to the
                                   % par file.
  fp=fopen(of_root+".par","a");

  ()=fprintf(fp, "#\n");
  ()=fprintf(fp, "#%s\n", id);
  ()=fprintf(fp, "#\n");
  ()=fprintf(fp, "# Conf(n,%d) [0=>68%%, 1=>90%%, 2=>99%%]\n",cnf_idx);

    ...
```

Parameter Manipulation: Monte-Carlo

This is an example of using Slang to access and manipulate fit parameters to carry out a monte-carlo simulation. (From J.Davis, avaailable at `http://space.mit.edu/CXC/ISIS/examples/ex_monte.sl`).

```
%  Purpose:
%       Carry out a Monte-Carlo search of the fit-function
%       parameter space to find a good set of starting parameters.
%
%  Notes:
%       A Monte-Carlo search may be helpful when examining
%       a complex chi-square space such as that associated
%       with the pileup model.
%
%       Each variable fit-parameter should be constrained
%       to lie within some reasonable min/max range.
%       If the allowed range is unconstrained, the Monte-Carlo
%       search will sample parameter values over the entire
%       range of representable double-precision numbers.
%
%  Usage:
%
%  e.g. to carry out 100 random trials with 'eval_counts':
%     monte_carlo (100, &eval_counts);
%
_debug_info = 1;
_traceback=1;
private define get_params ()
{
   variable i, n = get_num_pars ();
   variable p = Struct_Type [n];
   for (i = 1; i <= n; i++)
     {
        p[i-1] = get_par_info (i);
     }
   return p;
}

private define set_params (p)
```

```
{
   variable i, n = get_num_pars ();
   for (i = 0; i < n; i++)
     {
        set_par (i+1, p[i].value, p[i].freeze, p[i].min, p[i].max);
     }
}

private define linear_random (mn, mx)
{
   return mn + (mx - mn) * urand();
}

private define select_at_random (mn, mx)
{
   return linear_random (mn, mx);
}

private define random_params (p)
{
   variable i, n = get_num_pars ();
   for (i = 0; i < n; i++)
     {
        if (p[i].freeze == 0 and p[i].tie == 0)
          {
             p[i].value = select_at_random (p[i].min, p[i].max);
          }
     }
   return p;
}

private define test_params (p, evalfun)
{
   variable info;
   set_params (p);
   () = @evalfun (&info);
   return info.statistic;
}
```

```
public define monte_carlo (num, evalfun)
{
   variable bkp_verbose = Fit_Verbose;
   Fit_Verbose = -1;
   variable p, best_p, stat, best_stat;
   p = get_params ();
   best_stat = test_params (p, evalfun);
   best_p = get_params ();
   variable i = 0;
   loop (num)
     {
        if (bkp_verbose >= 0 and Isis_Batch_Mode == 0)
          () = fprintf (stderr, "monte_carlo:  %3d/%3d\tbest=%7.3f\r",
                        i, num, best_stat);
        stat = test_params (random_params (p), evalfun);
        if (stat < best_stat)
          {
             best_stat = stat;
             best_p = get_params ();
          }
        i++;
     }
   set_params (best_p);
   Fit_Verbose = bkp_verbose;
}
```

Options

Schizophrenia Current state of CIAO 2.2 as mix of command-line and slang (and chips, and ...) No consistent syntax, use of variables, functions. Can be painful, requires much therapy.

Get Smarter Parser work-in-progress may be able to unambigously tell when a statement is slang and when it isn't.

Lobotomy Explicitly set the input mode to command-line or slang, and change the prompt accordingly.

Unambiguous modes: Provides clean slang implementation, or standard command-line-interpreter,

Autonomous Collectives Develop targeted slang application packages, build a library. (e.g., param i/o; plotting; timing tools;)

Provides utilities for import to slang prompt.

Lacks coherent superstructure. No guarantee of compatibility, consistency, ability to interface with other collectives.

(Cf. multiplicity of IDL packages for plotting images, reading FITS)

Would require dual implementation to provide similar capability to command-line-interpreter.

Definitely good in many ways, and should be encouraged - fosters diversity and independence. Enables ad hoc applications to target limited audience.

Assimilation Move towards slang functional environment - organize Autonomous Collectives under one umbrella of interfaces and conventions.

Requires top-down definition of analysis system.

Requires incorporation of command-line capability into slang (opening libaries, writing importable modules; writing slang wrappers and functions).

Would benifit from an early tactical lobotomy.

A Miracle Define a new syntax for the command-line-interpreter mode which is complete and unambiguous, which provides variables and programming structures, which can be automatically parsed and translated into Slang, and which can be designed, implemented, and tested within a year.

Resistance (is futile.)