URL: http://cxc.harvard.edu/ciao3.4/slang−usermodel.html
Last modified: December 2006

AHELP for CIAO 3.4

usermodel

Context: slang

Jump to: Description Examples CHANGES IN CIAO 3.0 See Also

Synopsis

Creating Sherpa Usermodels with S−Lang

Description

This document describes how to use the S−Lang programming language to create a user−defined Sherpa model. Such a model can then be fit to your data in exaclty the same manner as any of the pre−defined models in Sherpa.

Creating a S−lang based model in Sherpa is a two step process. First, a S−lang function with specific inputs must be compiled, and then a special Sherpa/S−lang function must be called to register the model with Sherpa. Once created, a S−lang model is entirely equivalent to any built−in model in terms of usability. A simple example is shown below, for a power law model:

```
define slang_pow() {

  variable p, norm, Emin, Emax, dE, Result;

  if ( _NARGS == 3 ) {
     (p, norm, Emin) = ();
     Result = norm*(Emin^(−p));

  } else if ( _NARGS == 4 ) {
     (p, norm, Emin, Emax) = ();
     dE = Emax − Emin;
     Result = norm*(Emin^(−p))*dE;
  }

  Result = typecast(Result, _typeof(Emin));
  return Result;
}
```

and the Sherpa/S−lang command to register the model after it has been compiled with the evalfile() command is:

```
  sherpa> () = register_model("slang_pow",["power","norm"], 1,
                              [1.0,1.e−2],  % default values
                              [−10,1.e−20], % Minimum values
                              [10,1.e5],    % Maximum values
                              [1,1]);       % Both thawed by default
```

S−lang User Model Function Parameters

The S−lang function takes a variable number of arguments, depending upon the number of model parameters, and returns a vector of the evaluated model on the S−lang stack. The input arguments are:

Variable	Value
P_1	First function parameter value
P_2	Second function parameter value
P_i	i−th function parameter value
P_n	Last function parameter value
X	Vector of points at which to evaluate the function
X_max	(Possible) vector of bin maximum points

Integrated and Differential Models

One complication in writing a S−lang usermodel for Sherpa is that Sherpa models may be integrated (for use with binned data) or not (for unbinned or differential data). Sherpa, by default, creates all S−lang usermodels with integrate off, although this can be changed after the model is created using the <modelname> integrate on command. If integrate is off, the model is expected to caluculate a value at each input point X. If integrate is on, two vectors, Xmin and Xmax, will be passed to the function. In this case, the function should return the value of the model integrated over the each bin. The accuracy of the integration is up to the model author. Depending on the model and the input data, a simple evalution at one bin edge (or the center of the bin) and multiplication by the bin width (as shown in the example above) or an analytic integral may be needed. Note that the units output from the model are different in each case; if the model is not integrated, the units have a "per X−axis unit length", i.e. per keV or per Angstrom, while in the binned case they do not. Since the user can switch between wavelength and energy units at will in Sherpa (via analysis wavelength or analysis energy), the model writer must handle all these cases to have a complete model.

Dealing with binned data

A common mistake when using S−lang usermodels with PHA−type data is that PHA data are inherently binned, so the model have integrate set to "on". The code must integrated between the lower and upper edge of each detector channel to get the total photons in the channel. If this is not done, Sherpa will still calculate a result, with parameters that frequently will be close to the correct values except for a very wrong normalization.

Dealing with unbinned data

Sherpa can also use unbinned data, such as a list of X and Y points which has no inherent binning. Is is possible that the data may be binned, but the user wants to use the model with "integrate off." In which case the normalization will be incorrect, since the model will have different units than the data. In either case, Sherpa passes a single vector X to the function. In this case, the model should simply evaluate the function at each X point and return the resulting vector. Note that in the case of binned data with integrate off, the bin minimum (not the bin center) is passed to the function; this is how Sherpa treats all such cases.

Coding details

Since Sherpa allows any model to be set to integrate off, any S−lang usermodel must be able to deal with either binned or unbinned data. The simplest method is to check how many values have been passed to the routine; for a model with M parameters, there will be either M+1 (in the unbinned case) or M+2 (for binned data). If the M+1 values are passed, the routine should return the model evaluated at the requested points, and if M+2 values are given, the code should return the model integrated over each low to high bin.

Units

As noted above, the units of the independent axis (or axes) are vital. The routine get_axes(datasetNum) can be used to retrieve a structure containing the units for dataset datasetNum. When a model is evaluated in a fit, the currently active dataset number can be determined with the Sherpa routine get_dataset_number(). In the case of basic spectral models, the input units with be either keV (if analysis energy is set) or Angstroms (if analysis wave is set). The expected output in these cases is photons/cm**2/s in each bin (for binned data) or photons/cm**2/s/keV or photons/cm**2/s/A in the unbinned case. For more complex models, the values of the independent axes should be checked carefully.

S−lang types

One Sherpa requirement is that the datatype of the returned result be the same as the input vector. This can be enforced by using the following command before returning the result via

```
Result = typecast(Result, _typeof(Emin));
```

where Emin the input vector and Result the model's calculated values. This S−lang command simply forces the returned value to have the same type as the input.

Compiling the user model

The model can be compiled with the evalfile command, a S−Lang function which loads the given file into the S−Lang interpreter. To read in the S−Lang code stored in the file myusermodel.sl, the syntax is:

```
sherpa> () = evalfile("./myusermodel.sl")
```

The initial '() =' is used to ensure that the stack is properly cleared after execution, avoiding unwanted messages printed to the screen.

Registering the new model with Sherpa

After the new model code is compiled, it must be registered with Sherpa using the register_model function. The syntax is:

```
() = register_model("new model name",
                    ["parameter1","parameter2",...,"parmeterN",],
                    ModelDimension,
                    [p1_default,p2_default,...,pN_default],
                    [p1_minimum,p2_minimum,...,pN_minimum],
                    [p1_maximum,p2_maximum,...,pN_maximum],
                    [p1_thawed, p2_thawed, ...,pN_thawed]);
```

for a total of seven arguments. The first argument is a S−lang string which is the compiled function's name and will become the name of the new model (for example some of Sherpa's built−in model names are ngauss, xsapec, and delta2d). The second argument is an N−element string vector giving the names of each of the N parameters, such as "pos", "ampl" or "kT". The third argument gives the dimensionality of the model; spectral models are all one−dimensional, but a spatial model (such as the built−in model gauss2d) could have two. The next three arguments are all N−element vectors giving the default, minimum, and maximum values of each model parameter. The last argument is a N−element integer vector which gives the default status (either thawed or frozen) for each model parameter; either 1 for thawed or 0 for frozen is allowed. These last four arguments are optional; if they are set to NULL values, standard values with be used.

Example 1

```
() = register_model( "slang−blackbody", ["kT","ampl"], 1, [1.0, 1.0],
[0.001,0.0],[100.0,1.e10],[1,1])
```

This example registers a new one−dimensional function slang−blackbody, with parameters kT and ampl. The default parameter values are 1.0 and 1.0, with minima 0.001 and 0.0 and maxima 100 and 1.e10, respectively. Both parameters are initially thawed.

The S−lang code for the model, which is similar to the Sherpa bbody routine, is shown here:

```
define slang_blackbody() {

   variable kT, ampl, X, Xmax=NULL, savestack=NULL, EokT=NULL;
   if ( _NARGS == 3 ) (kT, ampl, X) = ();
   if ( _NARGS == 4 ) (kT, ampl, X, Xmax) = ();
   variable Result = Double_Type [length(X)];
   %  Are the X−axis units keV or Angstrom?
   if (_stkdepth() != 0) savestack = __pop_args(_stkdepth());
   variable curr_axis = get_axes(get_dataset_number());
   if (savestack != NULL) __push_args(savestack);
   if (length(curr_axis) != 1) {
      message("Error: dimension of dataset does not match that of model");
      return NULL;
   }
   if (strcmp(curr_axis.axisunits,"keV")==0) EokT = X/kT;
   if (strcmp(curr_axis.axisunits,"A")==0)   EokT = (12.3984/X)/kT;
   if (EokT == NULL) {
      message("X−axis units unknown, cannot calculate model.");
      return NULL;
   }
   variable gpL = where(EokT <= 1.e−4);
   variable gpM = where(EokT > 1.e−4 and X/kT <= 60.0);
   variable gpH = where(EokT > 60.0);

   if (strcmp(curr_axis.axisunits,"keV")==0) {
      Result[gpL] = ampl*kT*X[gpL];
      Result[gpM] = ampl*X[gpM]*X[gpM]/(exp(EokT[gpM]) − 1.0);
   }
   if (strcmp(curr_axis.axisunits,"A")==0) {   %  Run Wavelength model
      Result[gpL] = kT/(X[gpL]^3)/12.3984;
      Result[gpM] = 1.0/(X[gpM]^4)/(exp(EokT[gpM]) − 1.0);
   }
   Result[gpH] = 0.0;
   if (Xmax !=NULL) Result = Result*(Xmax−X);

   Result = typecast(Result, _typeof(X));
   return Result;
}
```

Note that the routine checks the units of the input X−axis to see if it is in keV or Angstrom units, and runs the appropriate model. The Sherpa routine get_dataset_number() gives the number of the currently evaluated dataset, which is passed to the get_axes() call to return a structure describing the independent axes. The __pop_args()/__push_args() calls are needed to clear the stack because it stack may not be empty depending on the circumstances of the call, and get_axes() requires a clear stack.

Example 2

```
() = register_model( "linemodel", ["ampl","intcpt"], 1, [2,0],
NULL,NULL,[1,0])
```

This example shows a how to register the S−lang function linemodel. In the register_model call, the parameter defaults for ampl and intcpt are set to 2 and 0 respectively, but the parameter minima and maxima are left for Sherpa to set. In addition, the first parameter (ampl) is set to be thawed by default, but the intcpt parameter is frozen by default.

The S−lang code for the model is shown here:

```
define linemodel() {
  variable ampl, intercept, x, xhi;
  variable size = 0;
  variable i = 0;
  variable y;

  % Pop inputs off of stack.  Model parameters
  % are popped off in the same order they
  % are listed when you call register_model.
  % The appropriate x array(s) were pushed on the
  % stack after the model parameters.

  if (_NARGS == 3) (ampl, intercept, x) = ();
  if (_NARGS == 4) (ampl, intercept, x, xhi) = ();

  size = length(x);
  y = Float_Type[size];

  if (__is_initialized (&xhi) == 0) {
    y = ampl * x + intercept;
  } else {
    for (i = 0; i < size; i++)
      y[i] = (ampl * x[i] + intercept) * (xhi[i] - x[i]);
  }

  if (Float_Type == _typeof(x)) y = typecast(y, Float_Type);

  % Push y array onto stack -- sherpa retrieves it from there
  return y;
}
```

It is a simple linear model with two parameters, ampl and intcpt. In the unbinned model (when xhi is not passed to the routine), it calculates all the y values simultaneously. As an example, when the data is binned, the code shows how each value can be calculated individually using a for loop. This method will usually execute slower than the vector notation, but will work just as well.

Example 3

```
() = register_model("delta_sin2d",["A","B","w"], 2, [1.0,100.0,1.e-2],
[0.0,1.0,1.e-20], [100.,1024.,1.e5], [1,1,1]);
```

This example shows a how to register a two−dimensional S−lang usermodel. In the register_model call, the parameter defaults for A, B, and w values are set to 1, 100, and 0.01 respectively, with minima 0, 1, and 1.e−20 and maxima of 100, 1024, and 1.e5. All three parameters are thawed by default.

The S−lang code for the (admittedly unphysical) model is shown here:

```
define delta_img(x) {  % Define a delta-function for imaging

   variable result;

   if (typeof(x) == Array_Type) {
      result = 0.0*x;
```

```
      result[where(x < 1)] = 1.0;
   } else {
      if (x < 1) result = 1.0;
   }
   return result;
}

define delta_sin2d() {

   variable A, B, w;
   variable Xmin, Xmax, Ymin, Ymax, dX, dY, Result;

   if (_NARGS == 5) (A, B, w, Xmin, Ymin) = ();
   if (_NARGS == 7) {
      (A, B, w, Xmin, Ymin, Xmax, Ymax) = ();
      dX = Xmax − Xmin;
      dY = Ymax − Ymin;
   }
   Result = A*delta_img(Ymin − B*sin(w*Xmin));
   Result = typecast(Result, _typeof(Xmin));

   return Result;
}
```

This model describes a line in X,Y space defined by the points where y = B sin(w*x). The amplitude at all points is A. Although clearly unphysical, this shows how a more complex multi−dimensional model may be created. The X and Y axes need not have the same units; Sherpa allows for mixed axes−−for example, CC mode data of a diffuse source may be analyzed where one axis is position and the other energy.

CHANGES IN CIAO 3.0

In CIAO 3.0 the function name was changed from sherpa_register_model() to register_model().

See Also

sherpa

atten, bbody, bbodyfreq, beta1d, beta2d, box1d, box2d, bpl1d, const1d, const2d, cos, delta1d, delta2d, dered, devaucouleurs, edge, erf, erfc, farf, farf2d, fpsf, fpsf1d, frmf, gauss1d, gauss2d, gridmodel, hubble, jdpileup, linebroad, lorentz1d, lorentz2d, models, nbeta, ngauss1d, poisson, polynom1d, polynom2d, powlaw1d, ptsrc1d, ptsrc2d, rsp, rsp2d, schechter, shexp, shexp10, shlog10, shloge, sin, sqrt, stephi1d, steplo1d, tan, tpsf, tpsf1d, usermodel, xs, xsabsori, xsacisabs, xsapec, xsbapec, xsbbody, xsbbodyrad, xsbexrav, xsbexriv, xsbknpower, xsbmc, xsbremss, xsbvapec, xsc6mekl, xsc6pmekl, xsc6pvmkl, xsc6vmekl, xscabs, xscemekl, xscevmkl, xscflow, xscompbb, xscompls, xscompst, xscomptt, xsconstant, xscutoffpl, xscyclabs, xsdisk, xsdiskbb, xsdiskline, xsdiskm, xsdisko, xsdiskpn, xsdust, xsedge, xsequil, xsexpabs, xsexpdec, xsexpfac, xsgabs, xsgaussian, xsgnei, xsgrad, xsgrbm, xshighecut, xshrefl, xslaor, xslorentz, xsmeka, xsmekal, xsmkcflow, xsnei, xsnotch, xsnpshock, xsnsa, xsnteea, xspcfabs, xspegpwrlw, xspexrav, xspexriv, xsphabs, xsplabs, xsplcabs, xsposm, xspowerlaw, xspshock, xspwab, xsraymond, xsredden, xsredge, xsrefsch, xssedov, xssmedge, xsspline, xssrcut, xssresc, xssssice, xsstep, xstbabs, xstbgrain, xstbvarabs, xsuvred, xsvapec, xsvarabs, xsvbremss, xsvequil, xsvgnei, xsvmcflow, xsvmeka, xsvmekal, xsvnei, xsvnpshock, xsvphabs, xsvpshock, xsvraymond, xsvsedov, xswabs, xswndabs, xsxion, xszbbody, xszbremss, xszedge, xszgauss, xszhighect, xszpcfabs, xszphabs, xszpowerlw, xsztbabs, xszvarabs, xszvfeabs, xszvphabs, xszwabs, xszwndabs

URL:
http://cxc.harvard.edu/ciao3.4/slang−usermodel.html
Last modified: December 2006