Advanced S-Lang Applications

John Houck

houck@space.mit.edu

MIT/CXC

Outline

- Two data analysis examples
- Debugging S-Lang scripts

Why Script Data Analysis?

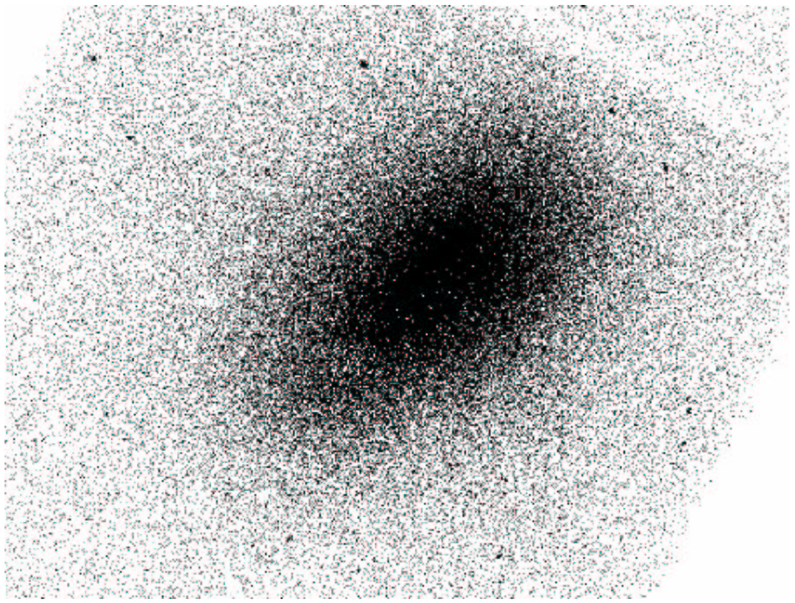
Some jobs are just tedious. . .

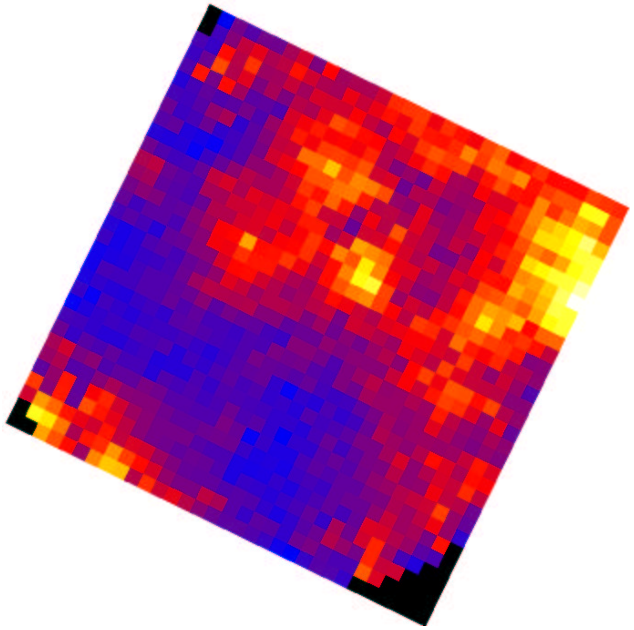
- making $\LaTeX$ tables of fit results

But without automation, some jobs might not get done:

- computing 2D confidence contours for computationally expensive models
- fitting many individual spectra

Mapping Extended Sources





- use adaptively sized spectral extraction regions (may overlap)
- 64x64 pixel map $\Longrightarrow$ 4096 spectrum fits
- read event file **once**
- extract and fit using S-Lang variables (no FITS files)

Choosing Extraction Regions

```
% Filter in stages:  Events => short_evts => evt
short_evts = filter (Events, reg_str);
n = length (short_evts.pi);
evt = short_evts;
forever
{
   if (n <= nmin and size == Tmap_Init.max_box_size)
     break;
   if (nmin <= n and n < nmax)
     break;
   size = guess_size (size, n, nmin, nmax);
   reg_str = sprintf (fmt, x0, y0, size, size, theta);
   evt = filter (short_evts, reg_str);
   n = length (evt.pi);
}
```

Extracting spectra

```
import("extract");    % user-defined module


public define extract_spectrum (events, region)
{
   variable f = @region.filter (events, region);
   %plot_events (events.x[f], events.y[f]);   cursor;
   return pi_bin (events.pi[f]);
}
```

events = struct with events (X, Y, PI, etc.)

region = struct describing the region
(e.g. ellipse, box, annulus, etc.)

pi_bin = user-defined in C

Writing Modules in C

To add a gamma function intrinsic, $\Gamma(x)$:

```
static double gamma_fcn (double *x) {
   return gsl_sf_gamma (*x);
}

static SLang_Intrin_Fun_Type Funs [] = {
   MAKE_INTRINSIC_1("gamma", gamma_fcn, D, D),
   SLANG_END_INTRIN_FUN_TABLE
};
SLANG_MODULE(gsl);
int init_gsl_module_ns (char *ns_name) {
   return SLns_add_intrin_fun_table (NULL, Funs, "__GSL__");
}
```

2D Confidence Contours

Computing 2D confidence contours can be slow, but a scriptable interface allows a workaround.

Because individual fits are independent, one solution is distributed processing

Apart from load-balancing issues,

$$N \text{ cpus} \implies N\text{x faster.}$$

A crude implementation requires only **ssh**.

Subdivide Among CPUs

Divide parameter grid into N slices

Assign one slice to each CPU

At each assigned grid point, each CPU will:

- Compute best-fit χ^2
- Save all fit-parameters

When finished, combine all output files.

Running on Many CPUs

To start a script running on N CPUs:

```
define start_task (host, cpuid, info)
{
   () = system (sprintf ("ssh -f %s %s %s %d",
                         host, script_path, host, cpuid));
   return 0;
}
() = map_cpus (&start_task, NULL);
```

On each CPU, the script also does:

```
() = system (sprintf ("renice 19 -p %d", getpid()));
```

A Single Slice

```
Conf_Map_Fail_Hook = &fail_hook;
Conf_Map_Save_Hook = &save_hook;

public define do_map_slice (y_slice, info)
{
   variable prefix, map;
   prefix = sprintf ("dir/conf_%d_%d",
                     info.hostid, info.cpu);

   Fp = fopen (prefix + ".txt", "w");
   map = conf_map_counts (X_Grid, y_slice);
   () = fclose (Fp); Fp=NULL;

   save_conf (map, prefix + ".fits");
}
```

Hooks Add Versatility

```
try_pars = get_params();

if (-1 == @fit_ref (&fit_info))
{
   if (fail_ref != NULL)
     {
        @fail_ref (ip1, ip2, @best_pars,
                   try_pars, fit_info);
     }
}

if (save_ref != NULL)
  @save_ref (fit_info);
```

Failure Recovery Hook

```
define fail_hook (p1, p2, start_pars, try_pars, fit_info)
{
   variable save_method = get_fit_method ();

   set_fit_method ("subplex;maxnfe=100");
   randomize;
   () = fit_counts (&fit_info);

   set_fit_method (save_method);
}
```

Custom Output Hook

```
define save_hook (info)
{
   variable pars = get_params();

   () = fprintf (Fp, "%15.6e", info.statistic);
   foreach (pars)
     {
        variable p = ();
        () = fprintf (Fp, "  %15.6e", p.value);
     }
   () = fputs ("\n", Fp);
   () = fflush(Fp);
}
```

View with ds9

```
public define ds9 (a)
{
   variable dims, file, fp, cmd, fmt;
   file = "/tmp/image.dat";
   fp = fopen (file, "w");
   () = fwrite (typecast(a, Float_Type), fp);
   () = fclose (fp);

   (dims,,) = array_info (a);
   fmt = "ds9 -array %s[xdim=%d,ydim=%d,"
       + "bitpix=-32,arch=littleendian] &";

   cmd = sprintf (fmt, file, dims[1], dims[0]);
   () = system (cmd);
}
```

Debugging

- print to the screen:
  `vmessage ("x= %S y= %S", x, y);`

- print to a file:
  `() = fprintf (fp, "x= %S\n", x);`

- **`_print_stack;`**

```
(2)[Array_Type]:Double_Type[3]
(1)[String_Type]:a
(0)[Integer_Type]:2
```

Tracing a Function

To trace a function:

```
_slangtrace=1;
_trace_function("name");
```

```
isis> _slangtrace=1;
isis> _trace_function ("who");
isis> who;
>>who (0 args)
 >>_apropos (3 args)
  [String_Type]:
  [String_Type]:
  [Integer_Type]:8
 <<_apropos (returning 1 values)
  [Array_Type]:String_Type[0]
 >>_apropos (3 args)
  [String_Type]:
  [String_Type]:
  [Integer_Type]:2
 <<_apropos (returning 1 values)
  [Array_Type]:String_Type[0]
<<who (returning 0 values)
isis>
```

Controlling Traceback Output

Control output when a script is interrupted:

- **`_traceback=n;`**
  have interrupt print trace information.
  Options are **`n=-1, 0, 1`**

- **`_debug_info=1;`**
  have trace include line number information.

Debugging Example

```
_traceback=1;  % want traceback
_debug_info=1; % include line # info

define stats (file, col)
{
   col = fits_read_col (file, col);
   variable s = moment (col);
   return s;
}
vmessage ("@ %S, line %S", __FILE__, __LINE__);
variable x = stats ("evt.fits", "ttime");
print(x);
```

Reading a Traceback Report

```
verus:˜> isis examples/debugme.sl
@ ./examples/debugme.sl, line 10
could not open the named file
S-Lang Traceback: error
...
S-Lang Traceback: fits_read_col
...
S-Lang Traceback: stats
File: ./examples/debugme.sl
  Local Variables:
        $0: Type: String_Type,     Value:   "evt.fits"
        $1: Type: String_Type,     Value:   "ttime"
        $2: Type: Undefined_Type,  Value:   Undefined_Type
called from line 11, file: ./examples/debugme.sl
verus:˜>
```

Interrupt Handling

In long-running scripts, its often necessary to catch interrupts:

```
define make_error ()
 {
     error ("Error condition created.");
     message ("This statement is not executed.");
 }
define test ()
{
    ERROR_BLOCK
      {
         _clear_error ();
      }
    make_error ();
    message ("error cleared.");
}
```

The Preprocessor

```
#iffalse  % (iftrue is the opposite)
   % enclosed lines are commented out
   x = [1:10];
   vmessage ("won’t print this message");
#endif

#ifnexists sum
% if ’sum’ hasn’t been defined yet, here’s a definition.
define sum (a) {
   variable tot = 0.0;
   foreach (a) { tot += (); }
   return tot;
}
#endif
```