The GALSPECT/BW pipeline: Up close and personal

J. E. Goldston

October 27, 2005

1 Introduction

The purpose of this document is to acquaint anyone who intends on spending a good bit of time processing, calibrating and gridding GALSPECT data with the finer vicissitudes of the reduction package. It is the intention of the author that the less interested/pressed user of the package *should* be able to use other documents (c.f. *The GALFA Cookbook*) to reduce a region of data. Certainly it would do a new user good to examine *The GALFA Cookbook* before delving into this document, particularly if one is interested in a scrumptious batch of deep-fried GALFA.

2 The General Overview

The goal of this data processing package is to take the time-ordered data that comes out of GALSPECT, observed with a basketweave-mode over a *single region* and turn it into a calibrated, gridded spectral (PPV) data cube. The guts of the first steps in this process are handled by a suite of programs designed by C. Heiles. These programs deal with the so-called LSFS and HDR areas of the data reduction, which is to say they do a lot of corrections to individual data sets. These programs are explained in gory detail in *GSR/PROCS/INIT/HDR AND /INIT/LSFS SOFTWARE* as well as *GENERATE MH AND LSFS FILES* and I refer the interested reader to those documents. The pipeline being described in *this* document calls upon the C. Heiles suite of codes to do the first step of reduction, but is mainly concerned with reducing blocks of data together to make maps. Note, then, that it is not necessary to do any reduction before running the suite of codes described in this document.

A flow chart is provided below in Fig 2. The chart goes from upper left to lower right, like a page of text. Each entry on the top of the arrow is a program that one who is reducing the data would directly call, and below each of these entries is the data product associated with each of these. All capital letters in the data products are stand-ins for numbers or names that are attached to specific products. Each of these programs call on data products generated earlier in the pipeline, and many of the call on directly antecedent products. The reduction package contains many other programs, but these are the only ones that need be directly called by the user.

The programs covered are split into 3 groups - CALIB, XING and GRID. CALIB is responsible for the zeroeth order calibration of the data, and its organization by day of observing or 'scan'. XING is responsible for the 'crossing-point' reduction - using the information gleaned by comparing the relative strengths of lines observed at the same position on different days to constrain the gain of each beam. GRID is responsible for taking fully corrected spectra and turning them into a data cube.

The plan here is to talk about each of the programs that are called in the flow chart sequence and most of the sub-programs that they call. I heavily recommend looking at these while examining the source code, side-by-side. Note that author makes no claims to non-self-plagarism, and some of the content below may be repeated in the *The GALFA Cookbook* , or in the author's award-winning, yet-to-be-written autobiography.

3 CALIB Programs

3.1 `make_dirs.pro`

To simplify the reduction process we set up a directory structure that is unique to each project – this helps the code be sure where everything is, and is the basis for the rest of the reduction. The directory structure is regularized by `make_dirs`, which is run as follows:

```
IDL> make_dirs, root, project, regions, days, nox=nox, curfitsdir=curfitsdir
```

`root` is the directory the whole thing falls under, `project`is the project name that GALSPECT used to write the fits files, `regions` are the names of the different regions done in the one project (can be a single entry), and `days` are the number of days each one takes (again, can be an array or a single number). The directory structure it generates is shown graphically in Fig. 1. The fits files will be automatically transferred into the $/fits$ directory. If one doesn't wish to transfer these fits files, one can set the `nox` flag, and the fits files will be left untouched. If one wants to look for fits files in a directory other than the one that IDL is running in, just set `curfitsdir='/directory/you /like/'`. Note that these names, `root` and `project`, are standardized throughout the pipeline, along with `scans` and `region`.

Note that this stage of the reduction, making the directory tree, needs only to be run a single time for a project. The rest of the reduction process will refer to a single *region* within this project, and would have to be run multiple times for multiple regions.

	generating codes	generated data products	description of task
CALIB	make_dirs.pro	(directory structure)	generates uniform directory structure for data products
CALIB	stg0_st.pro/stg0.pro	/rega_NN/galfa.DATE.PROJ.####.rega.sav /rega_NN/galfa.DATE.PROJ.hdrs.rega.sav	removes bandpass, doppler corrects, tags and truncates data from one day's observing of one region
XING	xgen.pro	/xing/regaAA_BB.sav	finds all crossing points and generates a structure of locations and relevant parameters.
XING	lxw.pro	/xing/regaAA_BB_l.sav	loads weighted spectra into crossing point structure
XING	xfit.pro	/xing/regaAA_BB_f.sav	fits for the relative gain at each crossing point
XING	lsfxpt.pro	/xing/rega_lsfxpt_NAME.sav	generates a least-squares fit matrix that relates crossing point gains to beam gain variability with time
XING	xg_assn.pro	/rega_NN/rega_NN_xing_NAME.sav /xga_NAME.sav /xingarr_NAME.sav	solves the least-squares fit-matrix and applies the beam gain solution to all spectra
GRID	todarr.pro	/todarr.sav	generates a single structure with all position information for entire region
GRID	gridzalfa.pro	/sprs_GRIDNAME.sav	generates a sparse matrix that relates the time ordered data to a user-defined set of grid pixels
GRID	make_grid.pro	/GRIDNAME.sav	generates the final grid from the sparse matrix and gain coefficients

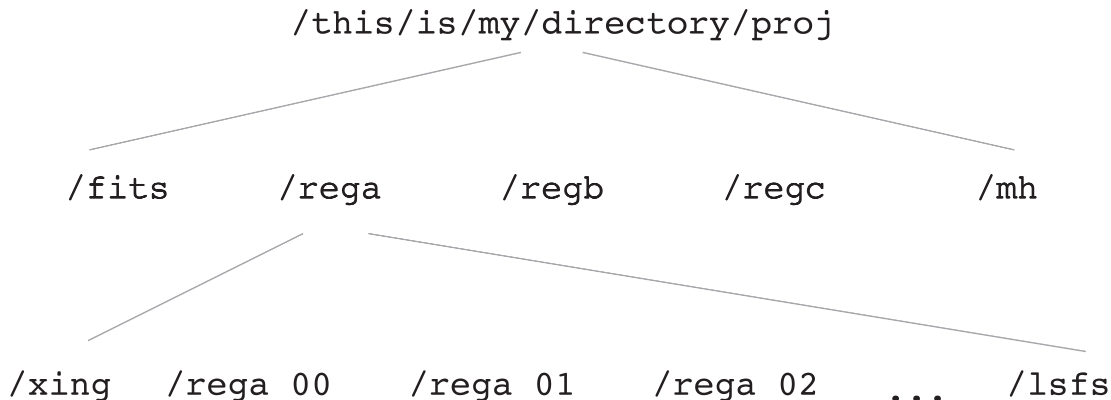

Figure 1: The file structure generated by `make_dirs, '/this/is/my/directory/', 'proj', ['rega', 'regb', 'regc'], [5,6,8]`

3.2 stg0.pro / stg0_st.pro

The stage 0 data reduction takes the data in raw form (fits files), removes the IF bandpass (see Heiles 2005, GALFA Technical Memo 2005-01 on how exactly this is done), does a rough gain calibration to put the data into temperature units, does a doppler correction to the LSR, finds the data that are during your BW scan and saves them to an appropriate folder, for a single day's observing of a single region. The data are organized by day numbers – if there are 11 days in your scan pattern your days will go from 00 to 10, the data for each day will reside in folders in */this/is/my/directory/proj/regx/regx*_00, .../*regx*_01, etc. In this case you would end up running some version of the stage zero reduction 11 times, one for each day. As a byproduct, each .fits file will have a corresponding .mh file that will live in */this/is/my/directory/proj/mh/* and each day will have .lsfs file that will live in */this/is/my/directory/proj/regx/lsfs*.

There are 2 ways to run the stage zero reduction process: `stg0.pro` and `stg0_st.pro`. `stg0_st.pro` is by far the easier of the two, as it relies upon a structure output from `BW_fm` for much of the information. If you are in a more complicated circumstance, or you do not have this structure available, use `stg0.pro`.

```
IDL> stg0, year, month, day, proj, region, root, $
     startn, endn, slst, elst, scan, cyc_time, lfn=lfn, nomh=nomh, $
     calfile=calfile, stops=stops, fitsdir=fitsdir
```

Some of these are a little subtle. `startn` and `endn` are the first and last files you are interested in reducing, as numbered by GALSPECT. `scan` is the day number you are interested in reducing (note that 'day' and 'scan' are used interchangeably in this text), and `cyc_time` is the cycle length of the basketweave. `slst` are the starting LSTs of the observation, as predicted by `BW_fm.pro`.

`elst` is the ending lst of the observation, and in most circumstances can just be found from the output of `BW_fm`, as is explained in the comments of the code. If, though, the observations were terminated early, but the spectrometer was left running for some time, it is best to put in the LST at which the BW was terminated, to avoid adding data to your map taken in some alien observing mode.

The inputs to `stg0_st` are simpler because a lot of the information is handled by the data in the outputs of `BW_fm`:

```
IDL> stg0_st, year, month, day, proj, scan, root, redst, nomh=nomh, delay=delay
```

`redst` is the output of `BW_fm` by the same name, and `delay` allows one to specify if this particular scan was started not on the first up-down-nod ('Λ') but at the beginning of a later scan. Note that if one had the scan truncated (as described above) and needs to choose an appropriate elst, one should use `stg0`, not `stg0_st`.

Both of the stage zero reduction codes call a sequence of relevant codes to do the work of the reduction. First, the mh files are generated. These are called through a code called `mh_wrap.pro`, which is a code in the `/procs/hdr` suite. Unless you provide it with the name of a previously generated LSFS file, through the calfile keyword, it will call a routine `lsfs_wrap.pro` which, given a list of file names, will generate as many LSFS files as their are SMARTF runs. Only the first of these LSFS files each day will be used to reduce the data for that day. After these two file sets have been generated, the code calls the awkwardly named `calcor_gs.pro` code.

`calcor_gs.pro` is responsible for applying the bandpass correction (through `lsfs_wrap.pro`) the temperature correction (again, through `lsfs_wrap.pro`), the doppler correction (through the .mh files), un-swapping any swapped receivers, tagging the data for bad receivers, and tagging the data with other bits of useful information, such as the the temperatures of the cals being applied. It also is responsible for only including data that are part of the BW scan, so that the rest of the reduction is not confused by extra data. To do these things `calcor_gs` calls a bunch of other yet smaller programs. For the bandpass correction `m1polycorr.pro` is invoked and for doppler correction, `dcs_wrap.pro` is called, which in turn calls `dop_cor_spect.pro`. Bad receivers are tagged with `whichrx.pro`.

The stage zero code generates two data products. One is the reduced data, packed along with the associated .mh data, information on which receivers are bad (a variable called rxgood), information on the wide-band spectrum and useful tags. These are saved in files with the format *reg_NN/galfa.DATE.PROJ.####.abc.sav*, each in a folder associated with the day that it was observed. The other data product is a single list of the information for a given day, including a über mh file that spans the entire day's worth of time. These are in the format *reg_NN/galfa.DATE.PROJ.hdrs.reg.sav*.

3.3 XING

3.3.1 `xgen.pro`

`xgen.pro` is responsible for finding all the crossing points. This is a relatively fast procedure, as the code only reads the .hdrs files, rather than the entire data sets. `xgen.pro` takes a standard set of inputs:

```
IDL> xgen, root, region, scans, dates, proj, goodx=goodx
```

like the following code, `xgen.pro` is primarily a wrapper for a more core piece of code, in this case `getx.pro`. It calls `getx.pro` for all possible combinations of scans and beams crossed with all other combiantions of scans and beams. It first does the auto section, which is to say beam-to-beam crossing within a single day, and then does the main section, for beam-to-beam crossing on days that are not alike. Note that on a given day we do not wish to allow all beams to cross each other - some beams never cross, and some beams cross in awkward points, when cals may be firing or when the telescope is slewing quickly. This is taken care of by a 7x7 matrix, `goodx`, which is set up for gear 6, normal basketweave scanning and can be superseded with a keyword of the same name. Note that only the upper triangle, `goodx[i,j]`, with $i < j$, is relevant. `getx.pro`, the core code, when handed a the appropriate mh files, will generate a structure (xarr) that contains a lot of different information about the crossing points. This information includes the day (or 'scan') number and beam number of each of the tracks that crossed, as well as time information, in which files the spectral data can be found and the positions of the crossing point. It also includes weights, which is to say how much emphasis to put on the data point before the crossing and how much to put on the one after the crossing. Currently this is just a linear weighting by distance from the crossing point. The structure also contains blanks for spectra to be loaded in and relative gains to be determined, that will be filled in later steps. `xgen.pro` Serves to feed this program the correct information to make crossing point structures and save them with the appropriate names. The data products are called *xing/regAA_BB.sav*

3.3.2 `lxw.pro`

After `xgen` is run, all of the spectra must be loaded into the crossing point files, with a code called `lxw.pro`. `lxw.pro` is called similarly:

```
IDL>  lxw, root, region, scans, proj
```

with all the inputs identically formatted. This takes a significant chunk of time, because each .sav file must be read a few times to load all of the spectra. As above, this is only a wrapper code. The core code that is being run is a code called `loadx.pro`. The way that this code gets the correct spectra from the correct files in a reasonable time (remember, IDL doesn't like doing loops), is *rather* complex. One thing it has to do is read spectra from 4 different places, the 4 files

where the spectrum before and after the crossing point is located. Usually this is only located in 2 files, as most crossing points do not exist between files, but some do. Note that this code calls a multipurpose code called `fixrx.pro`, which takes any dataset that has bad receivers and overwrites the offending data with its beam-pair. Since the spectra we are interested in are an average of these two polarizations, we don't want to average in any bad data. Of course, this reduces our SNR, but so be it. This procedure produces a similar data product to the last one, the only difference being that the slots for spectra are now filled with correctly weighted spectra. They are written in the format *xing/regAA_BB_l.sav*.

3.3.3 `xfit.pro`

The relative point-to-point gains must now be determined.

```
IDL>  xfit, root, region, scans, proj, noauto=noauto, conrem=conrem
```

`conrem` would only be set if the data had not had their continuum subtracted in the previous stages, and `noauto` would only be set if for some reason one did not want to compute the fit for crossing points within a single file. Both of these should be considered 'engineering' modes that should never need to be invoked.

`xfit.pro` follows the same structure as the previous two codes, but does not (for some reason) call a core code. It effectively plots the two spectra against each other and fits a line to the slope, thus determining the relative gain. It records this as well as any detected offset in the baseline, which is currently ignored. On occasion the fitting program flips out, with some part of the fit non-converging,so there is an error trap to deal with this - usually this come from user error. All of the original data, plus the gain and zero-point, less the spectra themselves, are saved in a structure called 'outx'. They are save in files called *xing/regAA_BB_f.sav*

3.3.4 `lsfxpt.pro`

`lsfxpt.pro` is the code responsible for engineering the 'equations-of-conditions' (X) matrix that fits all of the of the crossing point. It is based upon the idea that the ratio of the gains can be approximated as

$$R = \frac{G_{BD}(t)}{G_{B'D'}(t)} = \frac{1+\delta_{BD}(t)}{1+\delta_{B'D'}(t)} \simeq 1 + \delta_{BD}(t) - \delta_{B'D'}(t), \tag{1}$$

where B and B' are some arbitrary beams and D and D' are some arbitrary days. This allows us to set up our Y in the equation

$$Y = X \cdot C \tag{2}$$

as just

$$Y = \delta_A - \delta_B = R - 1, \tag{3}$$

which is linear in the $\delta_{BD}(t)$, and so can be solved with a set of linear equations. The C is a set of coefficients that determine the varying gain of each beam. The 'equations-of-conditions' matrix, that connects our data (Y) to the parameters we wish to fit (F) can be set up in a variety of different ways, controlled by the various keywords.

```
IDL> lsfxpt, root, region, scans, proj, $
degree, xarrall, yarrall, name,$
fourier=fourier, daygain=daygain, beamgain=beamgain
```

`root`, `region`, `scans` and `proj` are all standard inputs. `degree` is the degree of polynomials to fit to the varying gains of each beam and day. Set it to -1 to have no polynomial fits and to 0 and higher to have polynomial fits of those orders. `xarrall` and `yarrall` are the output matrices that are generated, and are only output here for diagnostics. `name` is the name of the fit, which allows the user to keep track of different attempts to fit the varying gains. It is not yet known how much the optimum parameters will vary from region to region. The `fourier` keyword allows the user to fit with sines and cosines by setting it to [a,b], where a is the lowest order (1 is a single period across the domain) and b is the highest order. Currently 1 is the lowest value that works for fourier (the zero-order fourier component, e.g. DC offset, can be done with zeroeth order polynomials). The `daygain` and `beamgain` keywords allow one to fit overall gains for each beam (which are equivalent to the cal values for that beam) and for an overall day. The code calls a piece of code called `makdom.pro` which, for each scan, generates a range over which the fourier components and/or polynomial coefficients can be evaluated. This range is expressed as a structure (mdsts) that can be read by `locdom.pro`, which is used to evaluate the elements of the X matrix. The X matrix must also contain constraints on the gains; since the gains are relative, the fits are equally good if all the gains are evaluated to be huge or tiny. We do this through a 'pinpoints' constraint. At a bunch of specific RAs, the total of all the fits for each beam and day is forced to be zero. These pinpoints are regularly spaced throughout the domain, and are proportional in number to the highest order (in fourier or polynomial) fit coefficients. The X and Y matrices that are generated by the program, along with the the mdsts structure, the length of the data (non-constraint) part of the Y array (ndata), the locations of the crossing points and the pinpoints are all saved in a file of the form */xing/reg_lsfxpt_NAME.sav*

3.3.5 `xg_assn.pro`

`xg_assn.pro` is designed to assign the gains to each point in the data set, given the output of `lsfxpt.pro`. The code has rather simple inputs:

```
IDL> xg_assn, root, region, scans, proj, fitsvars, name, cutoff=cutoff
```

`fitsvars` is an output for all of the variables that get generated while solving the matrix-inversion problem. It is good to examine this data to understand the goodness of your fit. `name` is the same as the name used in the `lsfxpt.pro` and names this set of gain files. `cutoff` allows one to set a cutoff value for the inverse of the weight matrix as determined by `lsf_svd.pro`. Note that this

code can take an *extremely* long time to run and may max out the memory of a computer. More than about 15 fourier coefficient terms in your X matrix (see `lsfxpt.pro`) may in fact top out a 2 GBs of RAM machine, and may take many hours to run. This procedure generates 3 data products. One is the crossing point gains, separated into their respective directories, in the files $/reg/reg_N N_x ing_N AME.sav$. Another is an amalgamation of all these data in a single file, called $/xingarr_N AME.pro$. The last contains all of the fitting data, and is called $/xga_N AME.sav$

3.4 GRID

3.4.1 todarr.pro

`todarr.pro` simply puts all the mh data together in one über-über mh-like file, for easy access. This structure is actually called 'mht' and it contains only the ras, decs and associated file names. These data are used in generating a grid. It is called as follows:

```
IDL> todarr, root, region, scans, proj
```

and generates */todarr.sav*

3.4.2 gridzalfa.pro

`gridzalfa.pro` has a storied lineage; gridzilla, AO_gridzilla, ao_gridzilla_GALFA and, now, the somewhat more sonorous gridzalfa. The basic product of `gridzalfa.pro` is a sparse matrix that relates time-ordered-data points (TODs) to grid pixels, without loading in any actual data. This matrix can then be multiplied by a data vector to produce a map with any binning or channel range desired, in `make_grid.pro`.

```
IDL> gridzalfa, root, region, proj, gridname, $
FWHM=fwhm, XGRID=xgrid, GRID=grid, $
IMSIZEX=imsizex, IMSIZEY=imsizey, REFRA=refra, REFDEC=refdec, $
GEOMETRY=geo, ASSOC=assoc, GFUNC=func, OBSERVER=observer , fd=fd
```

Clearly, there are many important parameters here in terms of defining the area of your map, the type of beam shape you wish to use and so forth. An example call might look like:

```
IDL> gridzalfa, '/share/galfa/', 'lwa', 'A2011', 'test1',  fwhm=3.2, $
xgrid=[1.,1.], grid=1.5, imsizex=1024, imsizey=512 , $
refra=[2.,15.,0.], refdec = [9.,45.,0.], geo='sin', gfunc='gau', observer='josh' $
```

The details of what each of the parameters means are detailed in the code itself, and exactly which to choose to have the best maps is yet to be optimized. The code has been optimized for speed –

11 hours of data takes about 30 seconds to run. Two somewhat subtle pieces of code are called in `gridzalfa.pro`. One is the suite of codes `llist_init.pro`, `llist_loop.pro` and `llist_read.pro`. The codes initialize, generate and read out a linked list of structures. These are implemented to speed up the data processing immensely. The other piece of code we implement is `sprsin.pro`, which is a routine that generates a sparse matrix, as in *Numerical Recipes*. It is this sparse matrix that is used in later processing. The data product that this step generates includes a structure that contains all the input parameters (fd), the sparse matrix (spmat) as well as the the normalized weight matrix (wnorm), used for normalizing the amplitude of the time-ordered data . It is in the format *sprs_GRIDNAME.sav*.

3.4.3 `make_grid.pro`

`make_grid` is the final step. It takes the sparse matrix generated by `gridzalfa.pro` and multiplies it by a vector of data it pulls from the reduced data files. This data vector is multiplied first by the gains determined by `xg_assn`. It is called as

```
IDL>  make_grid, root, region, proj, gridname, spcen, sprng, spavg, name=name
```

`gridname` is the same name that was used by `gridzalfa` and helps separated different gridding attempts. `spcen`, `sprng` and `spavg` are the center, range and averaging size for the spectrum that one wants to output. This can make a grid from 8192 spectral channels to 1 spectral channel wide. The `name` keyword allows one to use XING calibration by specifying a name. Leaving this unset will result in no XING calibration in the final data cube. The data cube is written out as both a .sav file and a .fits file.