

The GALSPECT pipeline version 2.2: GALSPECT with a vengeance

Joshua E. G. Peek (né Goldston)

August 29, 2006

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 GSR reduction package, as written in IDL. The *GALFA Cookbook* has more information about observations and other reduction paths as well. 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 over a *single region* and turn it into a calibrated, gridded spectral (PPV) data cube. Note that this version of the data reduction pipeline is functional for GALSPECT data taken with any observing mode, and is no longer restricted to modes monotonic in RA. 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, although the by-product of ‘datachk’ routines, such as the lsf and mh files can be used in this reduction - they are identical to some of the products of stg0, and so some computer time can be saved by moving these to the appropriate directory. Also note that the Archiver (see documents on this topic by M. Krco), will generate mh files that can be used in this process as well.

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 5 groups - CALIB, SPMOD, XING, GRID and AUX. CALIB is responsible for the zeroth order calibration of the data, and its organization by day of observing or 'scan'. SPMOD is responsible for dealing with minimization of baseline ripple in the data set. 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. AUX is a set of auxiliary codes that may be employed in reduction, and will be explained last, though the codes may be useful in other steps of the reduction.

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 this document while examining the source code, side-by-side; the code is well documented, and many questions on subtleties can be understood by reading the annotated code itself. That being said, some of the code involves some mind-bending array gymnastics, (and calling of C routines) and so may not be totally transparent to the reader (or, for that matter, the author). This code, though rather well refined at this point, may still be difficult for those not familiar with IDL to use, particularly when exploring new observational and reduction parameter space. Note that author makes no claims to non-self-plagiarism, 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 and/or thesis.

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. It is *completely* mandatory to set up this file structure in this way - otherwise the code will fail. The directory structure is regularized by `make_dirs`, which is run as follows:

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

`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). **NB:** It is recommended to name your regions with a short string of lowercase letters - names that use underscores, hyphens, capital letters or, heaven forbid, *spaces* may not



be supported in the software. 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/'`. The `tdf` flag allows for reverse compatibility to the original two-digit formatting that was the standard for all previous versions of the pipeline. The current version uses three digit formatting to label directories by day in the scan. Note that these names, `root` and `project`, are standardized throughout the pipeline, along with `scans` and `region`.

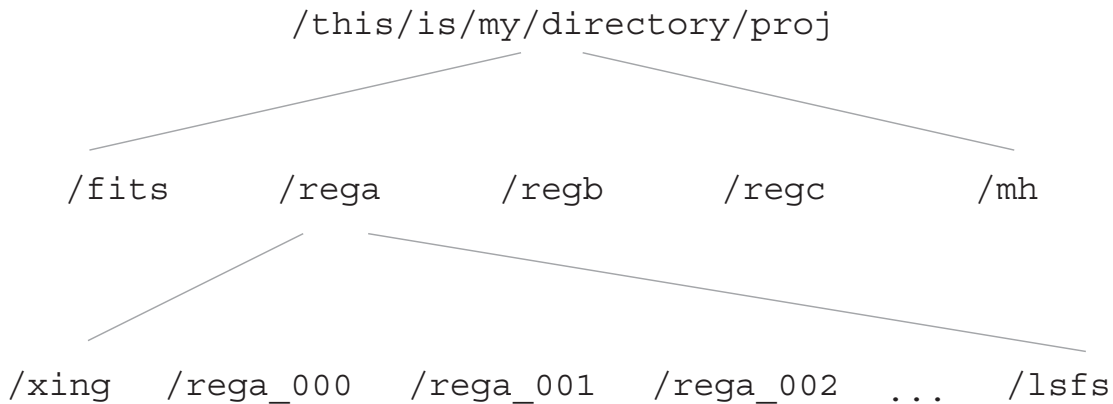


Figure 1: The file structure generated by `make_dirs`, `‘/this/is/my/directory/’`, `‘proj’`, `['rega', 'regb', 'regc']`, `[5,6,8]`

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. It is also important to realize that the reduced data itself will be imprinted with the original root directory you choose to run the code in. It is advisable to run this reduction in one directory, and not move it from place to place, as this will confuse the reduction.

3.2 stg0.pro

The stage zero 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 000 to 010, the data for each day will reside in folders in */this/is/my/directory/proj/regx/regx_000, .../regx_001*, 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*.

This is how to call `stg0.pro`:

```
IDL> stg0, year, month, day, proj, region, root, startn, endn, slst, $
      elst, scan, nomh=nomh,mhdir=mhdir, caldir=caldir, calfile=calfile,$
      stops=stops, fitsdir=fitsdir, userrxfile=userrxfile, tdf=tdf, odf=odf
```

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). `slst` are the starting LSTs of the observation. `elst` is the ending LST of the observation. If your observation was a standard BW observation, the relevant LSTs can be found in the output of `BW_fm.pro`, unless the observations were terminated early and the spectrometer was left running for some time; then it is best to put in the LST at which the observation was terminated, to avoid adding data to your map taken in some alien observing mode. `userrxfile` Allows one to specify a file (including the full path) that tells the code to ignore a bad receiver. These files can be generated with a code called `edbdrx.pro`, which takes as inputs a list of receivers and a list of start and end seconds (UTC) for when they are bad, and a file to write it to. This is intended to be used for single receivers that are bad over entire days, not on a second-to-second basis. Note that you will cause havoc if you specify two receivers on the same beam to be bad - there will be no receivers to average over for that beam.

`nomh` allows the user to specify that no new mh files need to be generated, `mhdir` specifies that mh files can be found in a specific non-standard directory. `caldir` similarly allows one to set a specific non-standard directory for the LSFS file, and `calfile` allows the user to specify a specific LSFS file to use, rather than generating one anew. `fitsdir` allows the user to specify a non-standard location for the raw fits files. `tdf`, if set, uses the old two-digit format for directories (not recommended unless reverse-compatibility is crucial) and `odf` saves the reduced data in the .sav data format, as in the previous version, rather than in the .fits format employed in GSR 2.2. This is also strongly advised against, particularly in areas with many crossing points.

The stage zero reduction code calls 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 `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 calcs 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`. On top of all this, a code called `deflect.pro` gets rid of the effect of any bad impedance matching in the cables that run from ALFA downstairs. This single fourier component ‘reflection’ can dominate any other ripple in the spectra.

The stage zero code generates three data products. One is the reduced data, saved in a fits file with the format `reg_NNN/galfa.DATE.PROJ.####.abc.fits`The second is an associated .sav file, containing 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_NNN/galfa.DATE.PROJ.####.abc.sav`, each in a folder associated with the day that it was observed. These two files contain all the data that was originally contained in a single .sav file in the previous versions of this pipeline; one can revert to this method, with the use of the `odf` keyword, though it is not advised. The last 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_NNN/galfa.DATE.PROJ.hdrs.reg.sav`.

3.3 SPMOD

The purpose of the SPMOD software is to get rid of ripples generated by reflections inside the Gregorian dome. Our tests have shown that the largest contribution to baseline ripple (after fiber reflections) is fixed over the course of a day in each receiver. We use this information, along with some sophisticated modeling of the HI we observe, to greatly decrease the contribution of baseline ripple. A subtlety with this approach is that we need to know the relative gains of the receivers before we can disentangle the HI from the background ripple, but was need to be able to remove the background ripple so that we can accurately determine the relative gains! Our solution is to jump-start the process by making a guess as to the relative gains of the beams first, by just comparing the HI from beam-to-beam each day. We then take that, use it to get rid of the ripple (SPCOR), use the ripple-removed data to do an accurate gain calibration (XING), and then go back and re-run the ripple removal process (SPCOR). So the true process is INIT, SPCOR, XING, SPCOR, GRID. It is crucial to mention that it is not yet known whether this has any useful effect upon data where the ALFA rotation angle changes substantially during the run or upon data where the telescope points significantly off of the meridian during the run. It is the author’s guess that this is a useful reduction to run for the latter situation, and not the former, but at this point this is only educated speculation.

3.3.1 spcor.pro

`spcor.pro` is responsible for doing this processing and calling all the relevant sub-codes.

```
IDL> spcor, root, region, scans, proj, noaggr=noaggr, userrxfile=userrxfile, $
```

```
spiter=spiter, xingname=xingname, tdf=tdf, odf=odf
```

The first four inputs are in the standard format. The `noaggr` keyword is for if you have already run the code and trust your aggregate spectrum file (`aggr.sav`) - aggregating the spectra takes some time, so this a good keyword to use if you have already generated your `aggr.sav` file - it is ignored if `spiter` is set. `userrxfile` is the same as is `stg0.pro`, and should be set if you have bad rx's. `spiter` should not be set the first time through SPCOR - it is the iteration number of the code, and if set will instruct the code to read XING and old SPCOR data. The second time through SPMOD it should be set to 1. `xingname` is also for the second time through SPCOR - set it to the name of your XING reduction (the `name` variable in `lsfxpt.pro` and `xg_assn.pro`). The `odf` and `tdf` keywords are the same as in INIT. **NB:**This re-reduction **only works** if the `lsfxpt.pro` is run without `daygain` or `beamgain` and with `degree` set to 0. The `fourier` keyword can be set however you like. This may be changed in later versions. `spcor.pro` will call three main codes to do the analysis - `aggr_spect.pro`, which aggregates spectra over the whole data set, `zogain.pro`, which determines the zeroth order gain corrections and `find_fpn.pro`, which finds the so-called 'fixed pattern noise', the dominant part of the baseline ripple which we are attempting to mitigate.

The data products from this code are `aggr.sav` and `spcor-##.sav`.

3.4 XING

3.4.1 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, xday=xday, tdf=tdf
```

like the following code, `xgen.pro` is primarily a wrapper for a more core piece of code, in this case `getx.pro`. `getx.pro` has been completely retooled for this release of the code, and now has a more robust algorithm for finding crossing points. It calls `getx.pro` for all possible combinations of scans and beams crossed with all other combinations 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 cal's 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. `xday` can be set to an equivalent matrix, but for days. If you do not wish to look for crossing points between specific days, set this to be a $N_{days} \times N_{days}$ array, with 1s at $[n,m]$ where you wish to compute crossing points between day n and day m and 0s where you do not. `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/regAAA_BBB.sav`.

3.4.2 `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, spiter=spiter, no_over=no_over, $
      file=file, tdf=tdf, odf=odf, no_auto=no_auto, xday=xday,
```

with all the inputs identically formatted. The keyword `spiter` allows the user to input an number of the `spcor.pro` iteration done to use to load corrected data into the crossing point code - you would typically use the number 0 here, to use the first `spcor.pro` data. The `file` keyword here refers to any `badrx` file you might wish to use to avoid loading corrupted spectra. The `no_auto` keyword allows the user to skip loading crossing points within a day (usually used for engineering) and `no_over` allows the user to not reload data that has already been loaded, if the program was interrupted. `lxw.pro` takes a significant chunk of time, because of the amount of file reading and writing required. As above, this is only a wrapper code. The core code that is being run is a code called `loadxfits.pro`, a much simplified version of `loadx.pro`. Note that `loadx.pro` is used when the data are stored in the older data format, which can be *excruciatingly* slow. As it is, this code can take hours to run. 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/regAAA_BBB.l.sav`.

3.4.3 `xfit.pro`

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

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

`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/regAAA_BBB.f.sav*

3.4.4 `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

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

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

$$Y = X \cdot C \quad (2)$$

as just

$$Y = \delta_A - \delta_B = R - 1, \quad (3)$$

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, $
big=big, tdf=tdf, time=time, blankfile=blankfile
```

`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. 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 zeroth 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. It is not yet known how much the optimum parameters will vary from region to region, but a good first guess might be to set `degree` to 0, to get the basic overall numbers, and `fourier` to [1, 3], to get a little bit of higher-order correction. Three new keywords have been implemented in this pipeline release. The first is `big`, which allows the code to handle situations where the number of crossing points is so large that the X matrix cannot be handled in memory. In this case, for each crossing point file $X^T X$ and $X^T Y$, are generated and then co-added to all following files. The second is the `time` keyword, which, if set, parameterizes the gain variations in UTC time, rather than RA. It is not clear to the author if there is ever a reason not to set this keyword, but it certainly must be set for any data set non monotonic in RA. The last new keyword is `blankfile`, which allows the user to input a file containing a list of time domains, as generated by `edblanks.pro`, which removes any crossing points that the user may think to be contaminated, and therefore may contaminate the gain calibration. `lsfxpt.pro` 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 (or times), 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`

3.4.5 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, big=big, tdf=tdf, time=time
```

`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`. `big` should be set if it was set on `lsfxpt.pro`, to do the correct matrix inversion. `tdf` reverts to the two-digit format. `time` assumes the crossing points are parameterized by time, rather than RA, and should be set if it was set in `lsfxpt.pro`. Note that the 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_NNN/reg_NNN_xing_NAME.sav`. Another is an amalgamation of all these data in a single file, called `/xingarr_NAME.pro`. The last contains all of the fitting data, and is called `/xga_NAME.sav`

3.5 GRID

3.5.1 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. This code needs to be run only once per reduction - even if later steps fail, the data product from this step need not be regenerated. It is called as follows:

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

and generates */todarr.sav*

3.5.2 sdgw.pro

`sdgw.pro` has a storied lineage; `gridzilla`, `AO_gridzilla`, `ao_gridzilla_GALFA`, `gridzalfa` and, finally, `sdgw`, which departs somewhat from the theme. The product of `sdgw.pro` is the final gridded data in two formats; `.fits` and `.sav` files. The complexities of how this code works are way beyond the scope of this document, except to say that it is a wrapper code that calls the brains of the operation, `sdgrid.pro`. It calls the code once to determine which files are needed in a given grid and the again for each data file that needs to be loaded. It is called as follows

```
IDL> sdgw, root, region, proj, gridname, lon0, lat0, spmax, $  
spmin, spres, imsize, projection=projection, _REF_EXTRA=_extra, $  
tdf=tdf, spiter=spiter, file=file, name=name, fwhm=fwhm, vel=vel, $  
allfiles=allfiles, savepath=savepath, gridfunc=gridfunc, odf=odf, $  
norm=norm, blankfile=blankfile
```

The first three parameters are as usual. `gridname` is any string you would like to identify the grid with. `lon0` is the center of the grid in longitude space in degrees. `lat0` is the same for latitude. `spmax`, `spmin` and `spres` define the maximum, minimum and binsize for the spectra to grid, which can be specified in spectral channel or in km/s, depending upon how you set the keyword `vel`. `imsize` is a 2-element array that is set to the size of the region in pixels. If the keyword `name` is set, crossing point calibrations with that name are applied to the data. `rs` allows the user to change the root of the data file, if the reduction was begun on another file system. Specify `file` to use a specific `badrx` file to eliminate bad rxs. Specify `savepath` if you wish to have the data save somewhere other than the main directory of the region. To use the spectral correction from SPMOD set `spiter` to the last iteration of `spcor` done, starting at 0. `tdf` and `odf` are as usual and `fwhm` specifies the FWHM of the beam, which can just be left to the default 3.35 arcminutes. `allfiles`, if set, skips the step in which the files to be included are verified - this will save a little time if you are certain that all your data files lie within your grid region, and will lose a lot of time if you specify it without this being true. `projection` specifies one of the projections from Calabretta

& Greisen 2002, A&A, 395, 1077 - see line 278 of `sdgrid.pro`. If this is not set, the default is the Cartesian projection. `arcminperpixel` is self explanatory, and is defaulted to 1. `gridfunc` allows the user to specify a gridding kernel, and is defaulted to a Gaussian kernel. `norm` allows the user to specify a normalization of the entire data set to divide by. Setting `norm` eliminates the LDS calibration of the data that would normally happen. It is a good idea to use `norm` if either you are looking at data very far off the static Galactic HI (e.g. HVCs), so that calibrating would be very noisy, or if you are doing many grids you wish to stitch together, where you would like to have a fixed calibration. In both of these cases, it is recommended that the user do a first pass on a normal sized grid that spans the static HI line, retrieve the normalization factor and then apply it to subsequent grids. `blankfile` allows the user to specify a file that contains information about bad seconds, and removes said data.

3.6 AUX

AUX contains a few codes that allow the user to do some fancy things with the data, and in particular allow the user to manipulate file structures using symbolic links, to cull and combine data sets.

3.6.1 merge.pro

`merge.pro` allows the user to combine two completely separate data sets into a new data set. This allows the user to do crossing point calibrations between two data sets. It is called as follows:

```
IDL> merge, root1, proj1, region1, days1, $
root2, proj2, region2, days2, $
newroot, newproj, newreg, odf=odf, tdf=tdf
```

The inputs are all in analogy to the simple `root`, `proj` and `region`, but specify the two original regions (1 and 2) and map to a new region (`new`). This is all accomplished through symbolic linking.

3.6.2 subday.pro

`subday.pro` is built on the same chassis as `merge.pro`, but has a slightly different goal. It extracts a subset of days the user wishes to examine from a single region and symlinks it to a new directory. It is called as follows:

```
IDL> subday, sds, root1, proj1, region1, days1, $
newroot, newproj, newreg, odf=odf, tdf=tdf
```

and is in complete analogy to `merge.pro`. `sds` is a list of days the user wishes to keep.

3.6.3 `dataplot.pro`

`dataplot` is a simple diagnostic tool that allows the user to plot all of the positions of the data in a region. It is called as follows

```
IDL> dataplot, root, region, scans, proj, $  
name, thin=thin, _EXTRA=ex
```

The only interesting keyword is `thin`, which allows the user to plot only 1/`thin` of the data points, to avoid overtaxing the code in the case of very large data sets. Note that any keyword that can be passed to the IDL routine `plot` can also be passed to `dataplot.pro`; e.g. if you wish to limit the y range, `/yzero` can be set.