IDL Software for LSS

Contents

1 Introduction

1.1 General tidbits

This document describes a set of utility routines, written in the IDL programming language, that are used to process, display and reduce Liquid Surface Scattering (LSS) data at the ChemMatCARS LSS facility. No knowledge of IDL, other than some details explicitly mentioned in the text, is required of the users (those who would like additional information, can obtain it from the IDL Help Menu.

In an attempt to reduce confusion, the following notation will be employed throughout the document:

• LARGE CAPS	Routine names.
• SMALL CAPS	Keywords, generic variable names.
• *Italics*	Expressions, function arguments, actual variable names.
• [ ] (Square brackets)	Depending on the context, denote an array, or a set of optional items

The meaning of "routine", "keyword" and the like is clarified below. None of the above, of course, means that one should use large and/or small caps as well as italics while typing actual IDL commands. In fact, it is perfectly OK to use lower case letters only, saves wear and tear on the Shift keys.

Throughout the document, the terms "**scan**" and "**frame**" are often used. "Scan" may cause some confusion since it may refer to:

1. A single SPEC scan, a group of data points collected through a single SPEC command and grouped together in the experiment's SPEC file.
2. A LSS scan, such as a XR (reflectivity) or GID scan (these will be dealt in detail later). A single LSS scan may include a whole group of SPEC scans (10-12 SPEC scans in a single XR scan is not rare).

In order to avoid confusion, the term "scan", with no qualifier will always refer to a single SPEC scan. LSS scans will always be mentioned with a qualifier, e.g. "XR scan", "GID scan" etc.

There is no such ambiguity regarding "frame", it simply refers to an area detector image corresponding to a single data point.

1.2 IDL Syntax tidbits

While, as mentioned above, no general knowledge of IDL is required of the users, a rudimentary knowledge of syntax is helpful. The basics are described below. Users with previous IDL exposure may well skip this section.

1.2.1 Constants and Variables

As any programming language, IDL utilizes constants and variables. Constants have a value, variables have a value and a name. "Value" is meant here in a general sense, it doesn't need to be a single value, it may be a vector, a 2D array or a higher dimensionality (up to 8 dimensions) array. It may be a numeric value, a character (string) value, a structure or a more exotic "creature".

In case of doubt as to what a given IDL variable represents, one can use the a call to HELP. IN the example below, three variables, X, Y and Z have been already defined and HELP is invoked as

```
IDL> help, x, y, z
X               INT       =        42
Y               FLOAT     = Array[3, 5, 189]
Z               STRING    = Array[4, 5]
```

(note that IDL> in the above is just the IDL command prompt, not part of the command)

Arrays, when created explicitly, are marked by square brackets, for example

```
IDL> arr = [3.5, 17, -3, 22.22]
IDL> print, arr
      3.50000      17.0000     -3.00000      22.2200
```

Character strings are denoted by an opening and closing apostrophe. For example, 'whatever' is a character string. '33' is also a string, distinct from the number 33. Instead of apostrophes one can use quotation marks. One or the other, but no mixing.

1.2.2 Routines

All the commands issued in IDL involve a call to an IDL routine. The routines come in two flavors, **procedures** and **functions**.

IDL **procedures** use the syntax:

PROCEDURE_NAME [, *Arguments*]

The square brackets are not part of the call syntax, only an indication that some or all of the arguments may be optional. The arguments may be constants, variables or expressions involving constants and/or variables. The number of arguments can be as low as zero, in some cases. For example, the command PRINT (which is a procedure) can be invoked simply as:

```
IDL> print
```

in which case a blank line is printed. It can also be provided with an explicit argument, or expression, e.g.

```
IDL> print, 17.2
      17.2000
IDL> print, 17.2^3 - 14*24
      4752.45
```

or a previously defined variable(s)

```
IDL> x = 21.7
IDL> y = x^2 - 5*x + 3
IDL> print, x, y
      21.7000      365.390
```

As can be seen from the last example, more than one argument may be specified. In fact, the number is unlimited, for the PRINT procedure.

IDL **functions** use the syntax:

Result = FUNCTION_NAME([,*Arguments*])

Where, again, the square brackets are not part of the syntax. The parentheses, on the other hand, are an integral part of the syntax and must be present even when no arguments are used.   *Result* is the value returned by the function (may be a numeric scalar or array, character scalar or array, other options exist). Said value can be assigned to a variable or provided as argument to another routine. For example, in the statement

```
IDL> print, exp(2)
      7.38906
```

the IDL exponential function EXP is used to calculate exp(2) and the result is submitted as an argument to the PRINT procedure. The function EXP takes a single argument, but other functions may take two or more arguments or, in some cases, no arguments at all. For example, in the statement

```
IDL> res = machar()
```

some machine dependent calculational parameters are returned by the function MACHAR which is called with no arguments. Note that the parentheses are still present.

To sum up, the main distinction between procedures and functions is that procedures perform some action while functions return some value. However, as will be shown later, procedures can also be made to return values, while functions can perform actions, so the distinction is not sharp.

As for the arguments, they also come in two varieties, **positional arguments** and **keywords**.

Positional arguments are the standard arguments from math. For example, the IDL Bessel J function is called using

Result = BESELJ(X, N)

where X is the independent variable (the input) and N is the order of the function. So, if we'll define Z = 17, ORD = 4, then BESELJ(Z,ORD) will return The value for the 4th order Bessel J function of 17, which won't be the same as BESELJ(ORD,Z). The role of a positional argument depends on its position in the routine call.

Keyword arguments have a name and are referred to by this name in a routine call. For example, the IDL PLOT routine, which will be encountered later, has (among others) a keyword named LINESTYLE, and a typical call to the routine may take the form

PLOT, X, Y, LINESTYLE = 4

In this case, the positional arguments X and Y provide the coordinates of the points to be plotted, and the keyword argument LINESTYLE takes the value 4. The values of keyword parameters do not depend on their position in the routine call.

1.2.3 More about keywords

In view of the content of the following chapters, some info regarding keywords needs to be mentioned, such as:

- Certain keywords are switches, used to turn an option on and off, and as such only two values, 0 and 1 (or, if one prefers, zero and not zero) are used. In this case an abbreviated notaion is used where instead of writing KEYWORD = 1 one writes /KEYWORD. For example the BESELJ function mentioned above accepts the keyword DOUBLE which is a switch specifying that the evaluation is to be performed in double precision. Thus, instead of writing

  Result = BESELJ(X, N, DOUBLE = 1)

  we can write

  Result = BESELJ(X, N, /DOUBLE)

  which saves couple keystrokes and wear and tear on the SHIFT key☺. Many of the keywords that'll be encountered in the following are of this type.

- Keywords can be used not only to transfer values to routines but also return values from the routines. When a keyword is provided with a name of a variable then, if said variable acquires a value within the routine, the value is returned to the caller. In such way procedures, which normally do not return values, can be made to do so.

- Keywords can be abbreviated to their shortest unambiguous length. For example, the routine PLOT mentioned above uses, among others, the keywords CHARSIZE and CHARTHICK (doesn't matter what they mean, for the moment). In usage, CHARSIZE can be abbreviated to CHARS, causing no problem. It cannot be abbreviated to CHAR, though, because CHAR may be an abbreviation of CHARTHICK as well. In case of doubt as to how far a keyword may be abbreviated, the rule is simple, as long as IDL doesn't complain it is OK, or, in the words of A. Clarke, "the only way to find the limits of the possible is by going beyond them".

1.3 Data formats

The LSS data, as generated by our software and (possibly) saved comes in two formats, **1D** and **2D**.

The 1D format is a [3,N] array, meaning an array of 3 columns and an arbitrary number of rows. The values present in the 3 columns are:

1st) Independent variable (i.e. scan variable) values
2nd) Actual data values. May be raw detector counts, normalized detector counts, or processed data.
3rd) Statistical uncertainties of the data values, evaluated based on the raw counting statistics as well as any additional processing that took place.

Thus a single row of the array is a triplet of values representing a single data point with its statistical uncertainty.

The 2D format is a three dimensional, [4,N,M] array. It should be considered as a 4-page "booklet", with each page being an [N,M] array (where M, N are the dimensions of the 2D data). Similarly to the 1D case, the pages contain:

1st) The horizontal coordinates of the 2D data.
2nd) The vertical coordinates of the 2D data.
3rd) Actual data values, can be anything 1D data values can.
4th) Statistical uncertainties of the data values. Again, those include counting statistics and any subsequent processing.

Various tools for the processing, displaying, saving and rereading data, both 1D and 2D, exist and some of them will be described in detail later.

2 Sample Session

In this chapter a brief exploration of our software is presented. All the commands used are shown and explained at a rudimentary level, more detailed information will be presented in the following chapters.

At places, some additional content is provided within grey-shaded “boxes”. This includes information which, while useful, is not necessary at first reading and may be skipped.

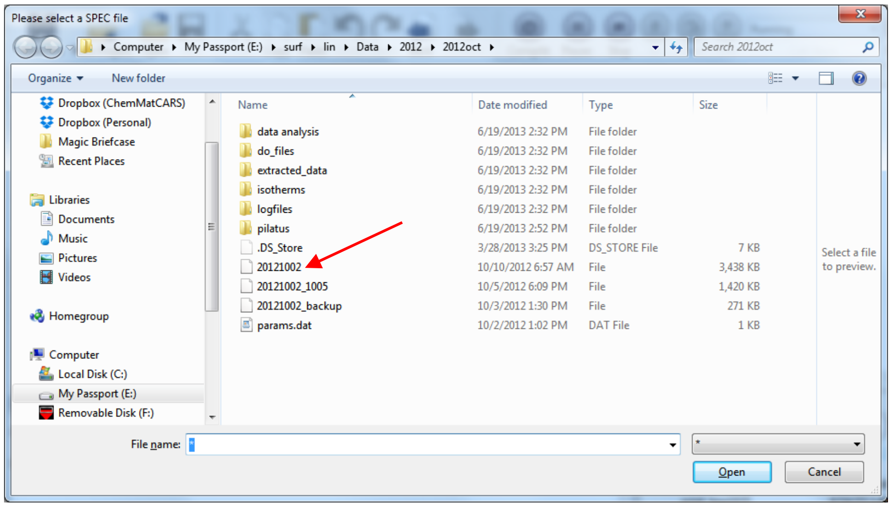
Wherever IDL commands result in screen or graphic display, a two column table is used, with the display on the left and the user input on the right (the other way around might be more appropriate, but this is our traditional way and tradition shall be respected☺)

2.1 Opening a session

Any LSS experiment has its SPEC file and the first step in an IDL LSS session is establishing a link between said file and IDL. This is done through the command

```
IDL> scan_info, /new
```

Following the command, a dialog window opens, asking the user to select a SPEC file. It is up to the user to know (or to ask somebody who knows) where the file is and how it is called. The typical file will be of type “file” (see example below).



Name	Date modified	Type	Size
data analysis	6/19/2013 2:32 PM	File folder	
do_files	6/19/2013 2:32 PM	File folder	
extracted_data	6/19/2013 2:32 PM	File folder	
isotherms	6/19/2013 2:32 PM	File folder	
logfiles	6/19/2013 2:32 PM	File folder	
pilatus	6/19/2013 2:52 PM	File folder	
.DS_Store	3/28/2013 3:25 PM	DS_STORE File	7 KB
20121002	10/10/2012 6:57 AM	File	3,438 KB
20121002_1005	10/5/2012 6:09 PM	File	1,420 KB
20121002_backup	10/3/2012 1:30 PM	File	271 KB
params.dat	10/2/2012 1:02 PM	DAT File	1 KB

In some rare cases (mostly with quite old data) following selecting the file another dialog window opens, asking the user to select a Pilatus data folder. In such case, one is to click on a subfolder called *Pilatus* within the same folder on which the SPEC file resides.

After selecting the file, SCAN_INFO will take some time (typically 5-20 seconds, depending on the file size and the speed of the computer) and then it’ll display file information as follows:

```
Filename    :     E:\surf\lin\Data\2012\2012oct\20121002

Date/time   :     Fri Sep 28 14:19:46 2012
   759 scans present

Listed local parameters:

Chi Phi In_Ht In_Rot Sample_H Sample_X Two_Thet Det_Th
Absorber Out_Th Out_Ht Out_Rot XOut_Ht XOut_Rot an_Sam_H Theta
Mir1Y Mir2Y CCD_X CCD_Y
```

At this point the link between IDL and the SPEC file has been established and the user can proceed to extracting, plotting and saving data.

Some comments:

1) The “switch keyword”, /NEW isn’t really needed when one selects a SPEC file the first time in a session. However, it does no harm and is necessary if and when one selects another SPEC file.
2) At any point during the session, only one SPEC file is linked to IDL.
3) Once /NEW is invoked (even if erroneously) there is no going back, a SPEC file must be selected. One can select same file as before, though.
4) If one knows the full (meaning including path) filename of the required SPEC file, then instead using the file search dialog, the command SPEC_INFO, FILE = *filename* (where *filename* stands for the required file) may be used.

2.2 Extracting XRF data

The primary routine used to extract X-ray Reflectivity (XRF) data is **PD_XR**. In addition, **PD_XR_VIEW** and **SCAN_PD_XR** (which is actually the “engine” underlying PD_XR) are used at times to verify data integrity and identify potential problems.

As mentioned before, a reflectivity scan consists of a group of SPEC scans and it is up to the user to know which SPEC scans make a proper group (that’s why records must be kept).

A function called **SCAN_FIND** may help in locating the reflectivity scans. The syntax is

```
IDL> list = scan_find(type = 'xrf',/show)
35,37-44
82-90
91-97
117-123
124-131
305-310
```

The column list printed as a result of the call contains the list of reflectivity groupings (the list is much longer here, it has been truncated for brevity). The *list* variable (any other name may be substituted here) receives the result of the function which is a string array containing the list. Checking

```
IDL> help, list
LIST            STRING    = Array[36]
IDL> print, list[2]
91-97
```

Each element of the list can serve as an input to PD_XR. One should not think of SCAN_FIND, though, as a replacement for record keeping. It is good but not perfect.

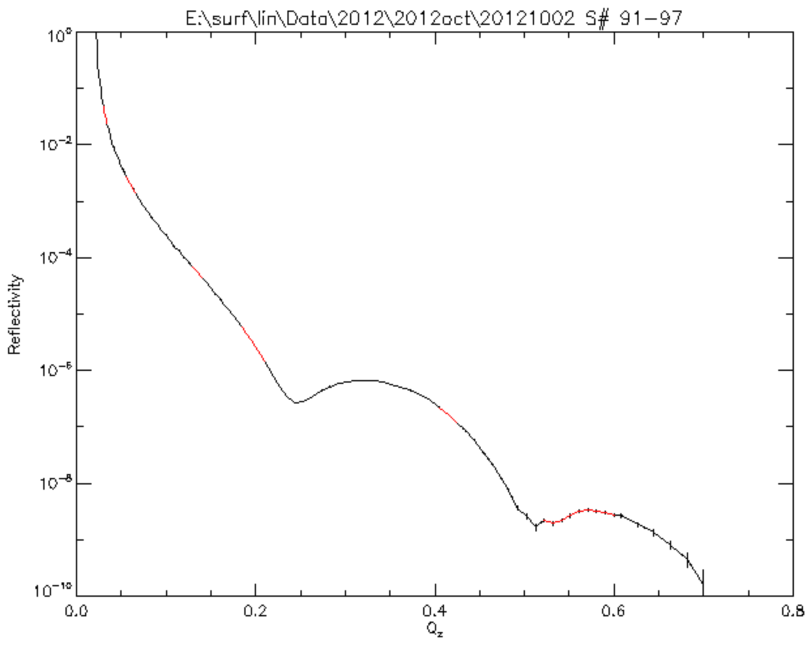
A list of the scans comprising a single reflectivity scan is the primary input to PD_XR. The list can be provided as a range, e.g. ’91-97’, combination of ranges, e.g. ’61-65,68,57-59’, or a numeric array, e.g. [25,26,28,29,30,23]. Note that the list can be given in any order.

The basic syntax of a PD_XR call is

PD_XR, SNUM, SLIT = [*hor*,*ver*], BACK_ANG = *angle*, RESULT = *resname*

where:

SNUM	-	A list of numbers representing a reflectivity scan, as explained above.
SLIT	-	Keyword, accepts a 2-element vector containing, in order, the horizontal and vertical "electronic slit" (i.e. signal region of interest), in **pixels**. Typical values for the Pilatus detector are [23,11], translating roughly to 4×2 mm (the Pilatus pixel size is 0.172mm), but users may use other values.
BACK_ANG	-	Accepts a value for the "sideways" offset angle for the background evaluation regions of interest. Given in degrees (typical value is ~0.5°).
RESULT	-	Returns the reflectivity data, in the standard 1D, 3 column, format (see Sec. 1.3). The first column contains the Q_z values, the second, reflectivity values, and the third, data errors. This keyword is not mandatory but, as there is no overhead involved (the data is evaluated anyway), it is a good idea to always use it. Note that *resname* can be anything.



Example:

```
IDL> pd_xr, '91-97', slit=[23,11], back=0.5, res=stuff
```

Here we combine 7 SPEC scans to generate the reflectivity curve on left. Following the command, the variable *stuff* contains the reflectivity data in the standard 1D format, see

```
IDL> help, stuff
```

```
STUFF           FLOAT     =
Array[3, 111]
```

Note that the names of some of the keywords were abbreviated. This is legitimate (see Sec. 1.2.3)

The color coding of the plot is meaningful, the red areas indicate patching regions between neighboring SPEC scans.

Of course, ultimately, the goal is not just to plot the extracted data but to save it in a form which can be later imported into any software the users may be utilizing for data analysis. This is done using the routine **WRITE_DATA**. The syntax is simple, for the data generated above one would do

```
IDL> write_data, stuff
```

Following the command, a dialog window opens, allowing the user to pick the save location and the save-file name. The data is saved in 3 columns, in a simple ASCII file (extension .txt)

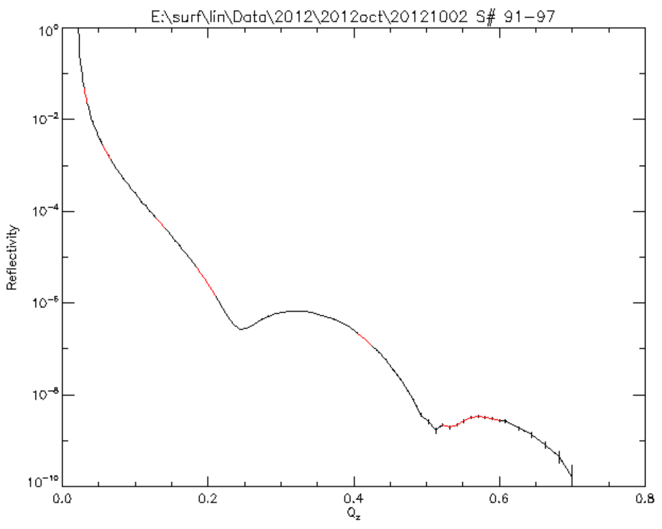
Often (more often than not) the user is interested not only in the raw reflectivity data but also in R/RF, i.e reflectivity divided by the Fresnel reflectivity predicted for the subphase being used. This is done with the aid of 3 additional keywords, two directly relevant plus one which, while not necessary, can make the task easier. The relevant keywords are:

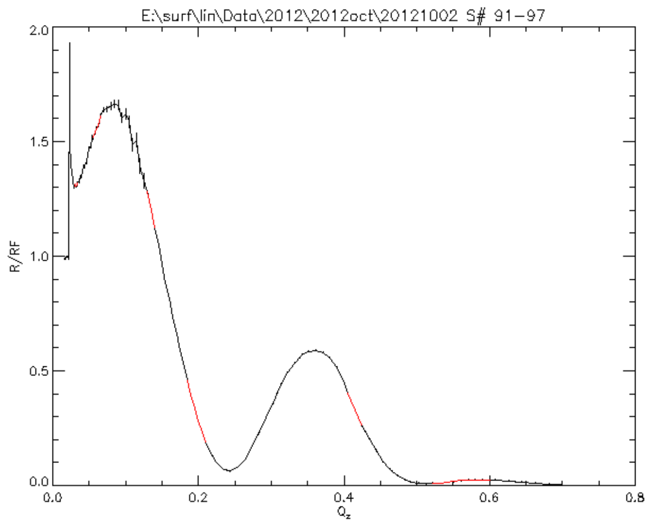
QCRIT	-	Accepts a value (from whatever source) for the critical subphase Q-value.
RFRESULT	-	Same as RESULT, for returning the R/RF data.

When a value for QCRIT is provided, PD_XR displays two plots. One is the reflectivity plot already encountered, the second is a plot of reflectivity divided by the Fresnel reflectivity, using the QCRIT value.

Since at times few different guesses for QCRIT are made, it is worthwhile to save the time involved in reevaluating the reflectivity, over and over again. This is done using the following keyword:

LAST	-	This is a switch keyword. When invoked, the last data processed is recalled from memory, instead of being reevaluated.





```
IDL> pd_xr, '91-97', slit=
[23,11], back=0.5, res=
stuff, qc = 0.022, rfres =
rstuff
```

Here, using QCRIT in the call, produced both plots shown on left. The "glitch' in the R/RF plot is a result of 0.022 being not quite the right value for QCRIT. In fact, with some experimenting it turns out that o.028 is nearly perfect.

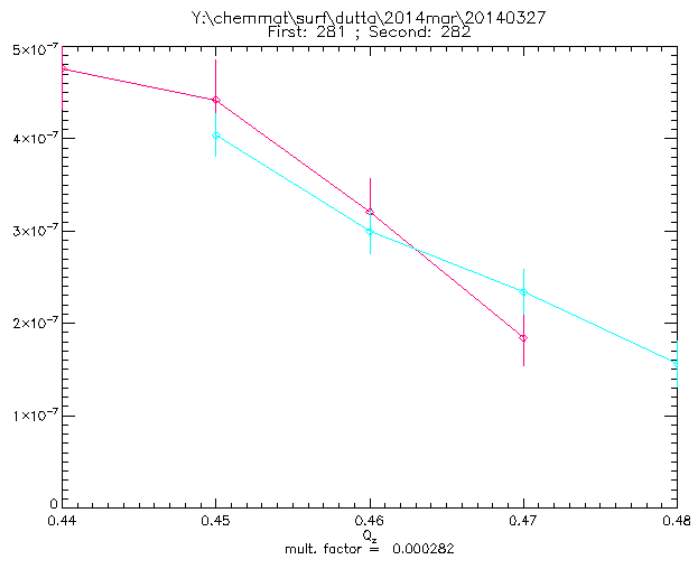
Both *stuff* and *rstuff* contain 1D data as evidenced by:

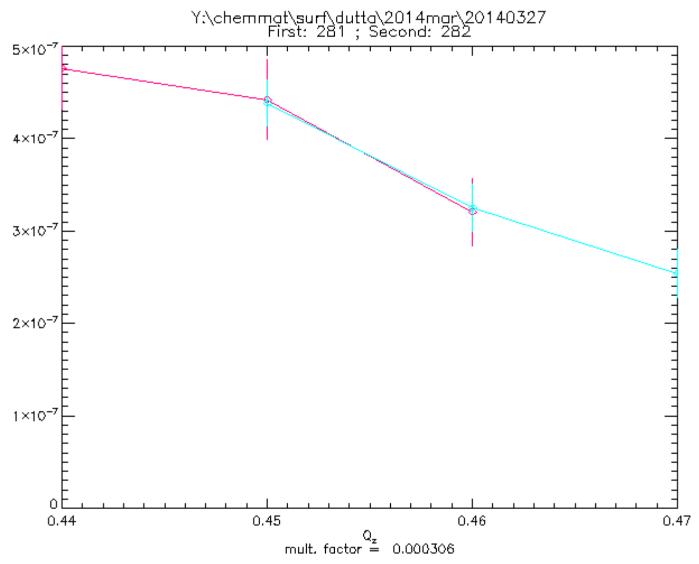
```
IDL> help, stuff
STUFF           FLOAT     =
Array[3, 111]

IDL> help, rstuff
RSTUFF          FLOAT     =
Array[3, 111]
```

Of course, both *stuff* and *rstuff* can be saved using WRITE_DATA.

In case of data mismatch in the overlap regions, the patching quality suffers. This is indicated by a jump in the size of the error bars in a plot section following an overlap region. In such case, the patching can be performed interactively, using the following switch keyword:

INTERACTIVE - If set, the routine stops during each patch action and displays a zoomed view of the overlap region. The user is then given the choice to accept the patch (typing “y”), reject it (typing “n”) or quit the interactive patching (typing “q”). If the choice is to reject the patch, the user is asked how many data points to drop. If the entered number is positive, say 2, the **first** 2 (or whatever was entered) points of the upper scan, counting from its beginning, are dropped. If the entered number is negative, say -2, then the **last** 2 (or whatever…) points of the lower scan, counting from the end, are dropped. The two scans are then re-patched and the process repeats till the user accepts the patch.





Example (from a different file)

```
pd_xr, '276-282',
sli=[23,11], bac=0.5, /int
```

the top plot on the left obtains, and IDL displays to the screen

```
OK (Y/N/Q)?
```

Poor match, so we type “n” and then IDL displays

```
Drop how many?
```

To drop one point from the data on the left (pink) we enter -1. Then, the bottom plot on left appears and the query

```
OK (Y/N/Q)?
```

is displayed again. Typing “y” accepts the choice.

Note: The above is repeated for each match in the reflectivity scan. The interactive matching can be aborted at any time by typing “q”.

Additional options for PD_XR appear in a separate chapter, dealing with this routine in detail.

2.2.1 ***Accessory reflectivity routines***

When additional information about the reflectivity scans is needed, one can use two accessory routines, PD_XR_VIEW and SCAN_PD_XR.

PD_XR_VIEW allows the user to see the 2D data, as collected on the detector, and to verify that the SLIT and BACK_ANG values (see p.8) chosen are adequate. The basic syntax is similar to PD_XR, e.g.

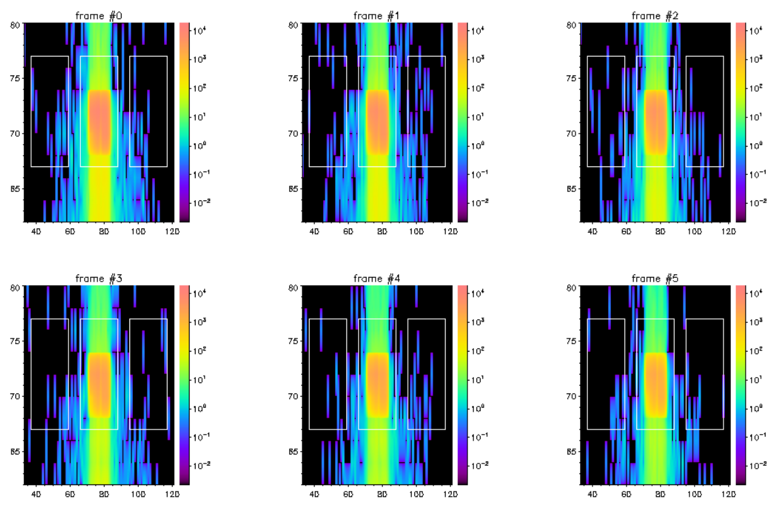
PD_XR_VIEW, SNUM, FNUM, SLIT = [*hor*,*ver*], BACK_ANG = *angle*

where:

SNUM	-	A single SPEC scan number (not a list, as before).
FNUM	-	A list of frame numbers (frames are the images corresponding to individual data points within the SPEC scan), following same rules as scan number lists (see p. 7). If this input is omitted, all the frames within the SPEC scan are displayed.
SLIT	-	Same as for PD_XR.
BACK_ANG	-	Same as for PD_XR.

Two optional keywords often used are:

LOG	-	A switch, when set, a logarithmic display scale is used.
SHAVE	-	Provided with an integer input (typically in the range of 3-9, some experimentation may be useful), "shaves" away the edges of the display, providing more room for useful information.



An example:

```
IDL> pd_xr_view, 92, '0-
5',sli=[23,11], back= 0.5,
/log, shave=7
```

Here we opted to display the first 6 frames (numbered 0-5) of the scan. The reflection peak is in the middle and the white rectangles indicate the regions of interest used to collect the signal (middle) and background (left and right).

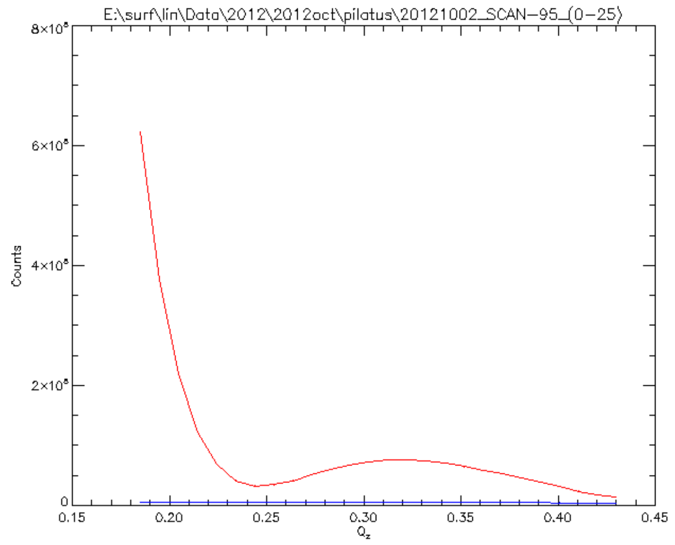
It can be seen, above, that the choice of the SLIT is proper, as it encompasses the whole reflection peak, with some room to spare. Also, the choice of the background angle is proper as the background regions of interest are close enough to the middle region but not overlapping it (in fact, the reflectivity routines won't let you set the background regions so as to overlap the center one).

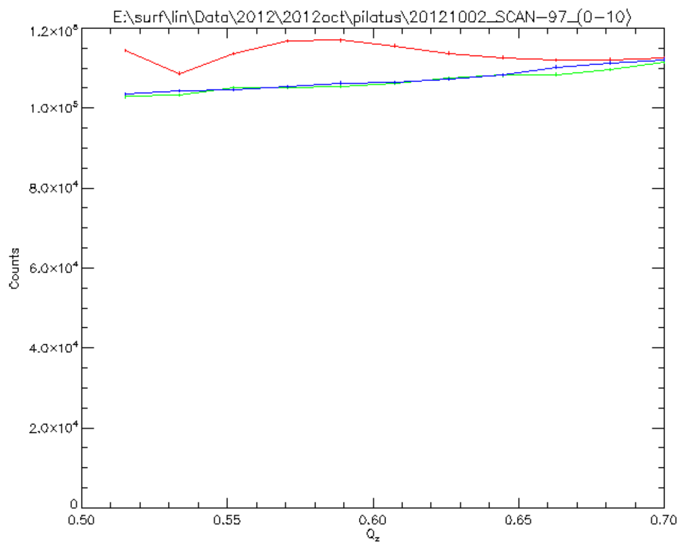
SCAN_PD_XR is used indirectly any time PD_XR is called (it is really the "engine" behind PD_XR). It can be used directly, however, to compare signal and background in the data. The basic syntax is:

dum = SCAN PD_XR (SNUM, SLIT = [*hor*,*ver*], BACK_ANG = *angle*, /SHOW)

where *dum* is an arbitrary variable name. The input SNUM is a single SPEC scan number (same as for PD_XR_VIEW) and SLIT and BACK_ANG have same meaning is for the previous routines. As for the last keyword:

SHOW - A switch, set to see a plot of the results.





Some examples:

```
IDL> dum = scan_pd_xr (95,
sli=[23,11],back=0.5,/show)
```

Here (top plot on left) we see the signal (red curve) and background (green and blue curves, merging) from scan #95, in the middle of the reflectivity scan. The signal is comfortably above background, though it is getting closer to it at the top end of the scan.

```
IDL> dum = scan_pd_xr (97,
sli=[23,11],back=0.5,/show)
```

In the bottom plot we've the signal and both background (green and blue correspond to left and right backgrounds) of scan #97, the last one of this reflectivity scan. At the high end of the range the signal merges with the background, indicating that continuing to higher Q_z values is pointless.

Note: in the (rare) case where the green and blue curves are significantly different, this indicates misalignment.

Two additional keywords, common to all the reflectivity routines, may be mentioned here. They're related to the location of the signal ROI. By default, this is the location of the "center pixel" which is stored in SPEC. In case of misalignment, though, the location may need correcting. Observing the peak location in PD_XR_VIEW will provide an indication, if the problem exists. The keywords are:

LOCATE	-	Switch, if set the center is being located and redefined for every SPEC scan.
FREE	-	Switch, if set the center is being located and redefined for every frame. More time consuming, so shouldn't be used unless absolutely necessary.

2.3 Extracting GID Data

The primary routine used to extract Grazing Incidence Diffraction (GID) data is **PD_GID**. Same as with reflectivity, a typical GID scan consists of a group of SPEC scans and it is up to the user to know which scans belong together.

SCAN_FIND (see p. 7) can be used but the user has to be aware that there are two varieties of GID scans. One is the classical two-slit GID, the other is our "Pinhole-GID". To find scans of the first variety, use "…TYPE = 'gid'…' in SCAN_FIND. For the second variety use "…TYPE = 'pgid'…". Most of our GID scans are of the second type.

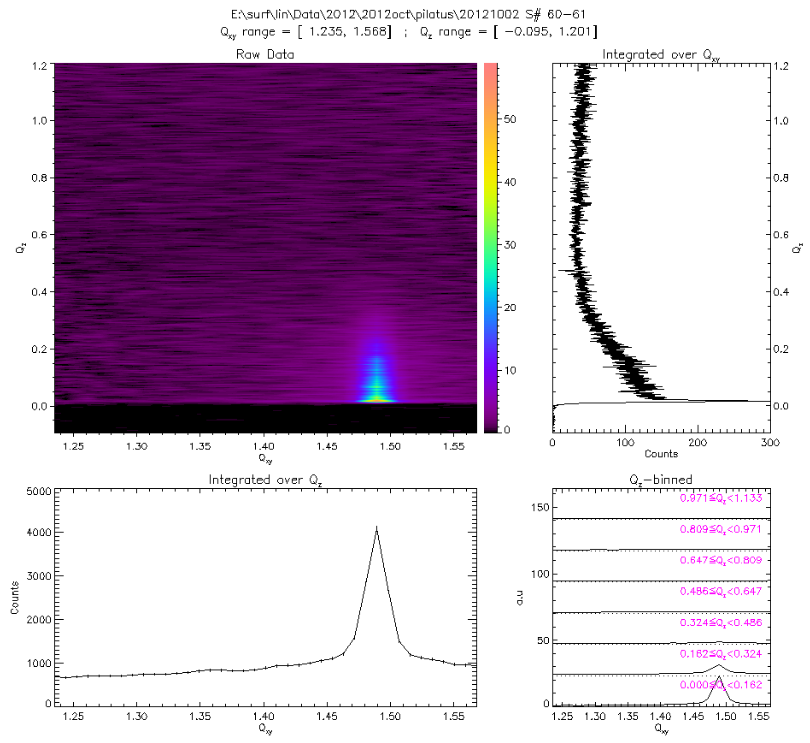
The primary routine used to extract Grazing Incidence Diffraction (GID) data is **PD_GID**. The basic syntax is

```
PD_GID, SNUM, SLIT = hsli, /BAD, [various return keywords]
```

Here:

SNUM	-	A list of numbers representing a GID scan (similar to the reflectivity case).
SLIT	-	The size of the horizontal "electronic slit", in pixels. For the PGID data type, SLIT should be used, else the data ends completely smeared. For the classical GID, where two hardware slits are used, SLIT should not be used (contributes nothing useful and may at times be detrimental.
BAD	-	This is a switch keyword which, when set, acts to eliminate "bad pixels", i.e. faulty pixels with an abnormally high count rate, which occur at times in Pilatus detector. Since the overhead for using BAD is very small, it should be always invoked, in PD_GID. In the (rare) cases when the application of BAD doesn't fully resolve the problem, a "stronger medicine" is available, in the form of BAD = *n* where *n* may be 2, 3, 4 etc (remember that /BAD is equivalent to BAD = 1).

As for the "return keywords", this'll become clearer following the example below



Example:

```
IDL> pd_gid, '60-61', sli=1,
/bad
```

The outcome of this command is the 4-part display on left. The image in the upper left corner is the full 2D GID data. The plots on top right and bottom left are the integrals of the 2D data, over Q_{xy} and Q_z, respectively. Finally, the multiple plot in the bottom right corner is also an integral over Q_z, but performed over multiple horizontal slices of the data (it is rarely used, but serves well to fill what otherwise would've been an empty corner☺)

Since there is a multiplicity of data sets, in the panel above, there is also a multiplicity of return keywords, as follows:

RESULT	-	Returns a 2D data set (see p. 5) corresponding to the image above.
XY_RESULT	-	Returns a 1D data set corresponding to the "Integrated over Q_{xy}" plot, in the top right corner.
Z_RESULT	-	Returns a 1D data set corresponding to the "Integrated over Q_z" plot, in the bottom left corner.

A typical command utilizing all these keywords would be

```
IDL> pd_gid, '60-61', sli=1, /bad, res = res, xy_res = xres, z_res = zres
```

In practice, most of the time, only Z_RESULT is of interest.

Any of these results can, in turn, be saved into file using WRITE_DATA (see p. 8). No need to specify whether the data is 1D or 2D, WRITE DATA will find it by itself. A warning, saving 2D data takes lots of space.

Some more useful keywords:

ANGLES	-	Switch, when set the data is evaluated in terms of the angles dth (horizontal) and beta (vertical), instead of the default Q_{xy} and Q_z.
LOG	-	Switch, when set, the 2D display is logarithmic.
LAST	-	Same as for PD_XR, causes last processed data to be reused
WNEW	-	Switch, opens new display window. Useful for comparing a number of data sets.