IRAM Memo 2016-2

Gridding spectra with **CLASS**

S. Bardeau[1], J. Pety[1,2]

1. IRAM (Grenoble)
2. LERMA, Observatoire de Paris

March, 1st 2018
Version 1.1

Abstract

In the sep16a **CLASS** release, the internal code of the `XY_MAP` command was refurbished to ensure that datasets larger than the available RAM memory can be processed by slicing the data in intervals of channels. The command is now also able to read direct or transposed input XY tables and/or to write direct (LMV) or transposed (VLM) output cubes. The latter feature is useful to easily reimport data when a 2nd step of baselining is desired on the gridded data cube. The online help was upgradeed accordingly (see in particular the output of `LAS> HELP XY_MAP MEMORY`).

This memo describes how the gridding is performed in **CLASS**, from the collection of spectra to the spectral cube. It focuses on the file access efficiency, memory consumption, and how the user choices can impact them.

Commands: `TABLE, XY_MAP, TRANSPOSE`
Keywords: gridding, transposition, memory, large files
Related documents: **CLASS** documentation

Contents

1 Quick overview

The gridding of spectra is done in **CLASS** with 2 commands and 3 files involved:

1. the spectra found in the input **CLASS** (.30m) file,
2. the table of spectra (.tab) created by the command `TABLE`,
3. the gridded spectra in a cube (.lmv) created by the command `XY_MAP`.

If the input table and the output cube fit both in memory at the same time, the command `XY_MAP` will read and write them at once: this is the ideal and most efficient case.

On the other hand, for larger datasets, the command splits the problem in slices of velocities (contiguous channels). With the way the input and output files can be ordered by default on the disk, reading or writing one slice may imply traversing entirely a file, and to repeat this process each time for each slice. This is known to decrease strongly the efficiency of the command `XY_MAP`.

1.1 Efficient table reading

This document shows that the default table ordering is unefficient (but there are good reasons for this ordering, see Section 2). For an efficient gridding process you should:

- give `XY_MAP` the right to allocate as much as memory as possible, *e.g.* 90% of the RAM (assuming you run a single **CLASS** session at a time), leaving typically 10% for the regular OS operations,
- if the problem still does not fit in memory, it is worth transposing the .tab table to .bat order before invoking `XY_MAP`.

This can be achieved with the commands

```
LAS> TRANSPOSE myfile.tab myfile.bat 21
LAS> SIC LOGICAL VOLATILE_MEMORY 90%
LAS> XY_MAP myfile  ! Will take myfile.bat if available
```

Note that the memory allocated during `XY_MAP` processing is freed as soon as the command terminates.

1.2 Efficient cube writing

The default cube order (.lmv) is efficient in terms of disk access, even for writing it at once or by slices. However, if you want to produce a .vlm cube (for *e.g.* reuse with `FILE IN myfile.vlm`), you should:

- give `XY_MAP` the right to allocate as much as memory as possible. If it can, the command will buffer the whole .vlm cube before writing it all at once at the end,
- if the problem still does not fit in memory, you must produce first the .lmv cube and then transpose it (`XY_MAP` does not support writing a .vlm cube by slices).

This can be achieved with the commands:

```
LAS> SIC LOGICAL VOLATILE_MEMORY 90%
LAS> XY_MAP myfile  ! Will produce myfile.lmv
LAS> TRANSPOSE myfile.lmv myfile.vlm 312
```

1.3 Efficient transpositions

If you need to perform one of the transpositions above, you have to know that the command `TRANSPOSE` faces the same problem of traversing the files by slices. For a better efficiency, you should give `TRANSPOSE` the right to allocate as much as memory as possible (the larger the available memory, the smaller the number of slices). This can be achieved with the commands:

```
LAS> SIC LOGICAL SPACE_GILDAS 4096  ! Megabytes
LAS> TRANSPOSE myfile.ab myfile.ba 21
```

The memory allocated by `TRANSPOSE` is also freed when the command terminates.

2 CLASS tables in details

CLASS can save a set of spectra with the command `TABLE`. It creates a 2D table using the Gildas Data Format (a header and rows of data). By default, the per-column format used is:

1. the X offset position,
2. the Y offset position,
3. the spectrum weight W (*e.g.* for different integration times),
4. first channel intensity

and so on for all channels.

Such a table is ordered Velocity-Position (hereafter: VP), *i.e.* X, Y, W and then the channels are contiguous in the file for each spectrum, and the spectra are concatenated one after the other. This has some advantages and disadvantages which are described in the table 1 and compared to the opposite Position-Velocity ordering (hereafter: PV).

Table 1: Advantages (+ or ++) and disadvantages (-) of VP- and PV-ordered Class tables

#	Velocity-Position		Position-Velocity	
1	++	natural ordering for Class Data Format to table conversion, *i.e.* the Class file (input) and table (output) are traversed once in parallel, the table may not fit in memory	-	would need non-contiguous access either to the Class file or to the output table in memory, the table must fit in memory
2	++	appending new spectra (`TABLE APPEND` mode) is straightforward and low-cost	-	`TABLE APPEND` is not easy, costs reading and duplicating old data
3	-	one needs to traverse the whole table to read the X Y W columns	+	X Y W arrays are contiguous, reading them has a marginal cost
4	+	the VP order is already the correct one for gridding (see section 3.1)	-	the PV order need to be transposed in an intermediate buffer before use in gridding
5	-	reading by block of velocities is unefficient, *i.e.* one needs to traverse the file as many times as the number of blocks	++	reading by velocity blocks is efficient as each block is a contiguous piece of file

From the items 1 and 2 in table 1, it is obvious that the result of the `TABLE` command must remain a Velocity-Position table. However, the item 5 tends to show that Position-Velocity tables may be interesting

for efficient[1] gridding, especially for tables which do not fit in memory and must be read by blocks.

Transposing a Velocity-Position table can be done with the command `SIC\TRANSPOSE`; the advantages and disadvantages are detailed in the table 2. The item 6 can be explained like this: both VP items 3 and 5 (table 1) and stand-alone transposition (table 2) need to traverse the input table several times. However, because the former are part of a larger problem (reading the table but also convolving, writing the cube, memory transpositions if any), they can use only a smaller amount of memory so they traverse the table more times (smaller blocks and more of them are needed) than the stand-alone transposition. In other words, and assuming the table does not fit in memory and that IO dominate the problem, VP gridding is slower than transposition + PV gridding. This is even more true if gridding is repeated several times on the same table. A demonstration of these conclusions is shown in table 3.

Table 2: Advantages (+) and disadvantages (-) of VP to PV stand-alone transposition

#		VP to PV
6	+	file-to-file transposition is a stand-alone task which can involve a maximum of the machine ressources, so that it is faster than the VP overheads (items 3 and 5) when performed within the `XY_MAP` command.
7	-	the transposed table is a duplicate of the same data, it doubles the disk usage,
8	-	the transposition must be repeated each time the original table changes or new data is appended.

Table 3: Gridding of a table of 61263 spectra x 20737 channels (4.8 GB). Each command was allowed to use only 2048 MB. The output cube (1.7 GB with the command defaults) is ordered LMV so that it is traversed only once for writing. The system cache was emptied before each run.

Action	Traversing table	Elapsed time
`XY_MAP` (VP)	1 (XYW) + 7 blocks	8.2 min
`TRANSPOSE` (VP to PV)	4 (in) + 1 (out)	5.0 min
`XY_MAP` (PV)	1	1.9 min

From these conclusions, there is no *unique* answer to what kind of table should be feed in the gridding engine. The answer is a mixture of efficiency when creating or appending the table, and reading it. However, if the whole table fits in memory, *i.e.* it can be traversed once, then it is probably better to use a VP table. Then, if it must be read by parts, the PV order should be prefered.

3 Gridding with `XY_MAP`

3.1 Gridding engine

The gridding engine in **CLASS** (command `XY_MAP`) is efficient partly because a single convolution kernel for a given location can be computed and used for all planes. This implies the innermost loop to be on channels (Velocity) and outer loops on X and Y (Position). In other word, the convolution engine works with Velocity-Position arrays in memory (hence item 4 in table 1).

[1] with nowadays technologies, *i.e.* hard-drives IO dominate other issues like memory copies or CPU computing.

3.2 Output cube in details

The gridding engine can write the resulting cube in either Position-Position-Velocity order (hereafter LMV) or Velocity-Position-Position (hereafter VLM). The default is LMV. The table 4 describes both possibilities.

Table 4: Advantages (+) and disadvantages (-) of LMV and VLM-ordered Class cubes

#		Position-Position-Velocity (LMV)		Velocity-Position-Position (VLM)
9	-	need transposing the convolved data before writing to the file	+	correctly ordered, convolved data can be written *as is*
10	+	the cube can easily be extended by block of velocities, in particular it may not fit in memory	-	the cube can not (reasonably) be extended on disk. If it fits in memory, it can be written in a temporary output buffer, with more memory cost.
11	+	default ordering in Gildas, efficient for the imaging tools (one image plane is contiguous in the file)	-	not the default in Gildas, many tools do not expect such order
12	-	need transposing for an efficient per spectrum analysis	+	the order is correct for spectrum-oriented tools like Class or Cube.

4 Implementation

4.1 Memory management

Several data buffers are needed in this problem. In particular, transposition buffers are needed to go to/from the internal gridding engine which works on velocity-first arrays. The buffers are:

1. the table data buffer (size $M_{td} = N_{posi} \times N_c$),
2. the table transposition/sorting buffer (size $M_{tt} = N_{posi} \times N_c$) if (and only if) the transposition of the table (.bat) and/or sorting (positions not sorted by Y offsets) are needed. The same buffer is used for both tasks.
3. the cube data buffer (size $M_{cd} = N_x \times N_y \times N_c$),
4. the cube transposition/sorting buffer (size $M_{ct} = N_x \times N_y \times N_c$) if and only transposition of the cube is needed (.lmv).

We call $M_T = M_{td} + M_{tt} + M_{cd} + M_{ct}$ the total amount of memory allocated to the buffers. Note that they all scale as N_c. Some other allocatable buffers are involved (*e.g.* X, Y, W columns read from the table) but they are neglictible in front of the data size, and they do not scale as N_c.

The command `XY_MAP` needs memory space to allocate these buffers. However, it does not allocate freely the memory: it ensures they fit in a limited memory space (`VOLATILE_MEMORY`, we do not consider here the way it is defined).

If $M_T <$ `VOLATILE_MEMORY`, the 4 buffers above fit in the dedicated memory space. We are in the best case where all the problem fits in memory. In particular, the output cube fits in memory and will be

written at once, whatever its order.

If $M_T >$ `VOLATILE_MEMORY`, the 4 buffers do not fit in the dedicated memory space. The problem will be iterated by blocks of channels along its velocity dimension. There will be n iterations, the buffers being divided by the same and correct amount. But, in this case, there are 2 possibilities:

- if the output cube is velocity-last (.lmv), it will be gdf-extended[2] by block directly on the disk. In other words it does not need to fit entirely in memory, only the current subset is loaded at a time. The minimum number of subdivisions needed to remain below `VOLATILE_MEMORY` is (floating point division):
$$n = (M_{td} + M_{tt} + M_{cd} + M_{ct})/\texttt{VOLATILE\_MEMORY} \quad (1)$$
- if the output cube is velocity-first (.vlm), it must fit entirely in memory. The libgio does not offer to write non-contiguous pieces of a cube (and we probably do not want this). This means we have to find a balance for the cube data buffer (scaled as N_c) and the 3 other ones (scaled as N_c/n). In this case, `XY_MAP` accepts to allocate up to 50% of `VOLATILE_MEMORY` for the whole data cube:
$$M_{cd} < 0.5 \times \texttt{VOLATILE\_MEMORY} \quad (2)$$
If the output cube needs more than 50%, `XY_MAP` stops with an error. Allocating more with leave too few space for the other buffers, resulting in much increased number of iterations, traversing the table more times, etc, leading to reduced efficiency. The recommendation is then to use a lmv cube and to transpose it afterwards for a better efficiency. The remaining part of the memory is then divided as usual between the other buffers to same and correct amount:
$$n = (M_{td} + M_{tt} + M_{ct})/(\texttt{VOLATILE\_MEMORY} - M_{cd}) \quad (3)$$

At this stage, n is the minimum (floating point) number of divisions needed. The number of channels processed per division is then:
$$n_c = \text{floor}(N_c/n) \quad (4)$$
Rounding to an integer number of channels is obviously needed as the internal engine can not process fraction of channels. Down-rounding is used because up-rounding could mean using more than `VOLATILE_MEMORY` per division. Because of the integer rounding, the actual number of divisions which will be performed is finally:
$$n = \text{ceiling}(N_c/n_c) \quad (5)$$
Note that the last division may process less channels than the other ones, because the total number of channels N_c may not be a multiple of n_c.

4.2 In practice

One can check how the memory management described above applies to his dataset by enabling the debugging messages before invoking the command `XY_MAP`. In practice, reusing the same dataset as in the Table 3, this gives:

```
sic message class s+d             ! Enable Debug messages
sic logical VOLATILE_MEMORY 6GiB  ! Custom memory buffer
xy_map myfile.tab
sic message class s-d             ! Disable Debug messages
```

which shows (some outputs have been hidden for clarity):

[2]This means a first block of the cube is written as a whole, with a consistent header, then each new block extends the last dimension of the data, always with a consistent header.

```
I-XY_MAP,  Reading columns X Y W
[...]
Table size: 61263 positions x 20740 values
[...]
Cube size: 148 by 142 pixels x 20737 channels
[...]
I-XY_MAP,  Creating file: myfile.lmv
I-XY_MAP,  Creating file: myfile.wei
D-XY_MAP,  Using at most VOLATILE_MEMORY = 6.00 GB
D-XY_MAP,  Mblocks: 3, MC: 9787, Size: 29361 20737
D-XY_MAP,  --------------------
D-XY_MAP,  Doing block 1/3, channels 1:9787, Nchan = 9787
D-XY_MAP>REALLOCATE>2D,  Allocated workspace of size: 9787 x 61263 = 2.23 GB
D-XY_MAP,  Reading table...
D-XY_MAP,  Sorting table...
D-XY_MAP>REALLOCATE>2D,  Allocated workspace of size: 9787 x 61263 = 2.23 GB
D-XY_MAP>REALLOCATE>3D,  Allocated workspace of size: 9787 x 148 x 142 = 784.6 MB
D-XY_MAP,  Convolving
D-XY_MAP,  Transposing to LMV...
D-XY_MAP>REALLOCATE>3D,  Allocated workspace of size: 148 x 142 x 9787 = 784.6 MB
D-XY_MAP,  Writing LMV cube...
D-XY_MAP,  --------------------
D-XY_MAP,  Doing block 2/3, channels 9788:19574, Nchan = 9787
D-XY_MAP>REALLOCATE>2D,  Workspace already allocated at an appropriate size: 9787 x 61263 = 2.23 GB
D-XY_MAP,  Reading table...
D-XY_MAP,  Sorting table...
D-XY_MAP>REALLOCATE>2D,  Workspace already allocated at an appropriate size: 9787 x 61263 = 2.23 GB
D-XY_MAP>REALLOCATE>3D,  Workspace already allocated at an appropriate size: 9787 x 148 x 142 = 784.6 MB
D-XY_MAP,  Convolving
D-XY_MAP,  Transposing to LMV...
D-XY_MAP>REALLOCATE>3D,  Workspace already allocated at an appropriate size: 148 x 142 x 9787 = 784.6 MB
D-XY_MAP,  Extending LMV cube...
D-XY_MAP,  --------------------
D-XY_MAP,  Doing block 3/3, channels 19575:20737, Nchan = 1163
D-XY_MAP>REALLOCATE>2D,  Allocated workspace of size: 1163 x 61263 = 271.8 MB
D-XY_MAP,  Reading table...
D-XY_MAP,  Sorting table...
D-XY_MAP>REALLOCATE>2D,  Allocated workspace of size: 1163 x 61263 = 271.8 MB
D-XY_MAP>REALLOCATE>3D,  Allocated workspace of size: 1163 x 148 x 142 = 93.2 MB
D-XY_MAP,  Convolving
D-XY_MAP,  Transposing to LMV...
D-XY_MAP>REALLOCATE>3D,  Allocated workspace of size: 148 x 142 x 1163 = 93.2 MB
D-XY_MAP,  Extending LMV cube...
D-XY_MAP,  --------------------
D-XY_MAP,  Writing weight image...

I-XY_MAP,  Time elapsed in command (total):           54.53 sec
I-XY_MAP,  Time elapsed reading XYW:                  36.94 sec
I-XY_MAP,  Time elapsed reading the table:             5.26 sec
I-XY_MAP,  Time elapsed sorting the table:             2.01 sec
I-XY_MAP,  Time elapsed transposing table or cube:     1.80 sec
I-XY_MAP,  Time elapsed convolving:                    5.63 sec
I-XY_MAP,  Time elapsed writing the cube:              2.80 sec
```

We find here some interesting informations. The input table provides $N_{posi} = 61263$ positions times $N_c = 20737$ channels (+ 3 values for X,Y,W). The output cube has $N_x = 148$ times $N_y = 142$ pixels per plane. Using `VOLATILE_MEMORY` = 6 GiB, `XY_MAP` chooses to split the processing in $n = 3$ blocks, using

$n_c = 9787$ channels per block (1163 in the last one).

In each block, we can see the allocation of buffers for each task, namely M_{td}, M_{tt}, M_{cd}, and M_{ct}. The total is 5.99 GB, which fits exactly in the `VOLATILE_MEMORY` limit. We see that because the .tab is ordered VP, the table does not need to be transposed. Note also that the first slice of the cube is written at the end of the first block. The others slices are then written by extending the cube.

Finally, the command shows a summary of the processing time per task[3]. The key point here is that reading the X,Y,W columns takes more than 50% of the total processing time. Again, this is because the table is ordered VP, so the columns are not contiguous in the file (spread all other the file), which must be traversed entirely for such a small information in comparison of the 20737 channels. It would have been almost instantaneous with a PV table. Note also that reading the table data in only 5.26 seconds here, because the system has saved the file in the system cache when we read the X,Y,W columns. This is possible because this example is small and fits in memory. For larger files (greater than system memory), reading (a part of) the table would cost again and again the same time in each block as it took reading X,Y,W. Finally, the main goal of the command which is to add and convolve the data at each pixels takes only about 10% of the overall processing time.

[3]They are faster here than in the Table 3 because the command was executed on a faster machine and with a larger memory buffer.