



Public

Hello [Mark RKidger](#)

[Log Out](#)

– [Create personal sidebar](#)

- Public Web**
- [Create New Topic](#)
- [Index](#)
- [Search](#)
- [Changes](#)
- [Notifications](#)
- [Statistics](#)
- [Preferences](#)

Webs

- [HCalSG](#)
- [HSC](#)
- [Hcss](#)
- [Hifi](#)
- [MOC](#)
- [Main](#)
- [Pacs](#)
- [Public](#)
- [Sandbox](#)
- [Spire](#)
- [TWiki](#)

[TWiki](#) > [Public Web](#) > [PacsDocumentation](#) > [SecondLevelDeglitching](#) (2013-01-14, MichaelWetzstein)

[Edit](#) [Attach](#)

Tags: [+ create new tag view all tags](#)

How to do Second Level Deglitching

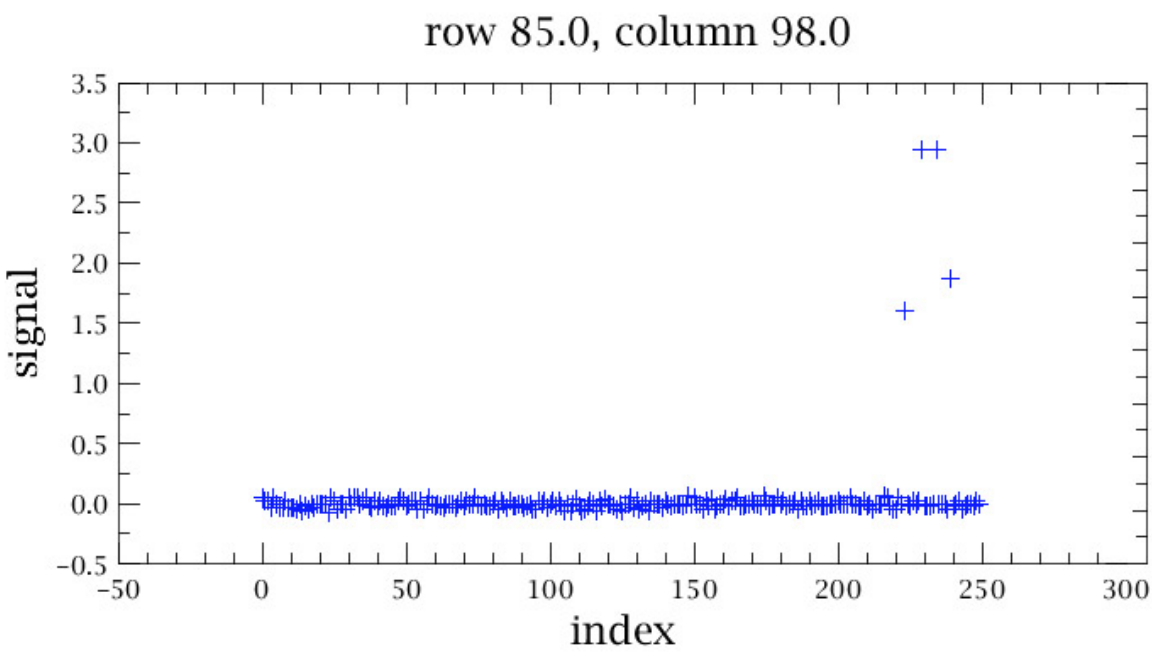
Contents

- [What is Second Level Deglitching?](#)
- [Prerequisites](#)
- [Command line syntax](#)
- [The most important syntax options](#)
- [A detailed look at the options of the MapIndex task](#)
- [A detailed look at the options of the deglitching task](#)
- [Avoid deglitching strong gradients](#)
- [Memory saving options explained](#)
 - [What is a MapIndex?](#)
 - [The size of the MapIndex](#)
 - [What can you do with the slim MapIndex: working with little memory](#)
 - [Iteratively deglitch large observations](#)
 - [Deglitching without MapIndex](#)
- [Diagnostics using the MapIndexViewer](#)
- [Optimizing the Sigclip algorithm with the MapIndexViewer for best deglitching results](#)
- [How to write your own Sigclip algorithm in jython](#)
- [Command line options for the MapIndex](#)

What is Second Level Deglitching?

Second level deglitching is a glitch removal technique that works on the final map of PACS photometer data. Instead of co-adding all signal contributions to a map pixel from the detector, second level deglitching collects the signal contributions in a vector per map pixel. Without glitches, all the signal contributions to a map pixel should be roughly the same, because their origin is the same location at the sky. Glitches will introduce significantly outstanding values and can thus be detected by sigma-clipping.

- Vector of roughly 250 signal contributions to map pixel (85, 98):



The actual deglitching process takes place in two steps:

1. Calculate the data contributions to each map pixel and store them in a vector (=data array) for every map pixel. This data is collected in a product called MapIndex (to be explained later).
2. Use the MapIndex product to loop over the data vectors and apply sigma clipping to each of the vectors. This is done by the task `lndLevelDeglitchingTask`

Prerequisites

To apply second level deglitching you need a Frames product that contains PACS photometer data. The Frames product should be processed up the stage where the data can be mapped (i.e. the end of Level 1). For a proper deglitching it is especially important that flux calibration has been done with the `PhotRespFlatfieldCorrectionTask`.

Command line syntax

Here is the most basic application of second level deglitching.

The syntax is the jython command line syntax. Lines starting with `"#"` are comments and usually don't have to be executed. During the startup of HIPE, most of the import commands are carried out automatically. Usually, you don't have to do the imports and you don't have to construct the tasks (but if you do, it does not harm).

However, for the full picture, these imports are:

```
# import the software classes
from herschel.pacs.signal import MapIndex
```

```
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask

Then you continue with:
# construct the tasks
mapIndex = MapIndexTask()
IIndLevelDeglitch = IIndLevelDeglitchTask()

# from the pipeline or previous processing get your
# Frames product ("frames" here). This is the input to the mapindex
# task. The output of the task is the MapIndex product ("mi")
mi = mapIndex(frames)

#the deglitching is the second step of the processing
map = IIndLevelDeglitch(mi, frames)
```

The most important syntax options

Especially the second step (`map = IIndLevelDeglitch(mi, frames)`) can be fine-tuned to optimise the deglitching process.

The most significant values to specify are these:

1. Do you want to produce a map as output or only flag the glitches and write the flags in form of a mask back into the Frames product?.
The options to do this are
`map = True` or `False`, `mask = True` or `False`
By default, both options are `True`.
2. You may customise the deglitching parameters by specifying a Sigclip algorithm and using it as an input for the deglitching task.

```
s = Sigclip(nsigma = 3) # define a new Sigclip algorithm
                        #which uses a 3 sigma threshold
s.setOutliers("both")  # detect both positive and negative glitches.
                        #Alternatives are "positive" (default for positive glitches only) and "negative"
s.setBehavior("clip")  # apply the clip to the data vector in one go (= don't use as box filter).
                        #if you prefer a filter, use s.setBehavior("filter") and s.setEnv(10).
                        #10 means, the boxsize will be 2*10+1 = 21
                        #please find more details in the Sigclip documentation
s.setMode(Sigclip.MEDIAN) #alternative is Sigclip.MEAN this defines the algorithm to detect outliers:
                        #either median and median absolute deviation or mean and standard deviation.
```

And now apply the new configured Sigclip algorithm to the deglitching:

```
map = deg(mi, frames, algo = s, map = True, mask = False)
```

An interesting option is also `algo = None`. This does not apply any deglitching, rather it only co-adds the data in the vector. This way, it creates an undeglitched map from the MapIndex. This is a rather fast algorithm. So if you already have a MapIndex and just want to map, using the `algo=None` option is faster than PhotProject.

If you don't specify the Sigclip alorithm, the default for second level deglitching is used, which is a clip with `nsigma = 3`, median algorithm and

both outliers (more or lesss what we have specified above).

You may test your Sigclip parameters interactively with the MapIndexViewer. So you don't have to guess the best Sigclip parameters. Please read the description of the MapIndexViewer further down to learn how this is done.

A detailed look at the options of the MapIndex task

The layout of the MapIndex determines the layout of the map itself. This is why the MapIndexTask uses many of the option that are also used by the PhotProjectTask (PhotProject is the task for a simple projection).

The most important parameters of MapIndex and how to use them:

1. inframes: the input Frames product.
2. outputPixelsize: this is the desired size of the map pixel in arcseconds (a square geometry is assumed). By default it is the same size as the Frames data array (3.2 arcsecs for the blue photometer and 6.4 for the red).
3. pixfrac: this is the fraction of the input pixel size. If you shrink the input pixel, you apply a kind of drizzle. pixfrac should be between 0 (non inclusive) and 1. 0.5 means that the pixel area is reduce to $0.5 \times 0.5 = 0.25$ of the original area, ie 1/4th of the unmodified pixelsize.
4. optimizeOrientation: set this value to True if you want the map rotated for an optimised fit of the data and thus smallest mapsize. After the rotation north may not be directed upwards any more. By default this parameter is set to False.
5. wcs: for a customised World Coordinate System. By default the wcs is constructed for a complete fit of the data.

If you want a special wcs, for example if you want to fit the map into another map afterwards, you may specify your own wcs. A good starting point is the [Wcs4MapTask](#), which creates the default wcs for the MapIndexTask (and also PhotProject). You can take the default wcs and modify it according to your needs:

```
from herschel.pacs.spg.phot import Wcs4mapTask
wcs4map = Wcs4mapTask()
wcs = wcs4map(frames, optimizeOrientation = False)

#now we have the default world coordinate system (wcs)
wcs.setCrota2(45) #rotate the map by 45 degees
wcs.setNaxis1(400)
wcs.setNaxis2(400) #force the map to 400X400 pixels. The data may not fit anymore
wcs.setCrpix1( 200 ) #make sure, the center of your data remains in the center of your map
wcs.setCrpix2( 200 )
```

Find the full set of options in the documentation of the Wcs ([herschel.ia.dataset.image.wcs.Wcs](#)).

6. slimindex: this is a memory saving option. Please read details in the section "Memory saving options explained". In a nutshell, slimindex stores the least possible amount of data in the MapIndex product. This means that as you create a map, some values have to be recalculated on-the-fly during the deglitching task. This costs processing time.

As a rule of thumb: use slimindex = True if you want to create only a glitchmask (map = False, mask = True in the deglitching). For this, the slim-MapIndex contains all necessary data.

On the other hand: If you want to create a map, use slimindex = False. This will enlarge the size of the MapIndex product from a slim-MapIndex to a full-MapIndex (which contains all bells and whistles), but it will deglitch and map fast.

If you don't have much memory , you may safely use slimindex = True and also create a map. Bring some more processing time!

The default value is `slimindex = True`.

The full call of the `MapIndexTask` with the described options looks like this:

```
mi = mapIndex(frames, optimizeOrientation = False, wcs = mywcs, slimindex = False)
```

A detailed look at the options of the deglitching task

The deglitching applies sigma clipping to the data that is stored in the `MapIndex`. The options for `lndLevelDeglitch` are:

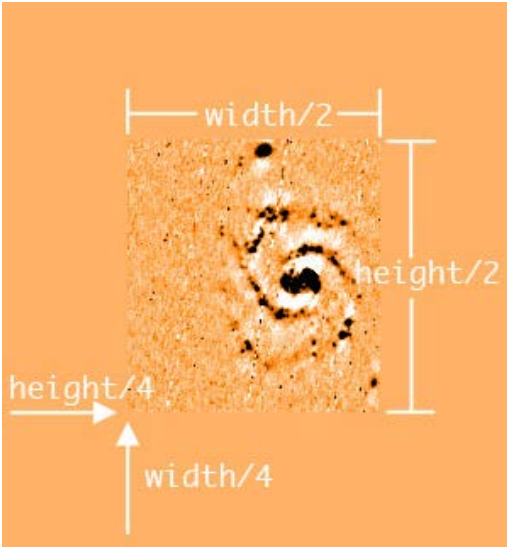
1. `index`: the `MapIndex` product that stores the map data.
2. `inframes`: the `Frames` product. Since the `MapIndex` only contains the references to the science data (and not the actual data themselves), the `Frames` product is needed as the container of the values that the `MapIndex` points to. If you specify in the task call to flag the glitches, the corresponding glitch mask is written back into the `Frames` product.
3. `map`: if you want a map as output of the deglitching task, use the default value `"True"`. Otherwise set this option to `False`.
4. `mask`: if you want a glitch mask as output of the deglitching task, use the default value `"True"`. otherwise use `False`. Otherwise set this option to `False`.
5. `maskname`: you may customise the name of the glitch mask, that is written into the `Frames` product. Default is `"2nd level glitchmask"`.
6. `submap`: specifies a rectangular subsection of the map and deglitching will only be done for that subsection. This implies that you know already what your map looks like. The values of `submap` are specified in units of map pixels.

For example:

```
# retrieve the size of the map from the MapIndex
width = mapindex.getWidth()
height = mapindex.getHeight()

# specify the first quarter of the map as [bottom left row index,
# bottom left column index, height (= number of rows), width (=
# number of columns)]
submap = Int1d([height/4, width/4, height/2, width/2])
```

Deglitching performed with the `submap` option:



Focusing the deglitching on a submap will accelerate the deglitching process. It can be useful to optimise the deglitching parameters on a submap first, before a long observation is completely processed.

- 7. threshold: a threshold value that is used in combination with the following parameter sourcemask. Default value is 0.5.
- 8. sourcemask: defines a submap that can have any shape. The sourcemask is a [SimpleImage](#) with the same dimensions as the final map.

The values of the sourcemask are compared to the threshold parameter and are treated as:

- value > threshold: this location is masked and will not be processed by the deglitching algorithm
- value < threshold: the deglitching algorithm will treat this map location

The sourcemask can be used to exclude bright sources from the deglitching process. A check on how the deglitching task has used the sourcemask is to use the output parameter "outmask". It returns the translation of the sourcemask as a 2d boolean array.

You can extract this array with

```
boolean_sourcemask = IIndLevelDeglitch.getValue("outmask")
#watch it with the Display tool
#convert the boolean values to 0 and 1 by putting them into a Int2d
d =Display(Int2d(boolean_sourcemask))
```

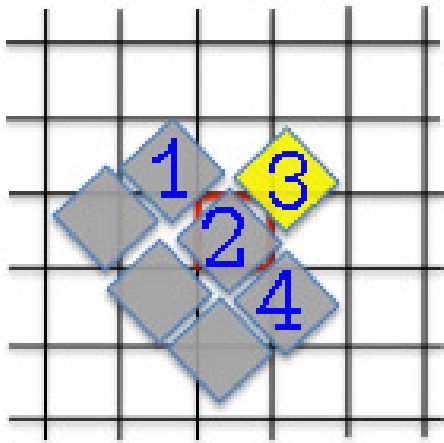
- 9. deglitchvector: allows one to treat strong flux gradients. Please read details in the section "Avoid deglitching strong gradients".
- 10. algo: a customized Sigclip algorithm. See [The most important syntax options](#) for more details.
- 11. weightedsignal: as for PhotProject, this value weights the signal contributions with the signal error (stored in the Frames product) when it co-adds the values to get a flux.
Default value is False.

Avoid deglitching strong gradients

If a small part of a source falls into one pixel, the default deglitching scheme may lead to wrong results. Look at the situation drawn in the following image: The grid in this image is the map you are projecting the Frames (detector) pixels, indicated with filled squares, onto. The yellow Frames pixel contains a small but bright source. Only a very small part of it maps onto the red-framed map

pixel (the fractional overlap that we call "poids" = weight is small). All other Frames pixels in the vicinity have a small signal value (indicated as gray).

- Mapping a small but strong source:



For the deglitching of the red-framed map pixel, the deglitchvector will be just the list of the signals s_n that fall into the pixel:

deglitchvector =[s1, s2, s3, s4,(all other timeindices)]

It is clear, that s3 dominates the list and will very likely be found as a glitch, and hence be sigma clipped. This is because by default the full signal values of the contributing Frames pixels are used for deglitching.

We can improve, if we do not take the plain signals, but weight them with the fraction of the overlap into the map pixel. We then add all contributions of one timeindex to a single value like this:

deglitchvector =[s1*p1/a1 + s2*p2/a2 + s3*p3/a3 + s4 *p4/a4 ,(all other timeindices)]

where s_n are the plain signal values, p_n are the overlaps and a_n the size of the detector pixels.

Because a glitch most likely appears only in one timeframe (i.e. within the Frames pixels, the glitches have sharp time profiles), the weighted contributions of the other time indices will be smaller and the glitch can be found. A real signal on the other hand, will be present in many time indices and thus not stand out of the weighted deglitchvector as much as a glitch.

The downside of this process is that along with the found glitch you throw away all signal contributions to that map pixel coming from a time frame. Because of the co-addition of the weighted signal, the real location of the glitch cannot be confined with more precision.

Use the weighted glitch with the option deglitchvector = "timeordered"

```
map = IIndLevelDeglitch(mi, frames, deglitchvector = "timeordered")
```

Memory saving options explained

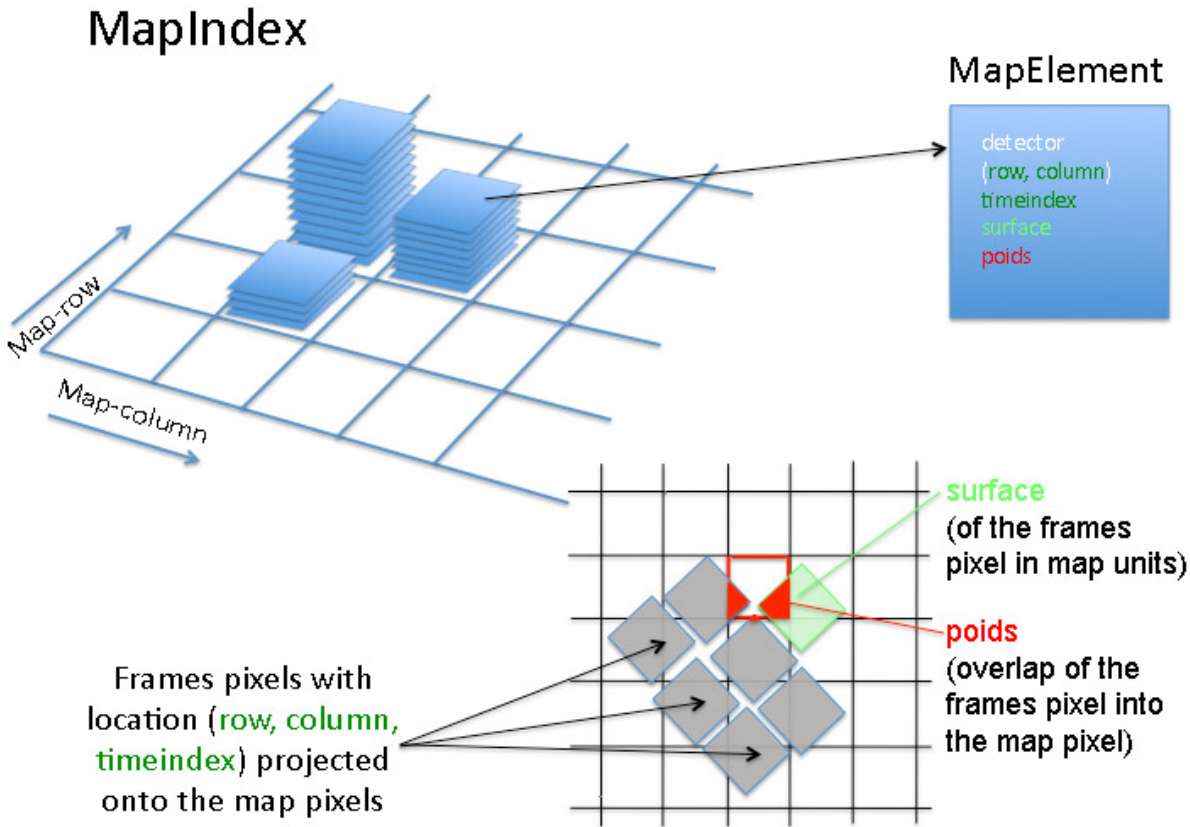
As soon as you execute the MapIndexTask, you will realise that it needs a lot of memory. There are reasons for this and it helps to understand the concept of the MapIndex in order to find the most efficient way to process your data.

What is a MapIndex?

The MapIndex is a 3-dimensional data array that has the same width and height as the resulting map. The third dimension contains references to the data in the Frames product, so that every flux contribution to a map pixel can be retrieved from the Frames product. In other words, the 3rd dimension contains information about where the data of each map pixel came from.

This third dimension is non-rectangular, because the number of detector pixel contributions differs from map pixel to map pixel. If you want to retrieve the data from the MapIndex, you get it returned as an array of MapElements. Please have a look at the image and see, what kind of data is stored in the MapIndex.

- MapIndex and MapElements:



The size of the MapIndex

If we want to store the full amount of data needed to project the flux values from the Frames product to the map, this is what we need for every flux contribution to a map pixel:

- the row, column and time index of the data in the Frames product
- the relative overlapping area of the Frames pixel that falls onto the map pixel (= the poids or weight value)
- the size of the Frames pixel in units of the map pixel size

Since this is more than one value, the set of information is put into a MapElement as a container. The MapElement is what you get from the MapIndex when you want to extract this information.

This full set of information is contained in a MapIndex product called the FullMapIndex. The FullMapIndex is one of the two flavours the MapIndex can have. You get it when you use the non-default option slimindex = False on the MapIndexTask.

Now, what is the size?

Assume a Frames product with 30000 time indices and a blue array with 2048 detector pixel. Assume further that every Frames pixel overlaps 9 map pixels when the flux is projected. We get a MapIndex size of nearly 12 gigabyte storing this information. Tweaks are obviously necessary! Besides compressing the data while it is stored, the most significant tweak is the slim MapIndex (the second flavour of the MapIndex. The java class s called SlimMapIndex). This contains a reduced set of information, namely only the row, column (encoded in one value: the detector number) and time information of the data. While the FullMapIndex uses 12 gigabyte per hour of observation, the SlimMapIndex needs only 2 gigabytes.

What can you do with the slim MapIndex: working with little memory

It is possible to perform the second level deglitching without creating a map. For the default deglitching, without the **timeordered** option, no information about pixel sizes or overlaps are necessary because only the values from the Frames product without weighting are used. In this way a glitch mask can be created.

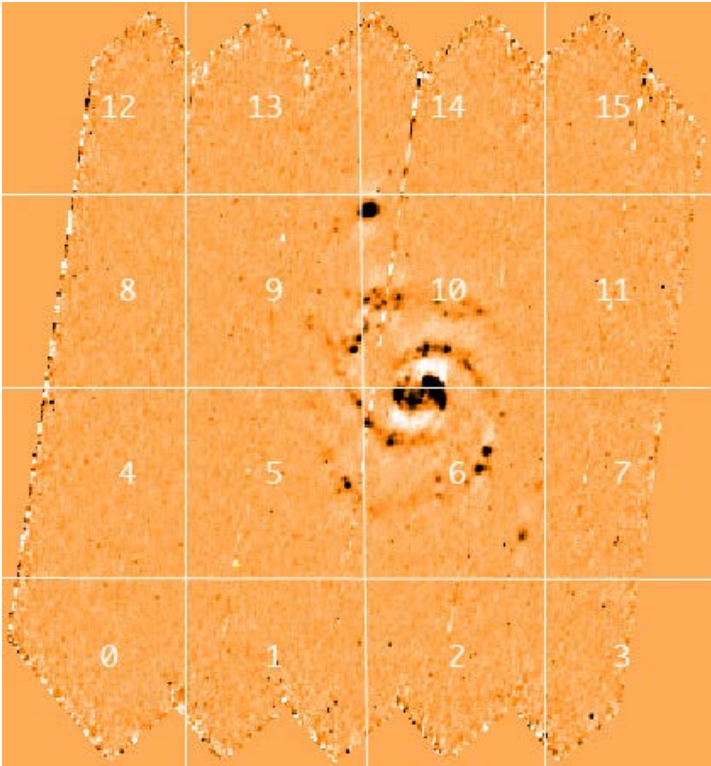
Although the second level deglitching task can also create a map out of the slim MapIndex, the necessary information about pixel sizes and overlaps have to be recalculated on-the-fly. This is inefficient. So, with the slimIndex the best mapping strategy is

- deglitch and create only the glitch mask and no map
- use the regular projection with PhotProject to create a map. PhotProject will take into account the glitch mask

Iteratively deglitch large observations

Another way to handle large observations is to divide the map into tiles and process one tile one after the other. The MapIndexTask supports this loop-based processing. To define the tiles, you overlay a chessboard like pattern over your final map and tell the MapIndexTask, how many rows and how many columns this pattern should have. Then, according to a numbering scheme, you may also tell the MapIndexTask, which of those tiles should be processed - default is all tiles. Have a look at the following image to understand the numbering scheme.

- The numbering scheme of the tiles for iterative deglitching.
This map is sliced into 4 rows and 4 columns. This results in 16 tiles.:



If you want to initiate a tile based processing, you have to know about four additional parameters. The first three are MapIndexTask parameters:

- no_slicerows: the number of rows for the chessboard pattern (default:1)
- no_slicecols: the number of columns for the chessboard pattern (default:1)
- slices: the numbers of the slices that you want to be processed. By default, all slices (= no_slicerows*no_slicecols) will be processed. But you can command to process only a subset.
- partialmap: this is a parameter, that has to be passed to the IIndLevelDeglitchTask. The value for this parameter is always None! In the first loop, None tells the IIndLevelDeglitchTask that it has to construct a new image with the correct size for your final map. In the following loops, the partialmap indicates to add the new data of the loop to the existing map, instead of creating a new one. See in the example below, how you achieve this.

Here is the necessary jython code (most lines are comments!):

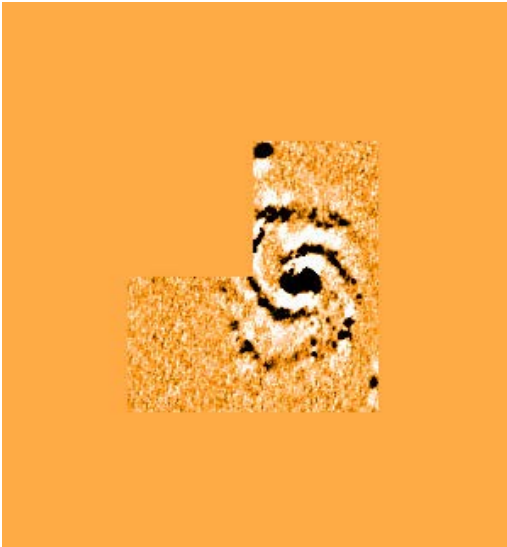
```
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mapindex = MapIndexTask()
deg = IIndLevelDeglitchTask()
img = None #img will contain your final map. This is important: The first image MUST be None!
d = Display()
counter = 0 #only if you want to save the MapIndices

for mi in mapindex(inframes = frames, slimindex = False, no_slicerows = 4, no_slicecols = 4, slices = Int1d([5, 6, 10])):
    #save the mapindex here, if you want
```

```
#name = "".join(["mapindex_slice_",String.valueOf(counter),".fits"])
#fa.save(name, mi)
#counter = counter+1
#
#note the parameter "partialmap = img" in the deglitch task
img = deg(mi, frames, algo = None, map = True, mask = False, partialmap = img)
#after the deg-step, the new data has been added to img
del(mi) #free your memory before the next loop is carried out
d.setImage(img) #this allows to monitor the progress
```

At the end of this loop, the variable img contains your deglitched map.

- Deglitching slices 5,6 and 10:



If you don't want a map but only a mask and also don't care for a progress display, the code simplifies even more. In this example, we also show how to apply a customized Sigclip algorithm:

```
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mapindex = MapIndexTask()
deg = IIndLevelDeglitchTask()
s = Sigclip(10, 4)
s.mode = Sigclip.MEDIAN
s.behavior = Sigclip.CLIP

for mi in mapindex(inframes = frames, slimindex = False, no_slicerows = 4, no_slicecols = 4):
    #for the mask only version, partialmap can be omitted
    deg(mi, frames, algo = s, map = False, mask = True)
    del(mi)
```

Deglitching without MapIndex

One of our latest achievements is the MapDeglitchTask. It virtually does the same as the lndLevelDeglitchTask, but it does not need a MapIndex as input. This way, it runs with much less memory usage. On the other hand, the time it needs to finish is as long as creating a MapIndex and then lndLevelDeglitch (plus a bit).

Internally, it implements a search algorithm, that collects the necessary data for deglitching Mappixel by Mappixel. Use it, if you don't have the memory to store the MapIndex. If you can store the Mapindex and need to deglitch and map more than once, it is more useful and efficient to still use the lndLevelDeglitchTask with the Mapindex.

```
from herschel.pacs.spg.phot.deglitching.map import MapDeglitchTask
mdeg = MapDeglitchTask()
s = Sigclip(10, 4)
s.mode = Sigclip.MEDIAN
s.behavior = Sigclip.CLIP

mdeg(frames, algo = s, deglitchvector = "timeordered", pixfrac = 1.0, outputPixelsize = 3.2)
```

The glitches are put into the frames products mask (that is still called "2nd level glitchmask" by default).

The MapDeglitchTask also has two parameters that we know already from the MapIndex. They are:

- pixfrac: helps you to drizzle. Values should be > 0.0.
- outputPixelsize: the size of the map pixels. Although a map is not created by the task, internally we have to know the size of the pixels of the map, so the task can calculate the overlap between frames pixels and map pixel. The value is in arcseconds, just like for the MapIndex.

Diagnostics using the MapIndexViewer and the command line

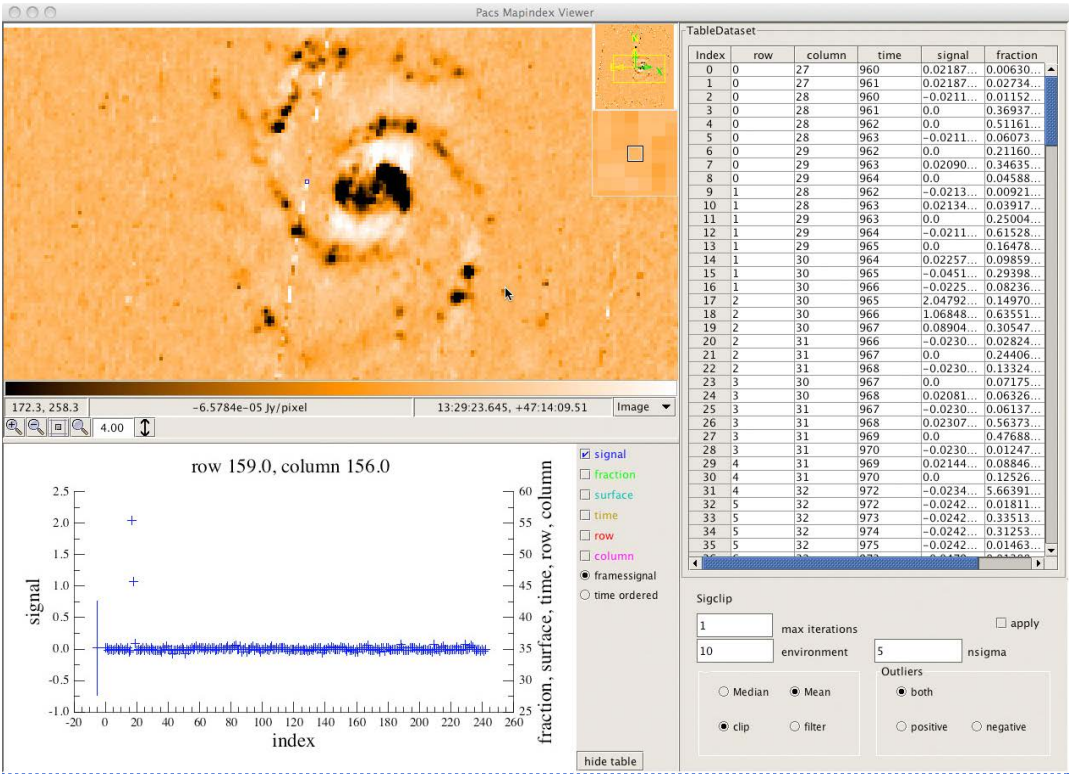
A MapIndex contains a lot of interesting data that is very useful to be analysed. The tool for doing this is the MapIndexViewer. It is called like this:

```
from herschel.pacs.spg.phot.gui.mapindexview import MapIndexViewer
MapIndexViewer(mi, frames)

# &#8230;feeds in the MapIndex mi and the Frames product. Or - better
MapIndexViewer(mi, frames, img)

# &#8230;provide also the map. If not, it must be calculated on the fly
```

- The MapIndexViewer GUI (click to enlarge):



In the top part of the GUI you see the map displayed by the Display tool. If you click into the map, a plot of the all values that a MapIndex contains for the selected map pixel plus the signal values from the Frames product is shown at the bottom. A click on the button "show table" displays the numerical values as a table on the right of the GUI.

Optimizing the Sigclip algorithm with the MapIndexViewer for best deglitching results

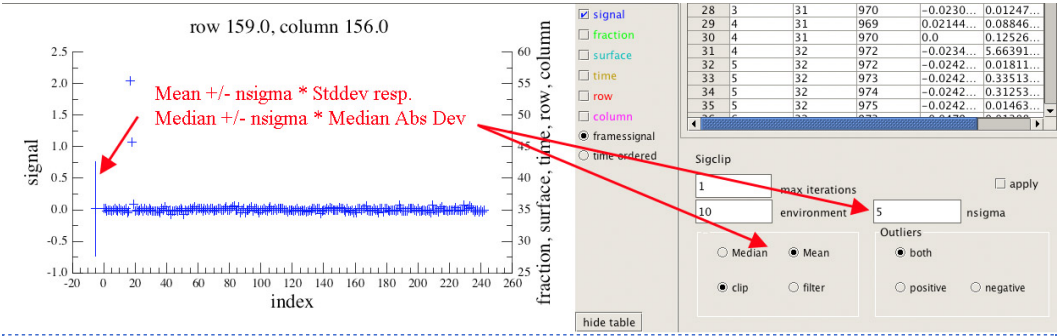
The bottom side of the table contains a panel, where you can test Sigclip parameters. Change the parameters and select the apply-checkbox. Then the Sigclip will be applied to the plotted signal. You may even choose the timeordered option for the signal. That provides an easy way to find the best Sigclip parameters without running the deglitching repeatedly. Just find glitches in your map and optimize Sigclip, before you deglitch the full map.

Maybe it is also worth to note at this point that you can use the sourcemask and/or the submap parameters, to deglitch different sections of the map with different Sigclip parameters. Just use the sourcemask/submap to divide the map into multiple regions and create multiple glitchmasks with different sets of parameters instead of only one.

There is also a permanent preview of the Mean +/- nsigma*Stddev, respectively the Median +/- nsigma*Median Absolute Deviation in the MapIndexViewer plot section. It is displayed at the x-axis index -5 as a value (Mean or Median) and an error bar (+/- nsigma*Stddev or +/- nsigma*Median Absolute Deviation).

The values nsigma and whether mean or median should be used, are directly taken from the Sigclip panel.

- Preview of the signal arrays Mean, Median and nsigma values (click to enlarge):



How to write your own Sigclip algorithm in jython

You may write your own Sigclip algorithm that can be passed to and be used by the `IIndLevelDeglitchingTask`. Two things are important:

- your algorithm has to be a jython class that extends the `numerics Sigclip`
- the jython class must implement a method called "of". This way it overwrites the "of" method in the `numerics Sigclip`.

Look at the necessary code:

```
from herschel.ia.numeric.toolbox.basic import Sigclip

class My_Own_Sigclip(herschel.ia.numeric.toolbox.basic.Sigclip):

    def of(self, vector):
        #
        System.out.println("using MY OWN algorithm")
        #
        #here you may write any code that does your job
        #only make sure it returns a Bool1d with the same
        #length as vector
        bvector = Bool1d(vector.size)
        return bvector
```

You have to do your implementation of Sigclip in the of-function. The interesting input parameter of that function is **vector**. It will contain the array of signal values (framessignal or timeordered) of a MapIndex row/column pair. The second level deglitch task will loop over the MapIndex and call your Sigclip for every row/column pair.

Your implementation MUST return a Bool1d that has the same length as the input vector. The convention is: the signal values will be treated as glitches at the indices where the returned Bool1d is **true**. That is all.

After you have finished your implementation, you have to make the class available for your hipec session. Here are two possibilities to do it:

1. load the code by opening the file with hipec, place the green hipec arrow at the first import (from herschel.ia.numeric.toolbox.basic import Sigclip) and then click **run** (repeatedly, until the arrow jumps below the last line of your code).
2. Save this jython class as a file. Use a directory that is already in the sys.path (like ./, your local working directory from which you start

hipe). From there import it into your hipe session.

Use it with the deglitch task as follows:

```
mySclip = My_Own_Sigclip()
img = deg(mi, frames, algo = mySclip)
```

Actually, you could use the above code for My_Own_Sigclip. It will do no deglitching, because all returned boolean values are **false** (Bool1d(vector.size) contains only false by default), but it should run out of the box.

Command line options for the MapIndex

On the command line, you may get the same information by using the MapIndex interface. For map location (map_row, map_col) of the map, the array of all MapElements is retrieved like this:

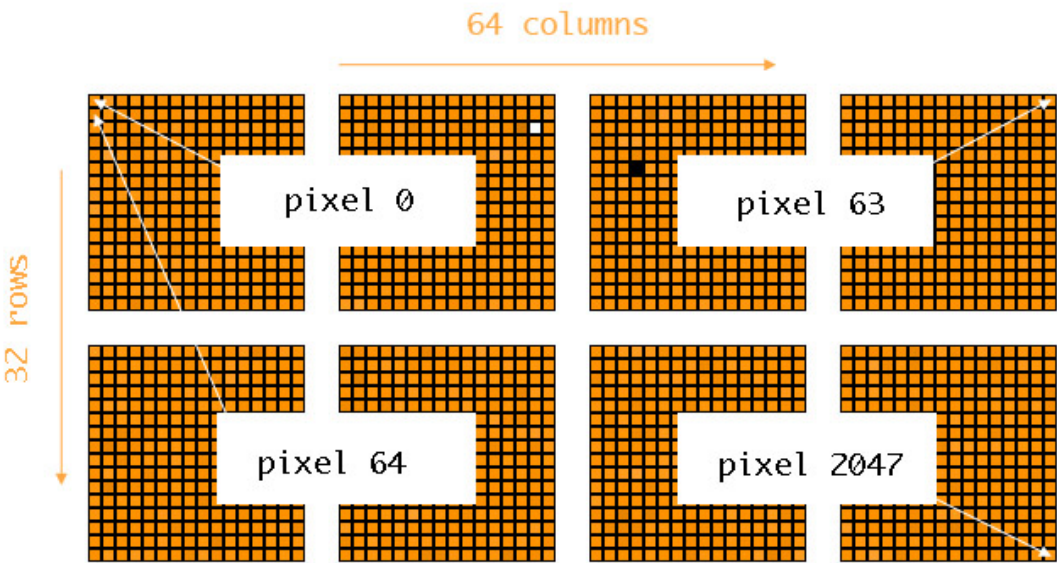
```
mapelement_array = mi.get(map_row, map_col)
print mapelement_array[0]
#> MapElement: detector 32, timeindex 1579, poids 0.0, pixelsurface 0.0

#or

print mapelement_array[0].detector # >32
print mapelement_array[0].timeindex # >1579
```

The mapelements are sorted in timeindex. This means mapelement_array[0] contains data with the smallest timeindex, the last mapelement in the mapelement_array contains values with the highest timeindex. To save memory, the detector_row and detector_column information of the data origin in the Frames product is encoded in the value detector. For the blue Photometer array the detector has values between 0 and 2047, for the red array the values are between 0 and 511. This is because the detector_row, detector_column values are compressed into single values.

- Detector numbering scheme for the blue photometer array:



The detector value can be translated to the actual detector_row and detector_column of the Frames data array using the method det2rowCol of the MapIndex product (FullMapIndex or SlimMapIndex):

```
detector_rowcol = MapIndex.det2rowCol(detector, frames.dimensions[1])
detector_row = detector_rowcol[0]
detector_column = detector_rowcol[1]
print detector_row, detector_column
```

Note: this is very complicated. A simplification, where detector_row and detector_column are directly stored in the Mapelements is on the way.

Please keep in mind that frames.dimensions[1] returns the size of the full detector array (32 or 64). This may not be the case if you work with a sliced Frames product. If you use the FullMapIndex, the surface and poids information is different from 0 (Remember? The SlimMapIndex stores only time and detector, the FullMapIndex also poids and pixelsurface).

```
print full_mapelement_array[0]
# >MapElement: detector 32, timeindex 1579, poids 3.1238432786076314E-4, pixelsurface 0.7317940044760934
print full_mapelement_array[0].poids
print full_mapelement_array[0].pixelsurface
```

The corresponding signal value is still stored in the frames product. To get access to the full set of the projection parameters, including the signal, please use this code:

```
detector_rowcol_n = MapIndex.det2rowCol(full_mapelement_array[n].detector, frames.dimensions[1])
signal_n = frames.getSignal(detector_rowcol_n[0], detector_rowcol_n[1], full_mapelement_array[n].timeindex)
```

What do you do, if you have created a SlimMapIndex and have to know the poids and surface values for some pixels?

There is a way to get to these values without recalculating a FullMapIndex. The values can be calculated on-the-fly with a task named




GetMapIndexDataTask. You use it like this:




```
from herschel.pacs.spg.phot import GetMapIndexDataTask
data_access = GetMapIndexDataTask()
full_mapelement_array = data_access(mapindex, map_row, map_column, frames, command = "mapelements")
```

Note: the use of the GetMapIndexDataTask is also very ugly and complicated. A way to access the MapElement array directly from the MapIndex product is on the way. If it is done, you will find it documented here.

-- [MichaelWetzstein](#) - 28 Jul 2010

▼ Attachments

!	Attachment	History	Action	Size	Date	Who	Comment
	DeglitchSubmap.jpg	r1	manage	40.7 K	2010-08-25 - 09:24	MichaelWetzstein	Deglitching slices 5,6 and 10
	DetectorNumbering.jpg	r1	manage	171.5 K	2010-08-25 - 09:25	MichaelWetzstein	Detector numbering scheme for the blue photometer array
	MapIndex.jpg	r1	manage	100.3 K	2010-08-25 - 09:23	MichaelWetzstein	MapIndex and MapElements
	MapIndexViewer3.jpg	r1	manage	337.4 K	2010-08-25 - 09:25	MichaelWetzstein	
	Map_M51_tiles_square.jpg	r1	manage	105.4 K	2010-08-25 - 09:24	MichaelWetzstein	The numbering scheme of the tiles for iterative deglitching. This map is sliced into 4 rows and 4 columns. This results in 16 tiles.
	SigclipSettings.jpg	r1	manage	137.6 K	2010-08-25 - 09:26	MichaelWetzstein	Preview of the Signals Mean or Median
	SignalVectorFor_Deglitching.jpg	r1	manage	53.4 K	2010-08-25 - 09:20	MichaelWetzstein	Vector of

							roughly 250 signal contributions to map pixel (85, 98)
	StrongGradients.jpg	r1	manage	26.6 K	2010-08-25 - 09:22	MichaelWetzstein	Mapping a small source
	StrongGradients_2.jpg	r1	manage	33.2 K	2011-10-12 - 15:11	MichaelWetzstein	
	Submap_M51.jpg	r1	manage	49.7 K	2010-08-25 - 09:19	MichaelWetzstein	Deglitching performed with the submap option

 [Edit](#) | [Attach](#) | [Watch](#) | [Print version](#) | [History](#): r11 < r10 < r9 < r8 < r7 | [Backlinks](#) | [Raw View](#) | [Raw edit](#) | [More topic actions](#)

Topic revision: r11 - 2013-01-14 - [MichaelWetzstein](#)