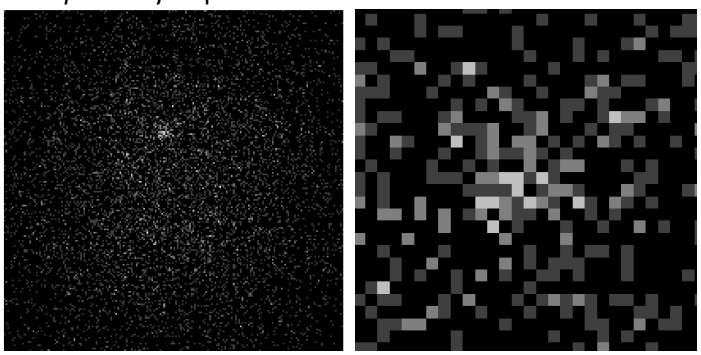
Gamma ray astrophysics project report – Luca Bonfiglioli, Antonio Grasso

Starting from a raw image, we observe that the region of the image corresponding to the source contains a higher density of pixels with greater intensity than the background ones. However, not all the pixels belonging to this region carry high intensity values: high intensity pixels appear scattered and mixed with low intensity ones.

Example – skymap:



Gaussian filter

The first step to accomplish is denoising the raw image: this can be done by applying a **Gaussian filter**. Before applying the Gaussian filter, we need to set $\boldsymbol{\sigma}$ and the **size of the kernel** $\boldsymbol{s}$.

For the sake of simplicity, we decide to bind s to σ according to the following formula:

$$s(\sigma) = \begin{cases} \max(5, \lfloor 6\sigma \rfloor), if\ \lfloor 6\sigma \rfloor\ is\ odd \\ \max(5, \lfloor 6\sigma \rfloor + 1), if\ \lfloor 6\sigma \rfloor\ is\ even \end{cases}$$

σ choice

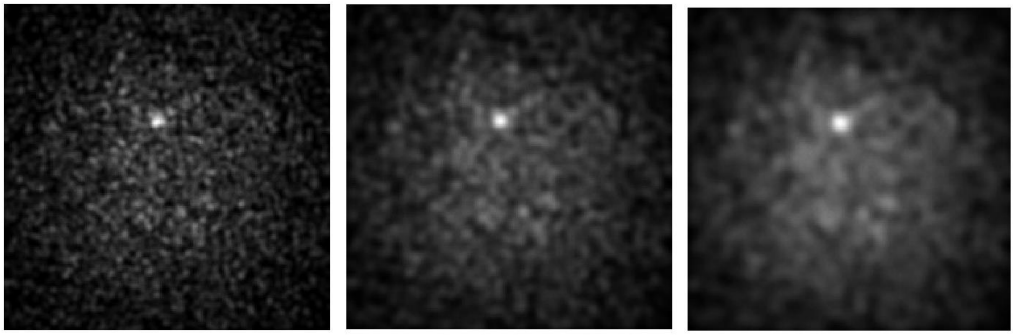
We now need to choose the value of σ in order to make it easier for us to detect the source: this depends on the shape of the source and on the background noise in the neighbourhood of the source. Thus, given a chosen σ, a single filtering may not be enough to detect the source, because that specific value may not be appropriate for that particular image. Therefore, we choose to apply multiple filters with increasing values of σ.

Starting from an image A filtered with σ_a, we want to obtain an image B filtered with σ_b: if $\sigma_b > \sigma_a$, this can be easily done by filtering A with $\sigma_c = \sqrt{\sigma_b^2 - \sigma_a^2}$. According to this formula, $\sigma_c \leq \sigma_b$: this means that $s(\sigma_c) \leq s(\sigma_b)$, so filtering with σ_c is less computationally expensive than filtering with σ_b.

Once chosen a range of σ values $[\sigma_1, \dots, \sigma_n]$, we can avoid filtering the original image n times with $\sigma_i, i = 1, \dots, n$ by applying n consecutive filters with $\tilde{\sigma}_i, i = 1, \dots, n$, where:

$$\tilde{\sigma}_i = \begin{cases} \sigma_1, if\ i = 1 \\ \sqrt{\sigma_i^2 - \sigma_{i-1}^2}, if\ i > 1 \end{cases}$$

Example – Gaussian filtering with $\sigma = 1.5, 2.5, 3.5$



We notice that as σ increases the source becomes more compact, its shape more circular and the pixel with the highest intensity tends to be around the center of this region.

Local maxima extraction

If the source is present, for each filtered image we observe that there exists a local maximum which is a good approximation of its location: the lower the σ, the better the approximation.

Thus, we need to **extract local maxima from each image**: to do so, we iterate on each pixel checking whether it is greater than all its neighbouring pixels or not. The maxima are then stored into a list as tuples $<(x, y), i>$, where (x, y) are the maxima coordinates and i is their intensity.

They are then **sorted by decreasing values of intensity**, because we know that if the source is present the corresponding local maximum will be among the most intense ones.

By increasing σ from σ_i to σ_{i+1} at each step, we expect some maxima found at the step $i + 1$ to remain within a small range from their position at step i. We want these corresponding maxima to be considered as a single maximum throughout all the iterations.

Useful data structures

For this purpose, we now introduce two simple data structures: **PointData** and **PointDataList**.

• **PointData**

It lets us store values related to a specific point in the image.

Attributes:

- $original$: a pair of coordinates initialised to a given tuple;
- $latest$: another pair of coordinates, initially equal to $original$;
- r: a numeric value initialised to a given positive value;
- $values$: a list of numeric values, initially empty.

Methods:

- $set(c, v, i)$: it takes as input a pair of coordinates c, a value v and an index i, setting $latest$ to c and the i-th element of $values$ to the maximum value between v and the previously stored value.
- $compatible(c)$: it takes as input a pair of coordinates c and it returns a Boolean value that is true if the Euclidean distance between c and $latest$ is less than r.

Example:

We initialise PointData with coordinates $(100, 100)$ and $r = 5$

$original$	$latest$	r	$values$
$(100, 100)$	$(100, 100)$	5	$\emptyset$

We perform the set operation with the coordinates $(101, 100)$, the value 30 and the index 0:

$original$	$latest$	r	$values$
$(100, 100)$	$(101, 100)$	5	$[30]$

We perform another set operation with the coordinates $(101, 102)$, the value 15 and the index 4:

$original$	$latest$	r	$values$
$(100, 100)$	$(101, 102)$	5	$[30, 0, 0, 0, 15]$

The parameters $original$ and r are intended to be read-only.

• **PointDataList**

It is a collection of PointData.

Attributes:

- $points$: a list of PointData, initially empty;
- r: a numeric value initialised to a given positive value.

Methods:

- $set(c, v, i)$: it takes as input the same parameters c, v, i as those of the set operation of PointData. This operation checks whether there is at least one element of $points$ compatible with the coordinates c or not: in the former case it performs the set operation with the parameters c, v, i on the closest compatible PointData; in the latter it adds a new PointData to $points$ and perform the set operation on it.
- $get\_point(c)$: it takes as input a pair of coordinates c returning the closest compatible element of $points$ if any is present, nothing otherwise.

Example:
Here we have a PointDataList already initialised with 2 PointData and some values:

$original$	$latest$	r	$values$				
$(100, 100)$	$(101, 102)$	5	30	0	0	0	15
$(120, 100)$	$(121, 99)$	5	0	2	15	20	0

We perform the set operation with the coordinates $(50, 80)$, the value 18 and the index 3. Since there is not any PointData compatible with $(50, 80)$, a new one will be created and added to the structure:

$original$	$latest$	r	$values$				
$(100, 100)$	$(101, 102)$	5	30	0	0	0	15
$(120, 100)$	$(121, 99)$	5	0	2	15	20	0
$(50, 80)$	$(50, 80)$	5	0	0	0	18	0

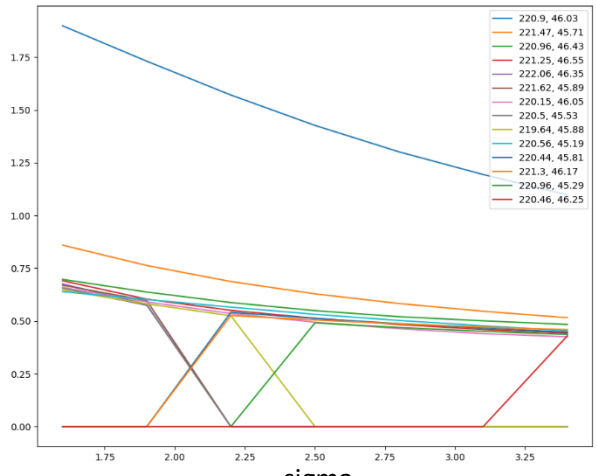
We perform the set operation with the coordinates $(100, 101)$, the value 30 and the index 5. $(100, 101)$ is compatible with $(101, 102)$, so it will perform the set operation on the first PointData:

$original$	$latest$	r	$values$					
$(100, 100)$	$(100, 101)$	5	30	0	0	0	15	30
$(120, 100)$	$(121, 99)$	5	0	2	15	20	0	0
$(50, 80)$	$(50, 80)$	5	0	0	0	18	0	0

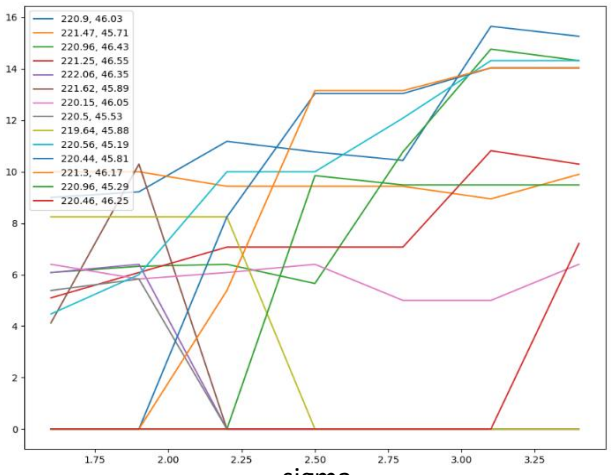
Features
For each filtered image we want now to compute **a set of features for some of the maxima** and store the computed values inside different PointDataList structures, one for each feature.
Every feature is associated with the following method:
- $update\_feature(maxima, i, point\_data\_list)$, where $feature$ corresponds to the feature name: it takes as input the sorted list of the maxima $maxima$, the index i corresponding to the i-th consecutive filtering of the image and the PointDataList $point\_data\_list$ to update with the computed values.
All the features must provide higher values for maxima which are more likely to correspond to the source and vice versa.
We have thought of two main features so far:
- **intensity**: the intensity of the maximum under consideration. This feature is very significative and carries a lot of information about the presence of the source.
- **isolatedness**: the sum of the distances between a maximum and the k nearest ones. Generally speaking, the source tends to be more isolated than the rest of the maxima, but the information provided by this feature is often useless.

Example – intensity and isolatedness
Each line represents the maxima trend across the sigma range. In the intensity graph (on the left) we can notice a blue line significantly higher than all the others, corresponding to the source.

Intensity



Isolatedness



Election algorithm

After computing all the features described above for each maximum, we end up with a fully populated PointDataList for every feature (i.e. intensity, isolatedness).

We now proceed with an election algorithm to find if there is a point which is more likely than the others to correspond to the source, among all the points belonging to the union of all the PointDataList. For every feature and for every step ($i = 1, \dots, n$), we consider the two highest values v_1, v_2 ($v_1 \geq v_2$) for that particular step:

- if $v_1 > kv_2$, v_1 gets w votes, where k and w are two positive constants depending on the feature;
- otherwise, the same number of votes w is assigned to a special candidate that represents the absence of the source.

There are 2 possible outcomes:

- the best scoring candidate is associated with a pair of coordinates which corresponds to the source;
- the best scoring candidate is the one representing the absence of the source, meaning that there is no source in the image.

Note: in order to make the algorithm work properly, v_1/v_2 should be **scale-invariant** for all the features.

As for the intensity values, their ratios are clearly not scale-invariant, since values tend asymptotically to the mean of all the pixels in the image (Gaussian smooth with $\sigma \to +\infty$). At the moment, **we do not have a clear idea about how to make those ratios scale-invariant**.

Configuration file

Along with the code, we provide a configuration file to set the most relevant parameters of the whole algorithm described so far.

The configuration file is a JSON contains the following main parameters:

- **dir**: a string representing the path containing the skymap files that the algorithm should analyse;
- **sigma_array**: a list of values $[\sigma_1, \dots, \sigma_n]$ corresponding to the $\sigma_i, i = 1, \dots, n$ values needed for filtering the images;
- **active_measures**: a list of strings representing the features that the algorithm takes into account;

Each feature is also associated to a parameter corresponding to the name feature (e.g. **intensity**, **isolatedness**). Each of these parameters contains the lists of specific parameters needed for that particular feature. Among these specific parameters there always must be four compulsory ones:

- **vote_thresh**: the value of k in the section "Election algorithm"
- **vote_weigth**: the value of w in the section "Election algorithm"
- **dist_thresh:** the value of r for the PointDataList corresponding to this feature
- **method**: the name of the method that computes the feature, that method will be called automatically if the parameter **active_measures** contains the name of this feature.

Skymap generation

Our code also provides the functionalities needed to the **automatic skymap files generation**. In particular, there are two main functions which leverage on the ctools library to generate the skymaps:

- $generate\_src\_data(start\_model, flow, n, start\_coords, radius, start\_seed)$: generates skymaps with one source, according to the following parameters:
    - $start\_model$: source and background model xml file.
    - $flow$: flow value used for the simulations.
    - n: number of skymaps to generate.
    - $start\_coords$: the coordinates around which the source will be placed.
    - $radius$: the maximum distance between $start\_coords$ and the source.
    - $start\_seed$: a positive integer that will be used as a seed for the simulation.
- $generate\_bkg\_only\_data(bkg\_only\_model, n, start\_seed)$: generates skymaps with background only, according to the following parameters:
    - bkg_only_model: background model xml file.
    - n: number of skymaps to generate.
    - $start\_seed$: a positive integer that will be used as a seed for the simulation.