Computer Vision - Final Report 2018/2019

Elena Ruggiano, Federico Oliva

06/03/2019

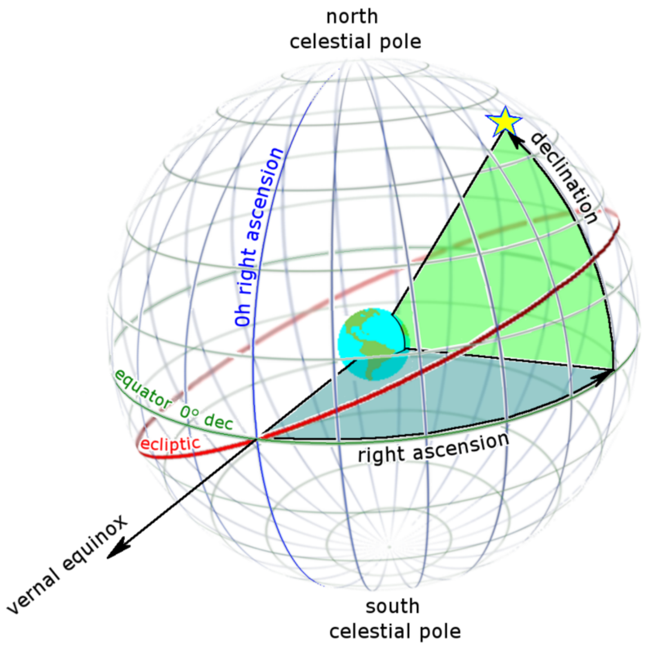
Contents

List of Figures

Chapter 1

Introduction

The goal of this project is to automatically analyze the sky maps of the CTA observatory in order to detect gamma ray sources. Given a sky map, this algorithm has to detect the gamma ray source in the map, obviously only if present. Knowing the coordinates towards the telescope is pointed, the algorithm has also to compute and return the coordinates of the gamma ray source detected.
Coordinates are expressed in the celestial coordinate system which is a 2D spherical reference frame in which each point is identified by two angular coordinates: the Right Ascension (RA), which is the angular distance between the zero hour circle and the hour circle passing through the considered point, and Declination (DEC), which is the angular distance between the celestial equator and the considered point.



The project consists of three steps:

1. **Generation of the maps**
   Two python programs (*map_ creator.py* and *recall.py*) have been implemented in order to create a large number of maps which are simulations of the real maps coming from CTA observatory. These maps have been generated accordingly to the specifics given by INAF, which will be presented in section 2.1: **source_generation_INAF**. The coordinates of the gamma ray source possibly present in each map are stored in a file called *Map.log*.

2. **Image analysis**
   A C algorithm based on OpenCV and CFITSIO libraries, in addition to many self-written functions, takes as input one map and returns the pixel coordinates of the gamma ray source, if present. Such algorithm uses some parameters whose values are not a priori fixed but will be set to optimal ones coming from the parameter tuning, implemented in the following step.

3. **Data analysis and parameters tuning**
   Given a set of N maps generated as in the first point, the python program *data_ analysis.py* compares the true gamma ray sources coordinates (stored in the *Map.log*) with the estimated ones obtained from the C analysis and quantitatively evaluates the performances of the C algorithm.
   The file *param_ tuning.py* calls the *data_ analysis.py* with different values of the C program parameters and determines which are those maximizing the performances of the algorithm. These values are selected to be the tuned parameters.

As final step a further set of maps (different from the set used for the parameters tuning) is generated and analysed using the tuned parameters. On this set of maps the detection algorithm's performances are computed.

Chapter 2

Generation of the maps

In order to completely understand the following contents some concepts should be clarified:

- **Map:** a map is *.fits* file containing all the information about a CTA telescope. These information range from the time duration of the observation to the sky coordinates pointed by the telescope. The *.fits* format is widely described at http://docs.astropy.org/en/stable/io/fits/. Briefly, a *.fits* file is a set of lists of different nature called *headers* containing all the image data. The *COUNTS* header contains a matrix whose elements are equal to the number of photons hitting each telescope's pixel during the sky observation.
- **Source:** a source is an observable gamma ray phenomenon that can be present in a map.
- **Background :** the background is the noise present in the region of sky pointed by the telescope. The greater it is, the tougher will be finding the source possibly present.

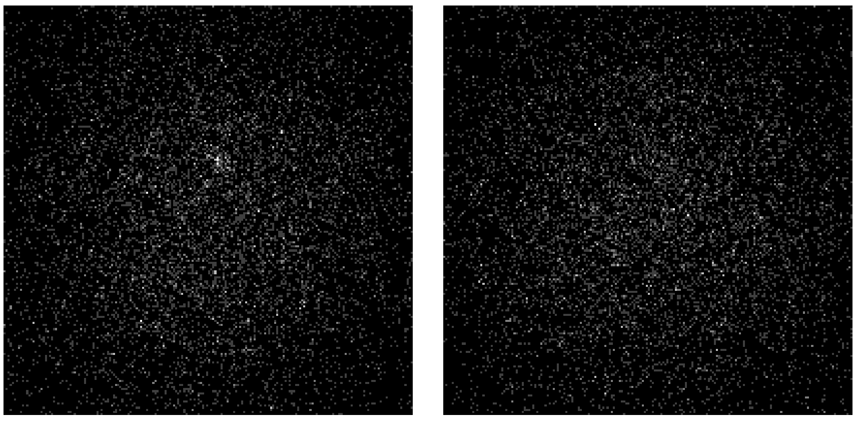

(a) *Map with a source* (b) *Map with only background*

Figura 2.1: *Examples of maps with and withouts sources*

2.1 map_creator.py

This file has been written as a python library in order to provide a bunch of functions allowing the user to create a map according to specific requirements.

Now a quick skim over the library functions will be presented:

find_coordinates function

```
int * position find_coordinates(FILE * logfile)
```

This function takes in input a file pointer where all the computed data and information are stored. This file pointer will be shared among nearly all the library functions creating a sort of final logfile containing all the coordinates of the sources within the generated maps (*Map.log*). From the command line input this function also retrieves the observation file *obs_crab.xml* within which the telescope pointing coordinates (in RA and DEC) are stored. This function reads these values and store them in an output array *position*.

source_generation_INAF

```
int * source_generation(int * position, int background_prefactor, int sigma, FILE *
    logfile)
```

This function is devoted to generate a map satisfying INAF requirements: the map has to contain one or zero sources. The source, if present, must be placed close to the pointing center (maximum distance 0.5°) and its intensity is defined with respect to a Signal to Noise Ratio *sigma* defined as

$$\sigma = \frac{S}{\sqrt{S+B}} \tag{2.1}$$

where $S = source\_intensity$ and $B = background\_prefactor$ which describes the minimum intensity of the sources with respect to the background. Fixing *sigma* and *background_prefactor*, the value of the source intensity is straight forward.
According to INAF specifications *sigma* has been set to values around 4. It must be highlighted that, in order not to generate different maps all with the same intensity of the source, the value of *sigma* has been randomly generated. Obviously the higher is *sigma* the brighter is the source with respect to the background, the easier is the source detection and viceversa.

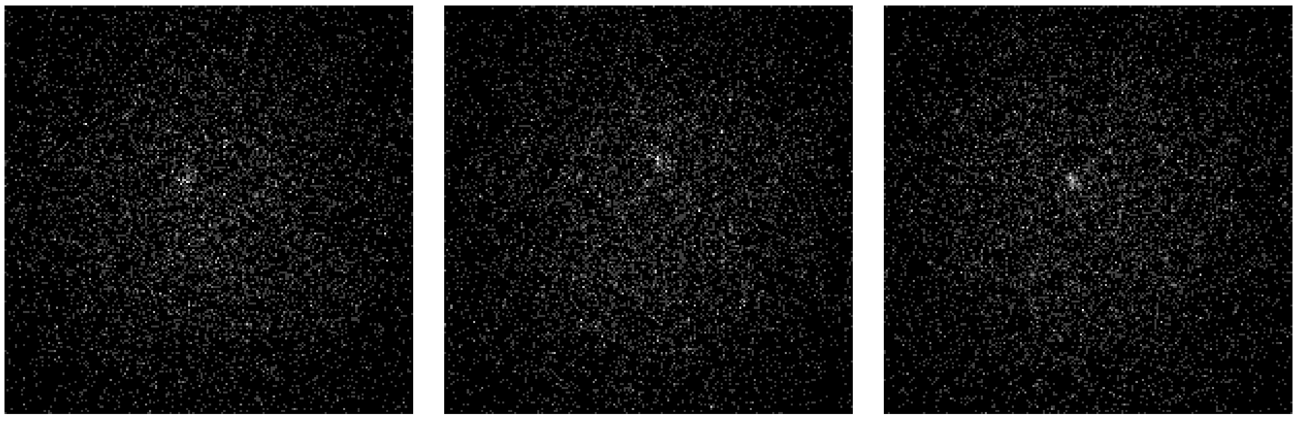

(a) *sigma 3 source* (b) *sigma 4 source* (c) *sigma 5 source*

Figura 2.2: *Examples of sources with different intensities*

backgoround_generation

```
float prefactor = background_generation(float min_val, float max_val, FILE * logfile)
```

This function takes as input the background range extrema and randomly picks a value within them. Finally, it writes this info in *Map.log*. The final value has a 4 digit precision.

sourcefile_generator

```
void sourcefile_generator(float * source_array, float background_prefactor, FILE *
    filename)
```

This function uses all the previously generated info to write an *.xml* file dscribing the sources intensity and positions. This file will be later used to create the *.fits* image of the source.
The *main()* function is nothing more than a wrapper of all these functions.

2.2 recall_inaf.py

If *map_creator.py* allows the user to create a single map then **recall_inaf.py** calls the previous program in order to generate a vast dataset. The calling syntax is the following:

- **recall_inaf.py** :

```
>> python recall_inaf.py RA_center DEC_center sigma n_maps
```

where *RA_center* and *DEC_center* are the coordinates of the pointing center of the telescope, *sigma* is the desired SNR and *n_maps* is the total number of maps desired.
Roughly speaking, the program clears the filesystem from previous files and then calls iteratively *map_creator.background_generation* , *map_creator.source_generation* , *map_creator.sourcefile_generator*.
Then, it uses the *ctools* library in order to generate the *.fits* image from the previously created *.xml* file. After this generation, all the notable files created are stored in dedicated folders.
After this procedure, the user has a dataset that can be used to test the algorithm, as it will be explained in the *data_analysis.py* section.

Chapter 3

Image analisys

The algorithm takes as input a *.fits* image (generated as in chapter 2). The values contained in the *COUNTS* header are stretched within the [0 : 255] intensity range and then stored in a *OpenCV CVImage* data structure.
Then, the algorithm performs all the steps described below in order to find the source position.
The output is provided as a file containing the number of detected sources and their coordinates expressed as pixels. Such coordinates will be subsequently converted in celestial ones by a specific python function taking as input also the telescope pointing coordinates.

3.1 Median filter

```
void median_filter(IplImage *input, IplImage *output)
```

A cross kernel with 5 elements is considered. The kernel centered into the generic *(i,j)* pixel is made by the following pixels: *(i-1,j)*, *(i,j-1)*, *(i,j)*, *(i+1,j)*, *(i,j+1)*. The intensities of these 5 pixels are stored in a vector and are ordered in an increasing way. The *(i,j)* pixel of the output image is set equal to the 4_*th* element of the ordered vector. This choice is different from what is usually done in the classical median filter in which the central element of the ordered vector (so the 3_*th* element in this case) is taken as output. This variation with respect to the classical median filter is done in order to kill single outliers of the background without diminishing too much the source intensity.

3.2 Gaussian smoothing

```
void gauss_smooth_fast(IplImage *input, IplImage *output, float sigma)
```

Gauss_smooth_fast performs the gaussian smoothing with the assigned sigma on the input image. At the end of the function the output image will be the smoothed one.
In order to speed up computations this function takes advantage of the separability property of the Gaussian function: a 2D Gaussian function can be expressed as the product of two 1D Gaussian functions:

$$G(x,y) = G(x)G(y) = \left(\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{y^2}{2\sigma^2}}\right) = \frac{1}{2\pi\sigma}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The Kernel of the 1D smoothing is a vector of dimension $2k+1$ where

$$k = \lceil 3\sigma \rceil$$

The *i-th* generic element of the Kernel has the following value:

$$G_\text{i} = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(i-k)^2}{2\sigma^2}} \qquad\qquad i = 0, 1, ..., 2k$$

After the Kernel assignment two scanning of the image are carried out: the first for the horizontal smoothing and the second for the vertical one. The smoothing is simply obtained by assigning to the value of the generic (i,j) pixel the weighted sum of the intensities of its horizontal/vertical neighbours, where the weights are given by the Kernel.

Horizontal smoothing:

$$Hor_{smooth}(i,j) = \sum_{v=-k}^{k} I(i, j+v) * G_{v+k}$$

Vertical smoothing:

$$Smooth(i,j) = \sum_{u=-k}^{k} Hor_{smooth}(i+u, j) * G_{u+k}$$

The edges of the image are treated in a specific way: only pixels inside the image are taken into account into the weighted sum and not all the Kernel is used. In this case the Kernel weights are divided by the normalization factor, which is given by the sum of only the considered Kernel elements: the sum of all Kernel elements is 1, but considering a fewer number of Kernel elements their sum must be remapped into 1 otherwise some intensity drops will appear on image edges.

3.3 Circle detection

```
void circle_detection(IplImage *input, IplImage *output, int values, int
    binarization_threshold, float accept_level, int r)
```

The input image is scanned by a circular window of given radius *r*. Let's focus the attention on a generic (i,j) pixel. The function counts how many of the pixels within the circle centered in (i,j) have an intensity greater then *binarization_threshold.* If the ratio between the number of counted pixels and the area of the circle is greater than *accept_level* then pixel (i,j) is classified as a possible gamma source (its intensity on the output image is set to 255), otherwise as background (its intensity on the output image is set to 0):

$$N/A \begin{cases} < accept\_level & output(i,j) = 0 \\ \geq accept\_level & output(i,j) = 255 \end{cases}$$

N: pixels above *binarization_threshold*
A: circle area

In case (i,j) is an edge pixel, A is the area of the portion of circle contained inside the image.
The variable *value* is a integer 2D array whose dimension is equal to the image size. This matrix stores the values of N for each pixel of the input image.

3.4 Blob analysis: labeling and area computation

```
int labeling_8(IplImage *input, int *L)
```

This function assigns an integer label to each blob of the binary image taken as input. L is a 2D matrix whose elements correspond to the input image pixels.
If the generic pixel (i, j) belongs to the background $L(i, j)$ is set to 0, otherwise $L(i, j)$ is set to the integer value of the blob that contains the considered pixel. Positive and consecutive integer values are used to label the blobs and an 8-connectivity is assumed.
This function returns the number of white blobs (so the number of possible gamma sources called *n_ sources*) present in the image.

```
void area_blobs(int *L, int n_sources, int *area, CvSize size)
```

Once the labeling is performed, the area of each blob is computed. These values are stored in the vector *area* of dimension *n_ sources.*

3.5 Intensity ratio

```
void intensity_ratio(IplImage *input, IplImage *output, int *L, int *area, float
    threshold, int n_sources, double *ratio)
```

After the circle detection some blobs could still be wrong background agglomerates which are not as intense as real sources. In order to remove these wrong blobs a further image elaboration is needed. The idea is to kill blobs whose intensity is not so larger than the background intensity.
This function consists of 4 steps:

1. Let's suppose to have a mask placed above the smoothed image. This mask has some holes in the same position where blobs have been detected. For each hole (or blob) the intensity value of its pixels are summed up (N. B.: intensities are read from the input image which is the smoothed one). All these sums are stored into an array of dimension *n_ sources.*

$$mu_k = \sum_{(i,j)\in Blob_k} I_{smooth}(i,j) \qquad\qquad k = 0, 1, 2, ..., n\_sources$$

2. Intensities contained in the array are divided by the area of the correspondent blob. In this way the obtained value associated to each blob doesn't depend on the size of the blob.

$$mu_k = mu_k / A_k \qquad\qquad k = 0, 1, 2, ..., n\_sources$$

3. The relative intensities of each blob is divided by the relative intensity of the background (which is the first element of the obtained array).

$$ratio_k = mu_k / mu_0 \qquad\qquad k = 1, 2, ..., n\_sources$$

So *ratio$_k$* is a quantity that tells how much blob k is brighter with respect to the background.

4. Finally if *$ratio_k$* is greater than *threshold* the blob continues to be classified as source, otherwise its binary classification is changed into background.
   $\forall(i,j) \in Blob_k$:

$$ratio_k \begin{cases} \leq threshold & output(i,j) = 0 \\ > threshold & output(i,j) = 255 \end{cases}$$

   After that a relabeling is performed and the value of *n_ source* is updated.

3.6 Center detection

Once the source image has been elaborated as in the previous steps a binary image is obtained with white blobs and black background. The next step is to compute the center of gamma ray sources which have been detected and to give as output the coordinates of such points.
Depending on the parameter *center_ type* chosen, there are three possible ways to determine the center point of the blobs:

1. *center_ type*='b': **Barycenter**

```
void barycenter(int *L, int n_sources, int *area, CvSize size, double *
    barycenter_real, int *barycenter_int)
```

   This function returns the barycenter position of each blob given the array *L* which contains the labels of the blobs. These coordinates, which are real values, are stored in *barycenter_ real* array in two consecutive elements of the array.

   $k = 1, ..., n\_sources$ $\qquad$ ($k = 0$ is the background blob):

$$barycenter\_real(2*(k-1)) = \frac{1}{A_k} \sum_{(i,j)\in Blob_k} i$$

$$barycenter\_real(2*(k-1)+1) = \frac{1}{A_k} \sum_{(i,j)\in Blob_k} j$$

   Barycenter_ int array contains the integer rounded version of the coordinates, which will be used just to provide the visualization of pixels corresponding to the barycenters.

2. *center_ type*='i': **Max Intensity**

```
void center_values(int *values, CvSize size, int *L, double *max_real, int *
    max_int, int n_sources)
```

   The aim of this function is to select the center of the blob not depending on its shape as in the previous case, but on the basis of the intensities of its pixels. Indeed in this case the pixel with the highest value of N (stored in *values* array) is selected as center of the blob. N is in fact the number of neighbours pixels with an intensity greater than *binarization_ threshold*. If two or more pixels of the considered blob have the same N value, the center is chosen to be the point with the middle position of these pixels. As for barycenter both real and integer center positions are stored.

3. *center_ type*='m': **Mean**

```
void mean_coordinates(double *bar, double *max, double *mean, int n_sources)
```

In this case for each blob both of the previous two steps are performed (barycenters and maximum intensity) and then the mean value of the obtained two points is selected as the center.

3.7 Remove separate blobs

```
int remove_separated_blobs(IplImage *input, IplImage *output, int *L, int *center_int
    , double * ceter_real, int n_sources, double *ratio, int center_distance)
```

The last step of the map analysis consists in removing those blobs which are too close to other blobs. Indeed it may happen that one gamma ray source is split in more then one blob during the analysis. To avoid this trouble this function makes the following:

- computes the euclidean distance (in pixels) between the centers of any possible pair of blobs;
- if the centers of the two considered blobs are too close (their distance is smaller or equal than *center_ distance*) then the blob which has a lower intensity (lower *ratio* value) is removed. This function returns the updated number of sources present in the image, which will be smaller than the input *n_ sources* in case of some removal has been performed.

Chapter 4

Parameters tuning and results

In the previous chapter the steps implemented by the C algorithm during the analysis of one single map have been described. Given one particular map, the performance of such algorithm varies depending on the values assigned to the following six parameters:

- *sigma* of the function *gauss_smooth_fast* (see section 3.2)
- *binarization_threshold*, *accept_level*, *r*, of the function *circle_detection* (see section 3.3)
- *threshold* of the function *intensity_ratio* (see section 3.5)
- *center_distance* of the function *remove_separate_blobs* (see section 3.7)

According to what described in Chapter 1, two sets of maps have been generated. The first set (made by 3000 maps) has been used to train the algorithm in order to make the parameter tuning, namely automatically find the values of the six thresholds that maximize the performances of the algorithm. Once found such values, the second set of maps (made by 10000 maps) have been used in order to find the final actual performances of the program.

4.1 Data analysis

```
>> python data_analysis.py Map.log fits_directory output.log center_type sigma
 accept_threshold radius binarization_threshold intensity_ratio min_source_dist
```

In order to mathematically evaluate the performances of the algorithm the python function *data_analysis.py* has been implemented. Such function has the following inputs:

- the address of the folder in which a set of maps is stored. In this folder the *Map.log* file is also stored. This file consists of a list of all the maps of the considered set. For each map the following information are stored: number of actual sources present in the map (0 or 1) and, in case of source presence, the actual RA and DEC coordinates related to the source.
- the directory path where all the *.fits* files are stored
- the six parameter values that will be used to call the C algorithm.

Given these input *data_analysis.py* analyses each map calling the C algorithm. A new log file called *est_map.log* is then generated containing a list of all the estimated coordinates. For each map the following information are stored:

- number of detected sources present in the map (0, 1 or more)
- in case of source/s detection, the estimated RA and DEC coordinates related to the source/s.

By comparing *Map.log* and *est_ map.log*, *data_ analysis.py* computes the value of the variable *error*. Such variable roughly speaking is the mismatch between the total number of sources present in all the analyzed maps and the number of estimated sources, so the smallest the error the better the performances of the C algorithm are. In particular the *error*, initialized to 0, has been incremented whether, for each map, one of the following situations occur:

- the C algorithm estimates one or more sources but the analyzed map doesn't have any source. In this case *error* is incremented by the number of estimated sources (false positives).
- the C algorithm doesn't estimate any source but the analyzed map contains one source. In this case *error* is incremented by 1 (false negative or misdetection).
- the analyzed map has the source and the C algorithm estimates the presence of one source but in the wrong position: the distance between the actual and the estimated source is greater than $0,5^\circ$ (INAF specifications). In this case both a false positive and a false negative error have been committed, so *error* is incremented by 2.
- the analyzed map has the source and the C algorithm estimates N sources all in wrong positions. In this case *error* is incremented by N (N false positives).
- the analyzed map has the source and the C algorithm estimates N sources one of which in the correct position and the other in wrong positions. In this case *error* is incremented by $N-1$ ($N-1$ false positives).

For each map having a source correctly detected by the C algorithm, *data_ analysis.py* computes also the differences (considered in absolute term, so always positive) between the actual coordinates of the source and the estimated coordinates. Once all the maps have been analyzed and all the coordinate differences computed the following values are find:

- *mean_ RA*: is the RA coordinate mean error.
- *mean_ DEC*: is the DEC coordinate mean error.
- *sigma_ RA*: is the standard deviation of the RA coordinate error.
- *sigma_ DEC*: is the standard deviation of the DEC coordinate error.

4.2 Parameters tuning

The python program *param_ tuning.py* calls in an iterative way the *data_ analysis.py* considering the first set of maps (the one with 3000 maps) and with different values of the six parameters taking as input by the C algorithm. In order not to make the computational time become too long the *param_ tuning.py* has been called six different times. Each time only one of the six parameters has been changed into a reasonable range.
The set of values given to the parameters that minimize the *error* is the following:

- $sigma = 2.3$
- $accept\_level = 0.8$

- $radius = 3$
- $binarization\_threshold = 23$
- $intensity\_ratio = 3.9$
- $center\_distance = 66$

4.3 Results

The set of tuned parameters has been used for the analysis (calling *data_ analysis.py*) of the second set of maps, the one made by 10000 maps. The following results have been obtained:

Noise level	wrong detections	total n maps	error percentage
$\sigma < 4$	19	262	7.25%
$4 \leq \sigma < 5$	51	2868	1.77%
$\sigma > 5$	16	1832	0.87%
only background	25	5038	0.50%
total dataset	111	10000	1.11%

Figura 4.1: *Algorithm performance on 10000 maps dataset*

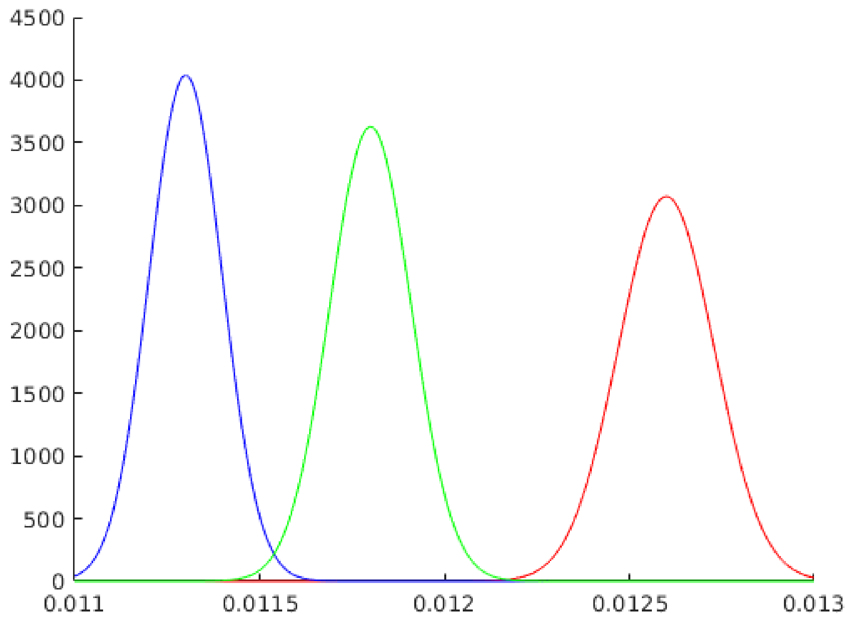
center type	mean RA	mean DEC	var RA	var DEC
b	0.0126	0.0263	0.00013	0.00038
i	0.0113	0.0250	9.88e-5	0.0003
m	0.0118	0.0255	0.00011	0.00033

Figura 4.2: *Statistic performance in sources estimation - same specifics of table 4.1*

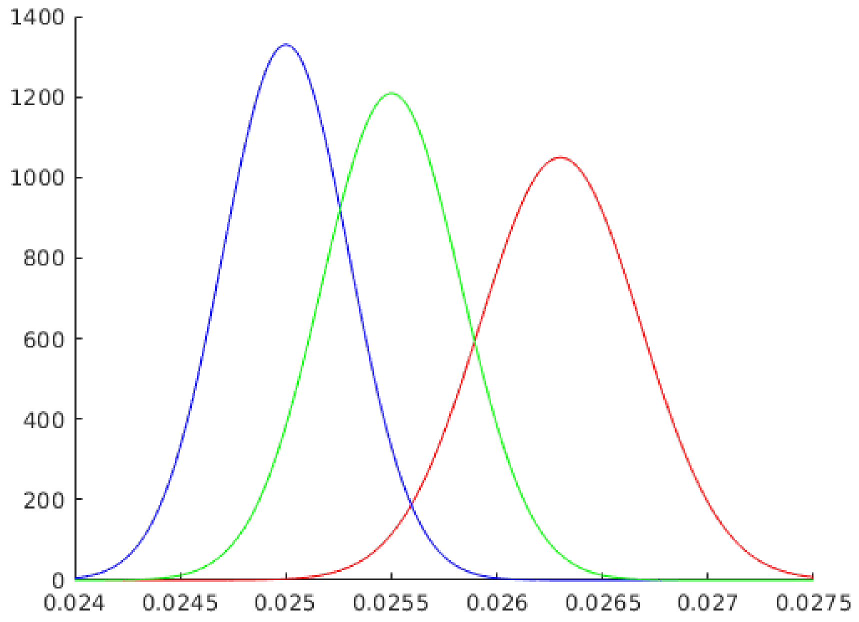
As it was expected the greater the level of noise, the higher the percentage of error that however is about 1% on the overall. As far as the *center_ type* is concerned, data shows that the best performances are obtained with the *i* option. Generally speaking the statistic performances are:

- mean RA: 0.0113
- mean DEC: 0.0250
- var RA: 9.88e-5
- var DEC: 0.0003

The statistics of the performances are shown in the figure below, both for the RA and DEC estimations



(a) *RA estimation*



(b) *DEC estimation*

Figura 4.3: *center_type options: red = 'b' blue = 'i' green = 'm'*