POLITECNICO DI TORINO

Master's Degree in mechatronics engineering

Master's Degree Thesis

Beehive monitoring with computer vision

Supervisors

Marcello CHIABERGE

Simone ANGARANO

Candidate

Cyrian FROISSART

2022

Summary

The bee is an essential species for the pollination of many crops; its abnormally high extinction rate is therefore particularly worrying. Domestic bee populations are being carefully observed to identify the reason for this decline. In this context, video hive monitoring aims to enable real-time monitoring while avoiding laborious human labour.

We'll Bee is a start-up seeking to automate certain hive monitoring processes. A tool used to count the number of entries and exits of bees to the hive already exists. The subject of this internship is the improvement of this system in order to determine the proportion of bees bringing pollen back to the hive. Since pollen is used to feed bee larvae, it is indeed a good indicator of the growth of the hive.

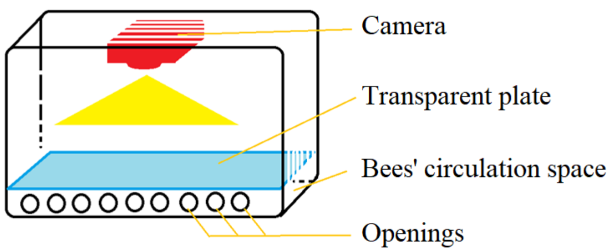


Figure 1: Observation system diagram

The observation system consists of a translucent box placed at the entrance to the hive (Fig. 1); openings to the outside and to the hive allow the bees to circulate, while a Plexiglas plate ensures that the bees walk at the bottom of this box. A raspberry Pi equipped with a camera attached to the top of the box films the bees at a speed of 15 frames per second. The lighting is natural so as not to disturb the behaviour of the bees. The raspberry is powered by a battery. Ultimately, the objective would be to make the system autonomous by charging the battery with a photovoltaic panel. The objective of this internship was to create a classification algorithm to separate the images of bees bringing pollen to the hive from those not bringing it back. To make such an algorithm useful, video acquisition, image segmentation and tracking of each bee from frame to frame are required. Finally, in order to be able to train an algorithm to classify bees, it is necessary to create a set of catalogued images.

Table of Contents

List of Figures

Literature review

In recent years, several studies have been carried out to prove the feasibility of video surveillance of bees.

Different methods of image segmentation are possible, either by comparing an image with a "large" number of images preceding it to bring out the changing pixels [1], or by using the colour differences between the bees and the background of the image [2] .

Tracking bees along a video is possible by comparing the actual position of a bee to a predicted position [3] . For this purpose, a bounding box is assigned to each position. Indeed, this comparison uses the notion of Intersection over Union (IoU) rather than a simple distance between the two positions.

Once images of bees were obtained, several classification methods were created to distinguish bees carrying pollen. The extraction of some parameters related to the colours and the shape of the bees can be enough to classify the bees with a satisfactory precision [1]. More advanced algorithms such as convolutional neural networks (CNN) can also be used to achieve the same goal, either by directly seeking to classify the bee image [4], or by searching for the presence of pollen bags in the picture [5].

The main difference between these studies and the work presented, is the goal to create a real-time algorithm able to classify bees using few computational power. As such, the accuracy of the algorithm is less important in this thesis than its capacity to be run by lightweight hardware.

The book “Hands on machine learning” [6] serves as a guide for all experiments related to classification algorithms.

Chapter 1

Video acquisition and pre-processing

1.1 Manual Distortion

The camera of the observation system is controlled by a raspberry Pi. The objective is that this raspberry can process the images and directly determine the number of bees carrying pollen in real time.

In order to obtain videos to train a classification algorithm, five-minute videos are recorded at regular intervals (at 5:00 a.m., 6:30 a.m., 8:00 a.m... until 8:00 p.m.). The videos are saved on an SD card plugged to the raspberry; during the thesis, the hive was in an area covered by a wifi network, it was then possible to download the videos via an SSH connection.

Since the camera has a wide angle of view, the image obtained is very distorted (Figure 1.1). Pre-processing is therefore necessary before analysing the image.

We observe a barrel distortion due to the camera lens. This distortion is inconvenient, but the use of such a lens is necessary to capture the bottom in its full length with a camera relatively close to it. There is also a much less present tangential distortion, due to the misalignment of the camera and the observed plane.

In order to be able to analyse the image, it must be deformed in such a way as to remove the distortions present (Figure 1.2).

The opencv library offers a function to correct these distortions. A transformation matrix is first formed by calling the initUndistortRectifyMap() function, then the transformation is applied to the image by calling the remap() function.

The transformation matrix is chosen by trial and error: several images are reformed using different parameters of the initUndistortRectifyMap() function (see appendix A). These are selected one by one, refining the image processing step by

Figure 1.1: Raw picture

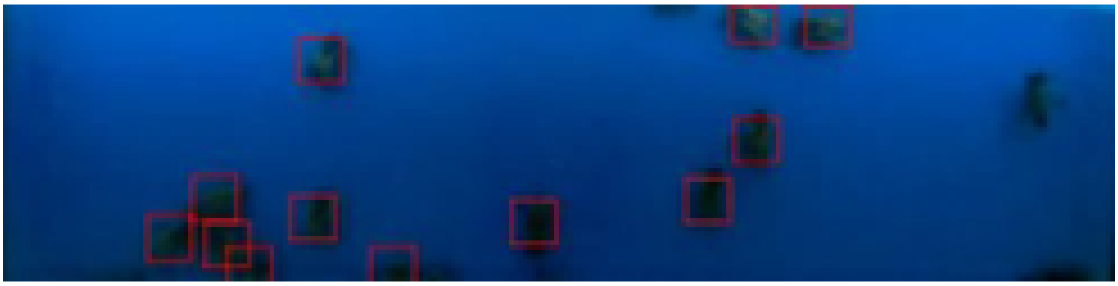


Figure 1.2: Picture undistorted and cropped

step.

1.2 Automatic distortion

The process of selecting appropriate distortion parameters is long and tedious. Moreover, it has to be done for each time a camera is set up. Indeed, the distortion observed and the fitting correction needed change significantly with the placement of the camera.

In order to avoid doing this task manually in the future, a program was made to find correct parameters more easily and by spending less time. This program needed to be as simple as possible, in order to be used by personnel with little to no training in optics and informatics if needed.

The distortion parameters are found using characteristic points in the picture and making some hypothesis. The border of the bee's circulation space (the bottom of the box) appear clearly curbed and we can suppose without risk that it should form straight lines on an undistorted picture (Figure 1.3). The hypothesis used is that all points belonging to the border should have the same Y coordinates. The Y coordinates of the upper point for the upper border and the lower point for the low border are kept in order to extend the picture. Indeed, if we were to diminish the picture size, some pixels would be lost (especially in the center of the picture) and some pieces of information with them.

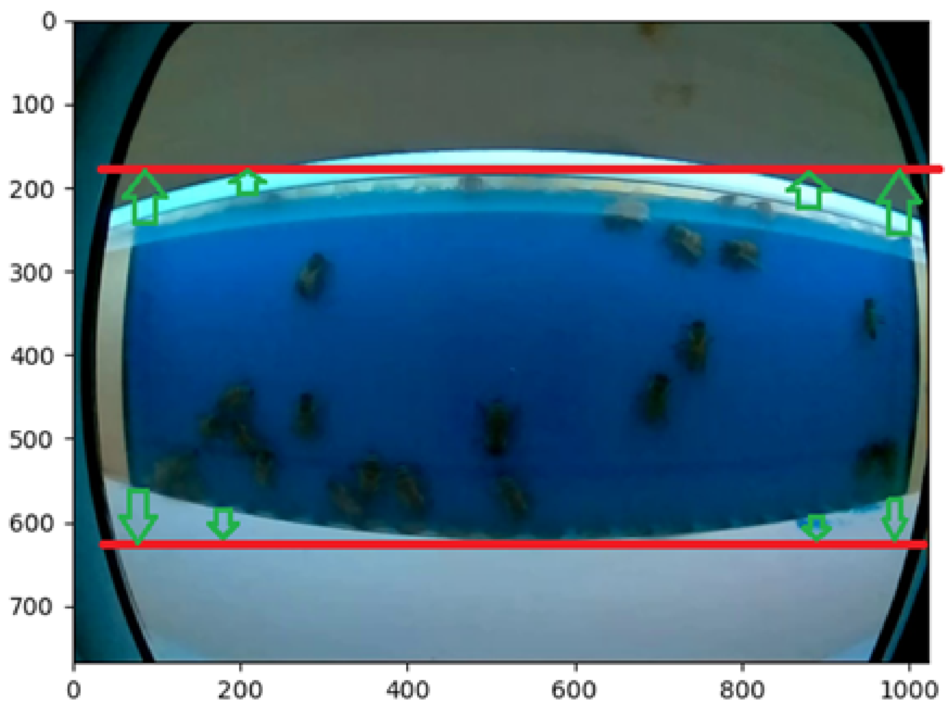


Figure 1.3: illustration of the hypothesis used; points of the border must be moved in order to obtain straight lines

Since OpenCV and the initUndistortRectifyMap() function use the Brown-Conrady model to describe to distortion of a picture, this model was also used for the program.

$$x_u = x_d + (x_d - x_c)(K_1 r^2 + K_2 r^4 + K_3 r^6) + P_1(r^2 + 2(x_d - x_c)^2) + \\ 2P_2(x_d - x_c)(y_d - y_c)$$

$$y_u = y_d + (y_d - y_c)(K_1 r^2 + K_2 r^4 + K_3 r^6) + P_1(r^2 + 2(y_d - y_c)^2) \\ + 2P_2(x_d - x_c)(y_d - y_c)$$

Here x_u and y_u are the coordinates of the points in the undistorted picture (i.e. the coordinates we want to reach in order to have straight borders). x_d and y_d are the coordinates of the points on the distorted picture, and r is the distance from the center of the distorted picture. K_i are the radial distortion coefficient, numbered 3; while P_j are the tangential distortion coefficient, numbered 2.

The picture is used to get y_d, and the straight border hypothesis fixes y_u. The equation of the Brown-Conrady model will be used to find the distortion coefficient needed.

To begin with, the operator has to select a number of points belonging to the borders of the distorted picture. The operator simply click on some points in a specific order and the coordinates of the points are saved thanks to the function ginput(). As the picture is more distorted away from the center, it is indicated to select the four corners of the bee's circulation space; but it is also necessary to select points along the whole borders. In order to help the operator, some evenly-spaced lines are drawn on the picture: the operator then need to select the intersection between those lines and the borders. (Figure 1.4)

Selecting the points lead to have all the y_{d_i} needed; in order to obtain straight horizontal lines y_{u_i} are set equal to the higher y_d for all the points of the high border, and equal to the lower y_d for all the points of the low border.

Once y_{d_i} and y_{u_i} are fixed, the distortion coefficients are to be determined using equations derived from the Brown-Conrady model:

$$y_i r_i^2 K_1 + y_i ri^4 K_2 + (2y_i + r_i^2)P_1 + 2x_i y_i P_2 + y_i r_i^6 K_3 = y_{u_i} - y_i$$

Where $y = y_d - y_c$ for practicality.

Please note that only the Y-coordinates are forced; letting the X-coordinates free is needed to correct the distortion. This means the pixels will tend to move away from the center, and not necessarily on a vertical "motion".

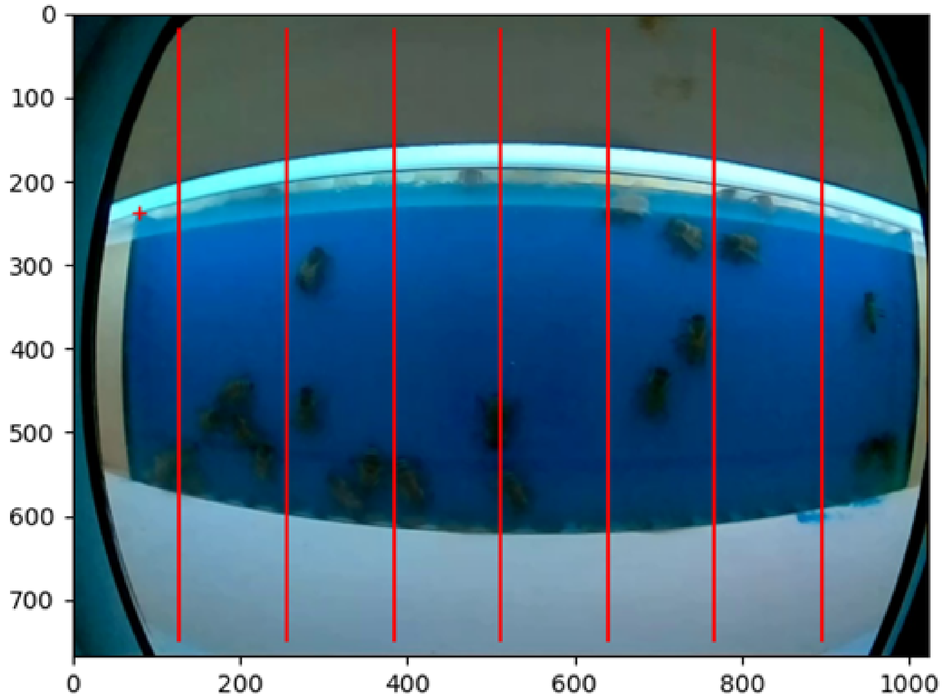


Figure 1.4: picture with guidelines for the selection of 18 points (14 intersection plus the 4 corners)

Having 5 unknown variables means we could theoretically find the distortion coefficients with a 5 equations system. This would mean the operator would have to select only five points. However, this approach didn't bring satisfying results, and more points had to be selected leading to have more equations than unknown variables. In order to find a result, the least-squares solution was used to find the result fitting approximately all the equation.

Now that the distortion coefficients are found, the picture is undistorted. The result, albeit far from perfect, shows much less distortion on the bee's circulation space and is clear enough to be used for the following works. (Figure 1.5)

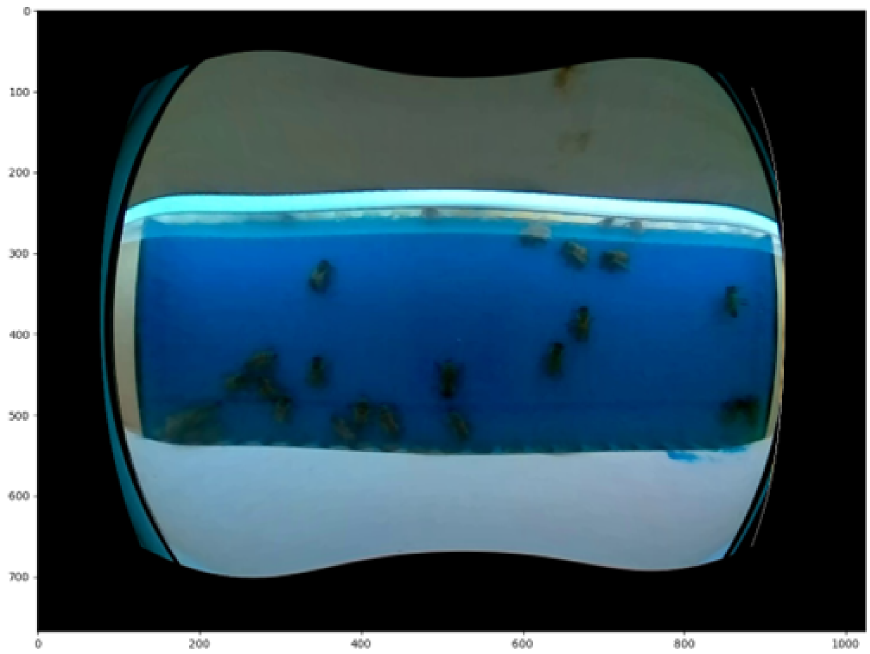


Figure 1.5: picture undistorted with the program

Chapter 2

Segmentation

2.1 Segmentation with blue background

Once the deformation of the image has been processed (Figure 2.1), it is necessary to make the segmentation of the image in order to differentiate the bees from the background. Succeeding in distinguishing between bees is also necessary to track of each of them.

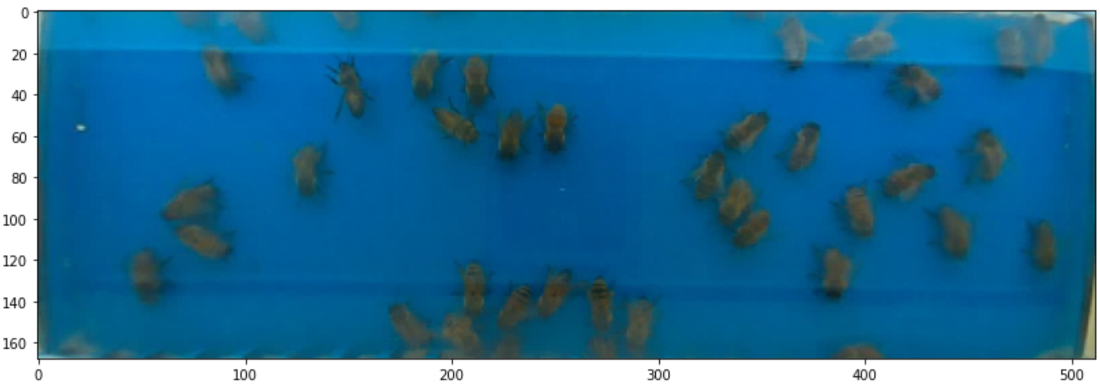


Figure 2.1: Picture img2 obtained after undistorting

The pixels are compared to the background colour. This colour is approximated by the variable bgd: a BRG vector which is the average of the BRG vectors of all the pixels in the image. The comparison is made via a cross product after normalization of the BRG vector. This normalization makes it possible to limit the influence of the luminosity, which varies with the distance from the edge of the box. A "res" grayscale image is formed (Figure 2.2), for which each pixel takes the value resulting from this calculation:

$$res[y, x] = 1 - np.dot(normalize(img2[y, x]), normalize(bgd)) \qquad (1)$$

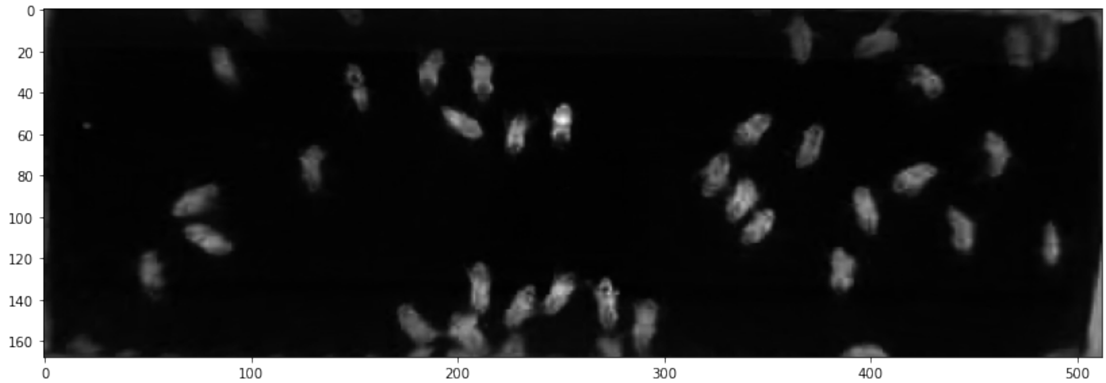


Figure 2.2: Picture res showing elements distinct from the background

In this image, some bees stand out sharply, while others are more difficult to distinguish. In order to make all the bees present as visible as each other, the pixels are compared to a variable threshold (adaptive thresholding). For this an integral image is used (Figure 2.3). This is a gray level image whose pixel value in (x,y) is calculated according to the formula:

$$Integral[x, y] = res[x, y] + res[x-1, y] + res[x, y-1] - res[x-1, y-1] \qquad (2)$$

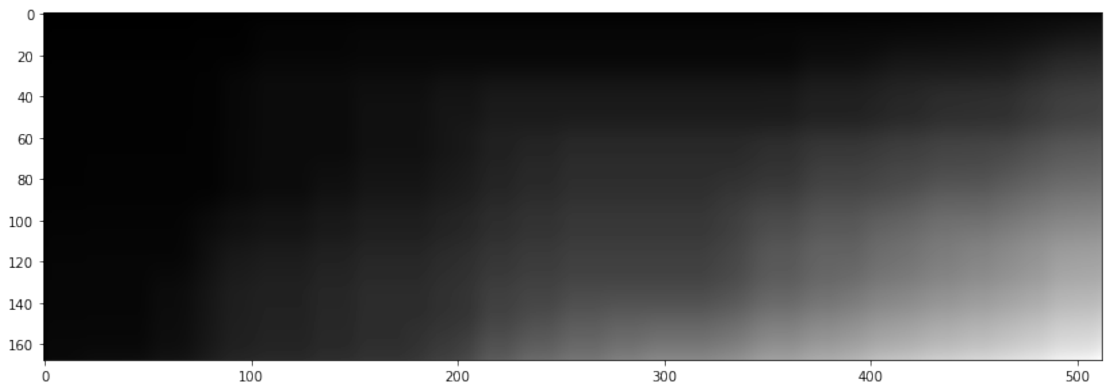


Figure 2.3: Integral picture

A stencil of 11x11 size is applied to “integral” in order to bring out the shapes of the bees present in the initial image (Figure 2.4). We then obtain a “res2” image in gray level where the bees form spots quite distinct from the background (Figure 2.5).

$$\begin{bmatrix} 1 & 0 & \dots & -1 \\ 0 & 0 & & \\ \dots & & \dots & \\ -1 & & & -1 \end{bmatrix}$$

Figure 2.4: Stencil applied to the integral picture

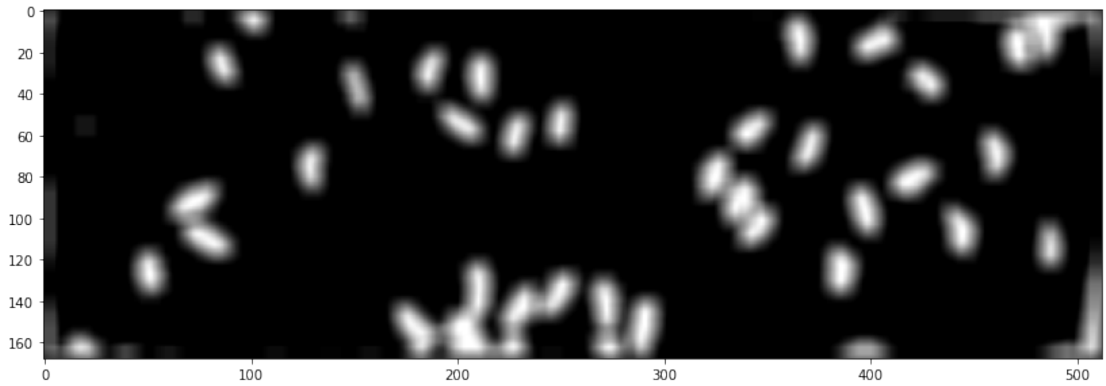


Figure 2.5: Picture res2 ; homogeneous area are more clearly outlined

Finally, the “res2b” image is formed by applying a 5x5 stencil to “res2”; it keeps only the pixel with the maximum value and sets the values of the other pixels to 0. “res2b” only has a few distinct points, which will be used to determine the position of the bees (Figure 2.6).

Once the image has been processed, we seek to determine the position of each bee in the image. A list is created, to which the coordinates of the pixels having a non-zero value in "res2b" are added, in descending order of value. If a pixel is too close to another already present in the list, then the coordinates of the new pixel are not added to the list. To do this, a square with a side of 20 pixels (a bounding box) is assigned to each of the two pixels to which we want to know if they are

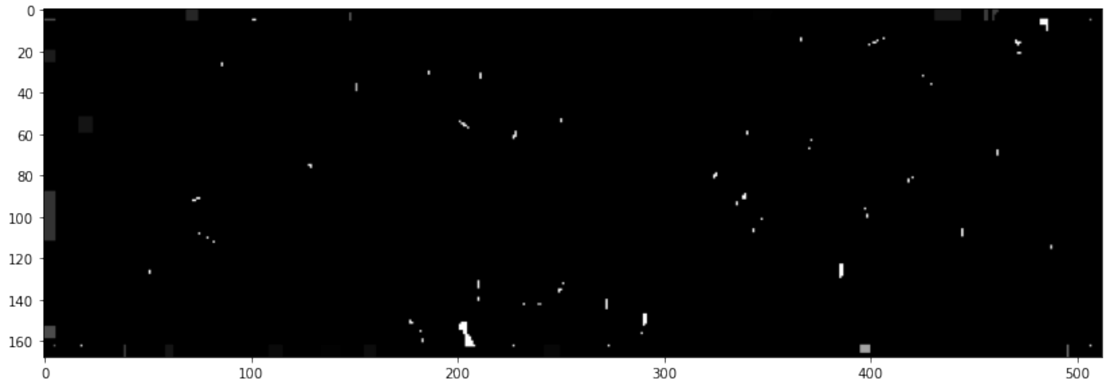


Figure 2.6: Picture res2b, mapping local maxima of res2

too close; then the Intersection over Union (IoU) is computed between these two squares (Figure 2.7).

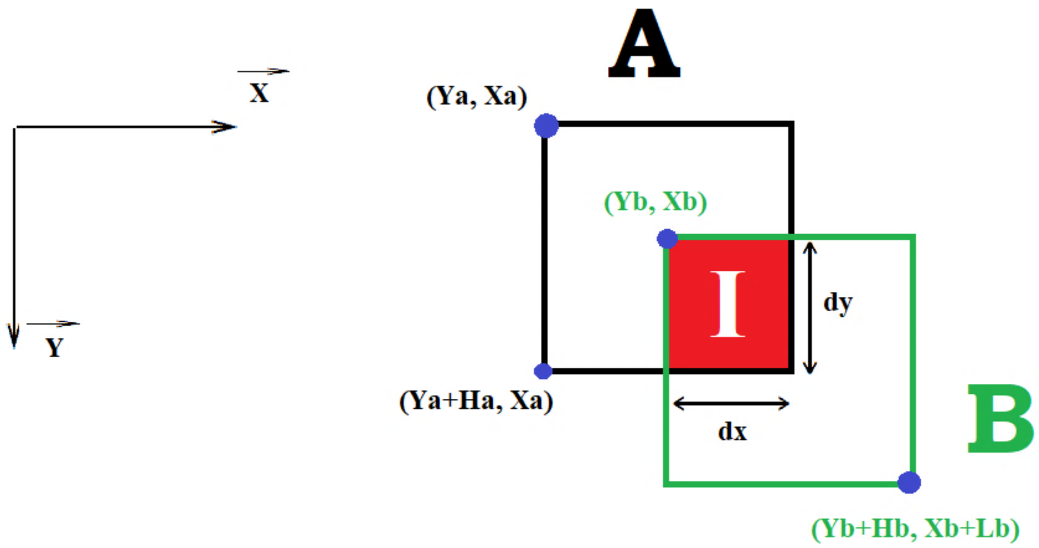


Figure 2.7: Illustration of the intersection between two bounding boxes. Beware, axis Y is oriented downward

A position vector P is associated with each bounding box according to the following format: P = [y, x, h, l], where y and x are the coordinates of the upper left point of the bounding box, h is its height and l its length. H and l can have a value other than 20 to prevent the bounding box from going out of the image.

The IoU of two bounding boxes A and B is computed as follows:

$$dy = min(y_a + h_a, y_b + h_b) - max(y_a, y_b) \qquad (3)$$

$$dx = min(x_a + l_a, x_b + l_b) - max(x_a, x_b) \qquad (4)$$

$$I = dx * dy(5) \qquad (5)$$

$$U = h_a * l_a + h_b * l_b - I \qquad (6)$$

$$IoU = \frac{I}{U} \qquad (7)$$

Obviously, the calculations (5) to (7) are only performed if an intersection exists; that is, if dx and dy are strictly positive.

If the IoU is superior to a determined threshold, the pixels are considered too close, and the new coordinates are not saved.

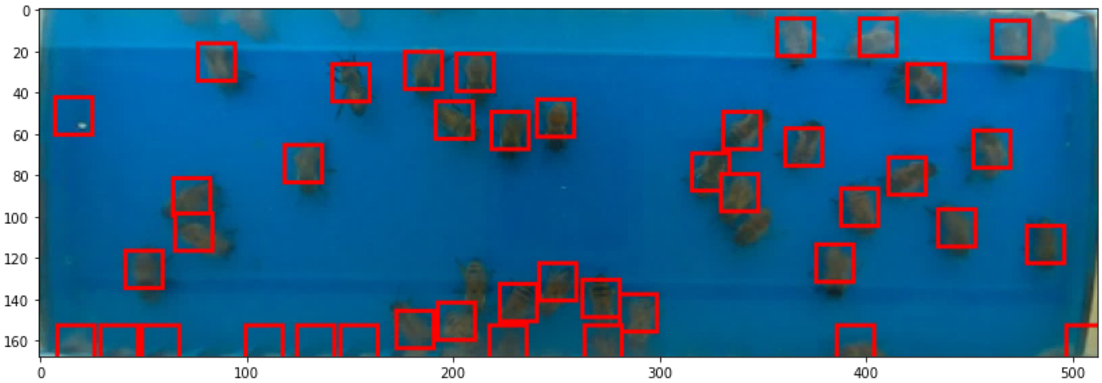


Figure 2.8: Picture obtained after bees' position identification on blue background through colour comparison

In figure 2.8 we can see the result obtained. Most bees have been recognized; the presence of the edges of the box in the image generates several false identifications. However, these artifacts are not a problem when analysing bees crossing the box.

2.2 Segmentation with white background

Since the videos obtained during the internship had a white background (Figure 2.9), the segmentation of the image had to be reworked.

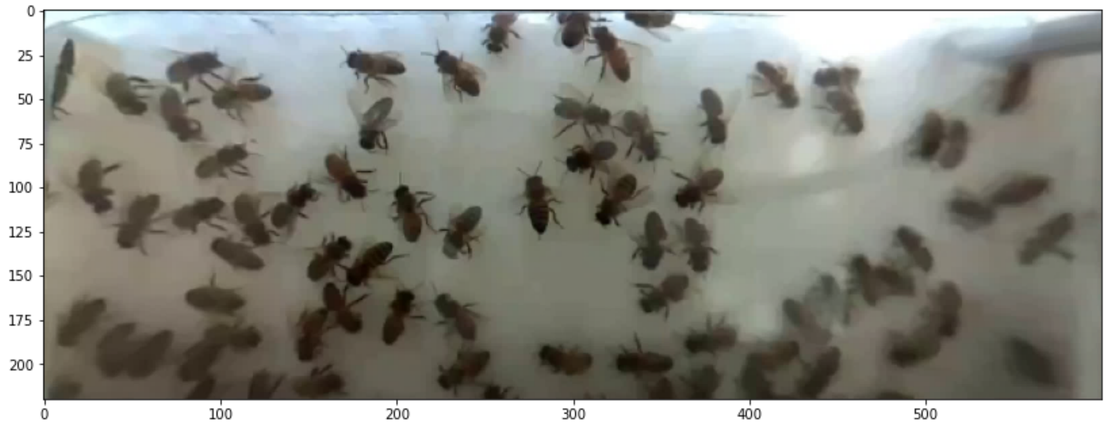


Figure 2.9: Picture distorted

Colour segmentation was performed following the same procedure. We note here that the use of a variable threshold makes it possible to bring out forms (Figure 2.11) that are almost invisible otherwise (Figure 2.10).

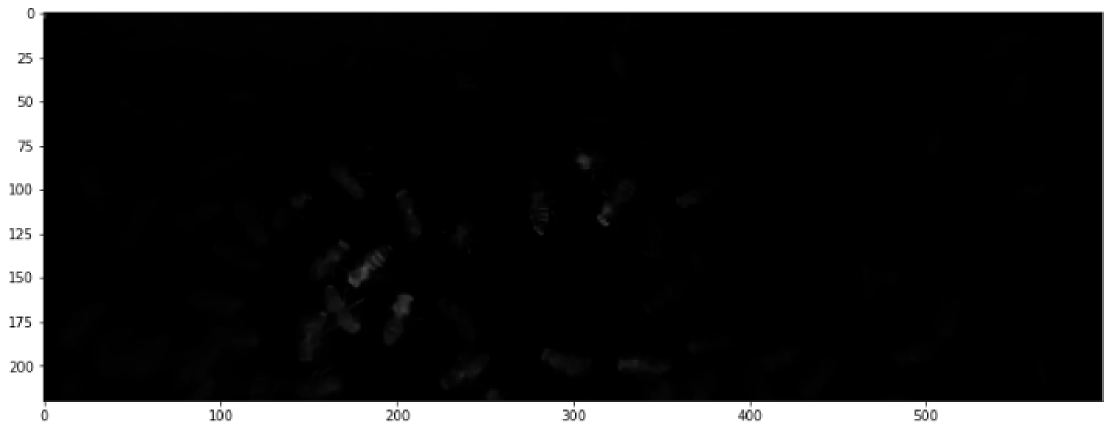


Figure 2.10: Picture res showing elements distinct from the background

Here the colour segmentation shows its limits: where reflections were present, the algorithm fails to distinguish the bees (rather gray on the image, therefore close to white) from the background (Figure 2.12, see top right).

Brightness seems to be a more appropriate criterion to segment the image in this case. Brightness is calculated as the average of the BRG values of a pixel.

A first test is made by whitening all the pixels whose luminosity is above a certain threshold (Figure 2.13). This threshold is relative to the average brightness of the image so that it is relevant all day.

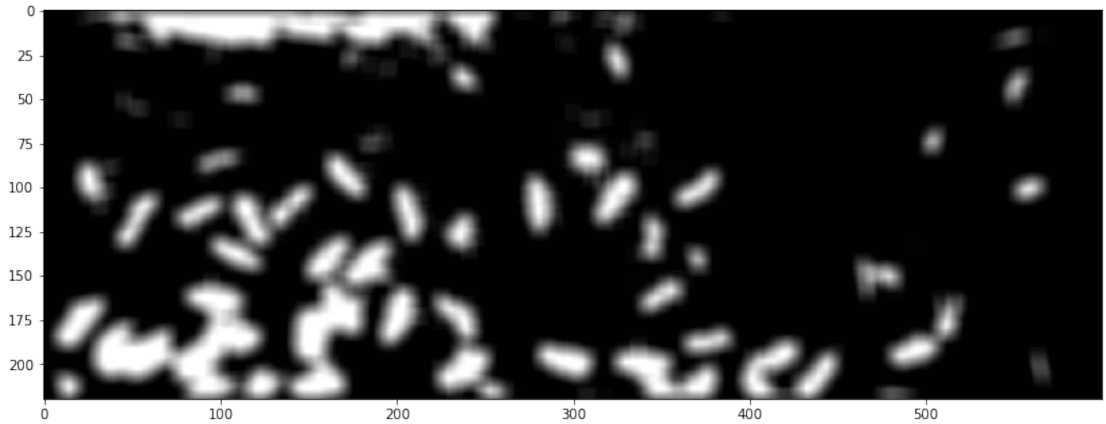


Figure 2.11: Picture res2

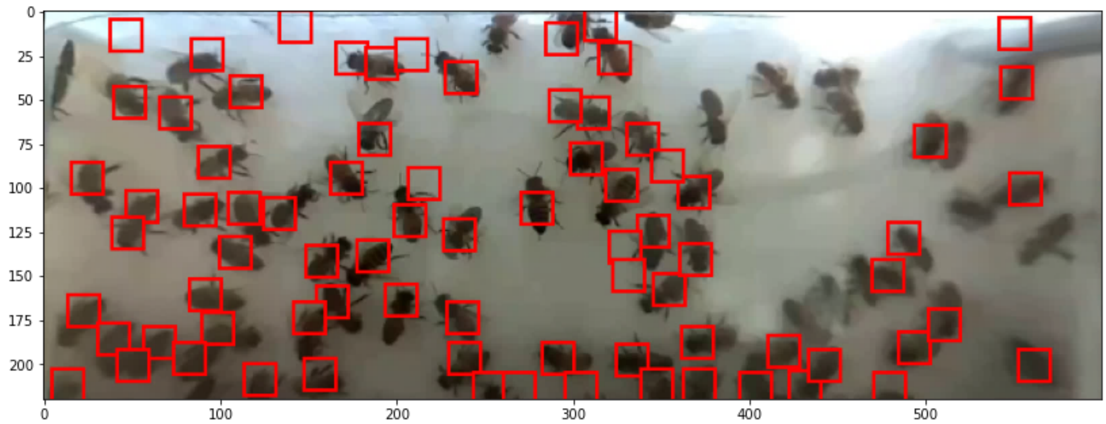


Figure 2.12: Picture obtained after bees' position identification on white background through colour comparison

However, choosing a single threshold for the whole image does not make it possible to obtain a correct segmentation: in the luminous zones, close to the exterior, some bees are associated with the background; while in dark areas the background is associated with bees.

In order to limit this problem, different segmentations on portions of images have been performed. Cutting the image into three horizontal bands for segmentation provides the best result. (Figure 2.14, see appendix B)

The procedure is then the same except for one detail: instead of looking for local maxima in the "res2" image (Figure 2.15), we look for local minima (Figure 2.16).

The segmentation is better, despite some bees still "invisible" due to a reflection on the plexiglass (Figure 2.17). Another flaw is the presence of duplicates: two

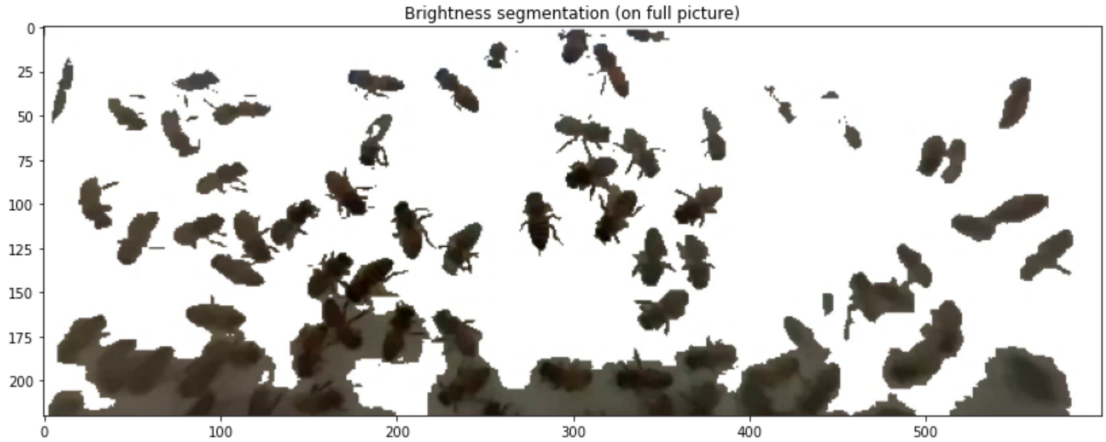


Figure 2.13: Picture obtained through thresholding the original picture with a single luminosity threshold

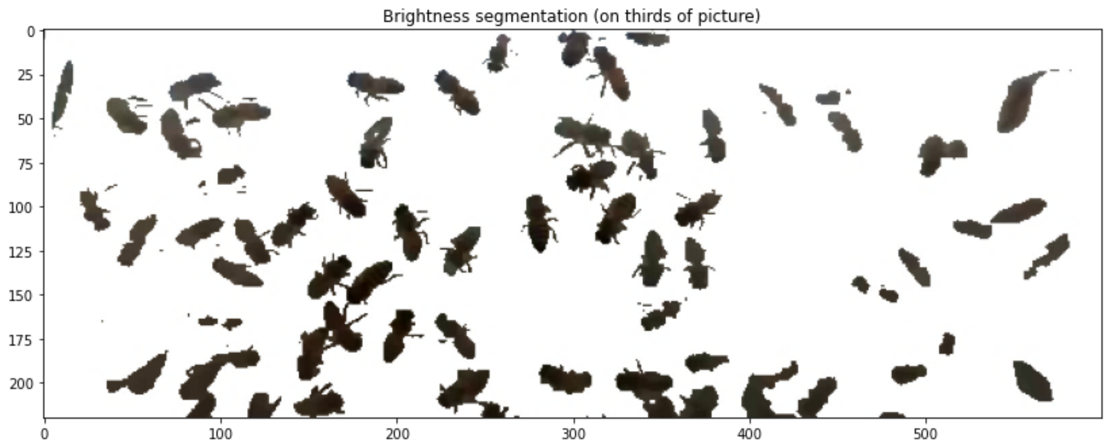


Figure 2.14: Result of a thresholding using three different thresholds

squares are sometimes assigned to the same bee. This problem is solved by changing two parameters: the size of the stencil applied to res2 to find the local minima and the size of the bounding boxes associated with the bees.

A series of analyzes was carried out with stencils ranging in size from 11x11 to 21x21, and bounding boxes from 20 to 40 pixels per side. The number of bees recognized indicates the most promising results; for example, there are 69 bees in this image. The best parameters are then chosen by checking the number of duplicates on each picture.

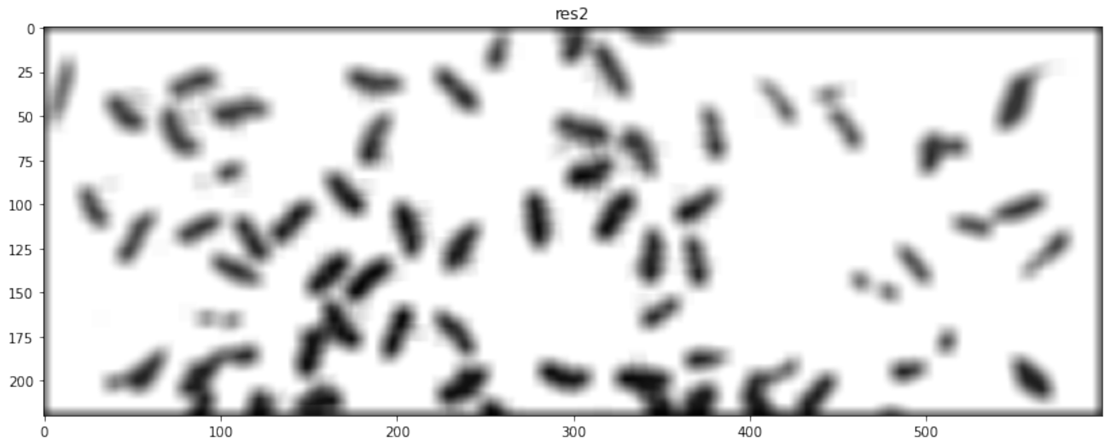


Figure 2.15: Picture res2

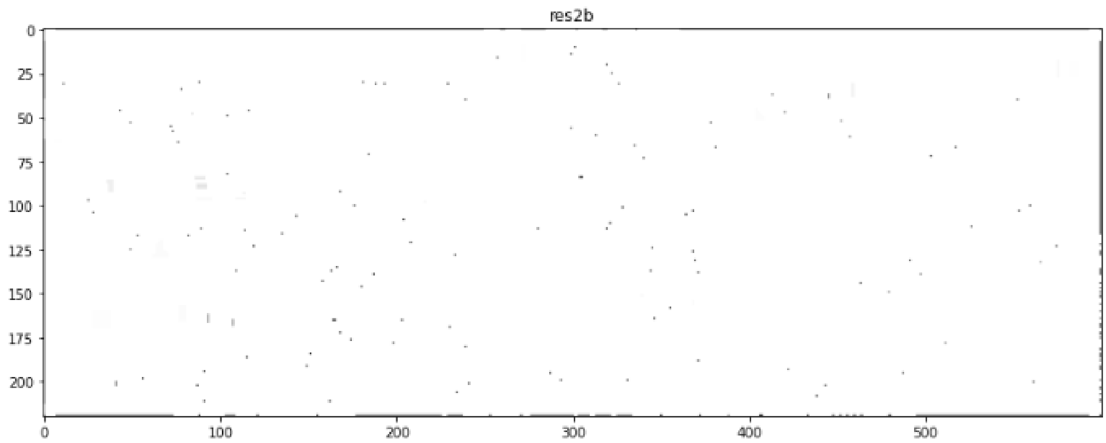


Figure 2.16: Picture res2b, mapping local minima of res2

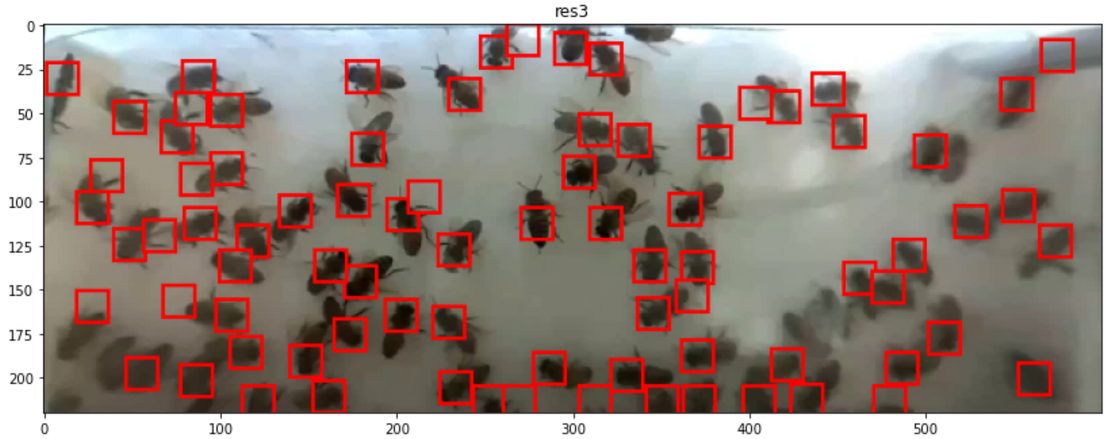


Figure 2.17: Result of bees' position identification on white background through luminosity thresholding

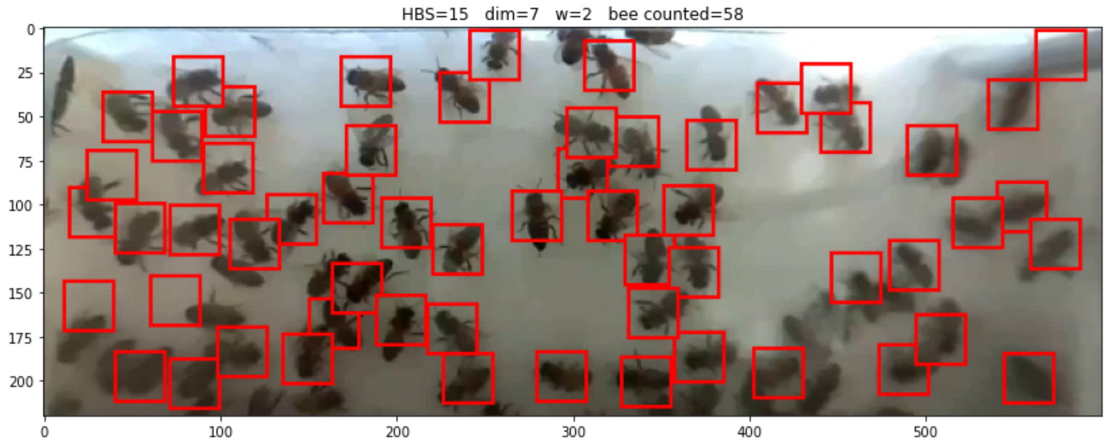


Figure 2.18: Result after a better choice of parameters

Chapter 3

Bee tracking

Once the position of each bee has been defined on a picture, it is necessary to follow the animal from one picture to another.

A "tracklet" list is created to contain the coordinates of the bees on several pictures. By analogy with a matrix, each row of the list contains the coordinates of the bounding boxes of a single bee, and each column n contains the coordinates of each bee on the nth image following the appearance of this bee in the field of view of the camera. The lines do not all have the same length, hence the use of a list (see appendix C).

The "tracklet" list is initialized with the coordinates of the bounding boxes of the bees of the first picture obtained. For subsequent pictures, the coordinates of each bee on the new picture are compared to the coordinates saved in the list when analyzing the previous image. If a bee is close enough to a recorded coordinate, the new coordinates are recorded on the same line after the old close coordinates (it is a bee continuing on its way). If, on the contrary, no coordinates correspond, a new line is created: it is a bee that has just entered the field of view of the camera.

Finally, if a bee leaves the field of the camera, its last coordinates recorded in the list will not correspond to any new position: in this case the line is deleted.

It may also be necessary to know if a bee should be analyzed (to classify it for example). A binary variable carrying this information has been added to the vector containing the first coordinates of the bees. It is worth 1 if the bee must be analyzed (typically if it comes from outside the hive) and 0 otherwise (the bee comes from the hive or has already been analyzed).

Example of figure 3.1: the tracklet list contains the positions of 12 visible bees in two pictures. The coordinate vectors are composed of the coordinates of the upper left point of the bounding box, and the length of its sides.

```
In [21]: track.tracklet

Out[21]: [[[327, 0, 20, 17, 1], [335, 10, 20, 16]],
          [[328, 51, 20, 20, 1], [330, 46, 20, 20]],
          [[306, 78, 20, 20, 0], [308, 72, 20, 20]],
          [[85, 77, 20, 20, 0], [84, 78, 20, 20]],
          [[360, 0, 20, 18, 1], [356, 0, 20, 19]],
          [[90, 98, 20, 20, 0], [88, 99, 20, 20]],
          [[228, 88, 20, 20, 0], [228, 94, 20, 20]],
          [[64, 95, 20, 20, 0], [62, 96, 20, 20]],
          [[129, 86, 20, 20, 0], [131, 87, 20, 20]],
          [[132, 15, 20, 20, 1], [133, 14, 20, 20]],
          [[165, 109, 20, 20, 0], [167, 109, 20, 20]],
          [[100, 109, 20, 20, 0], [104, 109, 20, 20]]]
```

Figure 3.1: example of a list containing bees' coordinates

Chapter 4

Dataset creation

The objective of this thesis was to make an algorithm allowing the classification of bees into two categories: those carrying pollen and those that do not.

Training an algorithm requires having a set of labeled data; a preliminary human work is therefore necessary. In order to facilitate this work and to obtain a set of data similar to those that the algorithm will have to process, a program has been written by applying the image processing and analysis seen previously to a video file.

A first simple approach was to record a bee image as soon as an animal was recognized. The bee image has a size similar to its bounding box, 20x20 pixels for the first video obtained, on a blue background.

The operator then had to indicate whether the bee was transporting pollen (positive case), or not (negative case). The program interface is shown in Figure 4.1. The program saved the image in a folder and under a name appropriate to the class of the bee.

It was also possible that the image was not clear enough to determine if the bee was carrying pollen, or that the recorded image did not contain a bee (the edges of the box being white, the colour segmentation separated them from the blue background). In this case, the image was deleted.

This approach had some flaws. First of all, since each bee is present several times in the dataset, the dataset is probably not representative of the observed population. Moreover, the deformation of the image not being perfectly compensated (in particuliar at the edges), keeping only the images of the bees close to the middle of the image allows to have a "cleaner" data set. The most important thing, however, is to have data close to what the algorithm will have to classify. Finally, the recorded images contain bees leaving the hive, which have no interest in being classified by the algorithm. Including them in the data set is therefore a waste of time for the operator.

A second program is created to avoid the flaws of the first. This time bee tracking is used so that each bee coming from outside is only “photographed” once, when it passes from the upper half to the lower half of the image. Images of bees leaving the hive are never recorded.

Two datasets are created by this program: one containing images of all the bees entering the hive, and the other containing as many images of bees carrying pollen as images of bees without pollen. Having a set in which both classes of bees are equally represented allows for better training of several classification algorithms.

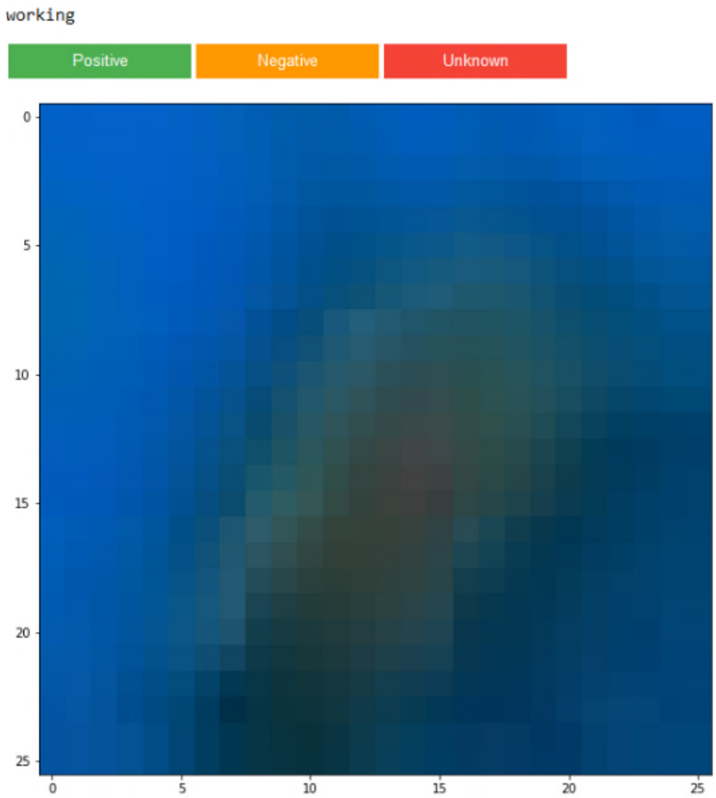


Figure 4.1: interface of the dataset creation program

Chapter 5

Classification

Classification consists of assigning a bee image to a class. In our case, two classes exist: the bees transporting pollen (positive cases) and those which do not transport it (negative cases).

Classification is done using an algorithm trained on a set of data; the first We'll Bee corporate videos were produced late, so we were unable to use our own datasets. A set of images produced by Ivan Rodriguez and associates, from the University of Puerto Rico [4], was used. This set contains 714 images, divided into 345 negative and 369 positive cases. This dataset was chosen because the images were similar to what was expected from images produced by We'll Bee: bees of the species Apis Mellifera had been photographed against a blue background (Figure 5.1).

However, there are two notable differences with the expected images. The resolution of the images of the set [4] is 180x300 pixels, much higher than the expected 20x20 pixels (Figure 5.2). Moreover, in this set all the bees have a close orientation (almost vertical); however, the orientation of the bees during the analysis of the videos are lot less regular (hence the use of square images).

It is customary not to use the entire data set to train an algorithm: this method presents the risk of not being able to see if the algorithm overfits the data on which it is training . It then makes an analysis that is very suitable for the training data but generalizes very poorly.

We prefer to separate the dataset into two portions (not necessarily equal): the training set (whose name clearly explains its use) and the verification set which is not used during training but after: it is used to check that the analysis performed by the algorithm can be generalized to other data.

The dataset was separated into a training set of 535 images and a verification set of 179 images. The split is done randomly using the train_test_split() function from the scikit-learn library, but the sets are the same for all tested algorithms.

Several types of algorithms for performing classification have been tested to determine the most efficient one.

Figure 5.1: a positive (left) and a negative (right) cases from the dataset [4]

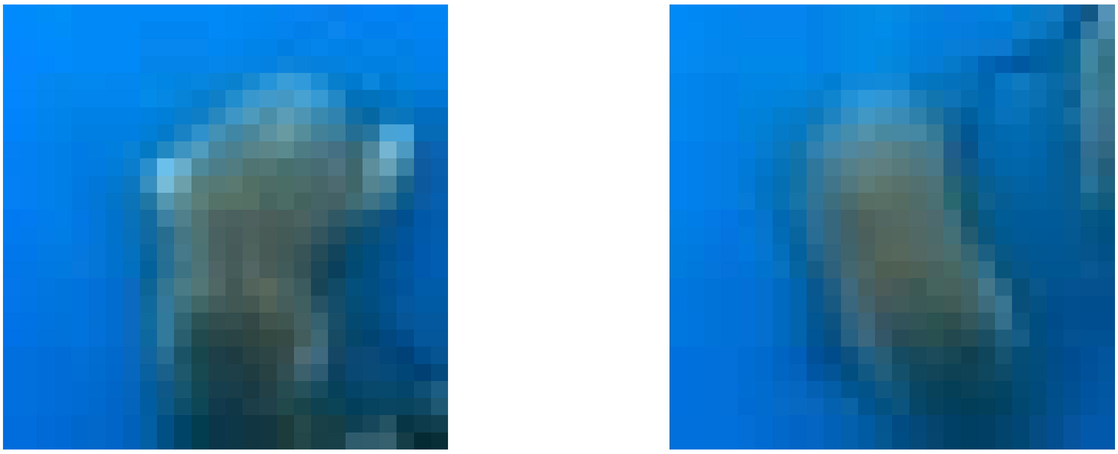


Figure 5.2: a positive (left) and a negative (right) cases obtained with the observation system

A first approach was to target the areas in the images of bees where pollen would possibly be visible. We hoped that this approach would eliminate noise and allow us to focus only on the important points of the images.

A "random forest" type algorithm was trained. It is an algorithm composed of several decision trees, each tree being formed of decision nodes. Each tree predicts a class for an analyzed image and the result of the "random forest" is the prediction given by the majority of the trees.

This algorithm succeeds in correctly classifying the images of the verification set in 84% of cases. Although this result is not extraordinary, it guarantees that the algorithm does not overfit.

The "random forest" type of algorithm was chosen more to determine the areas of interest in the image than to be applied directly to the classification problem. While it is difficult to understand the precise operation of the "random forest" in its entirety, one can obtain what information makes it possible to distinguish the two classes of bees most effectively. Each node is associated with a gini impurity index: the purer the groups at the exit of the node (the positive cases are less mixed with the negative cases), the lower this index is. A data importance is calculated from these indices to favor the data allowing an accurate classification.

A mapping of the importance of the pixels and their color channels was carried out: the clearer the pixels appeared on the mapping, the more they were useful to make a clear distinction between the two groups of bees (Figure 5.3).

The largest groups of pixels are at the bottom of the images, on the sides. This is the general location of the hind legs of bees, where pollen is accumulated if there is any. The right side seems more important than the left side, probably because of the lighting of the bees whose left side is more illuminated for this dataset. A third area, located in the upper third of the image, is also of some importance for unknown reasons.

We also observe a large disparity in the importance according to the color channels: the blue values are generally more important than that of red, while the importance of the green values is very small and seems to be randomly distributed in the image.

A study [1] indicating that brightness and color variance could be used to classify bees, these two quantities were measured for each image of the training set.

The color of the pixels is defined by a vector c = [B, R, G], where B, R and G successively indicate the intensity of the blue, red and green channel. The average color of an image i is µi = [µb, µr, µg]

The variance V of the colors of an image i is defined as being the sum of the variances v of its pixels:

$$v(i, y, x) = E((c - \mu_i)^2) \qquad (8)$$

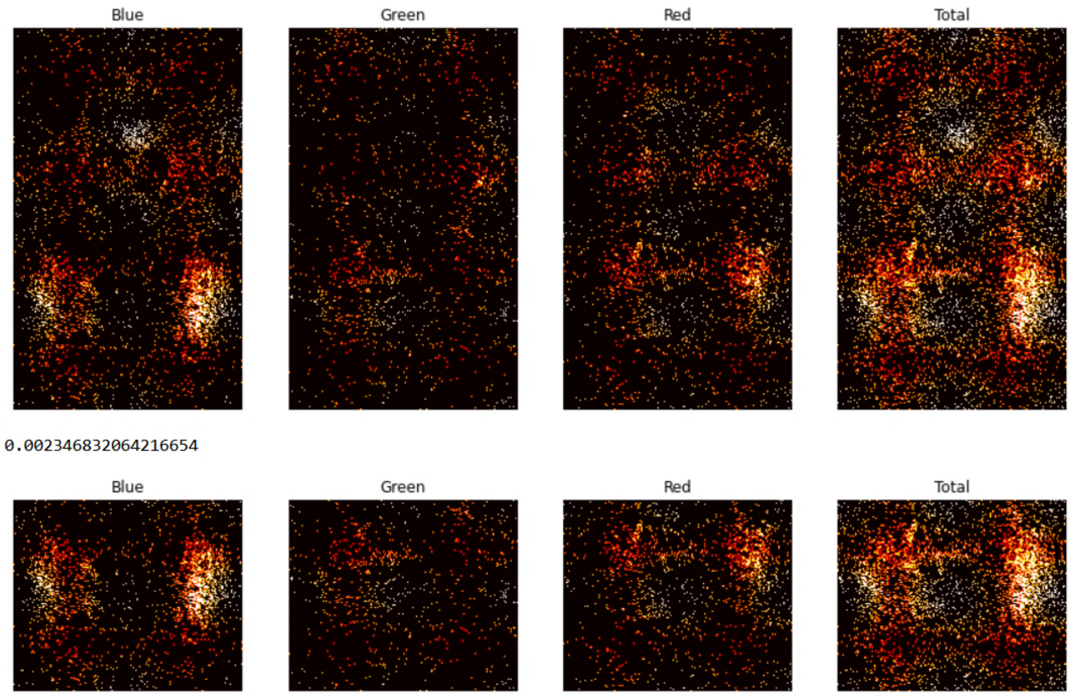


Figure 5.3: pixel importance mapping according to the random forest algorithm for each colour, and the sum of the three

$$V(i) = \sum v(i, y, x) \qquad (9)$$

There is a very slight difference between the two groups of bees; positive cases appear to have less color variance. However, the groups remain indistinguishable according to the sole criteria of luminosity and color variance.

5.1 SVM

Support Vector Machines (SVM) are a family of algorithms with a reputation for being both fast and suitable for binary classification (where only two groups exist).

An SVM seeks the hyperplane discerning the two groups with the largest possible margins. However, if the two groups are intermingled (as in figure 5.4) it is impossible to find a satisfactory hyperplane. We can then authorize certain transgressions of the margins: we speak of “soft margin”. An adimensional hyperparameter C establishes the acceptable degree of transgression: the larger C is, the more “rigid” the margins are. A hyperplane better separating the training data is usually found, but there is no guarantee that the boundary thus found does not generalize well.

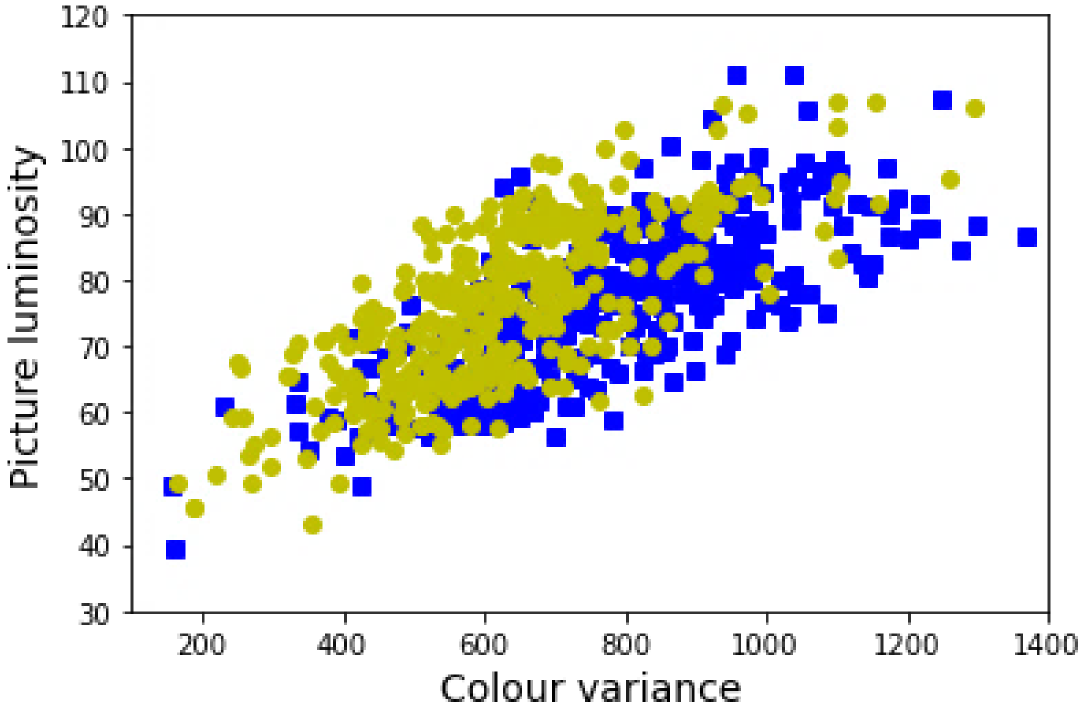


Figure 5.4: distribution of positive cases (yellow) and negative ones (blue)

Conversely, the weaker C is, the less the particular cases have importance; the risk here is to ignore an entire sub-group.

SVMs were trained by taking into account three data: brightness, color variance, and the average intensity of the red color in the lower half of the image. In order to find the best model different SVMs are trained with a C hyperparameter ranging from 10^{-12} to 10^{12}; then the accuracy of each model is calculated with the verification set.

The results are very disappointing: the best precision obtained is 63.7% for C=0.01.

This imprecision shows the shortcomings of the approach aimed at targeting precise areas and characteristics of the images: too much information had been eliminated to allow an effective classification.

This approach has therefore been abandoned, and the entire images are now provided to the algorithms. Other SVM models are then trained: an accuracy of 82.7% is achieved this time.

In order to further improve accuracy, various techniques were used to remove background noise from images.

A first technique consists in blackening the pixels where the blue channel (background color) is predominant (Figure 5.5). Accuracy rises to 83.8%.

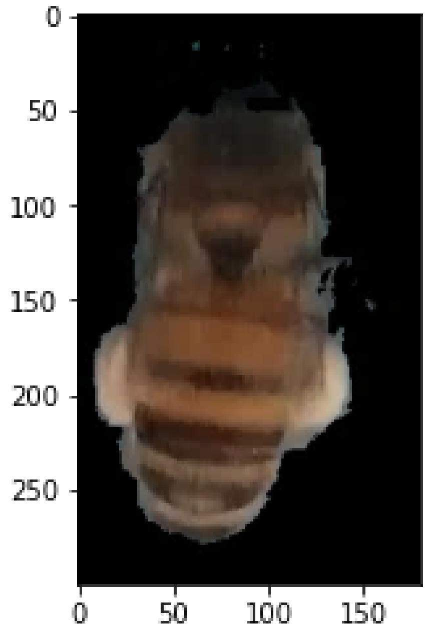


Figure 5.5: Example of a picture preprocessed by blackening pixels for which blue is the most important colour

A second technique is to use the proximity of a pixel's color to the background (as in color segmentation) to choose which pixels to darken. Surprisingly this slightly decreases the accuracy, which reaches 81.5%.

A last technique is tried, consisting in not transmitting the blue channel to the SVMs. Accuracy barely decreases, down to 82.1%. This result relativizes the importance of the blue channel, which nevertheless seemed to be the most useful color for an effective classification according to the analysis of the random forest.

These data processing only slightly affect the accuracy of the model, but demonstrate that an improvement in the classification remains possible. If the first technique seems better, it will still be necessary to test these techniques again on the data that we will produce to check their effect.

5.2 K-Nearest-Neighbourgh

Another model that looks promising is the K-Nearest-Neighbourgh (KNN). This algorithm keeps in memory the training data and classifies the images according to the class of their N nearest neighbors.

N is a parameter chosen by the programmer. In order to find the best parameter, several models are trained, with N ranging from 1 to 19. The optimal N seems to be 11, providing an accuracy of 83.8% (Figure 5.6). The precision drops for N greater than 17, indicating that a search beyond this value would be superfluous.

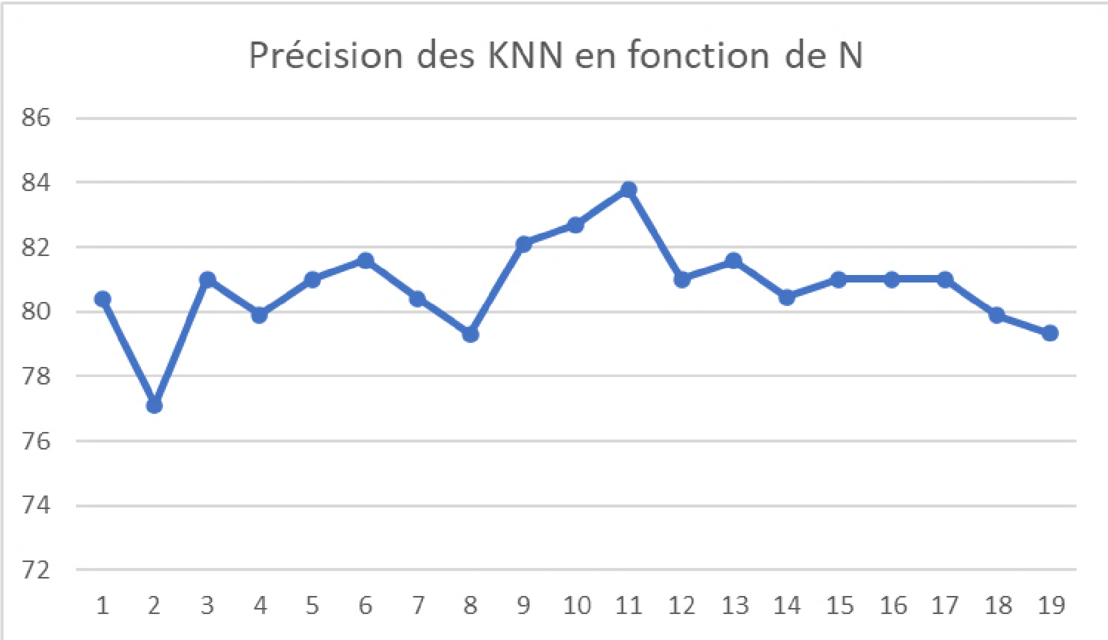


Figure 5.6: Precision reached by KNN algorithm as a function of N

5.3 Artificial Neural Network

Artificial Neural Networks (ANN) are models composed of several units called neurons organized in layers.

A neuron generally receives values that it combines before processing them through an activation function: the result of this function is the value that the neuron will transmit to the neurons of the next layer. Several activation functions exist, such as sigmoids, ReLU...

The first layer receives the raw data given to the model, while the last must provide values that can be easily interpreted. In the case of a binary classification, the last layer is composed of a single neuron. If the value returned by this neuron exceeds a threshold, the analysed picture is considered to be a positive case; otherwise, the picture is a negative case.

Many architectures are made possible thanks to the modularity allowed by neurons: it is possible to organize a model in many layers of neurons (deep learning), to pass certain information directly to the last layer (deep and wide, Figure 5.7). Finally, there are layers of neurons created for specific purposes.

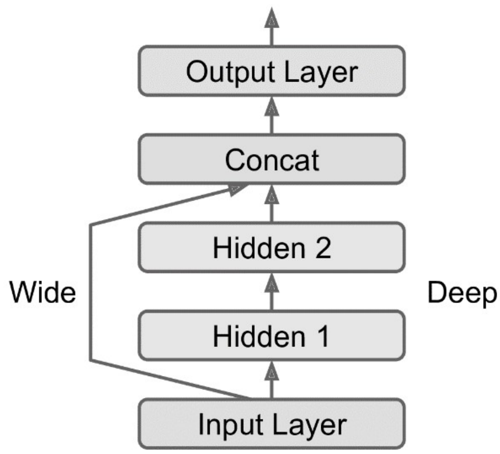


Figure 5.7: Example of a Deep & Wide architecture (source : [6])

Training of the model is necessary so that it can give correct results. The neurons combine the data received by assigning them different coefficients, and the training of the model aims to optimize them.

The training takes place in several phases, called epoch. At each epoch, the entire training set is analysed by the model, as if it were to classify the data. A backpropagation algorithm is then used to determine which coefficients caused the model to make good or bad predictions. These coefficients are then modified by an optimization algorithm in order to correct the errors of the previous epoch.

5.3.1 MLP

The first model tested on our classification problem is a Multi-Layer Perceptron (MLP). This is a relatively simple model: successive layers of neurons relay information in order to bring out relevant features of the picture so that the last layer can determine the class of the picture.

In an MLP, each neuron is linked to all the neurons of the previous layer: the layers are called “dense” (Figure 5.8).

A simple architecture was chosen: between the first layer and the last there are N layers composed of n neurons each.

Before building the model, it is interesting to simplify the data by reducing their dimensions. Indeed, for pictures of 180*300 pixels with 3 colour channels, we have to process data with 162,000 dimensions. The technique of Principal Component Analysis (PCA) is used to reduce the dimensions of our data by keeping the main information allowing to differentiate the data between them. Reducing the number

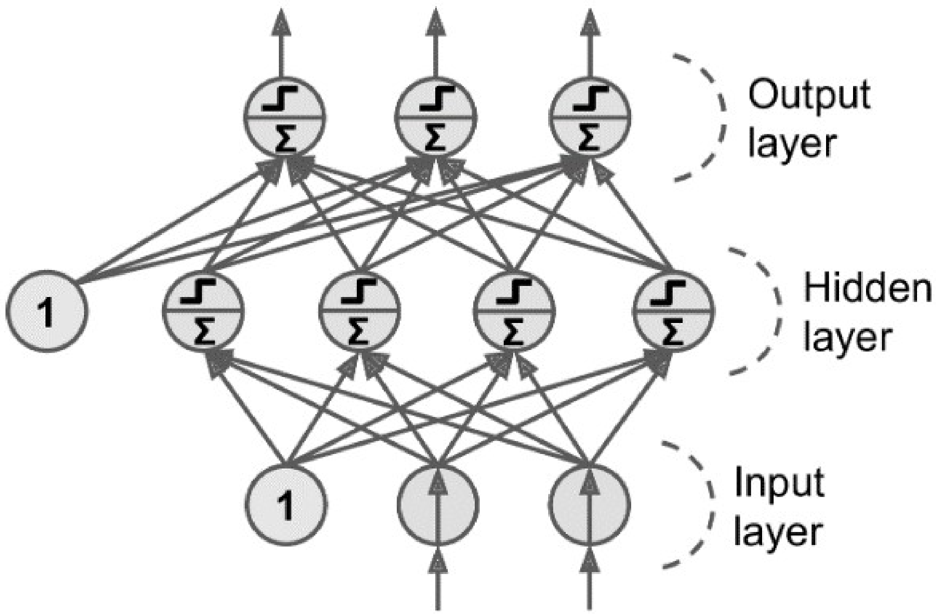


Figure 5.8: Diagram of an MLP architecture (source : [6])

of dimensions makes it possible to train an algorithm more quickly; moreover, a simpler architecture can be used, thus reducing the image analysis time (despite the time required to transform each image).

PCA assumes that the data is not equally distributed in the space of possible pictures with a resolution of 180*300 pixels, but that it is close to a hyperplane: by projecting the data onto a good hyperplane, it is possible to reduce the number of dimensions by losing little information (Figure 5.9).

The best hyperplane is chosen based on the variance of the training set preserved by the projection. It is possible to reduce the dimensions of our data several times until reaching a chosen fraction of the original variance.

A PCA was applied to the training set, with the limit of keeping 95% of the original variance (Figure 5.10). The transformation thus obtained makes it possible to reduce the number of dimensions of our data from 162,000 to 229. This drastic reduction is possible thanks to the great similarity between our images: a lot of information is redundant or useless (the blue background on the edges of the image. . .).

For information, here is an image extracted from the set [4] and its reconstruction after PCA (Figure 5.11). We notice that the bag of pollen remains visible, despite the alteration of the image

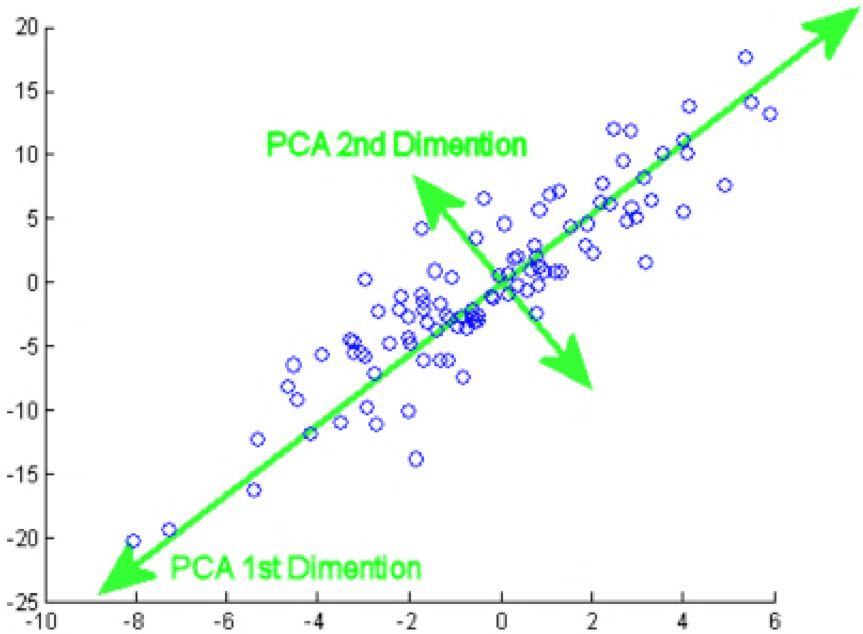


Figure 5.9: Illustration of a PCA: the data is not randomly distributed in the plane, and can be projected onto the first dimension with little loss of information (source : https://programmathically.com/)

A check is made to verify that the PCA does not affect the accuracy of our algorithms too much. An SVM is trained with the training set transformed by the PCA. Its accuracy is the same (82.7%) as the SVM trained on the untransformed set; the PCA does not imply a notable loss of precision in our case.

In an MLP several hyperparameters can be tweaked: the number of layers, the number of neurons they contain, but also the learning rate (lr). The latter affects the way the model is optimized: the larger it is, the more the parameters of the model will be modified at each epoch. A model trained with a high learning rate will therefore tend to converge quicker but will also be quick to miss the optimal solution by oscillating between two neighbouring solutions.

Keeping a fixed learning rate is not the most efficient way to optimize a model. It is preferable to use a large learning rate for the first epochs to quickly approach the optimal solution, then to gradually reduce the learning rate. However, finding a good fixed learning rate is a necessary step to find the maximum learning rate with a regressive learning rate [6].

Also, since the data used is not what will be produced by the hive observation system, the goal here is to get an order of magnitude of hyperparameters to use on

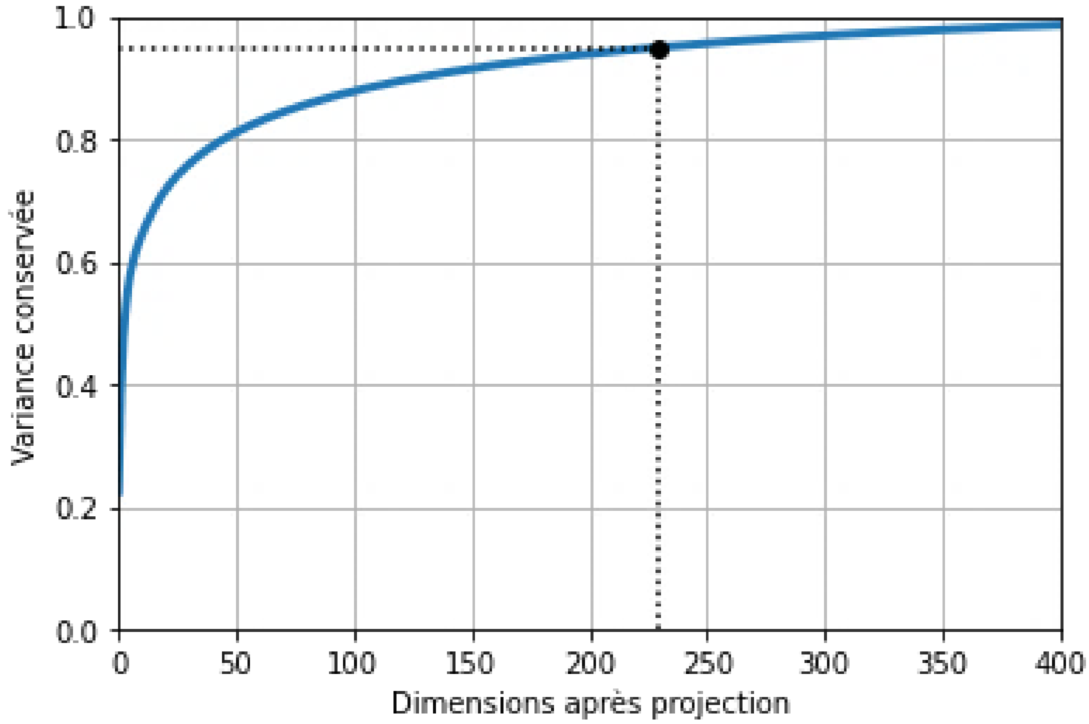


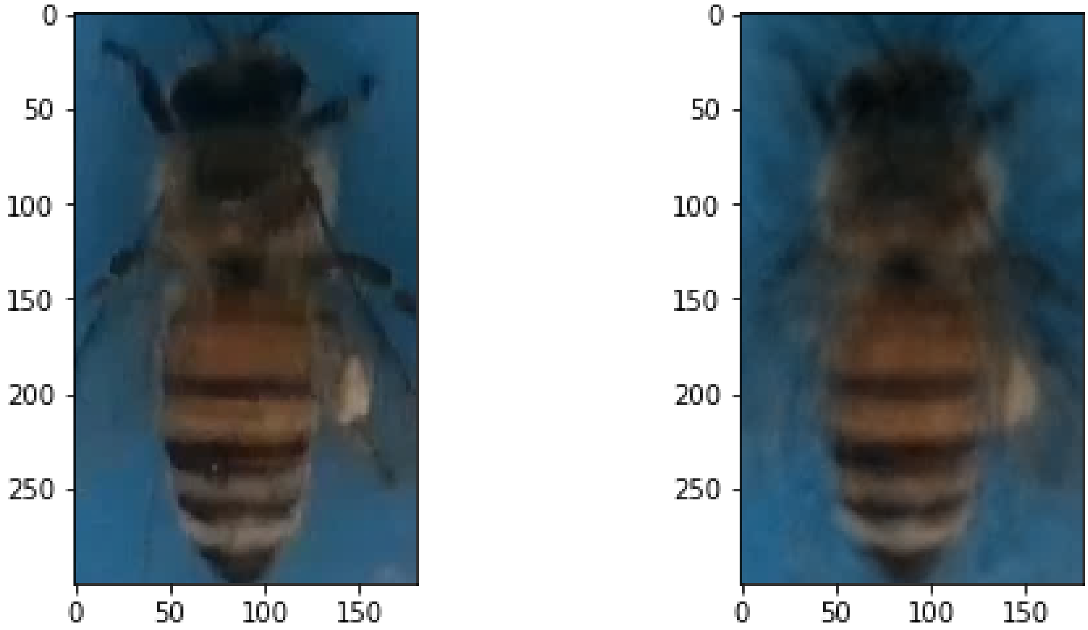
Figure 5.10: Graph of the maximum variance conserved by the dataset as a function of the number of dimensions of the hyperplane on which it is projected

a similar problem more than to create an optimal model.

A few trials with random hyperparameters are used to find a range in which to search for possible solutions. Models are trained with a number of layers ranging from 2 to 6, a number of neurons per layer ranging from 40 to 100 and a learning rate ranging from 10^{-4} to $10^{-}2$.

With models giving only the assumed class of a picture (SVM, KNN), precision was used to determine the best model. Now that we use models giving a probability that a picture belongs to a class, more relevant metrics are available.

This time the metric used to measure the effectiveness of the models is not their accuracy, but a score linked to their “binary crossentropy”. This quantity takes into account not only the class assigned to an image, but also the probability calculated by the algorithm that the image belongs to this class. The activation function of our last layer being a sigmoid, if the value returned by this layer is 0 or 1 the algorithm is “certain” that the analyzed image is a negative or positive case; if this value is between the two, a doubt exists. The use of the binary crossentropy encourages to select a model which is more sure of its classifications



(a) (b)

Figure 5.11: A bee picture (left) and its reconstruction after a PCA (right)

5.3.2 CNN

One type of model that performs well in image analysis is the Convolutional Neural Network (CNN). This model uses convolutional layers to extract information from the image before transmitting it to dense layers, similar to those present in an MLP.

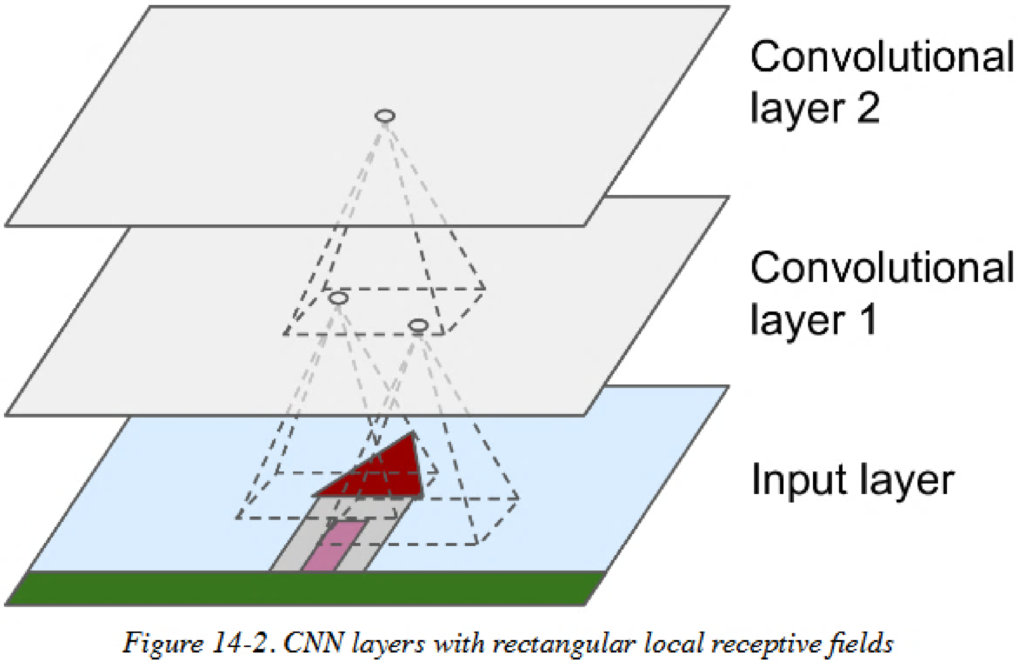


Figure 5.12: Diagram of principle of convolutional layers (source : [6])

Unlike dense layers, in convolutional layers each neuron is only connected to only certain neurons of the previous layer. Predetermined side squares (kernel) are analyzed by each neuron.

Another type of layers present in CNN are the pooling layers. These layers apply a stencil to the received data in order to reduce its size. Generally, the maximum value present in the stencil area is kept, but it is also possible to extract the minimum or average value. The pooling layers, in addition to reducing the dimension of the data, make it possible to make the algorithm more robust in the face of weak translations or rotations (Figure 5.13).

A relatively simple architecture was chosen to test a CNN (Figure 5.14). This architecture was created by Aurélien Géron in order to process the classification of the MNIST dataset (a set of 10*10 pixels pictures, showing handwritten digits). The images produced by Ivan Rodriguez and associates have a much higher resolution but training a model with a more complex architecture requires more memory than

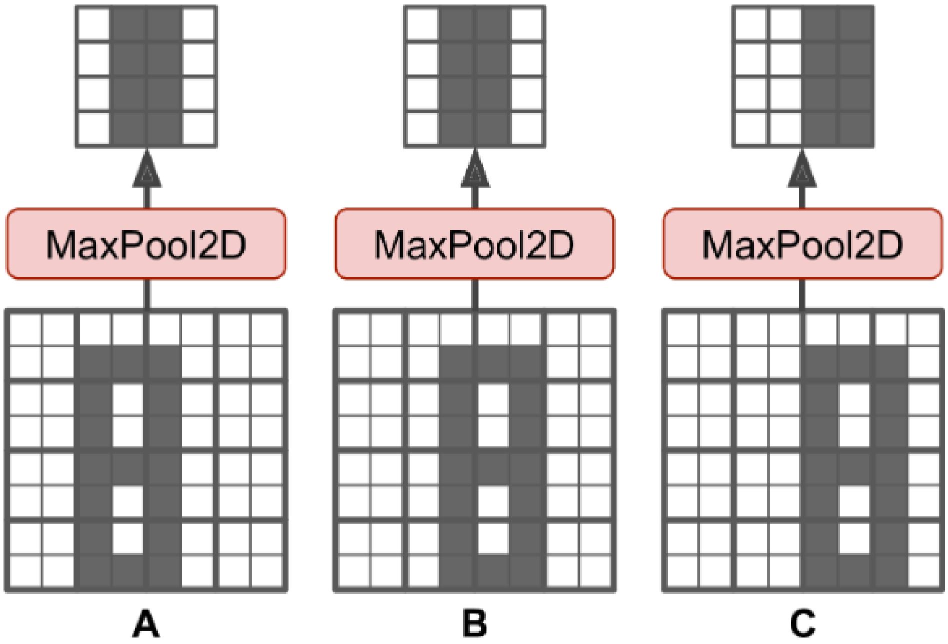


Figure 5.13: Effects of a pooling layer (source : [6])

we had available.

During the first training sessions of this model, it happened that the parameters did not converge. Moreover, when the training converged on a solution, the model sometimes classified all the images into the same category.

Hyperparameters that do not affect the architecture of the model have been modified this time. The activation functions of the dense layers have been changed to leaky ReLU in order to avoid having "dead neurons". These are neurons whose output is always the same regardless of the input data. With ReLU-like functions, it was possible that the model had this problem, as neurons could saturate at 0 (Figure 5.15).

Although there are other activation functions avoiding this saturation problem (ELU. . .), the leaky ReLU function has the advantage of being able to be computed simply. This simplicity makes it possible to create models that quickly classify data with limited computing power.

Another problem that could explain the behaviour of the models would be the vanishing gradient. This phenomenon can take place during training: the parameters of the model are less and less modified as the optimization algorithm progresses towards the lower layers of the model. To avoid this phenomenon, a "batch normalization" procedure has been implemented: after each layer, an

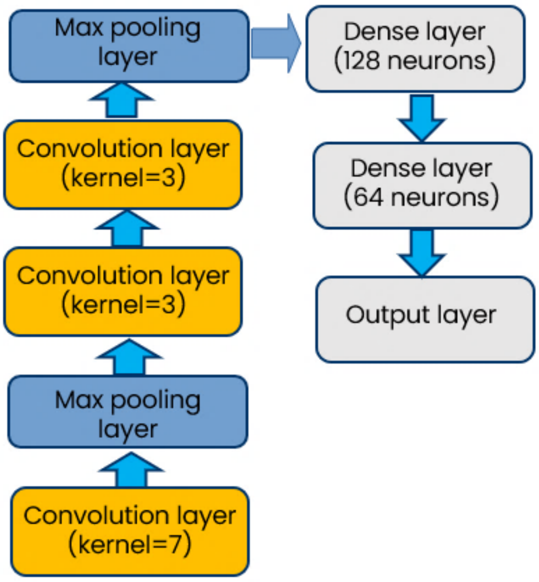


Figure 5.14: Tested CNN architecture

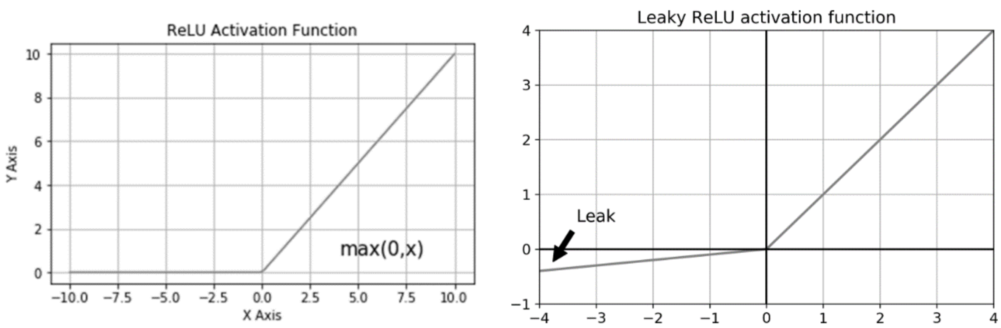


Figure 5.15: The two activation functions tested in dense layers (source : [6])

operation is added to center on zero (by creating an offset) and normalize the outputs of each neuron for the training set.

Finally, different optimization algorithms were tested: the Stochastic Gradient Descent used by default in Scikit-learn, the Nesterov Accelerated Gradient which adds an inertia to the gradients modifying the parameters of the model and the RMSprop which in addition to the inertia implements an adaptive learning rate. Using RMSprop provides the best results.

Once these choices are fixed, different learning rates are tested, ranging from 10^{-1} to 10^{-9}. Binary crossentropy is always the criterion for selecting the best model. This one is found with a learning rate of 10^{-5}, by training it on 5 epoch.

Beyond that, the model begins to overfit: it is more accurate on the training set but less on the verification set.

Conclusion

The initial objective of this internship was to create an algorithm capable of distinguishing bees bringing back pollen. Although this goal could not be fulfilled, all the preliminary steps were reworked or done.

The segmentation on bees on a different background and adapted to the different luminosities encountered during the day was carried out. Bee monitoring now includes the number of analyses a bee must pass. Finally, a dataset was created using the pictures produced by We'll Bee.

It now remains to create the classification algorithm. The pictures created being of much lower quality than those with which algorithms have been tested, the precision will undoubtedly be greatly reduced. It remains to be seen which algorithms allow real-time analysis, and whether it is possible to analyse each bee several times (if possible at each picture as in [5]).

Given the low proportion of positive cases observed, it will also be essential to adjust the classification thresholds of these algorithms in order to limit the number of false positives.

Appendix A

arguments effect of initundistortrectifymap()

The initUndistortRectifyMap() uses five arguments in order to estimate and correct the distortion on a picture. A, B, and E deal with radial distortion (Figure A.1 & A.2). C and D deal with tangential distortion, each along a different axis (Figure A.3 & A.4).

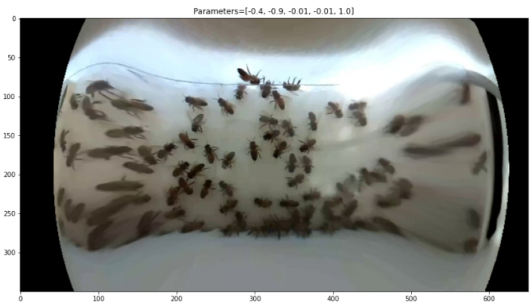


Figure A.1: overcompensation of the radial distortion, and creation of an handlebar distortion

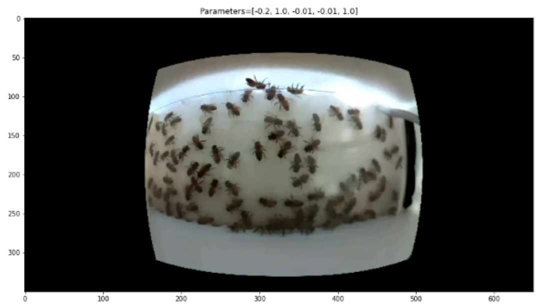


Figure A.2: amplified radial distortion

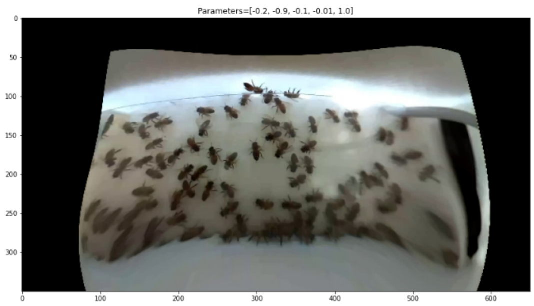


Figure A.3: C : tangential distortion created along the vertical axis

Figure A.4: D : tangential distortion created along the horizontal axis

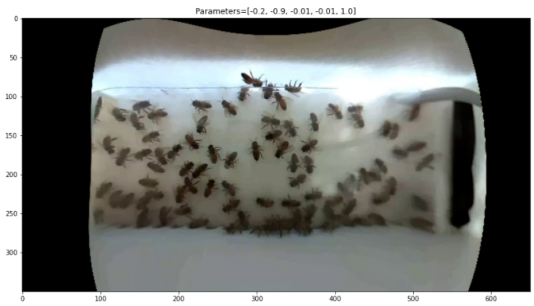


Figure A.5: picture obtained with correct parameters

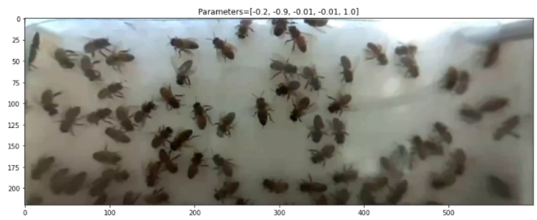


Figure A.6: image after distortion compensation and cropping

Appendix B

Choice of the number of thresholds for the segmentation by luminosity

When a single luminosity threshold is used to segment the whole picture, the result is unusable (see chapter 2.2).

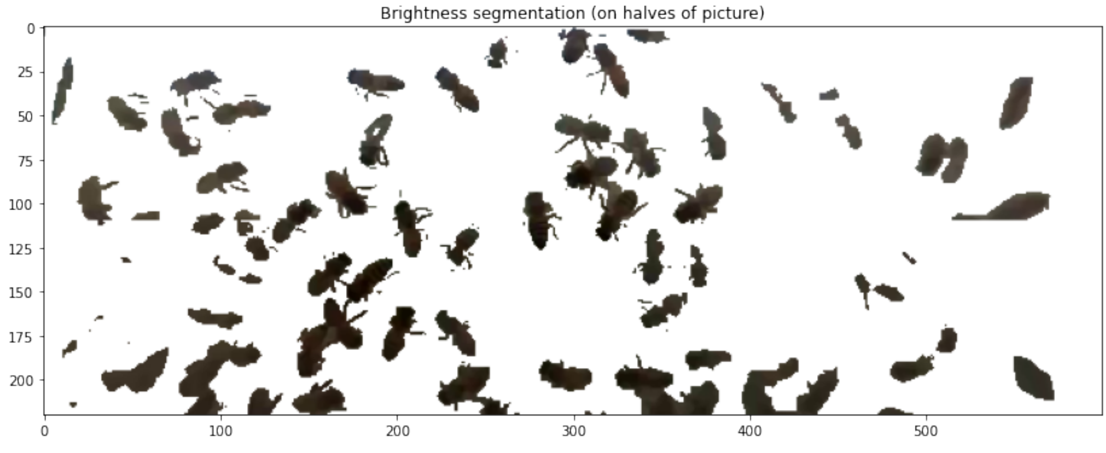


Figure B.1: result of a segmentation by half of picture

By using a different threshold for each half of the image, the segmentation is clearer despite some artifacts at the edge of the halves (Figure B.1).

Image processing by thirds gives good results. (Figure B.2)

If the image is processed in smaller fractions (in ninths in Figure B.3 to adapt

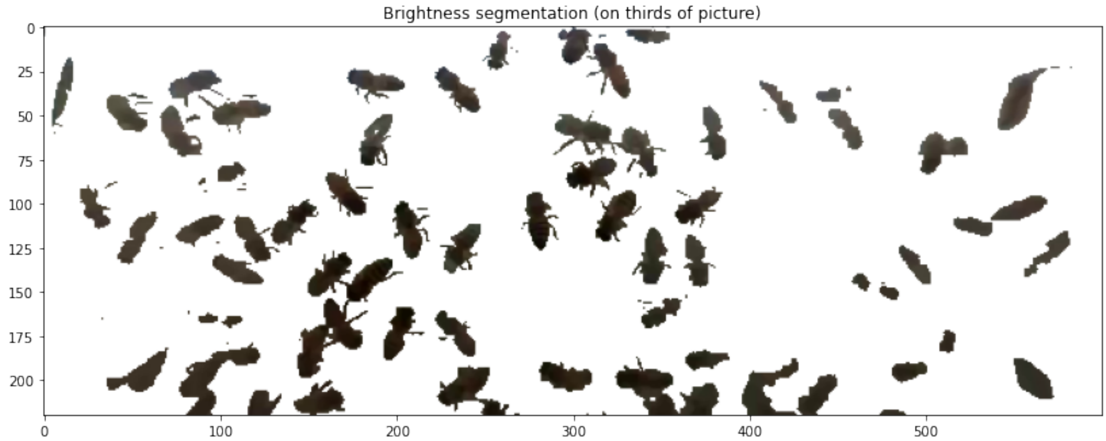


Figure B.2: result of a segmentation by thirds of picture

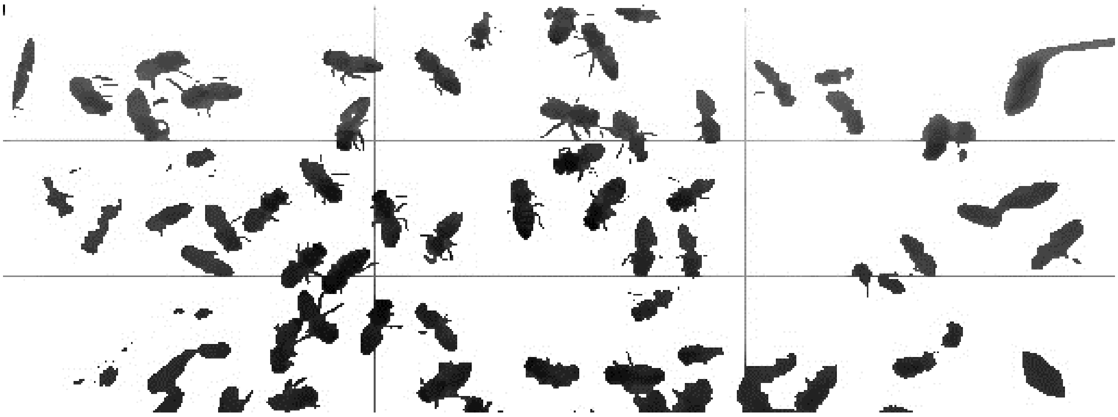


Figure B.3: result of a segmentation by ninth of picture

to a possible difference in luminosity along the horizontal axis), the segmentation is bad in the areas where few bees are present .

Appendix C

Example of a tracking list

Without a bee in the camera's field of view, the tracklet list is empty. When a bee is seen for the first time, a line is created with its position (A1). (Figure C.1)

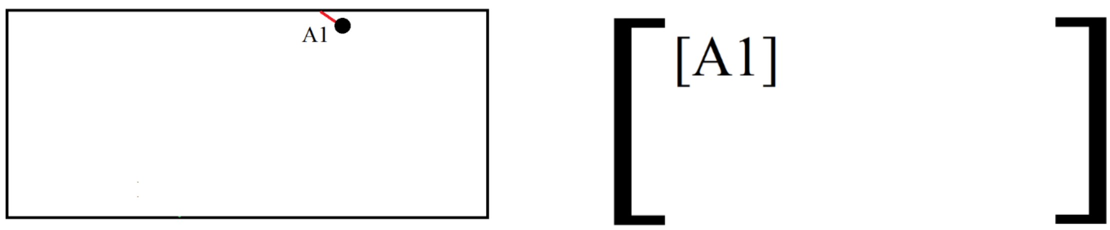


Figure C.1: trajectory of bees in the video and associated list (1)

The first bee continues on its way, its second position (A2) is recorded after the first. A second bee is detected, and a second line is dedicated to it, containing its position (B1). (Figure C.2)

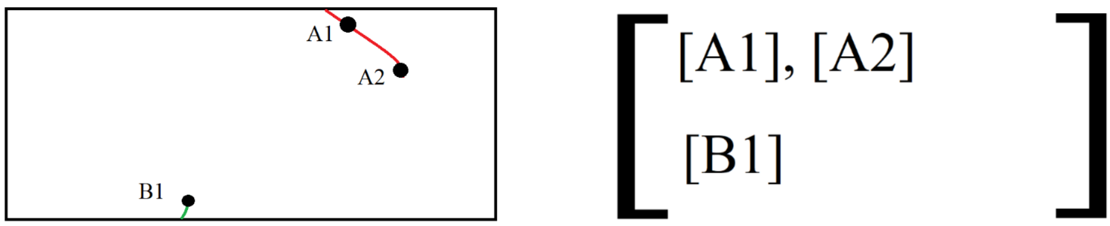


Figure C.2: trajectory of bees in the video and associated list (2)

The list is updated every frame, so the last item in a line is always the current position of a bee. (Figure C.3)

Figure C.3: trajectory of bees in the video and associated list (3)

Appendix D

Automatic undistortion program

```
import matplotlib
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import cv2
from skimage import data,io
import matplotlib.pyplot as plt
import numpy as np
from math import sqrt

%matplotlib qt

img=io.imread('first_frame.jpg')

plt.imshow(img)

xc=img.shape[1]/2 #center's coordinates (supposedly center of distortion)
yc=img.shape[0]/2

#select n points:
n=18

plt.imshow(img)
#plot some lines as guide:
for i in range (1, n//2-1):
    x=i*img.shape[1]/(n//2-1)
    yh=20
```

```
    yl=img.shape[0]-20
    plt.plot([x,x], [yh,yl], 'r-')




P=plt.ginput(n) #[(x1, y1),
                # (x2, y2),
                # (x.., y..),
                # (xn, yn)] pixel coordinates
```

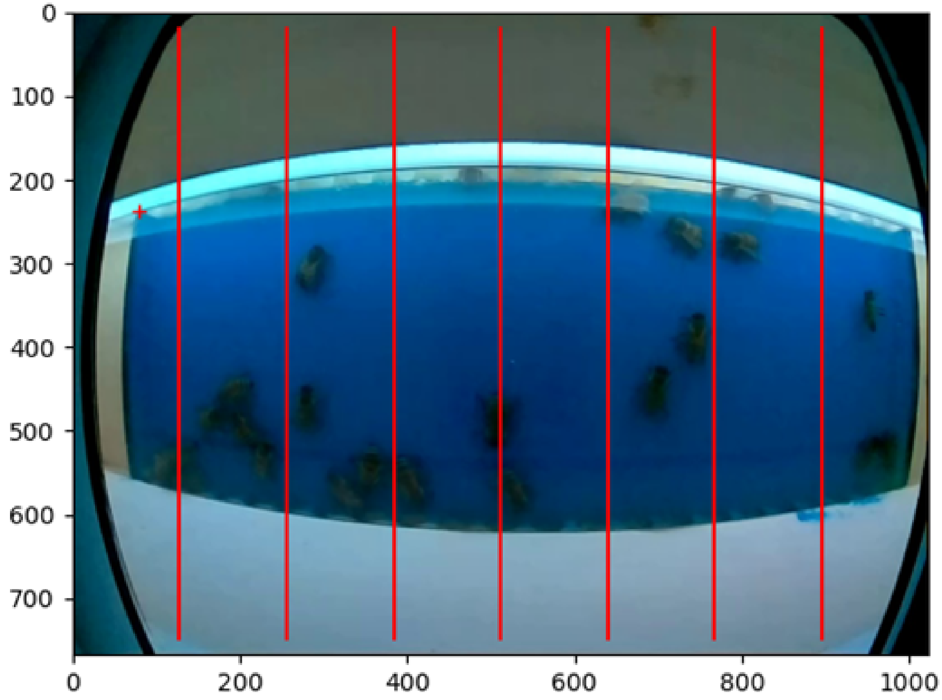


Figure D.1: picture with guidelines for the selection of 18 points (14 intersection plus the 4 corners)

```

# Compute parameters to undistort the picture ————————

focal=1000
x=np.zeros(n)
```

```python
y=np.zeros(n)
r=np.zeros(n)

for i in range (0, n):
    x[i]=(P[i][0]-xc)/focal
    y[i]=(P[i][1]-yc)/focal
    r[i]=sqrt(x[i]*x[i]+y[i]*y[i])


#fix some undistorted coordinates to make the problem solvable
#hypothesis: horizontal lines are straight

yuh=min(y[:n//2])
yul=max(y[n//2:])
yu=np.zeros(n)

for i in range (0,n//2): #points on top line first,
    yu[i]=yuh
for i in range (n//2,n): #then points on bottom line
    yu[i]=yul

A=[]
B=[]
focal2=3

for i in range(0,n):
    A.append([y[i]*r[i]**2, y[i]*r[i]**4, r[i]**2+2*y[i]**2, 2*x[i]*y[i], y[i]*r[i]**6])
    B.append((yu[i]-y[i])/focal2)

#Find the least square solution to a*x = b --------------------

X = np.linalg.lstsq(A, B, rcond=None)
X[0] # [k1, k2, p1, p2, k3]


#Use the computed parameters to undistort the picture ----------------

def func(params, cx,cy, x, y, w, h, focal):
    K  = np.array([[1000, 0,int(cx)], [0,1000, int(cy)], [0,0,1]])
    K2 = np.array ([[focal,0,int(cx)], [0,focal,int(cy)], [0,0,1]])

    map1, map2 = cv2.initUndistortRectifyMap(K, params, None, K2, (img.shape[1], img.shape[0]), cv2.CV_32FC1)
    img2 = cv2.remap(img, map1, map2, cv2.INTER_LINEAR)
    plt.figure(figsize=(14,14))
    plt.imshow(img2)
    return img2

```

```
img2 = func(-X[0], xc, yc, 0, 0, 1024, 768,700)
```

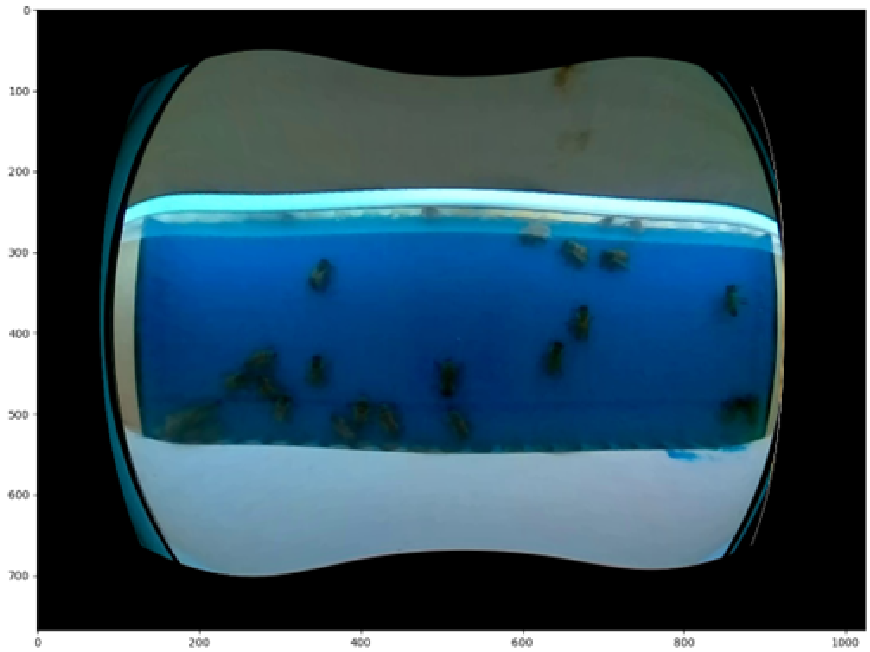


Figure D.2: picture undistorted with the program

Appendix E

Segmentation program

```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

import cv2
from skimage import data,io
import matplotlib.pyplot as plt
import numpy as np
import glob, os

# common functions ----------------------------------------------------------

def func(a,b,c,d,e, cx,cy, x, y, w, h, focal):
    K  = np.array([[1000, 0,int(cx)], [0,1000, int(cy)], [0,0,1]])
    K2 = np.array([[focal,0,int(cx)], [0,focal,int(cy)], [0,0,1]])
    a/=100
    b/=100
    c/=100
    d/=100
    e/=100
    map1, map2 = cv2.initUndistortRectifyMap(K, np.array([a,b,c,d,e])
    , None, K2, (img.shape[1], img.shape[0]), cv2.CV_32FC1)
    img2 = cv2.remap(img, map1, map2, cv2.INTER_LINEAR)
    return img2[y:y+h,x:x+w,[0,1,2]]

def normalize(x):
    x = x.astype(np.float32)
    l = x[0]*x[0] + x[1]*x[1] + x[2]*x[2]
    if l <= 0:return x
    return x/np.sqrt(l)

```

```
def bee_map(e): return max(0,((255 - e[2])*2 - e[1] + 2*e[0])//4)

def iou(r,s):
    dy = (min(r[0]+r[2],s[0]+s[2]) - max(r[0],s[0]))
    if dy <= 0:return 0
    dx = (min(r[1]+r[3],s[1]+s[3]) - max(r[1],s[1]))
    if dx <= 0:return 0
    return dx*dy/(r[2]*r[3])

def get_background_color(img):
    return np.mean(img[::2,::2], axis = (0,1))

def getFirstFrame(videofile):
    vidcap = cv2.VideoCapture(videofile)
    fourcc=cv2.VideoWriter_fourcc('H', '2', '6', '4')
    success, image = vidcap.read()
    if success:
        cv2.imwrite("first_frame_WBG.jpg", image)  # save frame as JPEG file
    else:
        print("could not open file")

# --------------------------------------------------------------------------------

path='C:/Users/Froissart/Code_Thesis/first_frame_WBG.jpg'
img = io.imread(path)

img2 = func(-20, -90, -1, -1, 100, 625, 500, 310, 365, 600, 220,400) #Undistorting and cropping
bgd=get_background_color(img2)
print("background:", bgd)
res = np.zeros((img2.shape[0], img2.shape[1]))

for y in range(res.shape[0]):
    for x in range(res.shape[1]):
        res[y,x] = 1 - np.dot(normalize(img2[y,x]), normalize(bgd)) # blue around [0.1, 0.5, 1]. [139,141,128] near center, [160.0,165.0,107.0] a bit better

integral = np.array(res>0.005,dtype = np.float32)

for y in range(res.shape[0]):
    for x in range(1,res.shape[1]):
        integral[y,x] += integral[y,x-1]
for y in range(1,res.shape[0]):
    for x in range(res.shape[1]):
        integral[y,x] += integral[y-1,x]
```

```
dim = 5
res2 = np.zeros((img2.shape[0], img2.shape[1]))
for y in range(res.shape[0]):
    for x in range(0,res.shape[1]):
        A = np.array([max(y-dim,0), max(x-dim,0)]) #[y-5, x-5]
        B = np.array([max(y-dim,0), min(x+dim,res.shape[1]-1)]) #[y-5, x+5]
        C = np.array([min(y+dim,res.shape[0]-1), max(x-dim,0)]) #y+5, x-5
        D = np.array([min(y+dim,res.shape[0]-1), min(x+dim,res.shape[1]-1)]) #y+5 x+5
        res2[y,x] = integral[D[0],D[1]] - integral[B[0],B[1]] - integral[C[0],C[1]] + integral[A[0],A[1]]


res2b = np.array(res2)

for y in range(res.shape[0]):
    for x in range(0,res.shape[1]):
        w = 2
        val = np.max(res2[max(y-w,0):min(y+w+1,res.shape[0]), max(x-w,0):min(x+w+1,res.shape[1])])
        if res2[y,x] < val:
            res2b[y,x] = 0 #keep only local maximums


res3 = np.array(img2)
sample = np.array(res2b)

rect = []

for i in range(10000):
    k = np.argmax(sample) #coordinates of the maximum
    y,x = k//res2.shape[1], k%res2.shape[1]

    if sample[y,x] == 0:break
    aux = [y-10,x-10,20,20]
    for e in rect:
        if iou(aux,e) > 0.2:break
    else:
        if x > 5 and y > 5:
            rect += [aux]    #x=x-10 y=y-10
            res3[max(y-10,0):y+10,max(x-10,0):x+10] = [255,0,0] #rectangle, centered on x,y
            res3[max(y-8,0):y+8,max(x-8,0):x+8] = img2[max(y-8,0):y+8,max(x-8,0):x+8]
    sample[max(y-10,0):y+10,max(x-10,0):x+10] = -1

```

```
#Segmentation --------------------------------------------------------------------------
    #trying three segmentations on thirds of picture

Y1third=int(img2.shape[0]/3)
Y2third=int(2*img2.shape[0]/3)
meanUp=np.mean(img2[:Y1third,:,:])
meanMid=np.mean(img2[Y1third:Y2third,:,:])
meanBottom=np.mean(img2[Y2third:,:,:])

test2 = np.array(img2)
a=0.6 #mean portion threshold

for y in range(Y1third):
    for x in range(0,img2.shape[1]):
        if np.mean(test2[y,x])>meanUp*a:
            test2[y,x]=[255,255,255]

for y in range(Y1third,Y2third):
    for x in range(0,img2.shape[1]):
        if np.mean(test2[y,x])>meanMid*a:
            test2[y,x]=[255,255,255]

for y in range(Y2third, img2.shape[0]):
    for x in range(0,img2.shape[1]):
        if np.mean(test2[y,x])>meanBottom*a:
            test2[y,x]=[255,255,255]

plt.figure(figsize=(14,14))
plt.imshow(test2)
plt.title('Brightness segmentation (on thirds of picture)')

# --------------------------------------------------------------------------------

test2G = np.zeros((img2.shape[0], img2.shape[1]))

for y in range(test2G.shape[0]):
    for x in range(test2G.shape[1]):
        test2G[y,x] = int(np.mean(test2[y,x,:]))


integral = np.array(test2G,dtype = np.float32)

for y in range(res.shape[0]):
    for x in range(1,res.shape[1]):
        integral[y,x] += integral[y,x-1]
for y in range(1,res.shape[0]):
    for x in range(res.shape[1]):
        integral[y,x] += integral[y-1,x]
```

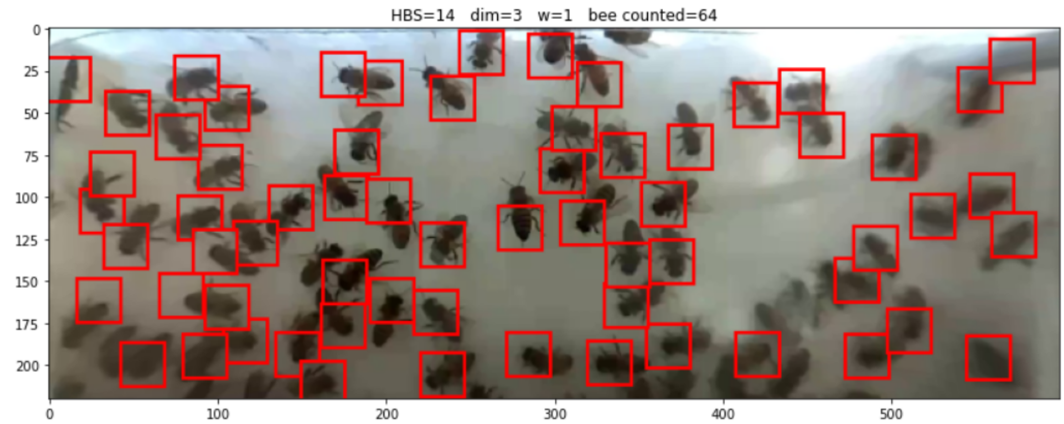
```
dim = 5
res2 = np.zeros((img2.shape[0], img2.shape[1]))
for y in range(res.shape[0]):
    for x in range(0,res.shape[1]):
        A = np.array([max(y-dim,0), max(x-dim,0)]) #[y-5, x-5]
        B = np.array([max(y-dim,0), min(x+dim,res.shape[1]-1)]) #[y-5, x+5]
        C = np.array([min(y+dim,res.shape[0]-1), max(x-dim,0)]) #y+5, x-5
        D = np.array([min(y+dim,res.shape[0]-1), min(x+dim,res.shape[1]-1)]) #y+5 x+5
        res2[y,x] = integral[D[0],D[1]] - integral[B[0],B[1]] - integral[C[0],C[1]] + integral[A[0],A[1]]

plt.figure(figsize=(14,14))
plt.imshow(integral)

plt.figure(figsize=(14,14))
plt.imshow(res2, cmap='gray')
plt.title("res2")

res2scaled=255*res2/np.max(res2)

plt.figure(figsize=(14,14))
plt.imshow(res2scaled, cmap='gray')
plt.title("res2scaled")

res2b = np.array(res2scaled)

for y in range(res.shape[0]):
    for x in range(0,res.shape[1]):
        w = 2
        val = np.min(res2scaled[max(y-w,0):min(y+w+1,res.shape[0]), max(x-w,0):min(x+w+1,res.shape[1])]) #np.max(res2[y-w:y+w,x-w:x+w])
        if res2scaled[y,x] > val:
            res2b[y,x] = 255 #keep only local maximums

res3 = np.array(img2)
sample = np.array(res2b)

rect = []

for i in range(10000):
    k = np.argmax(sample) #coordinates of the maximum
    y,x = k//res2.shape[1], k%res2.shape[1]
```

```
    if sample[y,x] == 0:break
    aux = [y-10,x-10,20,20]
    for e in rect:
        if iou(aux,e) > 0.2:break
    else:
        if x > 5 and y > 5:
            rect += [aux]     #x=x-10 y=y-10
            res3[max(y-10,0):y+10,max(x-10,0):x+10] = [255,0,0]  #rectangle, centré sur x,y
            res3[max(y-8,0):y+8,max(x-8,0):x+8] = img2[max(y-8,0):y+8,max(x-8,0):x+8]
    sample[max(y-10,0):y+10,max(x-10,0):x+10] = -1


    # Size of bounding boxes selection ------------------------------------------------
    #1st picture has 73 bees
for HBS in range(10,20):
    #restest3 = np.array(img2)
    #sampletest = np.array(test2__2d)
    res3 = np.array(img2)
    sample = np.array(res2b)


    rect = []
    #HBS=15 #Half Bee Size: half side of square

    for i in range(10000):
        k = np.argmax(sample)
        y,x = k//res2b.shape[1], k%res2b.shape[1]

        if sample[y,x] == 0:break
        aux = [y-HBS,x-HBS,2*HBS,2*HBS]
        for e in rect:
            if iou(aux,e) > 0.25:break #0.2
        else:
            if x > 5 and y > 5:
                rect += [aux]     #x=x-10 y=y-10
                res3[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = [255,0,0]     #rectangle, centré sur x,y
                res3[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2] = img2[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2]
        sample[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = 0 #don't search at this place again
    print('HBS=',HBS,'    bee counted=',len(rect))
    plt.figure(figsize=(14,14))
    plt.imshow(res3)
    plt.title('HBS='+str(HBS))

```

```
    #HBS=13 gives a good number, but there are some doublons. HBS=14 identifies more clearly the bess: see on left

# Parameters choice : Bounding boxes + kernels
-----------------------------------------------------------------------
    #1st picture has 73 bees

for dim in range(3,10): #5

    res2 = np.zeros((img2.shape[0], img2.shape[1]))

    for y in range(img2segG.shape[0]):
        for x in range(0,img2segG.shape[1]):
            A = np.array([max(y-dim,0), max(x-dim,0)]) #[y-5, x-5]
            B = np.array([max(y-dim,0), min(x+dim,res.shape[1]-1)]) #[y-5, x+5]
            C = np.array([min(y+dim,res.shape[0]-1), max(x-dim,0)]) #y+5, x-5
            D = np.array([min(y+dim,res.shape[0]-1), min(x+dim,res.shape[1]-1)]) #y+5 x+5
            res2[y,x] = integral[D[0],D[1]] - integral[B[0],B[1]] - integral[C[0],C[1]] + integral[A[0],A[1]]

    res2scaled=255*res2/np.max(res2)
    res2b = np.array(res2scaled)

    for w in range(1,5): #2

        for y in range(res.shape[0]):
            for x in range(0,res.shape[1]):
                val = np.min(res2scaled[max(y-w,0):min(y+w+1,res.shape[0]), max(x-w,0):min(x+w+1,res.shape[1])]) #np.max(res2[y-w:y+w,x-w:x+w])
                if res2scaled[y,x] > val:
                    res2b[y,x] = 255 #keep only local maximums

        for HBS in range(14,19): #10
            res3 = np.array(img2)
            sample = np.array(res2b)
            rect = []

            for i in range(10000):
                k = np.argmin(sample)
                y,x = k//res2b.shape[1], k%res2b.shape[1]

                if sample[y,x] >= 254:break
                aux = [y-HBS,x-HBS,2*HBS,2*HBS]
                for e in rect:
                    if iou(aux,e) > 0.2:break
```

```
                else:
                    if x > 5 and y > 5 and x<img2.shape[1]-5 and y<img2.shape[0]-5:    #avoid borders
                        rect += [aux]      #x=x-10 y=y-10
                        res3[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = [255,0,0]       #rectangle, centré sur x,y
                        res3[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2] = img2[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2]
                sample[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = 255 # don't search at this place again
            print('HBS=',HBS,'  dim=',dim,'  w=',w,'  bee counted=',len(rect))

            plt.figure(figsize=(14,14))
            plt.imshow(res3)
            plt.title('HBS='+str(HBS)+'  dim='+str(dim)+'  w='+str(w)+'  bee counted='+str(len(rect)))

    #
```

```
HBS= 14    dim= 3    w= 1    bee counted= 64
HBS= 15    dim= 3    w= 1    bee counted= 58
HBS= 16    dim= 3    w= 1    bee counted= 56
HBS= 17    dim= 3    w= 1    bee counted= 49
HBS= 18    dim= 3    w= 1    bee counted= 46
HBS= 14    dim= 3    w= 2    bee counted= 64
HBS= 15    dim= 3    w= 2    bee counted= 58
HBS= 16    dim= 3    w= 2    bee counted= 56
HBS= 17    dim= 3    w= 2    bee counted= 49
HBS= 18    dim= 3    w= 2    bee counted= 46
HBS= 14    dim= 3    w= 3    bee counted= 64
HBS= 15    dim= 3    w= 3    bee counted= 58
HBS= 16    dim= 3    w= 3    bee counted= 56
HBS= 17    dim= 3    w= 3    bee counted= 49
HBS= 18    dim= 3    w= 3    bee counted= 46
HBS= 14    dim= 3    w= 4    bee counted= 64
HBS= 15    dim= 3    w= 4    bee counted= 58
HBS= 16    dim= 3    w= 4    bee counted= 56
HBS= 17    dim= 3    w= 4    bee counted= 50
HBS= 18    dim= 3    w= 4    bee counted= 46
HBS= 14    dim= 4    w= 1    bee counted= 60
HBS= 15    dim= 4    w= 1    bee counted= 60
HBS= 16    dim= 4    w= 1    bee counted= 56
HBS= 17    dim= 4    w= 1    bee counted= 53
```


HBS=14 dim=3 w=1 bee counted=64
0
25
50
75
100
125
150
175
200
0
100
200
300
400
500

Appendix F

Tracking program

```
    # Read Video and Tracking ________________________________________

    import cv2
    from skimage import data,io
    import matplotlib.pyplot as plt
    import numpy as np
    import glob, os

def detect(img, mapx, mapy, crop, threshold, dim):
    img2 = cv2.remap(img, mapx, mapy, cv2.INTER_LINEAR)
    img2 = img2[crop[1]:crop[1]+crop[3],crop[0]:crop[0]+crop[2]] # crop=[xmin,ymin,width,height]
    res = np.zeros((img2.shape[0], img2.shape[1]))


def getFirstFrame(videofile):
    vidcap = cv2.VideoCapture(videofile)
    success, image = vidcap.read()
    if success:
        cv2.imwrite("first_frame_WBG.jpg", image)  # save frame as JPEG file

def normalize(x):
    x = x.astype(np.float32)
    l = x[0]*x[0] + x[1]*x[1] + x[2]*x[2]
    if l <= 0:return x
    return x/np.sqrt(l)

def bee_map(e): return max(0,((255 - e[2])*2 - e[1] + 2*e[0])//4)

```

```
def iou(r,s):
    dy = (min(r[0]+r[2],s[0]+s[2]) - max(r[0],s[0]))
    if dy <= 0:return 0
    dx = (min(r[1]+r[3],s[1]+s[3]) - max(r[1],s[1]))
    if dx <= 0:return 0
    return dx*dy/(r[2]*r[3])


def get_background_color(img):
    return np.mean(img[::2,::2], axis = (0,1))

def compare(r1, r2):
    return np.sqrt((r1[0] - r2[0])*(r1[0] - r2[0]) + (r1[1] - r2[1])
   *(r1[1] - r2[1]))


    #dim, w, and HBS to be set before
def detectbis(img, mapx, mapy, crop, threshold):
    img2 = cv2.remap(img, mapx, mapy, cv2.INTER_LINEAR)
    img2 = img2[crop[1]:crop[1]+crop[3],crop[0]:crop[0]+crop[2]] #
   crop=[xmin,ymin,width,height]
    res = np.zeros((img2.shape[0], img2.shape[1]))

    #image segmentation by thirds:
    Y1third=int(img2.shape[0]/3)
    Y2third=int(2*img2.shape[0]/3)
    meanUp=np.mean(img2[:Y1third,:,:])
    meanMid=np.mean(img2[Y1third:Y2third,:,:])
    meanBottom=np.mean(img2[Y2third:,:,:])

    img2seg = np.array(img2)
    a=0.6 #mean portion threshold

    for y in range(Y1third):
        for x in range(0,img2.shape[1]):
            if np.mean(img2seg[y,x])>meanUp*a:
                img2seg[y,x]=[255,255,255]

    for y in range(Y1third,Y2third):
        for x in range(0,img2.shape[1]):
            if np.mean(img2seg[y,x])>meanMid*a:
                img2seg[y,x]=[255,255,255]

    for y in range(Y2third, img2.shape[0]):
        for x in range(0,img2.shape[1]):
            if np.mean(img2seg[y,x])>meanBottom*a:
                img2seg[y,x]=[255,255,255]
```

```
    img2segG = np.zeros((img2.shape[0], img2.shape[1])) #segmented image in gray levels

    for y in range(img2segG.shape[0]):
        for x in range(img2segG.shape[1]):
            img2segG[y,x] = int(np.mean(img2seg[y,x,:]))

    integral = np.array(img2segG,dtype = np.float32)

    for y in range(img2segG.shape[0]):
        for x in range(1,res.shape[1]):
            integral[y,x] += integral[y,x-1]
    for y in range(1,res.shape[0]):
        for x in range(img2segG.shape[1]):
            integral[y,x] += integral[y-1,x]

    res2 = np.zeros((img2.shape[0], img2.shape[1])) #scan integral image for "flats"

    for y in range(img2segG.shape[0]):
        for x in range(0,img2segG.shape[1]):
            A = np.array([max(y-dim,0),             max(x-dim,0)]) #[y-5, x-5]
            B = np.array([max(y-dim,0),             min(x+dim,res.shape[1]-1)]) #[y-5, x+5]
            C = np.array([min(y+dim,res.shape[0]-1), max(x-dim,0)]) # y+5, x-5
            D = np.array([min(y+dim,res.shape[0]-1), min(x+dim,res.shape[1]-1)]) #y+5 x+5
            res2[y,x] = integral[D[0],D[1]] - integral[B[0],B[1]] - integral[C[0],C[1]] + integral[A[0],A[1]]

    res2scaled=255*res2/np.max(res2)
    res2b = np.array(res2scaled)

    for y in range(res.shape[0]):
        for x in range(0,res.shape[1]):
            val = np.min(res2scaled[max(y-w,0):min(y+w+1,res.shape[0]), max(x-w,0):min(x+w+1,res.shape[1])]) #np.max(res2[y-w:y+w,x-w:x+w])
            if res2scaled[y,x] > val:
                res2b[y,x] = 255 #keep only local maximums

    res3 = np.array(img2)
    sample = np.array(res2b)
    rect = []
    for i in range(10000):
        k = np.argmin(sample)
        y,x = k//res2b.shape[1], k%res2b.shape[1]
```

```
                if sample[y,x] >= 254:break
                aux = [y-HBS,x-HBS,2*HBS,2*HBS]
                for e in rect:
                    if iou(aux,e) > 0.2:break
                else:
                    if x > 5 and y > 5 and x<img2.shape[1]-5 and y<img2.shape[0]-5:    #avoid borders
                        rect += [aux]
                        res3[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = [255,0,0]        #rectangle, centré sur x,y
                        res3[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2] = img2[max(y-HBS+2,0):y+HBS-2,max(x-HBS+2,0):x+HBS-2]
                sample[max(y-HBS,0):y+HBS,max(x-HBS,0):x+HBS] = 255 #don't search at this place again
    return img2, rect



# class used to make the tracklet list ______________________________________

    class BeeTracker:
    def __init__(self, shape):
        ww, hh = shape[1], shape[0]
        K  = np.array([[1000, 0, 625], [0, 1000, 500], [0,0,1]])
        K2 = np.array([[400,  0, 625], [0, 400,  500], [0,0,1]])
        self.mapx, self.mapy = cv2.initUndistortRectifyMap(K, np.array([-0.2,-0.9,-0.01,-0.01,1]), None, K2, (ww,hh), cv2.CV_32FC1)
        self.tracklet = []
        self.index = 0

    def execute(self, img):
        self.last_img, stub = detectbis(img, self.mapx, self.mapy,[310,365,600,220], 0.05)
        self.index += 1
        return self.index, self.update(stub)

    def plot(self, folder):
        screen = self.last_img
        for tr in self.tracklet:
            for i in range(1,len(tr)):
                cv2.line(screen, (tr[i-1][1]+10, tr[i-1][0]+10), (tr[i][1]+10, tr[i][0]+10), (255,0,0))
            cv2.rectangle(screen, (tr[-1][1],tr[-1][0]), (tr[-1][1]+tr[-1][2],tr[-1][0]+tr[-1][3]), (255,0,0))
        io.imsave("{}/WBGimg{}.jpg".format(folder, self.index),screen)

    def update(self, stub):
```

```
        enter, exit = 0, 0
        match = []
        for k,tr in enumerate(self.tracklet):
            sscore, ind = dim*4, -1 #sscore=20
            for i in range(len(stub)):
                score = compare(tr[-1], stub[i]) #tr[-1]=last element of tr
                if score > sscore: continue
                sscore, ind = score, i
            if ind < 0: continue #if no correspondance between this tr[] and any stub[i], skip next
            for j in range(len(match)):
                if match[j][1] == ind:
                    if sscore < match[j][0]:  #if better proximity is found
                        match[j] = (sscore, ind, k) #"proximity", stub indice, tracklet(bee) indice
                    break
            else:
                match += [(sscore, ind, k )]
        ## metto i match in ordine ed elimino quelli doppi
        for m in match:
            self.tracklet[m[2]] += [stub[m[1]]]  #add new position
        new_tr = []
        for i,t in enumerate(self.tracklet):
            for m in match:
                if m[2] == i:
                    if t[0][4]==1 and t[-1][1] > self.last_img.shape[0]/2: #if the bee must be scanned AND it is gone through half the image
                        self.scan(t[-1])
                        t[0][4]=0 #remove "must be scanned" flag
                    break
            else:
                #questa catena viene eliminata
                if t[0][1]+t[0][3] < self.last_img.shape[0]/2 and t[-1][1] > self.last_img.shape[0]/2: #if y initial in upper half of pic, and y final in bottom half (remember y axis points downward!)
                    enter += 1
                if t[0][1] > self.last_img.shape[0]/2 and t[-1][1]+t[-1][3] < self.last_img.shape[0]/2:
                    exit  += 1
                continue
            new_tr += [t] #list of position kept if a match is found
        ##
        self.tracklet = new_tr
        for i,s in enumerate(stub):
            for m in match:
```

```
                if m[1] == i:break
            else: #a new bee entered the screen
                if s[1]< self.last_img.shape[0]/2:
                    s+=[1] #if bee comes from upper limit (outside),
     it must be scanned: flag=1
                else:
                    s+=[0] #this bee must not be scanned
                self.tracklet += [[s]]

        return enter, exit


    def scan(self,pos):
        dx=14
        dy=14 #1/2 height and 1/2 width bee picture
        pos[0]+=10
        pos[1]+=10 #center on bee
        if pos[0]-dx<0: #avoid borders to have full picture
            pos[0]=dx
        if pos[0]+dx>self.last_img.shape[1]:
            pos[0]=self.last_img.shape[1]-dx
        bee_image = self.last_img[pos[1]-dy:pos[1]+dy,pos[0]-dx:pos
     [0]+dx]
        folder="beepictures"
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/
     beepictures")) #number of files in folder
        io.imsave("{}/img{}.jpg".format(folder, index), bee_image)






# Parameters optimisation ----------------------------------------

Data=[]
#best parameters so far: 3, 7, 10 / 3, 7, 11
for w in range (3, 6):
    for dim in range (6, 12):
        for HBS in range (10, 15):
            getFirstFrame('2022_07_31_15_30_00.h264')
            img = io.imread("first_frame_WBG.jpg")
            cap = cv2.VideoCapture('2022_07_31_15_30_00.h264')
            track = BeeTracker((1296, 972,3)) ###img.shape
            #
            count=0
            while cap.isOpened():
                if track.index > 100:break
                ret, frame = cap.read()
```

```
                if ret == False: break
                print(track.execute(frame[:,:,::-1]))
                track.plot("./tracking_WBG")
            #
            #once the tracking is done on 100 pictures, check lenght
    of list
            m=0
            M=0
            for i in range (0,len(track.tracklet)):
                m+=len(track.tracklet[i])
                if len(track.tracklet[i])>M:
                    M=len(track.tracklet[i])

            m/=len(track.tracklet)
            print("w=",w,"    dim=",dim,"    HBS=",HBS,"    mean length=
    ",m,"    max length=",M)
            Data.append([w,dim,HBS,m,M])
            print(Data)
```

```
(97, (0, 0))
(98, (0, 0))
(99, (0, 0))
(100, (0, 0))
(101, (0, 0))
w= 5    dim= 6    HBS= 14    mean length= 55.59016393442623    max length= 101
[[3, 6, 10, 60.71232876712329, 101], [3, 6, 11, 56.68115942028985, 101], [3, 6, 12, 53.51470588235294, 101], [3, 6, 13, 57.6349
2063492063, 101], [3, 6, 14, 54.45161290322581, 101], [3, 7, 10, 71.43939393939394, 101], [3, 7, 11, 66.72307692307692, 101],
[3, 7, 12, 64.08196721311475, 101], [3, 7, 13, 58.847457627118644, 101], [3, 7, 14, 61.94642857142857, 101], [3, 8, 10, 68.187
5, 101], [3, 8, 11, 61.935483870967744, 101], [3, 8, 12, 58.47540983606557, 101], [3, 8, 13, 56.28813559322034, 101], [3, 8, 1
4, 59.49122807017544, 101], [3, 9, 10, 62.890625, 101], [3, 9, 11, 57.93650793650794, 101], [3, 9, 12, 56.868852459016395, 10
1], [3, 9, 13, 60.0, 101], [3, 9, 14, 60.214285714285715, 101], [3, 10, 10, 57.193548387096776, 101], [3, 10, 11, 54.0327868852
45905, 101], [3, 10, 12, 56.12068965517241, 101], [3, 10, 13, 57.38181818181818, 101], [3, 10, 14, 54.45454545454545, 101], [3,
11, 10, 52.676923076923075, 101], [3, 11, 11, 56.16949152542373, 101], [3, 11, 12, 53.732142857142854, 101], [3, 11, 13, 54.5,
101], [3, 11, 14, 47.62264150943396, 101], [4, 6, 10, 61.89041095890411, 101], [4, 6, 11, 58.89705882352941, 101], [4, 6, 12, 5
5.77272727272727, 101], [4, 6, 13, 56.333333333333336, 101], [4, 6, 14, 54.61290322580645, 101], [4, 7, 10, 69.5, 101], [4, 7,
11, 64.84615384615384, 101], [4, 7, 12, 60.26229508196721, 101], [4, 7, 13, 58.559322033898304, 101], [4, 7, 14, 59.76785714285
7146, 101], [4, 8, 10, 65.890625, 101], [4, 8, 11, 61.03225806451613, 101], [4, 8, 12, 57.16393442622951, 101], [4, 8, 13, 55.1
6949152542373, 101], [4, 8, 14, 56.719298245614034, 101], [4, 9, 10, 63.31147540983606, 101], [4, 9, 11, 57.95, 101], [4, 9, 1
2, 56.71186440677966, 101], [4, 9, 13, 59.14035087719298, 101], [4, 9, 14, 59.4, 101], [4, 10, 10, 58.88333333333333, 101], [4,
10, 11, 55.96610169491525, 101], [4, 10, 12, 57.339285714285715, 101], [4, 10, 13, 59.15094339622642, 101], [4, 10, 14, 56.3773
5849056604, 101], [4, 11, 10, 53.34426229508197, 101], [4, 11, 11, 56.82142857142857, 101], [4, 11, 12, 53.18518518518518, 10
1], [4, 11, 13, 55.01923076923077, 101], [4, 11, 14, 48.5, 101], [5, 6, 10, 62.6, 101], [5, 6, 11, 59.343283582089555, 101],
[5, 6, 12, 57.04615384615385, 101], [5, 6, 13, 57.693548387096776, 101], [5, 6, 14, 55.59016393442623, 101]]
(1, (0, 0))
(2, (0, 0))
```

Figure F.1: exemple of data printed

Appendix G

Dataset creation program

```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from ipywidgets import Button, HBox
from IPython.display import display
import glob, os

import  cv2
from skimage import data,io
import matplotlib.pyplot as plt
import numpy as np

def func(a,b,c,d,e, cx,cy, x, y, w, h, focal):
    K  = np.array([[1000, 0,int(cx)], [0,1000, int(cy)], [0,0,1]])
    K2 = np.array([[focal,0,int(cx)], [0,focal,int(cy)], [0,0,1]])
    a/=100
    b/=100
    c/=100
    d/=100
    e/=100
    map1, map2 = cv2.initUndistortRectifyMap(K, np.array([a,b,c,d,e])
    , None, K2, (img.shape[1], img.shape[0]), cv2.CV_32FC1)
    img2 = cv2.remap(img, map1, map2, cv2.INTER_LINEAR)
    plt.figure(figsize=(14,14))
    plt.imshow(img2[y:y+h,x:x+w,[0,1,2]])
    return img2[y:y+h,x:x+w,[0,1,2]]

def normalize(x):
    x = x.astype(np.float32)
    l = x[0]*x[0] + x[1]*x[1] + x[2]*x[2]
    if l <= 0:return x
```

```
    return x/np.sqrt(l)

def bee_map(e): return max(0,((255 - e[2])*2 - e[1] + 2*e[0])//4)

def iou(r,s):
    dx = (min(r[0]+r[2],s[0]+s[2]) - max(r[0],s[0]))
    if dx <= 0:return 0
    dy = (min(r[1]+r[3],s[1]+s[3]) - max(r[1],s[1]))
    if dy <= 0:return 0
    return dx*dy/(r[2]*r[3])

def getFirstFrame(videofile):
    vidcap = cv2.VideoCapture(videofile)
    success, image = vidcap.read()
    if success:
        cv2.imwrite("first_frame.jpg", image)  # save frame as JPEG file


#detection on blue background ---------------------------------

def detect(img, mapx, mapy, crop, threshold, dim):
    img2 = cv2.remap(img, mapx, mapy, cv2.INTER_LINEAR)
    img2 = img2[crop[1]:crop[1]+crop[3],crop[0]:crop[0]+crop[2]]

    color = normalize(get_background_color(img2))

    res = np.zeros((img2.shape[0], img2.shape[1]))
    for y in range(res.shape[0]):
        for x in range(res.shape[1]):
            res[y,x] = 1 - np.dot(normalize(img2[y,x]), color)
    mask = res > threshold
    integral = np.array(mask,dtype = np.float32)
    for y in range(res.shape[0]):
        for x in range(1,res.shape[1]):
            integral[y,x] += integral[y,x-1]
    for y in range(1,res.shape[0]):
        for x in range(res.shape[1]):
            integral[y,x] += integral[y-1,x]

    res2 = np.zeros((img2.shape[0], img2.shape[1]))
    for y in range(res.shape[0]):
        for x in range(0,res.shape[1]):
            A = np.array([max(y-dim,0),                   max(x-dim,0)])
            B = np.array([max(y-dim,0),                   min(x+dim,res.shape[1]-1)])
            C = np.array([min(y+dim,res.shape[0]-1), max(x-dim,0)])
            D = np.array([min(y+dim,res.shape[0]-1), min(x+dim,res.shape[1]-1)])
```

```
            res2[y,x] = integral[D[0],D[1]] - integral[B[0],B[1]] - integral[C[0],C[1]] + integral[A[0],A[1]]

    res2b = np.array(res2)

    for y in range(res.shape[0]):
        for x in range(0,res.shape[1]):
            w = 3
            val = np.max(res2[max(y-w,0):min(y+w+1,res.shape[0]), max(x-w,0):min(x+w+1,res.shape[1])])
            if res2[y,x] < val:res2b[y,x] = 0

    sample = np.array(res2b)
    rect = []
    for i in range(10000):
        k = np.argmax(sample)
        y,x = k//res2.shape[1], k%res2.shape[1]

        if sample[y,x] < dim*dim*0.5:break
        aux = [max(x-dim*2,0),max(y-dim*2,0),min(x+dim*2,img.shape[1]) - max(x-dim*2,0), min(y+dim*2,img.shape[0]) - max(y-dim*2,0)]
        for e in rect:
            if iou(aux,e) > 0.3:break
        else:
            if x > dim and y > dim and x < img.shape[1]-dim and y < img.shape[0]-dim:
                rect += [aux]
        sample[max(y-10,0):y+10,max(x-10,0):x+10] = -1
    return img2, rect

def get_background_color(img):
    return np.mean(img[::2,::2], axis = (0,1))

def compare(r1, r2):
    return np.sqrt((r1[0] - r2[0])*(r1[0] - r2[0]) + (r1[1] - r2[1])*(r1[1] - r2[1]))


# class used to make the tracklet list ----------------------------------------

class BeeTracker:
    def __init__(self, shape):
        ww, hh = shape[1], shape[0]
        K  = np.array([[1000, 0,ww//2], [0,1000, hh//2], [0,0,1]])
        K2 = np.array([[394,  0,ww//2], [0,394,  hh//2], [0,0,1]])
        self.mapx, self.mapy = cv2.initUndistortRectifyMap(K, np.array([-1,0.7,0.01,0.01,0.37]), None, K2, (ww,hh), cv2.CV_32FC1)
        self.tracklet = []
```

```
            self.index = 0

    def execute(self, img):
        self.last_img, stub = detect(img, self.mapx, self.mapy,[264+6,311+15,512-12,168-43], 0.05, 5)
        self.index += 1
        return self.index, self.update(stub)

    def plot(self, folder):
        screen = self.last_img
        for tr in self.tracklet:
            for i in range(1,len(tr)):
                cv2.line(screen, (tr[i-1][0]+10, tr[i-1][1]+10), (tr[i][0]+10, tr[i][1]+10), (255,0,0))
            cv2.rectangle(screen, (tr[-1][0],tr[-1][1]), (tr[-1][0]+tr[-1][2],tr[-1][1]+tr[-1][3]), (255,0,0))
        io.imsave("{}/WBGimg{}.jpg".format(folder, self.index), screen)

    def update(self, stub):
        enter, exit = 0, 0
        match = []
        for k,tr in enumerate(self.tracklet):
            sscore, ind = dim*4, -1 #sscore=20
            for i in range(len(stub)):
                score = compare(tr[-1], stub[i]) #tr[-1]=last element of tr
                if score > sscore: continue
                sscore, ind = score, i
            if ind < 0: continue #if no correspondance between this tr[] and any stub[i], skip next
            for j in range(len(match)):
                if match[j][1] == ind:
                    if sscore < match[j][0]:  #if better proximity is found
                        match[j] = (sscore, ind, k) #"proximity", stub indice, tracklet(bee) indice
                    break
            else:
                match += [(sscore, ind, k )]
        ## metto i match in ordine ed elimino quelli doppi
        for m in match:
            self.tracklet[m[2]] += [stub[m[1]]]  #add new position
        new_tr = []
        for i,t in enumerate(self.tracklet):
            for m in match:
                if m[2] == i:
```

```
                    if t[0][4]==1 and t[-1][1] > self.last_img.shape[0]/2: #if the bee must be scanned AND it is gone through half the image
                        self.scan(t[-1])
                        t[0][4]=0 #remove "must be scanned" flag
                    break
            else:
                #questa catena viene eliminata
                if t[0][1]+t[0][3] < self.last_img.shape[0]/2 and t[-1][1] > self.last_img.shape[0]/2: #if y initial in upper half of pic, and y final in bottom half (remember y axis points downward!)
                    enter += 1
                if t[0][1] > self.last_img.shape[0]/2 and t[-1][1]+t[-1][3] < self.last_img.shape[0]/2:
                    exit  += 1
                continue
            new_tr += [t] #list of position kept if a match is found
        ##
        self.tracklet = new_tr
        for i,s in enumerate(stub):
            for m in match:
                if m[1] == i:break
            else: #a new bee entered the screen
                if s[1]< self.last_img.shape[0]/2:
                    s+=[1] #if bee comes from upper limit (outside), it must be scanned: flag=1
                else:
                    s+=[0] #this bee must not be scanned
                self.tracklet += [[s]]

        return enter, exit

    def scan(self,pos):
        dx=13
        dy=13 #1/2 height and 1/2 width bee picture
        pos[0]+=10
        pos[1]+=10 #center on bee
        if pos[0]-dx<0: #avoid borders to have full picture
            pos[0]=dx
        if pos[0]+dx>self.last_img.shape[1]:
            pos[0]=self.last_img.shape[1]-dx
        bee_image = self.last_img[pos[1]-dy:pos[1]+dy,pos[0]-dx:pos[0]+dx]
        folder="beepictures"
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/beepictures")) #number of files in folder
        io.imsave("{}/img{}.jpg".format(folder, index), bee_image)

```

```python


# class used to make the labeling interface ________________________________

class Classifier:

    def __init__(self,imlist):
        self.path=imlist[0]
        self.imindex=0 #index of pic in imlist
        self.img=cv2.imread(self.path)
        plt.figure(figsize=(10,10))
        plt.imshow(self.img[:,:,[2,1,0]])
        #display_buttons()

    def display_buttons(self):
        buttonP=Button(description='Positive', button_style='success')
        buttonN=Button(description='Negative', button_style='warning')
        buttonO=Button(description='Unknown', button_style='danger')
        display(HBox([buttonP,buttonN,buttonO]))
        buttonP.on_click(self.P_case)
        buttonN.on_click(self.N_case)
        buttonO.on_click(self.O_case)

    def P_case(self,b): #positive cases are rare, so they are always kept
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/full_dataset/positives")) #number of files in folder
        io.imsave("full_dataset/positives/P{}.jpg".format(index), self.img)
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/balanced_dataset/positives")) #number of files
        io.imsave("balanced_dataset/positives/P{}.jpg".format(index), self.img)
        os.remove("beepictures/{}".format(imlist[self.imindex].split("\\")[1])) #delete picture from beepictures folder
        print("P{} saved".format(index))
        self.nextimg()

    def N_case(self,b): #saved in balanced dataset if there are no more negatives than positives
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/full_dataset/negatives"))
        io.imsave("full_dataset/negatives/N{}.jpg".format(index), self.img)
        indexP=len(os.listdir("C:/Users/Froissart/Code_Thesis/balanced_dataset/positives"))
```

```
        indexN=len(os.listdir("C:/Users/Froissart/Code_Thesis/balanced_dataset/negatives"))
        if indexN<=indexP:
            io.imsave("balanced_dataset/negatives/N{}.jpg".format(indexN), self.img)
        os.remove("beepictures/{}".format(imlist[self.imindex].split("\\")[1]))
        print("N{} saved".format(index))
        self.nextimg()

    def O_case(self,b): #other: image not clear enough for classification
        index=len(os.listdir("C:/Users/Froissart/Code_Thesis/Null"))
        io.imsave("Null/O{}.jpg".format(index), self.img)
        os.remove("beepictures/{}".format(imlist[self.imindex].split("\\")[1]))
        print("O{} saved".format(index))
        self.nextimg()

    def nextimg(self):
        #replace previous image by next in imlist
        self.imindex+=1
        self.path=imlist[self.imindex]
        self.img=cv2.imread(self.path)
        plt.figure(figsize=(10,10))
        plt.imshow(self.img[:,:,[2,1,0]])

    def test(self):
        print("working")
```

```
# First approach : save all bee pictures from video to /beepictures

        getFirstFrame("ViewBees.mp4")
        cap = cv2.VideoCapture("ViewBees.mp4")
track = BeeTracker(img.shape)

count=0
while cap.isOpened():
    if track.index > 100:break
    ret, frame = cap.read()
    if ret == False: break
    print(track.execute(frame[:,:,::-1]))
    track.plot("./results")
```

```
# Second approach : label the pictures in /beepictures
```

```

path="beepictures/"
imlist= glob.glob(os.path.join(path, '*.jpg'))

classi=Classifier(imlist)

classi.test()
classi.display_buttons()
```

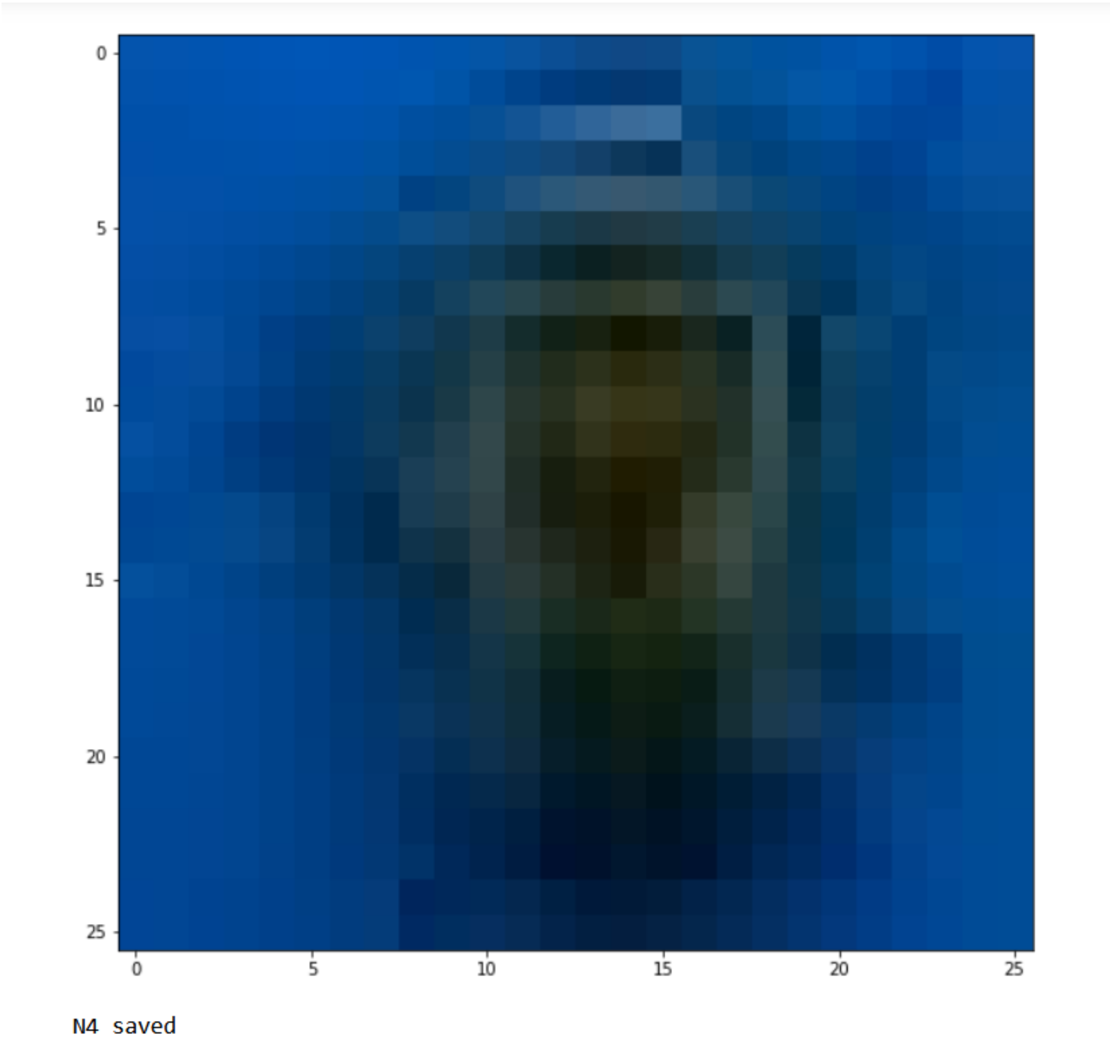


Figure G.1: once a picture is labelled

Bibliography

[1] Z. Babic et al. «Pollen bearing honey bee detection in hive entrance video recorded by remote embedded system for pollination monitoring». In: *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* (2016) (cit. on pp. 1, 24).

[2] Sai Kiran Reka. «A Vision-Based Bee Counting Algorithm for Electronic Monitoring of Langsthroth Beehives». In: *All Graduate Theses and Dissertations. 4960* (2016) (cit. on p. 1).

[3] Ngo et al. «A real-time imaging system for multiple honey bee tracking and activity monitoring». In: *Computers and Electronics in Agriculture* 163 (2019) (cit. on p. 1).

[4] Ivan F. Rodriguez, Remi Megret, Edgar Acuna, Jose L. Agosto-Rivera, and Tugrul Giray. «Recognition of Pollen-Bearing Bees from Video Using Convolutional Neural Network». In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2018, pp. 314–322. DOI: 10.1109/WACV.2018.00041 (cit. on pp. 1, 22, 23, 30).

[5] Cheng Yang and John Collins. «Deep Learning for Pollen Sac Detection and Measurement on Honeybee Monitoring Video». In: *2019 International Conference on Image and Vision Computing New Zealand (IVCNZ)*. 2019, pp. 1–6. DOI: 10.1109/IVCNZ48456.2019.8961011 (cit. on pp. 1, 38).

[6] A. Géron. *Hands-On Machine Learning with Scikit-learn, Keras, and TensorFlow*. O'Reilly, 2019 (cit. on pp. 1, 29–31, 34–36).