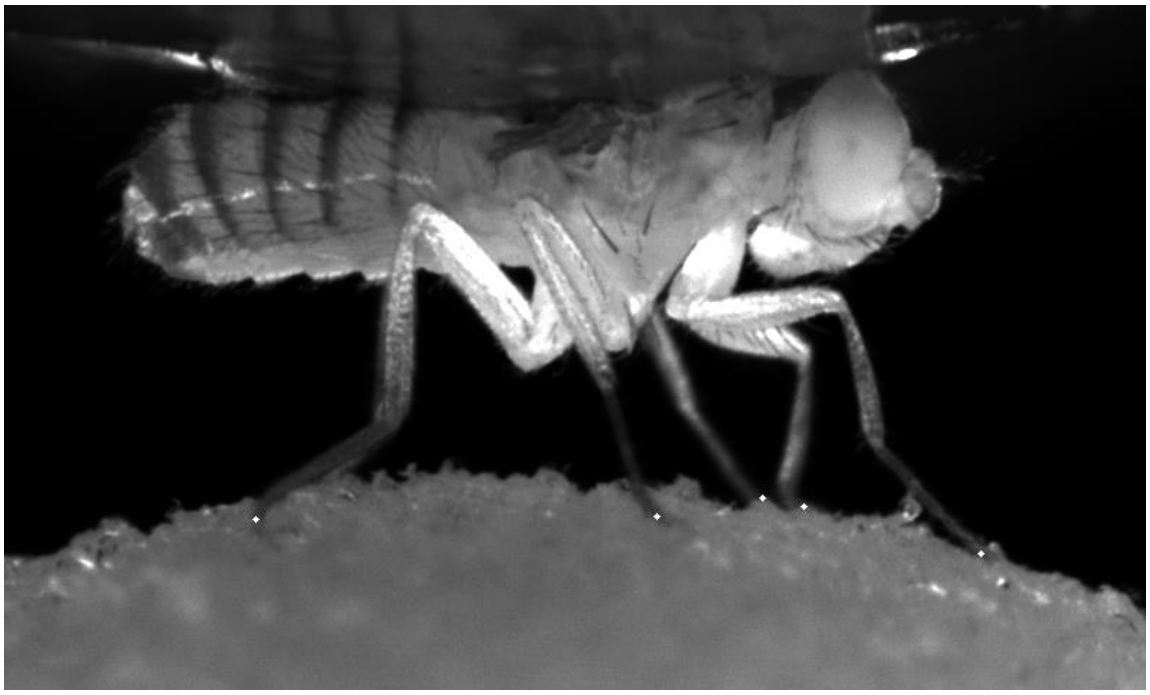
Claws detector
– Documentation –



BIOROB – EPFL

Lucas Massemin

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Table des matières

Introduction

Aim of the document

This document aims at explaining both functional and technical aspects of the claws detector program. The last part further explains how the program can be launched and exploited.

Modifications table

The table below keeps track of the different versions and should be modified in case of major changes.

Autor	Version	Date	Modification
Lucas Massemin	1.0	07/06/2018	Initial deployment

Aim of the detector

This program automatically detects the claws of a drosophila viewed from a certain angle.

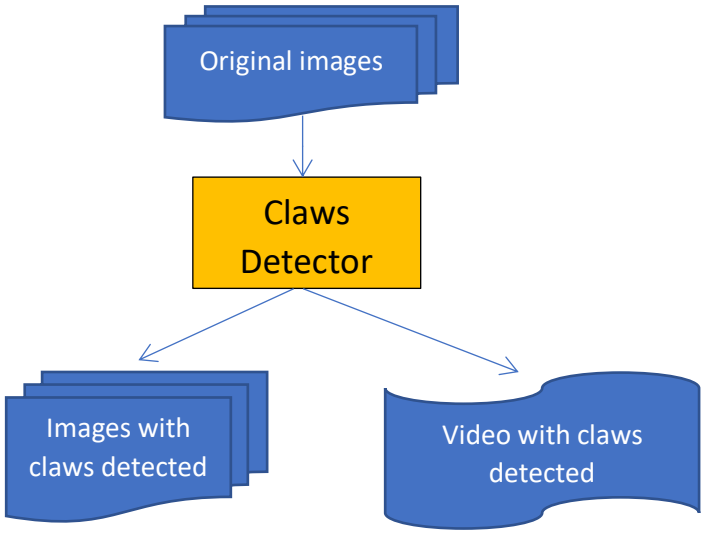


Figure 1 : The global use of a the claws detector

The detector has been designed to be as general as possible, however It is important to note that some parameters do depend on the image resolution, the drosophila anatomy and the ball appearance.

Consequently, major image changes might impact the accuracy and should be anticipated by carefully selecting the parameters.

Image Requirements

The images should be taken from approximately the same camera position but note that the method to spot the claws can easily be adapted to other view angles.

Claws detection method

The claws detector consists of two main phases.
The first one is the pre-processing of the image to get only interesting legs parts.
The second is the retrieval of claws coordinates using a connected components analysis.

Pre-processing pipeline

The pre-processing actions are presented below, in the order they are done.

- Calibration and automatic body detection
- Pseudo difference
- Precise detection of the ball background
- Detection of outer legs
- Detection of inner legs
- Merging of inner and outer legs

Calibration and automatic body detection

The program computes several masks useful for the claws detection. They are named in the following way:

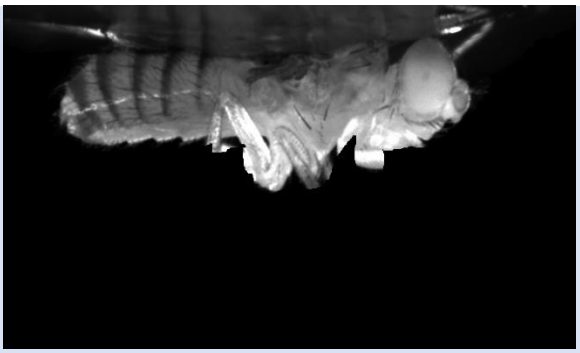
Region	Mask
the static body	`static_body_mask`
the legs	`legs_bottom_mask`
the 'extended' ball region	`ball_roi_mask`
the 'restrained' ball region	`ball_mask`
the right claw region	`right_mask`

When reading this section, remember that the width grows left to right, and the height grows top to bottom.

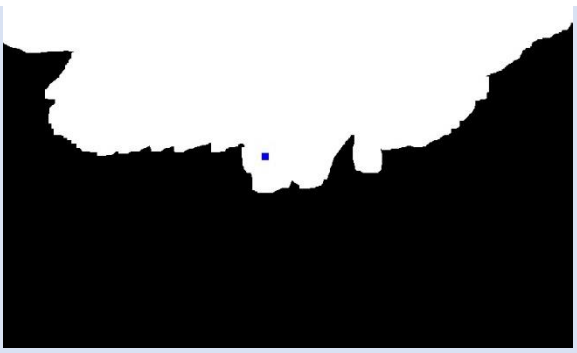
In addition to those masks, the calibration phase computes the approximated position of the thorax:

- Its width is the median of the widths of highest pixels of `static_body_mask`,
- Its height is their height – 50.

The thorax is displayed in the `static_body_mask` image as a blue point, note that the "-50" is resolution dependent



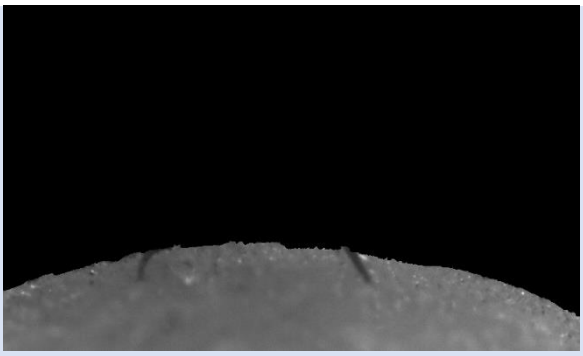
Original image & static_body_mask



static_body_mask

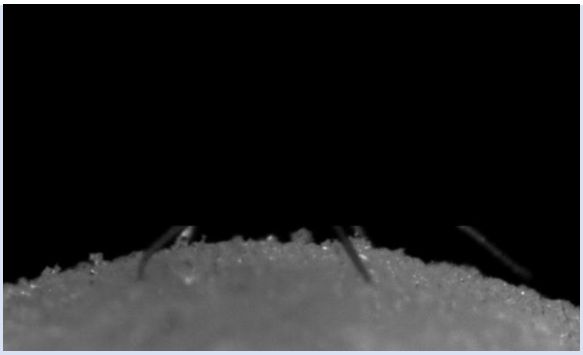
The `static_body_mask` and the `ball_mask` have been computed by doing the `bitwise_and` operation on 100 images thresholded binarily to detect pixels whose values are over 20.

This resulted in two big components, one for the ball and one for the body. The legs were suppressed by the black background. We associate the components to body or mask by looking at the heights of their centroids.



Original image & ball_mask

ball_mask

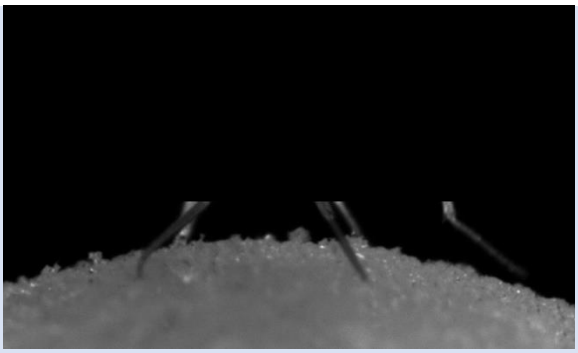


Original image & ball_roi_mask

ball_roi_mask

The `ball_roi_mask` is a mask that consider all the rows whose heights are above the height of the white pixel in `ball_mask` with minimum height – 20.

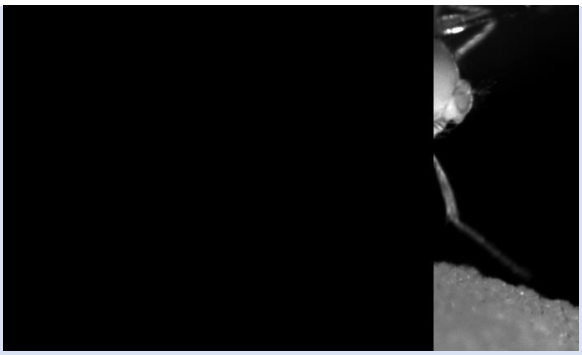
We retrieve 20 because we also want to include the irregularities of the ball. This is a ball-dependent safe value that should not change although “hardcoded”, it represents the max size of a ball irregularity.



Original image & legs_bottom_mask

legs_bottom_mask

The `legs_bottom_mask` starts considering rows when their height is beyond the mean of the lowest and the biggest heights of pixels in `ball_mask` and `static_body_mask` respectively.



Original image & right_mask

right_mask

The `right_mask` starts considering columns when they are 213 pixels over the width position of the drosophila thorax. Legs in the right part can be spotted if and only if they belong to this mask.

We note five points:

- This method requires images on which the legs are moving.
- The `static_body_mask` comes with noise at the top, due to the plate.
- The `static_body_mask` also captures segments of legs, see the claw at the top right corner.
- The `ball_mask` does not capture the irregularities at the top of the ball.
- The `ball_mask` also captures segments of legs.

Those masks are computed only once at the beginning and saved locally.

Pseudo difference

The goal of the pseudo difference operation is to retrieve the leg segments of the top right corner that have been declared as fixed by the `static_body_mask.`

The idea is to compare the current image with a reference image that has no leg in the top right corner.

Furthermore, as the fly can move a bit, we would like the operation to be robust enough not to consider a part of the head to be a leg. Consequently, the head in the reference image should be close to the position of the current image, but with no leg in the top right corner.

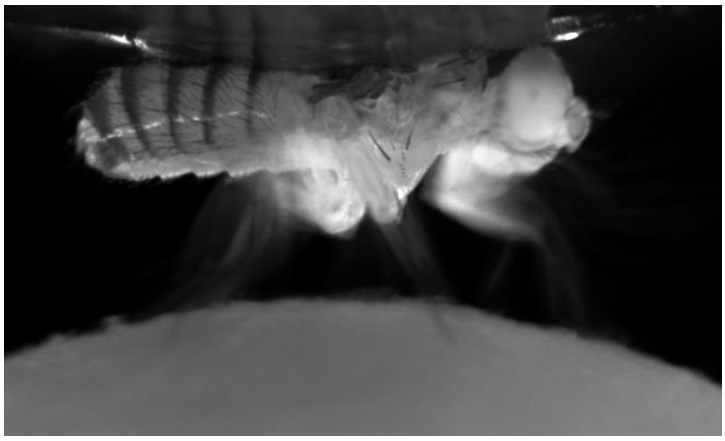


Figure 2 : A reference image obtained by meaning 100 images

To achieve this result, we exploit the fact that the head and the legs do not move at the same frequency.

As a consequence, taking the mean of the collection of images used to determine the masks should give something blurry (if the head moved) and bright for the head, whereas the legs that move a lot (especially in the top right corner) should be almost invisible (figure 2).

The reference image is computed only once during the calibration phase and is saved locally.

The pseudo difference roughly consists in subbing the reference image to the current image and keeping the pixels that are bright enough (here, higher than 40).

More precisely, the components consisting of the resulting pixels are dilated and eroded to fill the holes and are then kept only if bigger than a certain threshold (here 400). The evolution of the pseudo difference for pixels of the right part masked with `static_body_mask` is showed in figure 3, 4, 5, 6.

Figure 3 : Original image

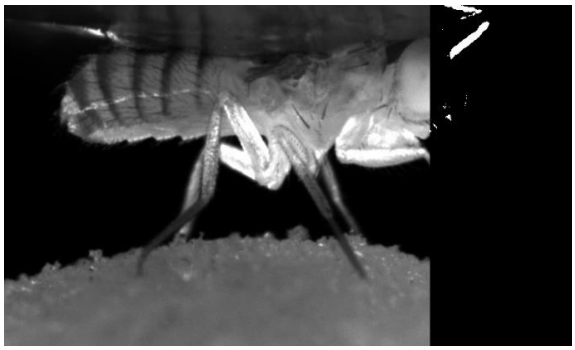


Figure 4 : The p-diff operation, subbing mask only

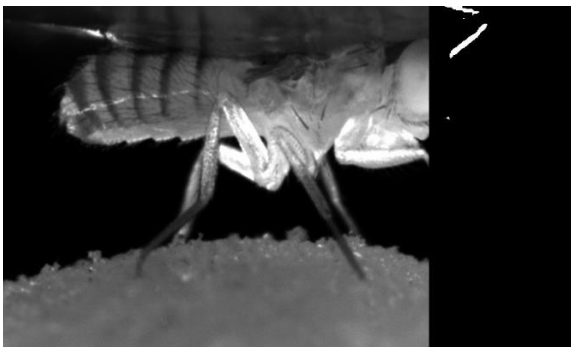


Figure 5 : The p-diff operation, subbing mask + dilate + erode

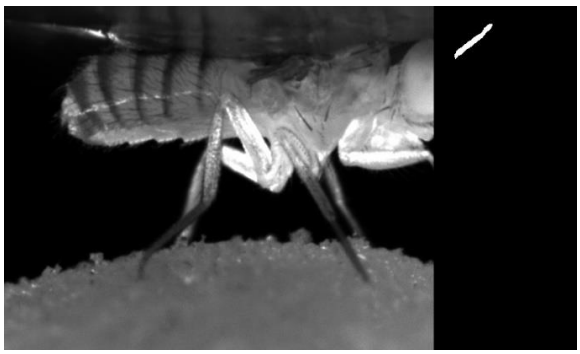


Figure 6 : The complete p-diff operation

The right part is now pre-processed, we simply construct a mask considering pixels in p-diff and not in `static_body_mask`.

Precise detection of the ball background

To spot the legs parts out of the ball at the bottom of the image, we need to distinguish them from the ball. To do so, we need to know exactly where the ball is, irregularities included.

The naïve approach consisting of thresholding fails miserably because some legs have almost the same colour as the ball, we consequently use another trick.

This operation focuses on the part defined by the `ball_mask`, which will thereafter be referred as 'the image'.

First, we use basic thresholding to spot the black background. The resulting image is shown in figure 8.

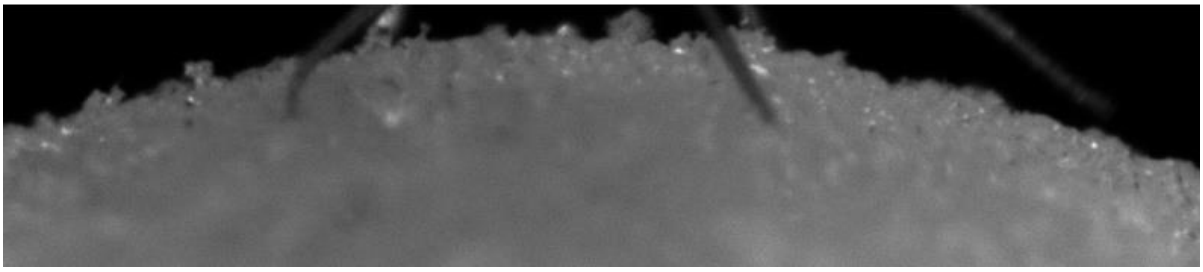


Figure 7 : Original ball_mask part

Figure 8 : ball_mask part thresholded

Figure 9 : ball_mask part thresholded, background components colorized (not done in the program)

We see in figure 9 that the legs touching the ball give birth to distinct background components, each colorized with its own colour.

A way to get rid of the legs, without losing the irregularities is to delete a black part if and only if it is contained between two different background components. Of course, doing so would also remove a huge part of the ball so we need to bound the number of pixels that we can delete at once.

This threshold corresponds to the apparent 'thickness' of a leg, and is set to 20.

We also need to define an axis along which we should search for labels.

Ideally, this axis would always be aligned with the surface of the ball. However, we cannot afford this computation and we simply approximate it using the horizontal axis. The work is quickly done by the `detect_exact_ball` function. It saves a lot of time and it is not too bad in practice as legs are mainly at the top of the ball. We also remove the components others than the ball (figure 11) who might exist if a part of a cut leg is too large (see figure 10).

The drawback of this trick is that a ball irregularity merging with a leg would be considered as a second leg if there is some background between them or would make the leg too thick to be deleted.

Figure 10 : The exact ball, no filter on components sizes

Figure 11 : The exact ball, components filtered

Detection of outer legs

'Outer' legs are the legs that are not superposed to the ball background on the image.

Now that we know precisely where the ball background lies, we can simply ignore it by setting all the pixels to 0.

We also ignore the fixed part, but we remember the result of the p-diff operation (figure 13).

We can binarily threshold to get a mask for the outer legs (we use 15 as a threshold value), but the ball edge seems shady and appears in our mask (figure 14). To solve this issue, we apply a median blur with a squared kernel of size 5 (figure 15).

Figure 12 : The original image

Figure 13 : The original image masked with p-diff + no ball

Figure 14 : Image of figure 13 thresholded

Figure 15 : Image of figure 14 after a median filter

Detection of inner legs

'Inner' legs are the legs that are superposed to the ball background on the image, as opposed to 'outer' legs.

They are simple to spot once we know the exact ball background mask because they are darker that the ball.

We simply do a double binary thresholding, between 5 and 70 (figure 16). As usual, the ball edge is a problem and we need to apply a mean blur (figure 17).

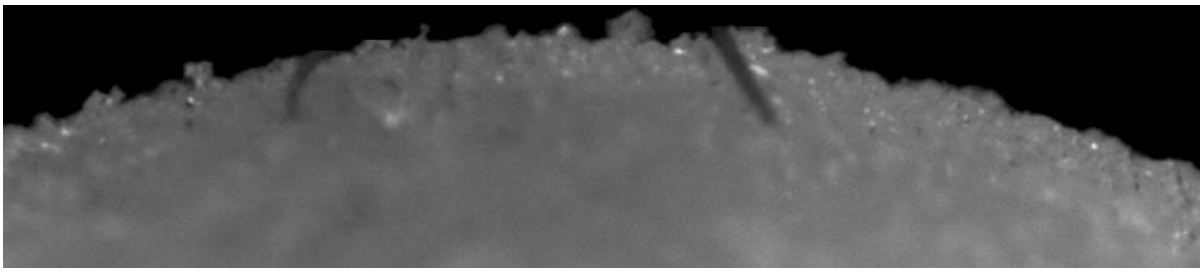


Figure 16 : Original imaged masked with the exact ball background

Figure 17 :Image of figure 16 thresholded between 5 and 70

Figure 18 : Image of figure 18 after a median filter

We can see that there are still invalid components left, but we later manage them at once for inner and outer leg.

Merging of inner and outer legs

We now aim at linking matching components for inner and outer legs together, we thus focus on a leg region define by the `legs_bottom_mask` and the right part.

Exploiting the previous results and after a median filter of 7, we obtain the result of figure 19.

We then dilate and erode to fill the holes, we also remove the connected components that are too small (figure 20).

Figure 19 : Inner and outer legs displayed together

Figure 20 : Merging done of inner and outer legs done

Claws coordinates using a connected components analysis

Once the original image is pre-processed, the claws detection algorithm can begin. As shown in figure 19 and 20, we only consider the claws region.

RLE encoding

The algorithm for claws detection heavily relies on RLE encoding, this subsection is a reminder of what is RLE.

RLE (or Run-Length-Encoding) is an encoding scheme that consists in representing each sequence of identical characters by the length of the sequence and the character:

[1, 1, 1, 2, 2, 5, 5, 5, 5, 6, 7] becomes [3122451617]

The RLE encoding obviously applies to images as well.

Algorithm

The algorithm first computes the connected components of the pre-processed image.

If it finds 6 connected components it means that there is one per leg and the algorithm simply returns the point of each component that is the farest to the thorax. This distance computation is done in the function `spot_claws_by_dist`, it is referred to as the classical method.

If the number of components is smaller than 6, it means that there is an intersection between the legs. Here, by intersection we mean that the legs overlay on the image.

In case of an intersection, we use the RLE to compute the number of legs in each row of the `legs_bottom_mask` part. The number of legs per label in a given row is simply the number of sequences of this label separated by black background.

The resulting sequence of numbers of legs (NOL) is noisy because of the image quality and the pre-processing. This means that there may be NOL too high (e.g. 5) or very close successive changes in the NOL. Invalid sequences of NOL are characterized by their very short length, which allows us to design a tool to delete those sequences and merge the others. The tool is used in the function `merged_rle`.

For each label, we compute a sequence of NOL over the rows. If this sequence contains a sequence of '2' longer than a given threshold then we can reasonably think that this label represents an intersection. Sequences of '2' not deleted by RLE merging can happen because of inadequate pre-processing, in that case they are usually short. Thus, the threshold is necessary so that this noise is not considered as an intersection.

In case an intersection is suspected, the program goes to the end of the last sequence of '2'.
Assuming one can only go from '2' to '1' or '0' in term of NOL, the program determines which leg ended.

If both ended, we remember the position of the one closest to the thorax, as it will not be detected by the method using distances.

If only one ended, the program approximates the end of the leg that did not end, compare it with the end of the other leg and remember the closest point to the thorax.

If we find enough intersections in the bottom part to reach 6 components, we do the classical method and return its result as well as the intersection points we remembered. Otherwise, we look for intersections in the right part.

If we find an intersection for the same label in both part, the results of the right part will get the priority on those of the left part.

Once we looked at the two parts, we return what we found as well as the result of the classical method.

Merging RLE

As said above, the RLE may be noisy. In that case the merging method cleans it by using the following two simple rules:

- only labels whose sequence length is below the error size can be ignored
- Ignore a label if and only if it is not a transition between two different labels (the adjacent valid labels must be the same)

We have the following result for a toy example :

```
        NORMAL RLE
labels [ 1  2  3  1  2  1]
pos    [ 0  1  8 11 18 20]
nums   [ 1  7  3  7  2  6]


        MERGED RLE
    labels [ 2  3  1]
    pos    [ 0  7 10]
    nums   [ 7  3 15]
```

Python Implementation

The implementation architecture is very simple and consists in a few functions. Those functions are embedded in a class ClawsTracker that can be instantiated by giving the calibration images.

Functions

The main functions are given in the table below.

Function name	Function behaviour
circled_claws_image(img_name)	Returns the image with the claws spotted and the coordinates of the claws
process_images(img_names, output_names)	Spots the claws in the images designed by 'img_names' and saves the results as designed by 'output_names'. Returns a list of the coordinates of the claws for each image
write_video(img_names, video_name)	Combines the image designed by 'img_names' in a video whose name is given by 'video_name'
video_from_scratch(self, img_names, output_names, video_name)	Applies process_images and write_video successively, returns the coordinates of the claws for each image.
spot_claws_advanced(labels, num_comp, bottom_part_height, right_part_width, thorax_point, min_separating_dist=10, min_dual_length = 8, error_size=5) :	Spots the claws of the drosophila and return their coordinates
distance(p1, p2)	Returns the manhattan distance between two 2D points
find_intersection_claw(comp_labels, thorax_point, num_to_find) :	Returns the coordinates of claws that cannot be spotted by the classical method
rle(array)	Run length encoding of the array given as a parameter
merged_rle(array, ignore_length)	Returns rle whose sequences shorter than 'ignore_length' have been ignored
detect_exact_ball(img, thickness)	Spots the exact ball position
spot_claws_by_dist(labels, thorax_point, nb_components)	For each component returns the coordinates of the point that is the farest from 'thorax_point'
clean_fixed_part(img, fixed_mask, from_width)	Retrieves the part of the image in fixed_part that is actually not fixed
get_masks()	Computes the masks and the thorax position
remove_comp_smaller_than(min_size, img)	Removes the components smaller than 'min_size' in the img
pseudo_diff(body_ref, current_body)	Computes the pseudo difference between current_body and body_ref

Speed

The processing of an image takes approximately 0.12 second. The approximated processing times are given in the table below for our above example.

Operation	Time (seconds)
p-diff	0.0080
exact ball mask computation	0.0140
outer legs pre-processing	0.0010
Inner legs pre-processing	0.0070
naïve merging (no dilatation/erosion/components removal)	0.0130
final cleaning	0.0079
Total pre-processing time	0.0460
Claws coordinates using a connected components analysis	0.0570
Total time	0.1090

Program execution

How to spot the claws in specific images

The program has no graphical interface to select the images to process. Instead, one should create two lists containing paths for input and output. The `process_images function` iterates on the input list, process the corresponding image and save the result in the input path. It returns the coordinates of the detected claws.

How to create a video from a sequence of specific images

One can combine images whose names are in 'img_names' and save it in a video called 'video_name' by calling the `write_video` function.

It is also possible to make a video from not yet processed input files by using the function `video_from_scratch.`

Example of execution

The following snippet of code builds two lists, 'inputs' defining where the images should be retrieved, and 'output' to indicate where the results should be stored.

It instantiates a ClawsTracker using the inputs (they are used to calibrate the ClawsTracker) and process the inputs using the `process_images` function. After that, it reuses the output images to build a video and name it after the 'video_name' variable.

Note that **the output folder must exist**, otherwise the programs silently fails. Moreover, a wrong inpu t file path would generate the error `<'NoneType' object has no attribute 'shape'>.` Any existing file located at a path in the 'outputs' variable will be overwritten.

```
from claw_tracker import ClawsTracker
import cv2


range_size = 100

# images from path 'examples/camera_0_img_0.png' to 'examples/camera_0_img_99.png'
inputs = ["examples/camera_0_img_" + str(i) + ".png" for i in range(range_size)]

# images from path 'test/0.png' to 'test/99.png'
outputs = ["test/" + str(i) + ".png" for i in range(range_size)]

video_name = "legs_spotted.avi"

# instantiate tracker and calibrate it with input images
ct = ClawsTracker(inputs)

# process input images and get the coordinates
coords = ct.process_images(inputs, outputs)

# write a video consisting of the processed images
ct.write_video(outputs, video_name)

#... or do the above in one line
coords = ct.video_from_scratch(inputs, outputs, video_name)
```