# Undergraduate Research on Yellowjacket Flight Patterns

By Marc Roizman, advised by Dr. John Ringland
Completed Sept 2019 - June 2020

## 1. Objective

The objective for this research is to analyze the flight patterns of yellowjackets in recorded videos. Using machine learning techniques, we hope to be able to extract useful information from the frames of these videos, such as 3-dimensional location and direction (angle). In order to analyze movement patterns, it is essential to be able to obtain the relevant data at a very high level of precision.

---

## 2. Footage

So far, the research has primarily been conducted using 2 videos:

1. A YouTube video (7:39 length) recorded by user *Oakheart*, entitled "Messing with Yellow Jacket Wasp Nest - Flight Patterns". In their video, a branch is dropped onto a yellowjacket nest, and dozens of them swarm out to inspect the new object. Video is 1080p 30fps, shows many yellowjackets, somewhat blurry at times
2. A video (1:02 length) recorded by Dr. Ringland, entitled "new_yellowjackets.mp4". Not online. Yellowjackets fly around a table with some food on it. Video is 1080p 60fps, only a couple yellowjackets, details are very clear.

---

## 3. Image Processing

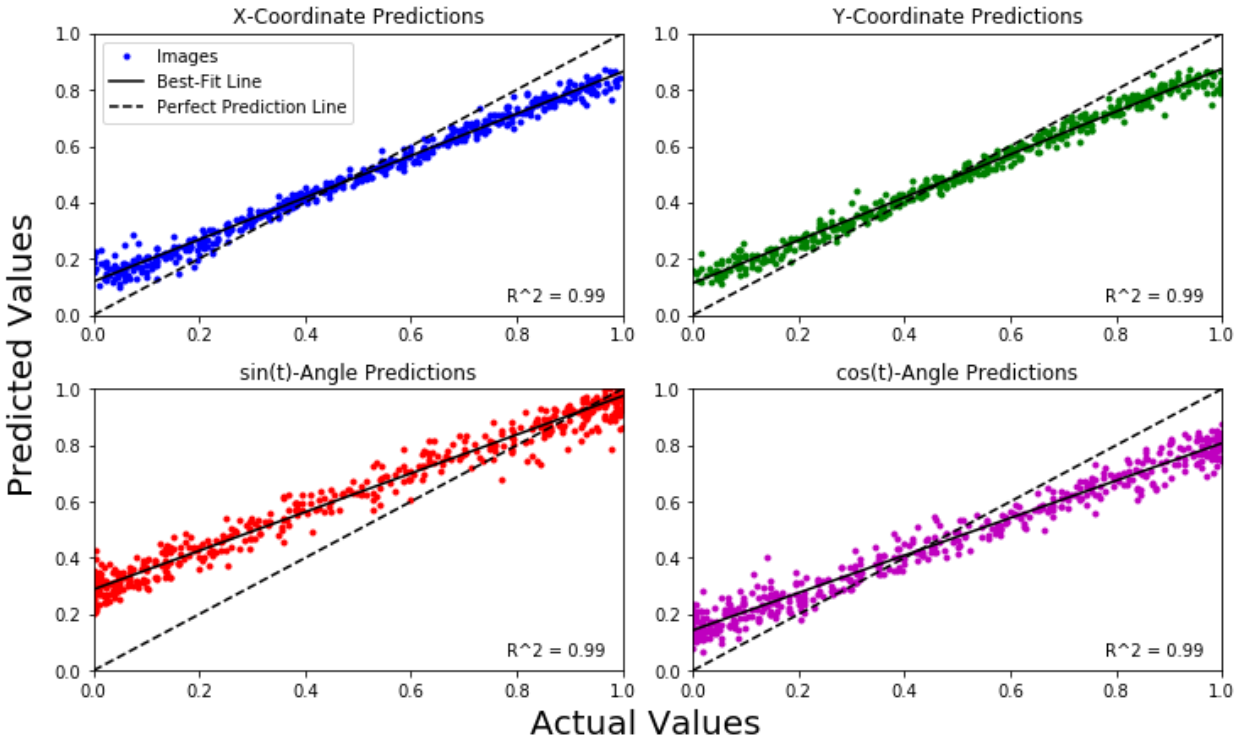
Originally I generated "differences" of frames in the video (image_diff.py), however this approach was flawed in that it resulted in images with overlapping yellowjackets.

The approach which we are using now is to generate an "average" frame based on some subset of frames in the video (I chose the first 100 frames), and then take the difference between every frame and the average (image_avg.py). This gives a set of mostly black images with the yellowjackets clearly visible in white. On the right is a zoomed in example of one of these frames. This approach may need adjusting in a video with a volatile background.

---

### 4. CNN Detection (Unsuccessful)

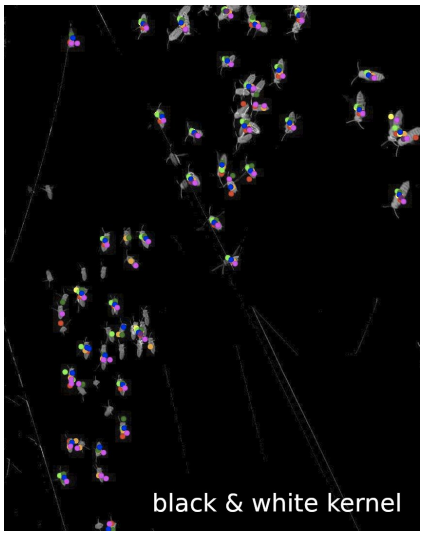
Our first idea was to use a Convolutional Neural Network to predict certain continuous features from yellowjacket images (more detail on full approach in Fall 19 Research Report v2.pdf). Given an image, I used TensorFlow and Keras RetinaNet to train a CNN to predict x-position, y-position, and the sine and cosine of the yellowjacket's angle. This approach ultimately fell short due to slow convergence. Running the CNN on a set of "ideal" artificially-generated test images (essentially a best-case scenario for image detection), the test error was still somewhat large. The below graphs reflect our test accuracy after 200 epochs of training on 1,500 images.



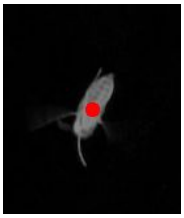

---

### 5. Kernel Detection (Successful)

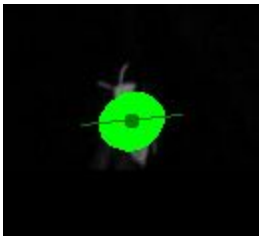
Another attempt at detecting orientation involved the use of kernels (kernel_convolution.py). By creating 8 black-and-white kernels, one for each cardinal orientation (N,NE,E,...,NW), and convolving each one with the yellowjacket images, we hoped to be able to get an output reflecting the yellowjacket's orientation. Unfortunately,



the kernel outputs are not different enough to conclusively determine which direction each yellowjacket is facing. As the image shows, many yellowjackets have a large number of dots covering them (multiple different dots on a yellowjacket indicate that it was incorrectly detected as facing multiple different directions).



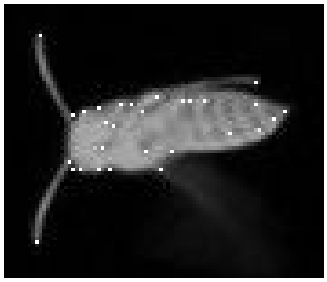
We later returned back to this kernel convolution method in an effort to detect the approximate *location* of yellowjackets within an image. This was very successful—convolving a white circle kernel with the images of the yellowjackets, then thresholding the output, helped find the approximate locations. From there, just using the centroid of this output proved to be an extremely effective way to identify yellowjackets.



Following this, I tried a simple linear regression over all of the points in the convolution blobs. The hope was that this line would be a reasonable approximation for the yellowjacket's angle. Unfortunately, the output is too amorphous to get a good sense of orientation. The example image shows how this method failed to capture orientation accurately.

---

### 6. Corner Detection (Successful)



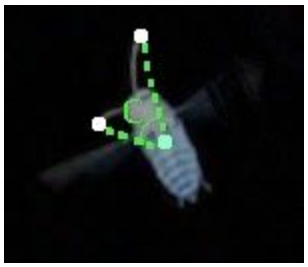
CV2 has a built-in corner detection function which uses the Shi-Tomasi Corner Detector. The image to the right shows the 50 strongest corners which this detector identified on an example image. My first idea was to perform a linear regression over the corners, to try to get an approximate major axis. While this was unsuccessful, I noticed in the process that specific points on the yellowjacket always seemed to get picked up: the tip of each antenna, the root of the antenna (where it meets the head), and the stinger. Additionally, the antenna tips and the stinger seemed to be both very "strong" corners, and very distant from the centroid.

---

## 7. Guessing Orientation from Corners

The methods described in this section are implemented in corner_detection.ipynb

The first thing to look for are the **antenna tips**. These tend to be the most unique points identified. getTipScore takes in an image along with a list of pairs of points as an input. The output is a list of scores (for each pair), calculated as follows:



$$Score = (15° < C < 120°) * [D + (1 - B) + (1 - R)]$$

**C = Interior angle** formed by drawing lines from centroid to points. This should be between 15° and 120°, otherwise the pair is given a score of 0.

**D = Average distance** from test points to centroid (normalized w.r.t farthest points). This should be as large as possible.

**B = Average brightness** in neighborhood near each test point (normalized w.r.t brightest points). The size of the neighborhood can be adjusted. This should be as small as possible, hence why (1-B) is used in the score computation.

**R = Relative difference in distances:** $R = \frac{|distance\ from\ P_1\ to\ centroid - distance\ from\ P_2\ to\ centroid|}{distance\ from\ P_1\ to\ centroid + distance\ from\ P_2\ to\ centroid}$ . This should be as small as possible, hence why (1-R) is used in the score computation.

Once the tips are identified (by choosing the maximum score), we're interested in identifying the **antenna roots**, or the points where the antennae meet the head. getRootScore takes in an image, a list of pairs of points, and a pair of tip points as an input. The output is a list of scores (for each pair), calculated as follows:

$$Score = (C < 60°) * [(1 - D) + (1 - B) + (1 - R)]$$

**C = Interior angle** formed by drawing lines from centroid to test points. This should be below 60°, otherwise the pair is given a score of 0.

**D = Average distance** from test points to each *tip* (normalized w.r.t farthest points). This should be minimized; one would expect the roots to be very close to the tips, compared to other points around the image.

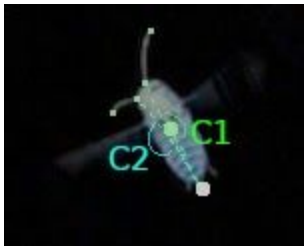
**B = Brightness proximity to 0.5*MaxBrightness**. Uses neighborhood brightness near each test point. $B = 2 * |avgBrightness - 0.5|$. B should be as small as possible (indicating a value close to 0.5*MaxBrightness).

**R = Relative difference in distances:** $R = \frac{|distance\ from\ P_1\ to\ centroid - distance\ from\ P_2\ to\ centroid|}{distance\ from\ P_1\ to\ centroid + distance\ from\ P_2\ to\ centroid}$ . This should be as small as possible, hence why (1-R) is used in the score computation.

Finally, given the tips (again choosing the max score), we can obtain a guess for the **stinger** point. Once we have this, it's trivial to compute the orientation. getStingerScore takes in an image, a list of points, and a pair of root points as an input. The output is a list of scores (for each point), calculated as follows:

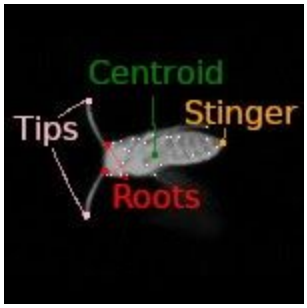
$$Score = (C_1 > 90°) * (C_2 > 90°) * [D + (1 - B) + (1 - R)]$$

**C1 = Interior Angle** formed by drawing lines from centroid to a root and to the stinger test point. Should be larger than 90°, otherwise Score = 0.
**C2 = Interior Angle** formed by drawing lines from centroid to a root and to the stinger test point. Should be larger than 90°, otherwise Score = 0.
**D = Distance from test point to centroid** (normalized w.r.t farthest point). Should be large, since stinger is far from centroid.
**B = Brightness proximity to 0.5*MaxBrightness**. Uses neighborhood brightness near each test point. $B = (3/2) * |avgBrightness - (1/3)|$ . B should be as small as possible (indicating a value close to 1/3*MaxBrightness).




**R = Relative difference in distances:** $R = \frac{|distance\ from\ Root_1\ to\ stinger - distance\ from\ Root_2\ to\ stinger|}{distance\ from\ Root_1\ to\ stinger + distance\ from\ Root_2\ to\ stinger}$ . This should be as small as possible, hence why (1-R) is used in the score computation.

To the right, you can see a diagram explaining the output of corner_detection.ipynb. Antenna tips are color-coded pink, roots are colored red, and the stinger is yellow. Additionally, the centroid will be green, and very small white dots show other corners which OpenCV picked up.

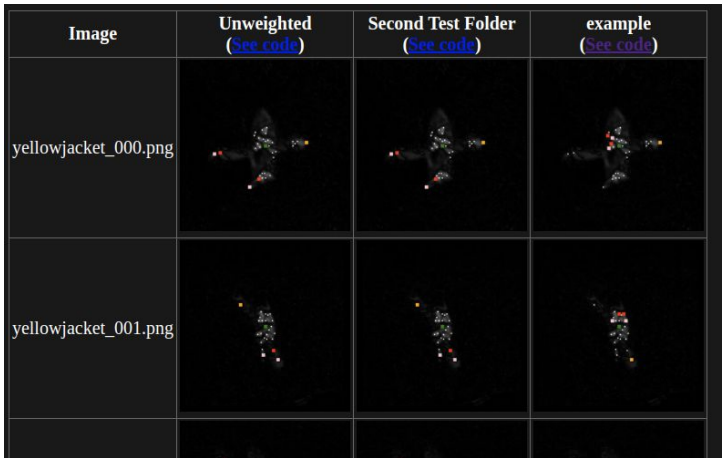



The formulas to detect all of this are purposely kept relatively generic, and more importantly, adjustable. The parameters are not heavily tested, and anytime a specific value was chosen, it was probably by visual inspection. These are meant to be adjusted, and any future work would likely require this. I would particularly recommend trying to experiment with weights/coefficients for the terms, as clearly some parameters are more important than others.

---

## 8. Future Work

The next phase of this research project will involve tweaking the model parameters and weights to better detect the different components of the



yellowjacket. To make this easier, I've created create_html.ipynb. This will allow you to generate side-by-side comparisons of various formula changes as an html page (shown on the right). One could try out several different weights and compare the outputs directly, side-by-side.

An example workflow might look like this:

- Begin with a new mp4 video of yellowjackets
- Use image_avg.py to generate black and white frame-by-frames
- Run the frames through corner_detection.ipynb to generate processed frames
- Use create_html.ipynb to generate an HTML page containing your processed frames, or multiple different sets for comparison. Keep everything in an "html" folder!