Climber recognizer: a Signal, Image and Video project

Roman Simone
DISI, Università degli Studi di Trento

1. Introduction

This project have two main goals:

1. **Development of Route Recognition Software:** This first objective consists in identify indoor climbing routes simply by looking at a picture and knowing the color of holds on the route. In addition to a small demo, a graphical interface has been implemented that allows the user to crop the image to choose only the region of interest and then to choose the color of the holds on the route and then through image processing to identify the holds on the route and a line that represents it.
2. **Development of Climber Tracking System:** The secondary objective is to establish a robust tracking system for climbers, using a video of the climb as input. Two distinct modes are employed for this purpose:
   - **Optical Flow Mode:** Employing optical flow techniques to track the movement of the climber within the video.
   - **Background Subtraction Mode:** Employing background subtraction methodology to isolate and track the climber against the dynamic backdrop.

All the project was developed in Python using `OpenCV` library, which is highly specialized in computer vision tasks and in frame-by-frame analysis. All the source code can be checked in my GitHub.

2. Route Recognition

This first application aims, given an image as input, to detect climbing holds and compute a line that represents the route. In order to perform this, three key steps are performed:

1. **Mask Generation**: Initially, the application generates a mask to isolate the climbing holds within the image.
2. **Contour Detection and Connection Establishment**: Following the mask generation, a function is utilized to detect the contours of these holds. Subsequently, connections between these holds are established based on their proximity.
3. **Route Visualization**: Finally, another function is employed to visually represent the climbing route. This involves drawing a line representing the route and which hypothetically the climber should follow.

2.1. Mask

The first step is the creation of the mask. Taken the input image the first process is the application of a **Gaussian blur filter** to the original image. This is done to reduce noise in the image and make the image ”smoother” and the colors more uniform.

After that, the blurred image is converted from RGB (Red, Green, Blue) to **HSV (Hue, Saturation, Value)**. This is done because it is easier to work with color in HSV than RGB.

Obtained the image in hsv it’s possible to create a **binary mask** where pixels that have a color between lower_range and upper_range (this range represent the color of the route) are set to white (255) and all other pixels are set to black (0). This is done to isolate pixels in the image that are of a certain color and so make it easy to recognize the contours of them.

The last operation that is performed is to **dilate** the edges of the found holds this is done because climbing holds are often fouled with chalk and thus the color of them is altered by reducing their area.

The figure 1 shows all the transformations explained.

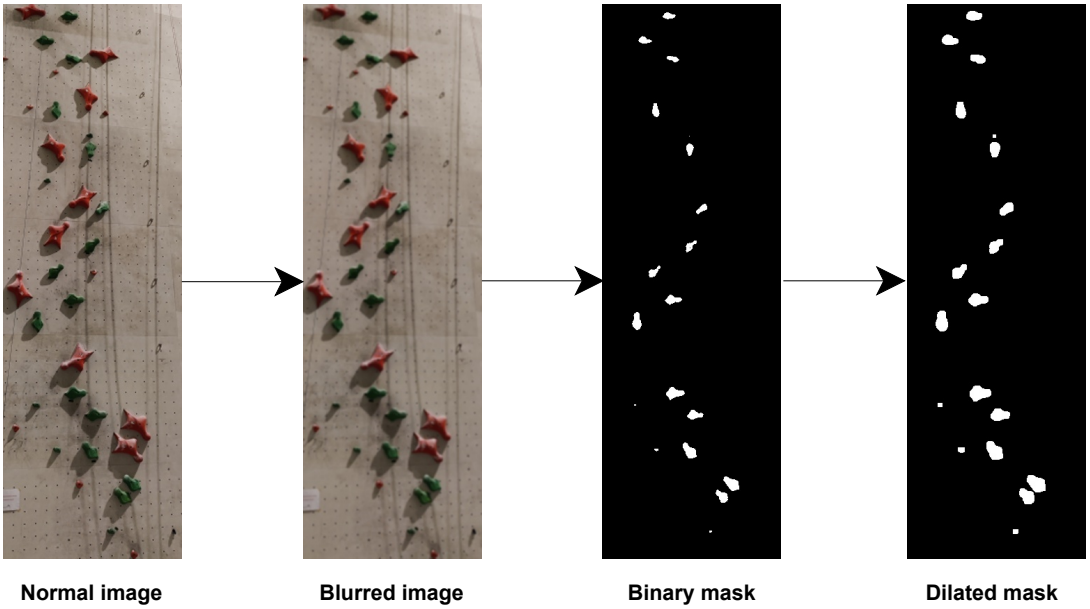

Fig. 1. Mask creation

2.2. *Detect contours*

The second step is to identify climbing holds. In order to detect the edges of the holds I use the **canny** function which by using the intensity gradient can detect the edges, the result of the canny algorithm is a binary image where the edges are white and everything else is black.

After doing so, through the function **findContours** you can obtain the contours of the holds. The result of the function is a list of found contours. Each contour is represented as an array of points.

From the list of contours just obtained, filtering is done to remove areas that are too large or too small, and a new list is created with the centers of the contours that will be needed later. With the list of contours filtered, the contours can be drawn on the original image with the function **drawContours**.

Finally as a last step thanks to the previously calculated contour centers it is possible to draw a line between the nearest climbing holds to simplify the view of the route with the function **line**.

The figure 2 shows all the transformations explained.

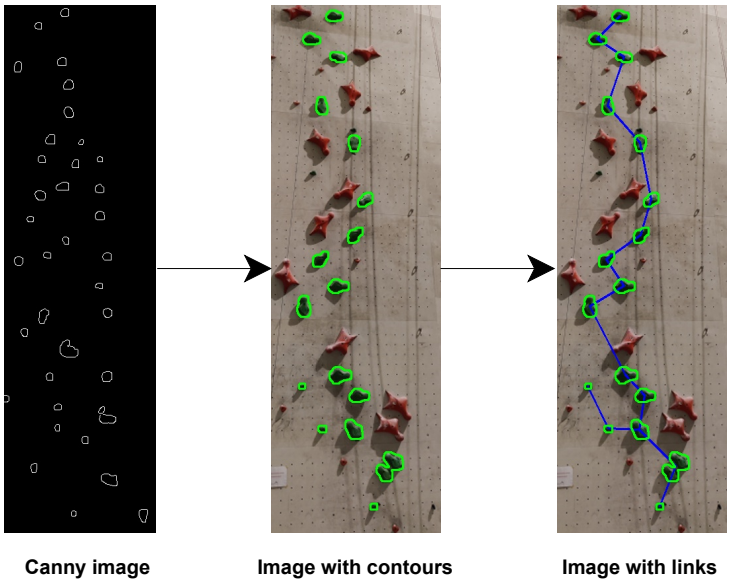

Fig. 2. Detect contours

2.3. *Route line*

The third and final step is the processing of the **route line**. Usually in a climbing book or charts it is the only information shown. The idea is to take x number of climbing holds (the centers of the holds contour) and calculate the midpoint between them. Then, by scrolling through the holds in order of height, one can connect these midpoints and create a hypothetical line that the climber must follow to make the route.

The figure 3 shows an example of route line.

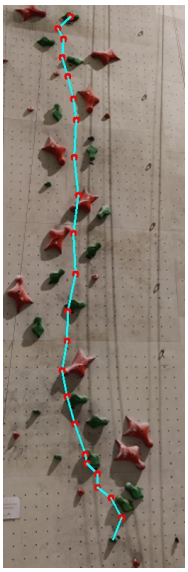

Fig. 3. Route Line

3. Climber Tracking System

This second application aims to detect (create the bounding boxing) a climber given a video as a input. Several assumptions are made regarding the input video:

- The video must be stable
- The lighting should be constant without any fluctuations in intensity or changes in color
- The distance should not change

Initially, the optical flow method was used for detection. However, this approach tends to introduce flickering issues. Therefore, in addition to the optical flow, the background subtraction method (bgsubtractor) was also proposed as a solution. While the latter method may be less precise, it effectively mitigates flickering problems encountered with optical flow.

3.1. *Optical flow*

The optical flow method is a fundamental technique in video processing used to estimate the movement of objects within a sequence of images or frames from a video. It operates by tracking the displacement of pixels between consecutive frames, providing a displacement vector for each pixel that indicates both the direction and speed of its movement over time.

The underlying principle of optical flow relies on the assumption that neighboring pixels in an image tend to move in a similar manner. By analyzing changes in pixel intensities between frames, optical flow algorithms can discern patterns of motion within the video.

The approach that I use is **Gunnar Farneback**, this method method consists of several steps:

1. **Image Pyramids**: The algorithm begins by constructing image pyramids for each frame. These pyramids consist of a set of scaled images (or levels), where each level represents a smoothed and downsampled version of the original image. This is done to handle different scales of motion.
2. **Calculate Optical Flow at Each Level**: For each level, the following steps are performed:

(a) Compute the spatial gradient.

(b) Compute the temporal gradient.

(c) Compute the product of the two derivatives.

(d) Apply Gaussian smoothing to the product.

(e) Use the smoothed derivatives to solve a polynomial expansion at each pixel. This results in flow vectors representing the motion at each pixel.

3. **Interpolation and Upsampling**: After obtaining flow vectors at each level, the algorithm performs interpolation and upsampling to refine the flow field, especially for smaller scales of motion.

4. **Final Flow Field**: The final dense optical flow field is obtained by combining the flow vectors from all levels of the image pyramid.

3.1.1. Code overview

The developed code processes the input video, analyzing it frame every 10 frames (to mitigate flickering issues).

For optical flow estimation, the `cv2.calcOpticalFlowFarneback` function from the OpenCV library was used.This function requires several parameters including:

- `prev`: a temporally previous frame in 8-bit single-channel format
- `next`: a frame temporally subsequent to the previous frame, which has the same type and size as the first

In order to have a 8-bit images, both input frames are converted to grayscale. This simplification enhances computational efficiency and allows for accurate optical flow calculation. So, the function returns a NumPy array containing the optical flow vectors computed for each pixel of the input images. These optical flow vectors represent the motion of pixels between the two frames of the video.

Once the flow is obtained, the x and y components of the optical flow (fx, fy) are taken and the angle and magnitude are calculated.

Next, the code converts the angle and magnitude to the **HSV** (Hue, Saturation, Value) color space. It creates a blank HSV image with the same dimensions as the optical flow and sets the Hue channel to angle, the Saturation channel to 255 (maximum saturation) and the Value channel to magnitude, limiting the maximum to 255. After that the HSV image to the BGR (Blue, Green, Red) color space using the `cv2.cvtColor` function and saves the resulting image.

The next step is to convert the BGR image to grayscale and apply a threshold to create a **binary mask** where the only white pixels are where there is motion. The mask is then subjected to a morphological closure operation (dilation followed by erosion) to better show the body contours.

Finally, the last step is to find the contours in the mask through the `cv2.findContours` function and **draws a bounding rectangle** around the largest contour (which ideally is the climber's body) in the original image.

The figure 4 shows the image in hsv, the mask and the final result.

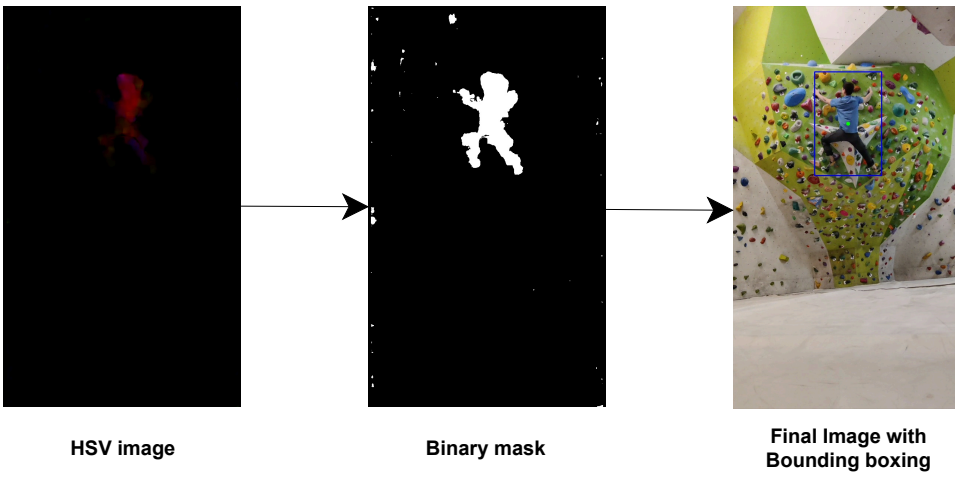

Fig. 4. Optical flow

3.2. Background Subtraction

Background subtraction is a widely used method for segmenting a scene into background and foreground components, particularly for isolating moving parts. In my application, I employed the OpenCV function `createBackgroundSubtractorKNN`, which operates as follows:

1. **Initialization**: The background subtractor object is initialized with default parameters, setting the stage for subsequent processing.
2. **Parameter Learning**: Utilizing an adaptive learning algorithm, the background subtractor continuously updates its internal model of the background. This dynamic learning process considers recent pixel values to adapt to changes in lighting conditions and gradual variations in the background over time.
3. **KNN Algorithm**: The K-nearest neighbors (KNN) algorithm plays a pivotal role in classifying pixels as either background or foreground. For each pixel, the algorithm examines its recent history and compares it with neighboring pixels. By selecting the k-nearest neighbors based on pixel values, the algorithm determines whether the pixel represents background or a moving object (foreground).
4. **Foreground Mask**: The resulting output of the function is a binary mask. Pixels corresponding to the foreground are set to 255, while those representing the background are set to 0. This binary mask provides a clear distinction between foreground and background elements in the scene, enabling further processing to identify and track moving objects within the video sequence.

3.2.1. Code overview

The developed code as first step applied a **Gaussian blur** to the frame image using the `cv2.GaussianBlur` function. This step is essential to reduce noise and enhance the overall smoothness of the image.

Next, background subtraction is performed to isolate a moving object from the background within the video sequence. This is achieved using the `backSub.apply()` method, which yields a **binary image**. In this binary image, pixels corresponding to the foreground are set to 255, while those representing the background are set to 0.

Following background subtraction, a morphological operation is executed, comprising dilation followed by erosion. This process helps in closing small holes and gaps within the image, thereby refining the segmentation of the moving object.

Subsequently, the `cv2.findContours` function is employed to detect and extract the contours of the moving object present in the binary image.

As a final step, drawing a bounding rectangle around the largest outline (which is ideally the climber's body) in the original image takes place

The figure 5 shows the mask and the final result.

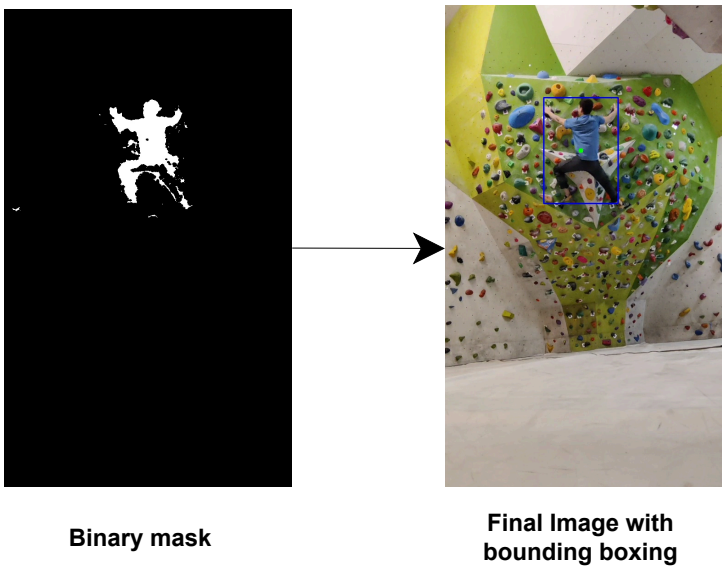

Fig. 5. Background Subtraction

4. Conclusion

As illustrated in this report, there were two objectives: the development of an application for recognizing climbing holds and route, and the tracking of climbers.

- For **holds recognition**, the objective was to identify climbing routes given an input picture and the color of the holds. This involved creating a visual representation of the route using a line and outlining the holds. This goal was achieved, but with some complications. One of the main challenges encountered was the presence of dirt, such as chalk or rubber residue from climbing shoes, on the holds. This dirt often obscured smaller holds, making them difficult to recognize.

  A potential **future improvement** could involve route recognition without relying on color as input. This would introduce an additional level of complexity, as it would require initially identifying holds solely based on their outline. Subsequently, methods such as segmentation could be employed to determine their color and identify routes. Recognizing the outline of sockets without color information is difficult using only image processing techniques. However, by incorporating advanced algorithms like YOLO, which utilize Convolutional Neural Networks (CNNs), more accurate and efficient detection in real-time applications could be achieved.

- For **climber tracking**, the objective was to create bounding boxes around the climber using a video as input. This goal was pursued through two distinct approaches: optical flow and Background Subtraction.

  Optical flow, while slightly more accurate, suffered from slower processing speeds, resulting in flickering issues during real-time video analysis. On the other hand, **Background Subtraction**, although marginally less accurate, produced smoother real-time video output. However, both methods encountered critical issues. It was imperative for the video to be as stable as possible, with minimal fluctuations in environmental brightness, to ensure reliable tracking performance.

  A potential **future improvement** is the use of YOLO for object detection. Integrating YOLO could alleviate the need for video stability and potentially enhance performance. By harnessing YOLO's advanced object detection capabilities, tracking accuracy and robustness could be significantly improved, even in dynamically changing environments.