Documentation

Function Documentation with sources

video_processing.py

- **mp4_to_list_of_arrays**: Extracts frames from an MP4 video and returns them as a list of NumPy arrays of shape (H, W, 3).
- **list_of_arrays_to_mp4**: Writes a list of video frames (NumPy arrays) to a video file in the specified folder.
- **linear_stretch_colors**: Linearly stretches pixel intensities in the input image so that the minimum value becomes 0 and the maximum becomes 255.
- **extreme_stretch_colors**: Converts an image so all non-zero pixel components become 255 and black remain 0. All pixels are then either pure red, green, blue or any combination of them, including back and white.

activity_detection.py

- **saliency_heatmap_list_of_arrays**: Applies static saliency detection to each frame and returns a heatmap overlay. It works by converting to grayscale, then applying FFT to it and to a Gaussian blur of the grayscale image, then substracting the two. Source of the OpenCV module: Saliency Detection: A Spectral Residual Approach by Hou and Zhang (2007).
- **keep_most_red_pixels**: Keeps the most red pixels above a defined threshold.
- **background_subtraction_list_of_arrays**: Builds a Gaussian model of each pixel color by updating it frame by frame, allowing to highlight movement. For instance if a pixel color changes too much (too much being determined by the mean and variance of the model's pixel), it updates the Gaussian model. Source: Adaptive background mixture models for real-time tracking by Stauffer, C., & Grimson, W. E. L. (1999).

user_selection.py

select_rectangle: Allows the user to draw a rectangle on a frame using the mouse by entering the (x,y) position of the top left corner of the rectangle and a (width, height) tuple. It then draws a green rectangle on the frame input.

difference_of_intensity_of_superpixels.py

difference_of_intensity_superpixels_matrix: Process video frames to detect motion using simple superpixel comparison frame over frame consecutively with matrix operations (sums...) instead of for loops which is faster.

ST_corner_detection_and_LK_optical_flow.py

process_optical_flow: Process Shi-Tomasi corner detection and Lucas-Kanade optical flow on a list of numpy arrays (frames). Periodically resets some points of interest by adding new high-quality ones. Source: An iterative image registration technique with an application to stereo vision by Lucas, B. D., & Kanade, T. (1981).

ST corner detection review:

- Compute the image brightness gradients and collect them inot a matrix
- Compute its eigenvalue. They measure how strong the change is in two perpendicular directions.

LK optical flow principles:

- Take a small window around each point. The goal is to look in the next frame to see where this window has moved.
- We assume the window looks almost the same, just shifted.
- To find the shift, it compares the brightness values in the window from the old frame to candidate windows in the new frame. It chooses the displacement (dx, dy) that best aligns them (often minimizes the difference). It often starts by detecting stronger movements with a downscaled version of the image and then zooms closer.

gaussian_blur.py

gaussian_blur_list_of_arrays: Applies Gaussian blur to each frame of a video using specified kernel size and sigma. The kernel determines the neighborhood size for the weighted average around each pixel. Higher values result in more blurring with farther neighbours but it becomes more computationally expensive. Sigma is the standard deviation of the Gaussian kernel. Higher values result in more blurring with farther neighbours.

Further explaination following: The idea is to blur the image and then subtract the result from the original image. This enhances areas of rapid intensity change, which correspond to edges. Blurring is typically done using a Gaussian blur, which is equivalent to convolving (i.e. applying a weighted sum operation) the image with a Gaussian function. This is a low-pass filter, because the Fourier Transform of a Gaussian is another Gaussian, and the convolution suppresses high-frequency components (fine details or noise) in the image.

In two dimensions, the Gaussian function is symmetric and defined as:

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

where x and y are the horizontal and vertical distances from the center (or the coordinates in the convolution matrix), and σ is the standard deviation that controls the spread (or "width") of the blur.

This function generates a smooth, bell-shaped surface with concentric circular contours centered around the origin. The values of this Gaussian surface form a convolution kernel, which is applied to the image thus getting a final image $L(x, y) = g(x, y) *$

$f(x, y)$ with g being the convolution kernel and f the original image in 2D. The kernel assigns higher weights to pixels near the center and progressively smaller weights to those farther away, with values beyond 3σ typically ignored due to their negligible contribution.

During convolution, each pixel in the image is replaced by a weighted average of its neighborhood, where the weights come from the Gaussian function. The pixel at the center of the kernel gets the highest weight, and surrounding pixels contribute less depending on their distance.

block_matching_and_optical_flow.py

- **block_matchig_list_of_arrays**: Splits frames into blocks, tracks how each block shifts between consecutive frames by searching for the best match (defined by sum of absolute differences) among all blocks of the same size in the next frame within a predefined search area. This is a basic block-matching motion algorithm, a simpler alternative to optical flow.
- **optical_flow_farneback_list_of_arrays**: Applies fast dense optical flow (Farneback) to visualize motion between frames. Farnebäck's idea is to approximate the neighborhood around each pixel with a quadratic polynomial $x^T Ax + b^T x + c$ with A representing curvature b the slope and c the offset of the pixel of x coordinates, then track how that polynomial changes between two frames thanks to a polynomial expansion of $(x - d)^T A(x - d) + b^T (x - d) + c$. Then solving for the displacement d giving a displacement vector.
- **optical_flow_farneback_compression_list_of_arrays**: Faster Farneback optical flow with noise suppression (don't take into account small movements due to drone moving a bit for instance) and frame downscaling.
- **track_red_pixels_with_optical_flow**: ST corner detection and KB optical specifically for a range of red pixels.
- **optical_flow_rectangle**: Track a colored rectangular object in a sequence of frames using LK optical flow and log its position, velocity, and acceleration.

difference_of_gaussians.py

difference_of_gaussians_list_of_arrays: Computes the Difference of Gaussians (DoG) for each frame in a list of video frames. This method consists in subtracting a Gaussian-blurred image from the original image. The blur is applied using a Gaussian kernel, which gives more weight to nearby pixels and less to distant ones. Blurring acts as a **low-pass filter**: it smooths the image by preserving low-frequency details and removing sharp changes. Subtracting the blurred version from the original keeps only the **high-frequency components**, such as edges and small details.

KB_saliency_detector.py

kadir_brady_saliency_list_of_arrays: Fast approximation of Kadir-Brady saliency using entropy. The image must first be converted to **grayscale**. The algorithm then computes the **Shannon entropy** around each pixel, using a defined box size to determine the neighborhood for the calculation.

1. A **histogram** of pixel intensities is built within this box.

2. The histogram is **normalized** to obtain the **probability distribution** of intensities.

3. The **entropy** is then computed as:

$$-\sum_i p_i \log_2 p_i$$

   where p_i is the proportion of pixels with intensity i.

This entropy value reflects how **diverse or uncertain** the region is, that is to say how much **information** it contains.

mser.py

mser_list_of_arrays: Detects MSER (Maximally Stable Extremal Regions) in each frame and overlays them.

Each pixel is assigned to a class based on whether it falls within a certain threshold, effectively dividing the image into two groups.

In some cases, the image can be divided into a grid, and the thresholding can be computed locally in each cell. This allows for adaptive thresholding, which is more effective when lighting conditions vary across the image.

SIFT.py

- **SIFT_list_of_arrays**: Detects and tracks SIFT keypoints across a list of video frames using optical flow.

To detect motion, we need to find correspondences between images, that is, to determine which parts of one image match parts of another. This is necessary because differences between images can result from the movement of the camera, the passage of time, or the movement of objects.

The first step in SIFT is to detect points of interest and compute their dominant gradient direction. Around each keypoint, a histogram of gradient orientations is built, where each direction is weighted by its magnitude. The most prominent peak in this histogram represents the dominant orientation of the local patch. To ensure rotation invariance, SIFT then aligns the descriptor to this dominant direction before comparison, so the keypoint is always described relative to its own orientation. SIFT is also scale-invariant, meaning it works correctly even if the image is zoomed in or out. However, the algorithm requires detecting many features in each image, and it doesn't directly compare images to find changes. Instead, it processes each image independently, focusing on the local features that can later be matched across frames. Source

manual_rectangle_tracker.py

- **manual_rectangle_tracker**: Manually tracks a rectangle around a user-selected point across frames thanks to template matching (block-matching algorithm).

The block-matching algorithm works by dividing each frame of a video into small rectangular regions (blocks), and then, for each block in one frame, **searching for the best match** in the next frame within a predefined search area.

The algorithm compares the block with candidate blocks in the next frame using an **error function**, such as: Sum of Absolute Differences (SAD), Sum of Squared Differences (SSD) or Normalized Cross-Correlation (NCC).

The best match (i.e., the one that minimizes the error) gives the **displacement vector**, which estimates the motion of the block between frames. This motion reflects the movement of the **underlying object or region**, assuming brightness consistency and small temporal changes.

Template matching is sensitive to changes in **scale, rotation, and lighting**.

- **CSRT_rectangle_tracker**: Tracks a rectangle of defined position and size in the image and displays its center and trace on the frame.

The CSRT tracker (*Discriminative Correlation Filter with Channel and Spatial Reliability*) is a visual object tracking algorithm that uses Discriminative Correlation Filters (DCFs), with specific modifications to improve robustness to deformation, rotation, and partial occlusion.

At the core of CSRT is the idea of correlation filtering. A DCF learns a filter f that, when convolved with the image, gives a strong peak where the object is located and low response elsewhere. This is typically done in the Fourier domain to speed up the convolution operation (since convolution becomes multiplication in the frequency domain).

Unlike basic DCFs that use raw grayscale pixels, CSRT computes features from multiple image channels: HoG (Histogram of Oriented Gradients), Color Names (quantized color labels) and Intensity values (grayscale).

Each channel improves robustness to lighting changes and background noise.

A key innovation in CSRT is the introduction of a spatial reliability map. Not all pixels in the template are equally helpful for tracking (e.g., pixels near the object's edge or background can be misleading). CSRT computes a binary or continuous-valued spatial mask that weights the importance of each spatial location during training of the correlation filter.

This means the filter focuses more on reliable, central, object-specific regions, and less on uncertain or background areas, leading to fewer tracking failures when the background changes.

YOLO.py

detect_and_draw: Uses YOLO and a query to draw a rectangle around the object corresponding to the query in the first frame.

Result description (video documentation)

The following summarizes the functions used in each video result and provides a rating of the outcome relative to the expected result. This makes it possible to reproduce the results.

I used a first short video named `video_sample_1-uhd_3840_2160_25fps` to test my first algorithms.

- `1_difference_of_gaussians_1.mp4` is made by separating the video into frames (`mp4_to_list_of_arrays`) and then computing the DoG for each frame with `difference_of_gaussians_list_of_arrays` with default parameters, then `linear_stretch_colors` and then patching the frames back together (`list_of_arrays_to_mp4`). Note: I won't mention the mp4 to frames and the frames to mp4 steps as they are redundant from now on.
- `1_SIFT_0.07_10.mp4` is built with the `SIFT_list_of_arrays` function. The parameters used for `SIFT_list_of_arrays` are `contrastThreshold=0.07` and `nfeatures=10` and all the other parameters default. A variant is `1_SIFT_10_3_0.07.mp4` with `contrastThreshold=0.07`, `nfeatures=10` and others default in order to get a few strong features. `1_SIFT_100_3_0.07.mp4` is the same but with 100 features.

Then I used `video_sample_3-1080p.mp4` and then `video_sample_4-1080p.mp4` because it seemed to have more lighting variation and pose more of a challenge.

- `3_SIFT_0_3_0.07.mp4` is built with `SIFT_list_of_arrays` like before but with not limit on the amount of features.
- `4_background_substraction.mp4` uses `background_subtraction_list_of_arrays` to show which pixels change in consecutive frames. Low values are black → red → orange → yellow → white for high values.
- `4_difference_of_gaussians_linear_stretch.mp4` is a DoG as before followed by a `linear_stretch_colors`.
- `4_difference_of_intensity_of_superpixels_0.01_0.1.mp4` uses `difference_of_intensity_of_superpixels_list_of_arrays` with parameters `0.01` and `0.1`. I then tried many different resolutions of superpixels to find one that is not too computationally intensive but still precise and doesn't catch too much noise such as in `4_difference_of_intensity_of_superpixels_0.005_0.1.mp4` and others but ended up focusing on the one with parameters `0.002` and `0.5`.

- `4_difference_of_intensity_of_superpixels_0.002_0.3_from_difference_of_gaussians_linear_stretch.mp4` is the same as before but with a preprocessing step including DoG. It does not seem to work. Variants include `4_difference_of_intensity_of_superpixels_0.02_0.5_from_difference_of_gaussians_linear_stretch.mp4` but overall keep too much noise.
- `4_difference_of_intensity_of_superpixels_0.005_0.3_from_4_KB_saliency_detector.mp4` applies `difference_of_intensity_of_superpixels_list_of_arrays` over a `kadir_brady_saliency_list_of_arrays` heatmap. It works worse than the following solutions.
- `4_difference_of_intensity_of_superpixels_matrix_0.004_0.9_from_4_keep_only_color_0_255_0_from_difference_of_intensity_of_superpixels_matrix_0.002_0.35.mp4` is where I tried to combine multiple `difference_of_intensity_of_superpixels_matrix` with different parameters but it doesn't seem to bring anything more.
- I also tried image segmentation with k-means with multiple `k` values but it did not work well as you can see in `4_kmeans_2_-1_from_4_difference_of_intensity_of_superpixels_matrix_0.002_0.5.mp4`.
- I then tried to further highlight the detected movement with `4_local_color_propagation_7_7_30_from_4_difference_of_intensity_of_superpixels_matrix_0.002_0.5.mp4` by grouping moving pixels together to fill the gaps between the rear and front of the car. But it doesn't seem to work well.
- `4_motion_tracking_optical_flow_from_cut_local_color_propagation_10_10_70_from_4_difference_of_intensity_of_superpixels_matrix_0.002_0.5.mp4` is mostly ST corner detection and KB optical flow (`process_optical_flow`) applied after `difference_of_intensity_of_superpixels_matrix`.
- `4_mser.mp4` is built with `mser_list_of_arrays` but it doesn't seem to work well.
- `4_optical_flow_farneback_from_difference_of_intensity_of_superpixels_0.002_0.35.mp4` is `optical_flow_farneback_list_of_arrays` after `difference_of_intensity_of_superpixels_matrix`.
- `4_saliency_heatmap.mp4` is built with `saliency_heatmap_list_of_arrays` but I did not make any use of it. I highlighted the most moving parts in `4_saliency_heatmap_most_red_0.00001.mp4`.
- `4_track_red_pixel_with_optical_flow_from_cut_local_color_propagation_10_10_70_from_difference_of_intensity_of_superpixels_matrix_0.002_0.5.mp4` is the best I could get. It first applies

`difference_of_intensity_of_superpixels_matrix` to the original video with parameters `0.002` and `0.5`. Then I manually cut it so that we can see the car as soon as the video starts (see `video_sample_5-1080p.mp4` for such a cut video). Otherwise the optical flow doesn't work. Then I applied `track_red_pixels_with_optical_flow` with default parameters which tracks specifically red pixels with ST corner detection and KB optical flow. This function also saves the positions to a log file (see function docstring)

Then for **manual tracking**:

- I used `select_rectangle` to draw a bright green rectangle around the object on the first frame so that the first frame is replaced with this new one. Then the best I could get is with `optical_flow_rectangle` applied to `CSRT_rectangle_tracker` which can track a rectangle of a defined size and origin (top left corner of the rectangle) as well as its center. Thus for the 2 functions to work together, you need to save the size and top left corner coordinates of the rectangle you want to follow. See the final results here `5_optical_flow_rectangle_from_CSRT_rectangle_tracker.mp4`.
- I tried following the rectangle with a template matching technique as in the `manual_rectangle_tracker` function but the results were not so consistent and depended on the rectangle you chose.

Finally for **YOLO** pre-processing, I used `20250314_163308.mp4`:

In order to detect and track a certain object category, I first tested a pipeline that is similar to the previous manual tracking by using `YOLO_with_CSRT` which uses `detect_and_draw` and then `CSRT_rectangle_tracker`. See result: `YOLO_CSRT.mp4`. The issue is it can lose track of objects.

Thus `detect_objects` solves this issue by running YOLO detection independently on each frame. It is a bit longer, but because of how this project was built by separating a video into frames, the main bottleneck is the RAM on consumer PC.