Design document for Optical Flow

1. Goals and Overview

A proposed application for the Blue-ROV is to maintain off-shor fish farms with limited human assistance. Fish farms are enclosed by a cylindrical net or fence, which repeats a diamond fence pattern all around the enclosure. Methods are being researched for how the Blue-ROV can navigate this fence and track its relative movement in real time. This project focuses on the method of using Sparse Optical Flow to find the change in position of the fence over time.

The goal is to use the optical flow method on the robot's sonar images to find transform data on how much the objects in the image move between a set of sonar image frames.

2. Explanation of Optical Flow

In a nutshell, optical flow method consists to:

- Read a frame
- Find all the key points of that frame based on intensity
- Track those key points along the next frames

Because there is motion between the observer (the sonar) and the scene (the fence), this should draw the trail of motion of the key points. From this trail, the actual position of the robot will be known.

2a - Optical Flow on a pegboard Experiment

The 2019 paper on *Underwater Terrain Reconstruction through Forward-Looking Sonar Imagery* by Jinkun Wang, Tixiao Shan and Brendan Englot [1] demonstrates the initial potential that sparse optical flow can be used on a repeating pattern. The most important takeaways from this paper:

- The ROV moves at a relatively low speed.
- A-KAZE feature detector extracts features from the first frame.
- “New features are introduced if they are not in close proximity to current features” (3473).
- Feature stops being tracked when: “the distance between descriptors computed at previous and current feature locations is larger than a designated threshold” **or** “the minimum eigenvalue c
- riterion isn’t satisfied in the Lukas-Kanade method”. <u>We will use this criterion.</u>
- Tracking history of the features is visualized.
- Tracking is error-prone when the feature is far from the sonar.
- Paper assumes that although there is differentiating intensity between sonar frames, and optical flow works on the assumption that there is no change in intensity between frames, the method holds well under high-frequency returns and a slow speed of the ROV.

For reference, these details are found in section II-C of the paper.

2b - Research on Lukas-Kanade Optical Flow

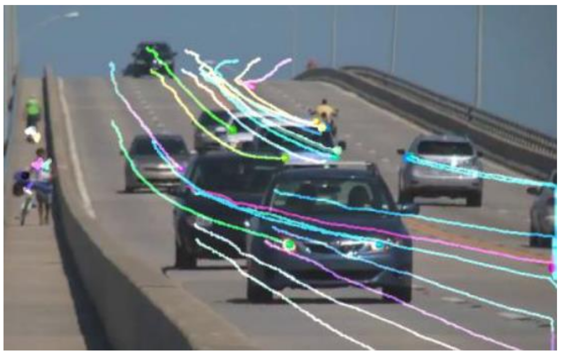
There are two big types of optical flow: Sparse and Dense. This project tried using dense, but there is no trail of motion left behind, so like the paper before us, <u>now only uses Sparse.</u>

Lukas-Kanade Optical Flow is a visual tool for the apparent motion of known features. It is the most used method of Sparse Optical Flow.

There are two assumptions:

1. The pixel intensities do not change between consecutive frames
2. Neighboring pixels have similar motion

Lines are drawn on the apparent motion paths which looks like *Figure 1.*



<u>Figure 1</u>: Lukas-Kanade Optical Flow applied to a video of moving cars [2]

It works by taking a small square of pixels (all the pixels in this window should have the same motion according to the second assumption) and there is a calculation that uses each point in the square as a variable under the same kinematic motion.

In python, the function *CalcOpticalFlowPyrLK*() performs this calculation and returns the new feature locations on the image [2].

2c - Picking Parameters for Lukas-Kanade Optical Flow

What the function *CalcOpticalFlowPyrLK()* does is not interesting to develop here: we can refer to [3] to satisfy any mathematical curiosity.

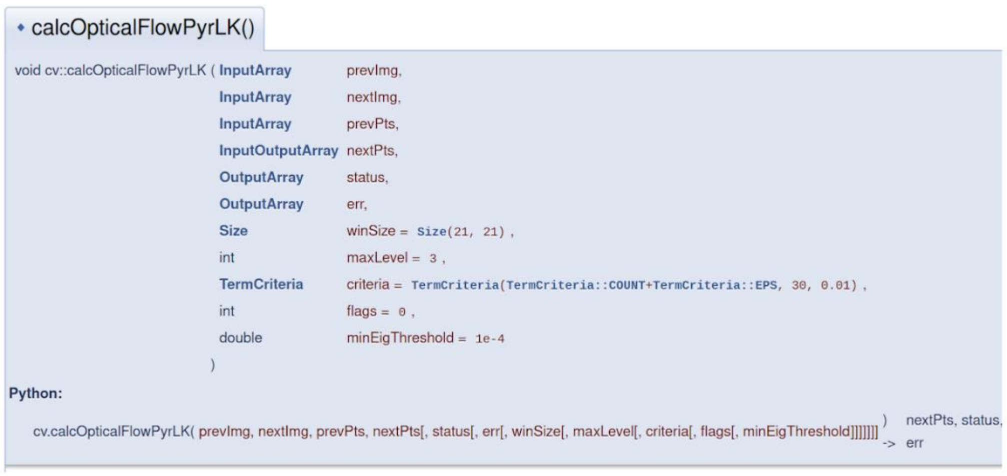
What is interesting to understand is its parameters:

- Flags: can set criteria for an error (if not set, there is a default error measure).

Three options can be considered:

3. ***OPTFLOW_USE_INITIAL_FLOW**:* uses initial estimations, stored in *nextPts*; if the flag is not set, then *prevPts* is copied to *nextPts* and is considered the initial estimate.
4. ***OPTFLOW_LK_GET_MIN_EIGENVALS**:* uses minimum eigenvalues as the error measure (see *minEigThreshold* description)
5. No flag set: *L1* distance between patches around the original and a moved point, divided by the number of pixels in a window, is used as an error measure.

- Min Eig Threshold: boosts performance. Filters out features that don't create eigenvalues above the minimum threshold.
- nextPts (input/output): input only if you already have them and want to know that status and err.
- *status (output):* if flow from the previous features was found, status =1. Otherwise, status = 0.
- maxLevel: maximum pyramid levels (starts at 0).



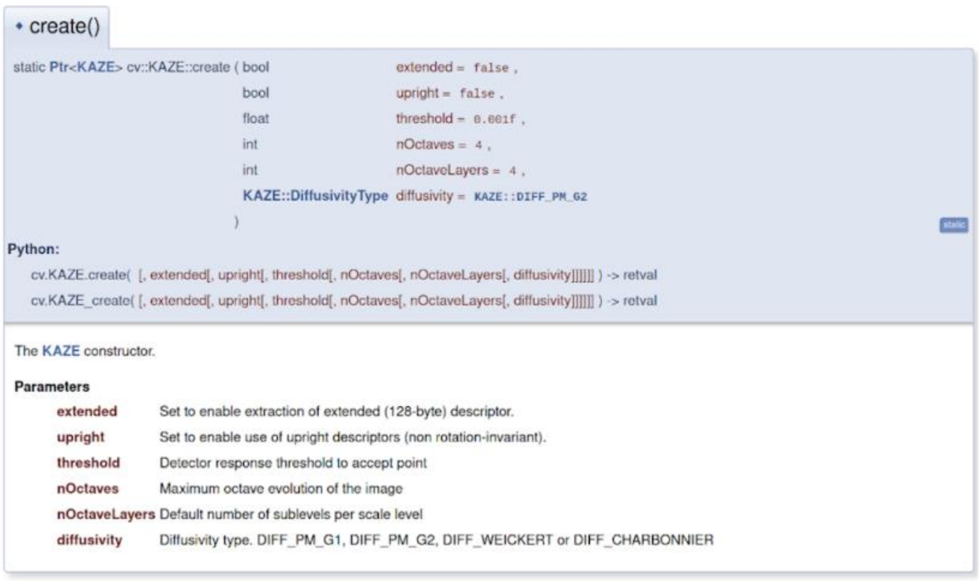
```
◆ calcOpticalFlowPyrLK()

void cv::calcOpticalFlowPyrLK ( InputArray          prevImg,
                                InputArray          nextImg,
                                InputArray          prevPts,
                                InputOutputArray    nextPts,
                                OutputArray         status,
                                OutputArray         err,
                                Size                winSize = Size(21, 21) ,
                                int                 maxLevel = 3 ,
                                TermCriteria        criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01) ,
                                int                 flags = 0 ,
                                double              minEigThreshold = 1e-4
                              )
Python:
   cv.calcOpticalFlowPyrLK( prevImg, nextImg, prevPts, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]] )  -> nextPts, status, err
```

Figure 2: Details about calcOpticalFlowPyrLK() function []

3. Explanation of A-KAZE Feature Tracking

A-KAZE is the chosen method to detect the features.

In the python files, the function *cv2.KAZE_create()* is used []: this function returns a ‘detector’ that will find the key points. Here are the parameters of this function:



• create()

```
static Ptr<KAZE> cv::KAZE::create ( bool                   extended = false ,
                                    bool                   upright = false ,
                                    float                  threshold = 0.001f ,
                                    int                    nOctaves = 4 ,
                                    int                    nOctaveLayers = 4 ,
                                    KAZE::DiffusivityType  diffusivity = KAZE::DIFF_PM_G2
                                  )
```

static

Python:

```
cv.KAZE.create(  [, extended[, upright[, threshold[, nOctaves[, nOctaveLayers[, diffusivity]]]]]] ) -> retval
cv.KAZE_create( [, extended[, upright[, threshold[, nOctaves[, nOctaveLayers[, diffusivity]]]]]] ) -> retval
```

The KAZE constructor.

Parameters

extended	Set to enable extraction of extended (128-byte) descriptor.
upright	Set to enable use of upright descriptors (non rotation-invariant).
threshold	Detector response threshold to accept point
nOctaves	Maximum octave evolution of the image
nOctaveLayers	Default number of sublevels per scale level
diffusivity	Diffusivity type. DIFF_PM_G1, DIFF_PM_G2, DIFF_WEICKERT or DIFF_CHARBONNIER

Figure 3: Details about the KAZE_create() function []

- Image: picture to find points from.
- Mask: for writing the new points onto.
- Keypoints: Optional. Can input a vector of keypoints and just get all their data and descriptors as output.
- Descriptors: descriptors for the input keypoints if they’re used.
- Octaves: number of scale levels in pyramids.
- Octave layers: each layer adds more change in luminance (ie Gaussian blur).
- Diffusivity: certain flow function that controls diffusion process.

4. Script Design

Class OpticalFlowNode():

- **__init__()** : Init all the class variables.
- **init_node()** : Init the node, fetch all paramaters from ROS. All parameters are explained in the yaml file.
  Args: ns (str, optional): The namespace of the node. Defaults to "~"

- **get_image()** : Crop the image in order to just have the fence.
  Args: data from the M1200d
  Returns: the cropped image in color and in grayscale
- **colors_array()**: Create array of random colors for drawing purposes.
- **callback()** : Keypoints are defined every 50 frames. For the 'good' keypoints, optical flow method is applied.

5. Verifying Good features/flows

Chose the features is a crucial step. How to find out if keypoints are good ones? 2 methods have been used in field:

- Eliminate/don't track features that don't have a good match
- Only track features that don't move more than a threshold between frames

The first one was first used with Brute-Force Matcher on the detected keypoints and the optical-flow-found keypoints. However, when it physically worked in the script, it seemed that this was not reliable in the case of the sonar (either because of the intensity scale or because of the moving objects).

The difference between matches could be found, but they all over 0.8 and many in the high 1's or low 2's. Which makes it hard to tell what a proper threshold is in the sonar application.

Option 2 is now applied but it is not yet handled perfectly. The error criterion L1 has still to be tuned (its current value is arbitrary).

6. Results

Here are the results you should have running the current codes.

- For the USMMA data:

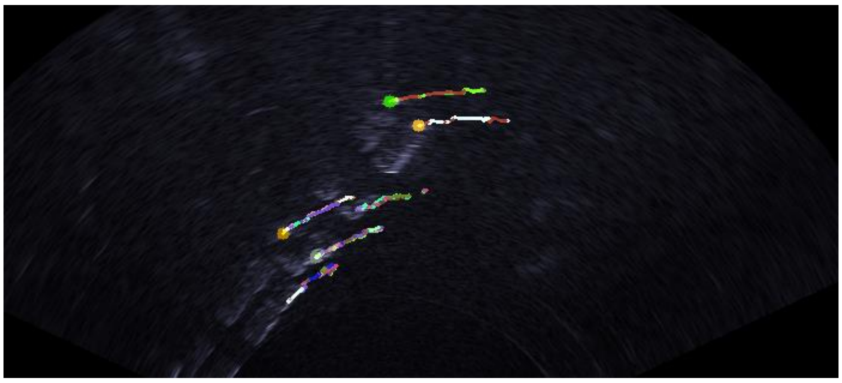


Figure 4: Optical Flow applied on USMMA data

The corners of the rectangles are well detected and tracked according to the movement of the sonar. Each track has the same direction: it is coherent.

Note: each trace is supposed to have a color, but this function does not work yet, that is why they are multicolored.

- For the fence data:

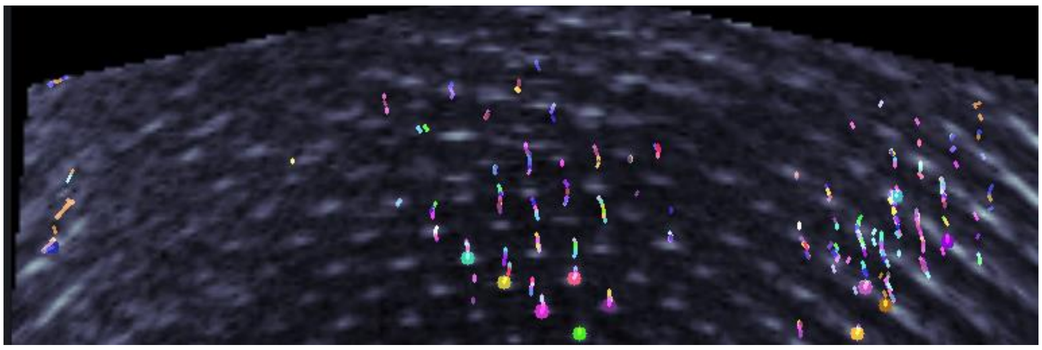


Figure 5: Optical flow applied on the fence data

After cropping the image, the detector finds mostly the central keypoints. Once again, the keypoints follow the same direction.

7. What still needs to be done

✓ Tune the error criterion.

✓ Solve the color problem: assign to each keypoint a color, which it must keep until it is replaced by another keypoint.

✓ There might be a more robust or efficient way to verify which points are good and should be tracked.

✓ A more robust transform estimation. Improvements to the cycling of points (when new ones are taken in, if there should be a maximum number of points being tracked, if points should be dropped before their 30 frames are up or after their 30 frames are up, etc).

8. Sources

[1] J. Wang, T. Shan, and B. Englot, Underwater terrain reconstruction from forward-looking sonar imagery. In 2019 *International Conference on Robotics and Automation (ICRA)* (pp. 3471-3477). IEEE. 2019.

[2] OpenCV, *Optical Flow*. [Online]. Available on: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

[3] OpenCV, *Object Tracking*. [Online]. Available on: https://docs.opencv.org/4.x/dc/d6b/group__video__track.html

[4] OpenCV, *cv::AKAZE Class Reference*. [Online]. Available on: https://docs.opencv.org/3.4/d8/d30/classcv_1_1AKAZE.html