Parameters Data

Optical Flow

◆ 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
```

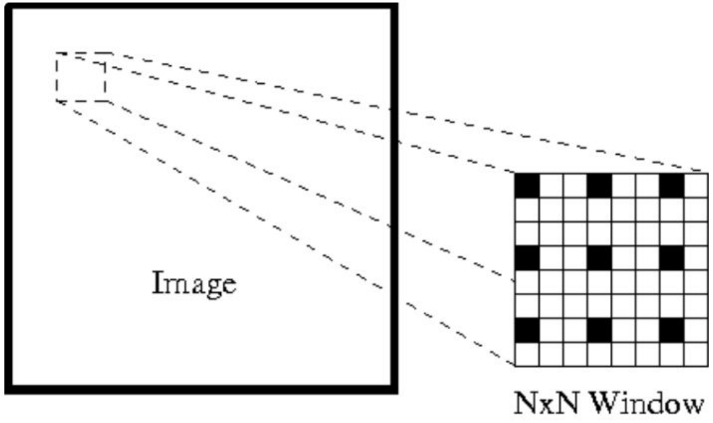
nextPts (input/output) = input only if you already have them and want to know the status and err

Status (output)= If flow from the previous features was found, status=1. Otherwise status=0.

maxlevel= maximum pyramid levels (starts at 0)

Windows in CalcOpticalFlow

First, under these assumptions, we can take a small 3x3 window (neighborhood) around the features detected by Shi-Tomasi and assume that all nine points have the same motion.



Lucas-Kanade: Optical flow is estimated for the black pixels

Window size can be adjusted in parameters

Iterations in CalcOpticalFlow

$$\sum w^2(x, y)\ [\nabla I(x, y, t)\, v + It(x, y, t)]^2$$

Where $w(x, y)$ denotes a window function that gives more influence to constraints at the center of the neighbourhood.

Compute Iterative LK at highest level
For Each Level I [7];

- Take flow $u(i\text{-}1)$, $v(i\text{-}1)$ from level i-1
- Up sample the flow to create $u^*(i)$, $v^*(i)$ matrices of twice resolution for level i.
- Multiply $u^*(i)$, $v^*(i)$ by 2
- Compute It from a block displaced by $u^*(i)$, $v^*(i)$
- Apply LK to get $u'(i)$, $v'(i)$ (the correction in flow)
- Add corrections $u'(i)$, $v'(i)$ to obtain the flow $u(i)$, $v(i)$ at I th level, i.e., $u(i)=u^*(i)+u'(i)$, $v(i)=v^*(i)+v'(i)$

Iterations:

How many times the math of Lucas Kanade is done within each window

More iterations = more robust

Criteria in opencv calc: can have max iteration count or threshold for smallest window movement

(continued)

◆ 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
```

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

Min Eig Threshold: boosts performance. Filters out features that don't create eigenvalues above the minimum threshold

Error and Flags

In calcOpticalFlowPyrlk

3 options for flags, change what err is

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

(Source: details hovering over the function in VS Code

A-KAZE Feature Detection

◆ 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

cv2.KAZE_create()

Octaves: Number of scale levels in pyramid

Octave layers: each layer adds more change in luminance (ie Gaussian blur)

Diffusivity: certain flow function that controls diffusion process

Descriptors

- Extra info about keypoints so computer knows if two are "the same point" or different points
- Example: Letters h and b both have a 3-point intersection. The intensity of the 3x3 window around them describes if they are similar or the same. That's how the computer would know that the keypoints in the intersection look the same, but because of the neighborhoods they are not the same feature.

Feature Matching to compare descriptors and match up the key points has multiple options

Brute Force vs FLANN based

BF vs FLANN Matching

- Brute Force tries every match and sees which is the best one
- FLANN Is fast approximations, based on being the best match nearby, so it computes quicker but will b less accurate.
- Since the features in the sonar images already look very similar, I suspect Brute Force is the better choice to implement for this project.

◆ detectAndCompute()

```
virtual void cv::Feature2D::detectAndCompute ( InputArray                 image,
                                                InputArray                 mask,
                                                std::vector< KeyPoint > &  keypoints,
                                                OutputArray                descriptors,
                                                bool                       useProvidedKeypoints = false
                                              )
```

Python:

```
cv.Feature2D.detectAndCompute( image, mask[, descriptors[, useProvidedKeypoints]] ) -> keypoints, descriptors
```

Detects keypoints and computes the descriptors

Image = picture to find points from

Mask = for writing the new points onto

Keypoints = Optional. Can input vector of keypoints and just get all their data and descriptors as output

Descriptors = Descriptors for the input keypoints if they're used

References

- https://www.cs.huji.ac.il/course/2006/impr/lectures2006/Tirgul9_opticalFlowEnd.pdf
- https://nanonets.com/blog/optical-flow/
- https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html
- https://www.ccoderun.ca/programming/doxygen/opencv_3.2.0/classcv_1_1AKAZE.html
-