Robotic Obstacle Avoidance with Optical Flow

Wang Feixuan

27/08/2016

1. Introduction

Optical Flow is the pattern of apparent motion of objects, surfaces and edges in a visual scene caused by the relative motion between an observer and the scene. With one single camera(and no other sensors) and optical flow method, the mobile robot can detect and avoide obstacles in realtime successfully.

2. Optical Flow

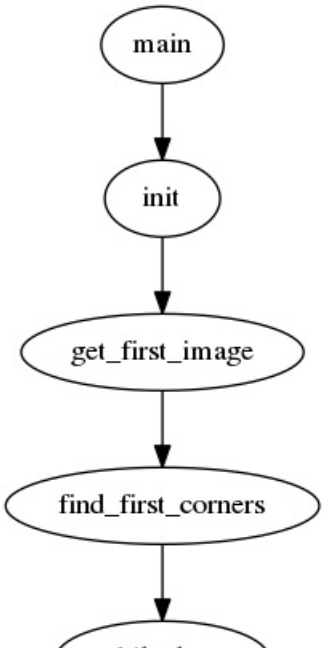
Optical flow uses two consecutive frames and find the same points and thus get their motion vectors. Many algorithms have been come up with mainly in categories: dense and sparse. The former computes all the points of the frame, usually all the pixels, putting them into smaller grids. Classic algorithms on dense matrix are the Horn-Schunck algorithm(rather old, depleted in opencv 3, can be found in opencv 2 docs here), and the Gunnar Farneback's algorithm(implemented in opencv 2 and opencv 3 with samples in opencv 2, more information in opencv 3 docs here), and more. The latter, which is adopted on sparse matrix and requires much less computation, finds corners first and then only finds the location of all the corners(instead of every point) in the next frame. Thus it's important to find the right corners. I use opencv functions instead of building the wheels myself in that frames got by camera with opencv are in the format of opencv `Mat`, and it's always quicker to do computation on this data structure with opencv functions than computing every pixel by oneself.

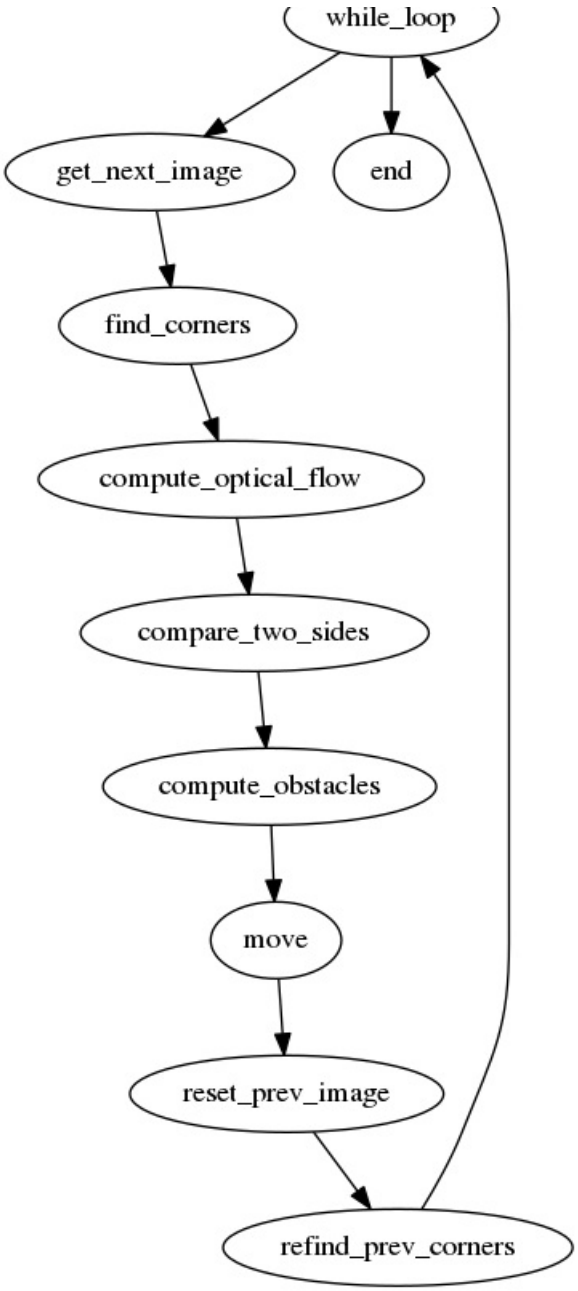
3. Algorithm

Here's how I implemented optical flow for obstacle avoidance on the eyebot.

I use opencv 2.4.x and C++ for the code as many functions are not supported well in C(opencv 3 doesn't even have C API). Modifications for the raspberry pi system are in the Raspberry_Pi3 directory.

- Flow Chart





1. init

initialize camera(video capture)
set camera resolution to QVGA(or QQVGA for faster computation)
print some information on LCD screen
set some parameters for opencv functions

2. get frames and find corners

As optical flow uses relative motion, get one frame from the camera before the while loop. Then in the while loop, for every frame, first convert it to grayscale and then adopt **Shi-Tomasi** algorithm with opencv `goodFeaturesToTrack` function and compare with those from the last frame. Some corners may be lost, especially when the movement is rather fast. Thus corners need refinding every frame. Note that the result of `goodFeaturesToTrack` is in the integer type, and should be made more exact using `cornerSubPix`.

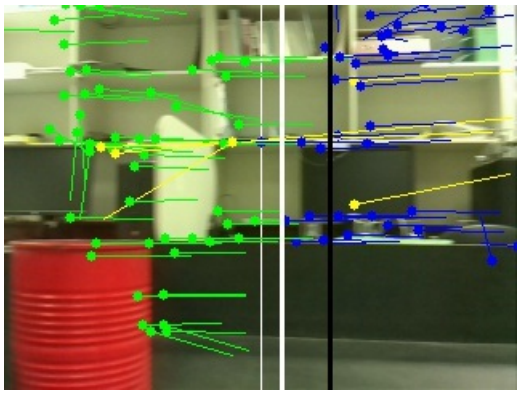
3. compute optical flow

Then get all the optical flow vectors from the two frames and their corners with opencv function `calcOpticalFlowPyrLK`. This method uses the pyramidial implementation of `Lucas-Kanade` algorithm.

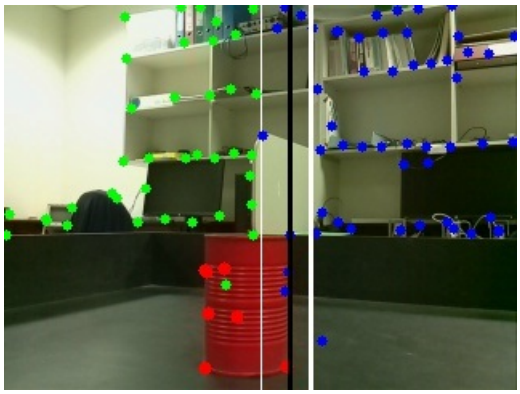
When the eyebot moves too fast, L-K method can be rather inexact. Thus, use the pyramidial implementation of L-K method to get a more exact result.

4. Compute potential obstascles

The screen is cut into two parts: left and right(for UAVs four). Corners and their optical flows are categorized into left and right and their numbers and sums of moduli of all the optical flow vectors are added. Note that optical flows with too-long modulus are ignored are noise.



The green and blue dots are (current) corners and the lines are their optical flow vectors. The yellow lines are optical flows that are too large. In this frame nearly all the optical flows are large, which indicates that the eyebot has moved very fast.



In the picture above, green dots and blue ones are the corners in the left and right side of the frame and the red dots are potential obstacles. Around this frame the eyebot moves rather slowly as no distinct optical flow vector lines can be seen.

Obstacle are computed easily: if the modulus of a corner's optical flow vector is larger than the average modulus of all the optical flow vectors on the other side, then the corner might belong to an obstacle.

To display frames on the LCD screen, first save the image to a temporary file(absolute path needed) with `imwrite` and then use `LCDImage` from eyebot API. Using opencv window or movedWindow can be rather slow.

To avoid more noise lower down the camera. But with a good implementation and better weight parameters the corners in the back should not be so big a problem as the corners there should move much slower than the obstacles in the front.

5. Move with Balance Strategy

With all the information of the motion field, the eyebot then adopts **Balance Strategy** to decide the next orientation. If the balance strategy is smaller than a threshold, then the eyebot goes straight; otherwise, an angle is computed and the orientation is drawn on the screen, and the eyebot takes a turn accordingly.

The basic equation for the turning angle is:

$$\Delta(F_L - F_R) = k\left(\frac{\sum\|w_L\| - \sum\|w_R\|}{\sum\|w_L\| + \sum\|w_R\|}\right)$$

where `w` can be either the sum or the average of moduli of optical flow vectors on each side. In my code I use different weights for both the sum and the average of all the optical flows and the sum of those of potential obstacles for a better result.

The next orientation is drawn on the screen.

6. Move

Use eyebot API V-Omega functions to go straight or move. Note that always wait for a while or use `VWDriveWait()` after making a turn. `VWStop()` and `VWExit()` are specially written for stopping and directly exiting the program.

4. TODO

- `psd.cpp` uses infrared for better detection as optical flow can only detect obstacles in the camera view. Optical flow cannot detect obstacles right in front of it as thus the weighted sum of left and right can be very similar.
- `final.cpp` has an interactive GUI and better format of code, but the dense method is not finished. Also, the driving part of it needs testing as when I tested it all of the pis got stalled with continuous signals, which didn't happen when I tested the pi before with `psd.cpp` (which is strange as the driving part of code didn't change).
- Parameters for calculating the optical flow and finding good corners to track may need further adjusting for better results.
- computation of `FOE` (focus of expansion) on both dense and sparse matrix

5. References

[1] 秦峰, 刘甜甜, 尤海鹏, 麦宇庭, 赵黎明, and 陈言俊. “基于图像识别的水下机器人自主避障系统.” 兵工自动化 11 (2012): 24.
[2] 肖雪, 秦贵和, and 陈筠翰. “基于光流的自主移动机器人避障系统.” 计算机工程 39, no. 10 (2013): 305–308.
[3] 赵海, 陈星池, 王家亮, and 曾若凡. “基于四轴飞行器的单目视觉避障算法,” 2014.
[4] Born, Christof. “Determining the Focus of Expansion by Means of Flowfield Projections.” In In Proc. Deutsche Arbeitsgemeinschaft Fur Mustererkennung DAGM’94, 711–719, 1994.
[5] Negahdaripour, Shahriar, and Berthold KP Horn. “A Direct Method for Locating the Focus of Expansion.” Computer Vision, Graphics, and Image Processing 46, no. 3 (1989): 303–326.
[6] O’Donovan, Peter. “Optical Flow: Techniques and Applications.” The University of Saskatchewan, 2005. [7] Plyer, Aurélien, Guy Le Besnerais, and Frédéric Champagnat. “Massively Parallel Lucas Kanade Optical Flow for Real-Time Video Processing Applications.” Journal of Real-Time Image Processing 11, no. 4 (April 22, 2014): 713–30. doi:10.1007/s11554-014-0423-0.
[8] Prazdny, K. “Determining the Instantaneous Direction of Motion from Optical Flow Generated by a Curvilinearly Moving Observer.” In 1981 Technical Symposium East, 199–206. International Society for Optics and Photonics, 1981. [9] Souhila, Kahlouche, and Achour Karim. “Optical Flow Based Robot Obstacle

Avoidance.” International Journal of Advanced Robotic Systems 4, no. 1 (2007): 13–16.
[10] Tistarelli, M., E. Grosso, and G. Sandini. “Dynamic Stereo in Visual Navigation.” In Computer Vision and Pattern Recognition, 1991. Proceedings CVPR’91., IEEE Computer Society Conference on, 186–193. IEEE, 1991.
[11] 基于光流法的视觉避障系统研究 Visual Obstacle Avoidance System Based on Optical Flow Method.” Accessed July 27, 2016.