

# OPTIC-FLOW BASED ESTIMATION OF ANGLE OF ATTACK

SEMESTER PROJECT

BRICE PLATERRIER

SECTION MICROTECHNIQUE

FEBRUARY 2015 / 2016

ASSISTANTS:

JULIEN LECOEUR

LABORATORY OF INTELLIGENT SYSTEMS (LIS)

PROF. DARIO FLOREANO

EPFL

FACULTE SCIENCES ET TECHNIQUES DE L'INGENIEUR (STI)

INSTITUT D'INGENIERIE DES SYSTEMES (I2S)



## Abstract

The Ywing is a VTOL (vertical take off and landing) drone capable of flying at any speed between its cruise speed and hover by varying its angle of attack. The angle of attack (angle between wing chord and airflow) cannot always be approximated by pitch angle (angle between wing chord and horizon) because the drone is not always flying at constant altitude or undergoes some wind. The goal of this project was to use optic flow measurements from a wide field of view camera embedded on the drone to estimate the angle of attack. Indeed, in translation flight, all optic flow vectors point to the current direction of flight, which can be used to estimate angle of attack. The estimation of angle of attack was implemented on the onboard controller using optic flow vectors computed on the camera, and validated using motion tracking systems.

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Introduction</b>                              | <b>1</b> |
| <b>2</b> | <b>Method for Direction of Motion Estimation</b> | <b>3</b> |
| 2.1      | Omnidirectional Camera Model . . . . .           | 3        |
| 2.2      | Mapping Optical Flow to Unit Sphere . . . . .    | 6        |
| 2.3      | Derotation of Optical Flow . . . . .             | 10       |
| 2.4      | Finding the Focus of Expansion . . . . .         | 11       |
| 2.5      | Computational cost . . . . .                     | 15       |

# 1 Introduction

Monocular vision sensors are probably the most widely used devices in embedded systems, where the payload is a serious parameter to take into consideration, even more so for flying robots. Depending on the application, whether it be obstacle avoidance or egomotion, several such devices may be used on the embedded system.

Previous work from A. Briod [1], show that the egomotion estimates greatly benefit (statistically) from the addition of numerous pinhole cameras on the body of the robot. Nevertheless, this method entails a calibration procedure to estimate the viewing direction of each sensor with respect to the body of the robot, for which an automatic estimation thereof was proposed in [2]. Hence, another solution is to consider using omnidirectional cameras.

An omnidirectional camera provides a wide field of view of at least 180 degrees. In our case of interest, the dioptric camera that was used consists in a combination of shaped lenses (fisheye lenses) and typically can reach a field of view slightly larger than 180 degrees. Thanks to the calibration of a model of the camera, mapping each camera image pixels to the 3D scene, these devices enable the sampling of many points on the camera image without the need to re-estimate the corresponding viewing directions each time they are changed. More importantly, a wide field-of-view is preferred for optic-flow-based egomotion estimation because it is necessary to distinguish clearly the direction of motion out of multiple optic-flow measurements. It thus allows for more robustness and redundancy in a variety of environments and conditions.

The optical flow (or optic-flow) is the apparent motion of objects, surfaces and edges in a visual scene and can be obtained either from the variation of pixel intensity (Lucas-Kanade method) or pattern displacement. This approach has two advantages compared to feature-based techniques. (1) The optic-flow can be estimated from low-resolution sensors, which are typically small and lightweight, allowing for a low computational overhead. (2) The optic-flow measurements can be measured from any scene whether recognizable features are present or not, which enables its use in both indoor and outdoor environments.

But the use of optic-flow measurements has several drawbacks compared to feature tracking when it comes to ego-motion estimation. (1) The optic-flow is caused by the relative motion between the camera and the scene from which some visual cues are extracted. This makes the measurements particularly sensible to outliers induced by large uniform textures (e.g. white walls) or moving objects in the environment. (2) The visual cues which are used for optic-flow computation vary at each timestep causing the distance to the scene to change. Thus, the estimation of the correct scaling of the velocity of the drone may be difficult and typically requires additional sensors such as inertial measurement units [1] or sonar sensors.

Several approaches were proposed for inferring the direction of motion from optic-flow measurements. A first approach is to rely on coarse epipolar constraints to estimate the position of the focus of expansion, which is the single point from which the optic-flow field diverges. In this case, the sampling of a high number of optic-flow measurements

at each timestep and the computation of their corresponding epipolar constraints enable the refinement of the estimated position of the FOE [3]. Another approach would be to proceed to optic-flow and an inertial sensor fusion [1]. Relying on translational optic-flow direction constraint and derotation [4], this method enable the estimation of the direction as well as the scaling of the velocity.

However, the above methods exhibit some disadvantages. (1) The computation of epipolar constraints from raw optic-flow measurements is limited to antipodal points, thus reducing the interest in wide field-of-view cameras. (2) The estimation of the scaling factor using extended kalman filters (EKF) is not necessary in our particular case and requires inertial measurements units which are not present on the camera board. This method would thus entail the transmission of the pre-processed data from the camera chip to the autopilot. Hence, it would be necessary to estimate the additional delay due to preprocessing and transmission. In this case, the computation of optical flow measurements and viewing direction, the projection on the unit sphere, and the derotation, could be performed on the camera chip while the remaining computations (EKF predictions and updates) would be achieved by the autopilot.

In this report, we propose a method for estimating the direction of motion based on high frequency and unscaled optic-flow measurements. We characterize this method by means of a wide field-of-view camera on which a hundred of optic-flow measurements are computed and a motion capture system. Section II describes the theoretical background and the actual algorithm implemented on the camera board. Section III presents the experimental setup used for characterization. Section IV details the results of the experiments conducted in a synthetic environment.

## 2 Method for Direction of Motion Estimation

### 2.1 Omnidirectional Camera Model

All modern fisheye cameras are central, and hence, they satisfy the single effective focal point property. The reason a single effective viewpoint is so desirable is that it allows us to generate geometrically correct perspective images from the pictures captured by the omnidirectional camera. When the geometry of the omnidirectional camera is known, that is, when the camera is calibrated, one can precompute this direction for each pixel. Therefore, each pixel can be mapped onto a plane at any distance from the viewpoint to form a planar perspective image. Additionally, the image can be mapped onto a sphere centered on the single viewpoint, that is, spherical projection.

Omnidirectional camera systems cannot be described using conventional pinhole model because of the very high distortion induced by the imaging device. Indeed, in our case, the model should take into account the multiple refractions caused by the lenses of the fisheye camera. A unified model was proposed in [5]. To overcome the lack of knowledge of a parametric model for fisheye cameras, a Taylor polynomial, whose coefficients and degree are found through the calibration procedure, is used.

Let  $p$  be a pixel point of your image, and  $(u, v)$  its pixel coordinates with respect to the center of the omnidirectional image. Let  $P$  be its corresponding 3D vector emanating from the single effective viewpoint, and  $(x, y, z)$  its coordinates with respect to the axis origin. The function estimated by the calibration process maps an image point  $p$  into its corresponding 3D vector  $P$ :

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \\ f(r) \end{bmatrix} \quad (1)$$

where  $\begin{cases} r = \sqrt{u^2 + v^2} \\ f(r) = a_0 + a_1 r + a_2 r^2 + a_3 r^3 + a_4 r^4 + \dots \end{cases}$

and the parameters to estimate are  $a_0, a_1, a_2, \dots$ . Although increasing the polynomial may yield better accuracy, we used 4<sup>th</sup> order polynomials, as a good trade-off between accuracy and complexity of the polynomial model.

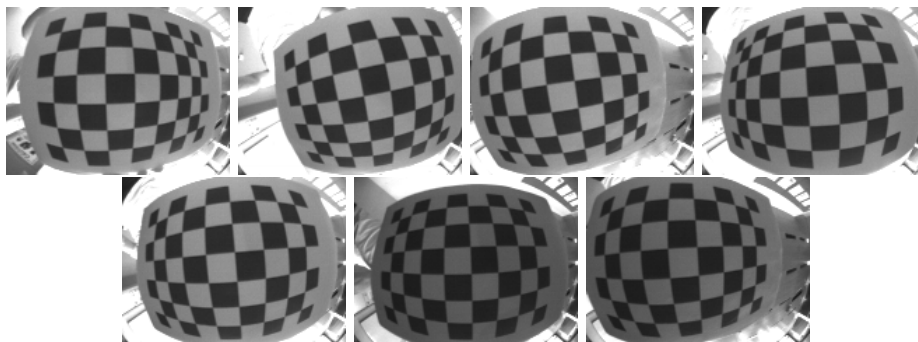
However, as the camera and lenses axes are never perfectly aligned, the model is extended so as to model these errors through an affine transformation:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} c & d \\ e & 1 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix} \quad (2)$$

which relates the real distorted coordinates  $(u', v')$  to the ideal undistorted ones  $(u, v)$ .

This approximate model of the camera allows for more scalability in the number of optical flow measurements and more flexibility on their location, without the need to proceed to another calibration. Furthermore, the viewing directions corresponding to each pixel can be pre-computed as part of an initialization procedure to speed up subsequent computations.

The proposed calibration procedure relies on the use of a chessboard to automatically locate feature points on the camera images. Unfortunately, the resolution of our camera ( $160 \times 120$ ) was too low for the edge detection algorithm to perform correctly its task. This issue may be avoided by using circles instead of the squares of the checkerboard. Indeed, this would yield more easily distinguishable corners. Considering this was not part of the current toolbox, we simply located 35 corners manually for each image, prior to calibration. We used 7 images taken from the camera with varying positions and orientation of the chessboard so as to cover most of its field of view (Fig. 1).



**Figure 1: Training images used for camera model calibration** - The orientation and position of the chessboard were changed from one image to another to most of the field of view. The images are given in grayscale by the camera and exhibit a resolution of  $160 \times 120$  pixels.

In the end, we obtained a subpixel average error of 0.34 pixels and the following model, which relies on a 4<sup>th</sup>-order Taylor polynomial:

$$\text{Polynomial: } \begin{cases} a_0 = -6.66.10^1 \\ a_1 = 0.00 \\ a_2 = 6.42.10^{-3} \\ a_3 = -2.31.10^{-5} \\ a_4 = 2.73.10^{-7} \end{cases} \quad \text{Center: } \begin{cases} x_c = 56.23 \\ y_c = 77.64 \end{cases} \quad \text{Matrix: } \begin{cases} c = 1.00 \\ d = -0.00 \\ e = -0.00 \end{cases}$$

As we can see from the coefficients of the matrix in (2), there is not significant misalignment between the camera and the lenses. Hence, we can neglect this part of the model if necessary. Nevertheless, the estimated location of the center is not exactly the actual center of the  $160 \times 120$  grid of pixels. The reprojections of the points used for calibration appear on Figure 2.

This model was implemented as a structure which is passed as parameters to any function that needs it. The implementation of the projection from camera image to the (3D) scene is shown in Code 1.



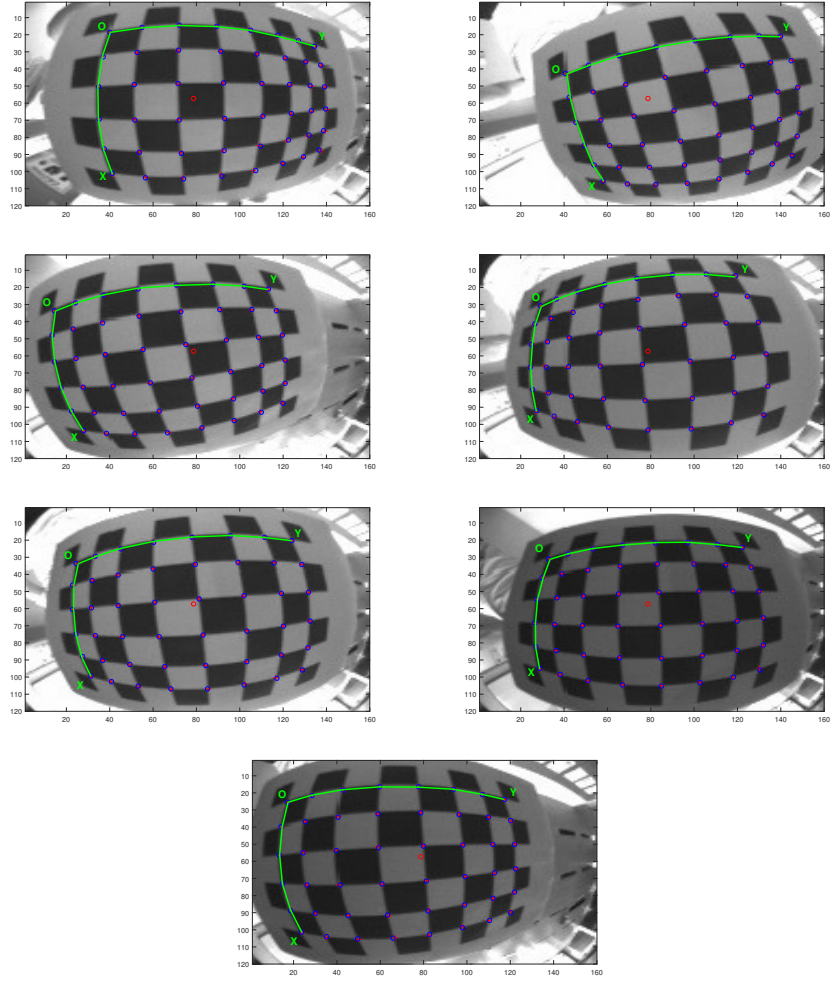


Figure 2: Reprojection on the camera image after calibration

Code 1: From pixels to 3D directions

```
1 void cam2world(float *xp, float *yp, float *zp, float u,
2               float v, cam_model *cam)
3 {
4     float *pol      = cam->pol;
5     float xc        = cam->xc;
6     float yc        = cam->yc;
7     float c          = cam->c;
8     float d          = cam->d;
9     float e          = cam->e;
10    uint8_t length_pol = cam->length_pol;
11
12    float invdet      = 1/(c-d*e);
13
14    // back-projection of u and v
15    *xp = invdet*(    (u - xc) - d*(v - yc) );
16    *yp = invdet*( -e*(u - xc) + c*(v - yc) );
17
18    float r = sqrt(SQR((*xp)) + SQR((*yp)));
19    *zp      = pol[0];
20
21    float r_i = 1;
22
23    // compute z from polynomial model
24    for (uint8_t i = 1; i < length_pol; i++)
25    {
26        r_i *= r;
27        *zp += r_i*pol[i];
28    }
29 }
```

## 2.2 Mapping Optical Flow to Unit Sphere

Now that the model of the omnidirectional camera is known - and the pixels can be mapped onto a plane or a sphere -, we can deduce the viewing direction corresponding to a given pixel. However, the particular geometry of the omnidirectional camera induces high distortions in the image. Hence, the optical flow, which is computed using Lucas-Kanade method, can be projected onto the unit sphere using the method presented in [6]. Mapping the optical flow directly on the unit sphere allows us to simplify further processing required for the estimation of the location of the focus of expansion, through the straightforward use of spherical coordinates. However, though more natural, the

sphere still induces radial distortions and may not be appropriate for all types of motions, as exposed by Shakernia et al.

The optical rays are described in spherical coordinates  $(\rho, \theta, \Phi)$  from the center of the camera, where  $\rho$  is the magnitude,  $\theta$  is the azimuth in the  $X$ - $Y$  plane, and  $\Phi$  is the polar angle between the ray and the  $Z$ -axis. Given an image point  $(u, v)^T$ , we first compute the corresponding ray (or back-projection ray)  $b = (x, y, z)^T$  using equations (1) and (2), and normalize it to unit length  $s = b/\|b\|$ . The spherical coordinates of the "unitized" back-projection ray are given by:

$$\begin{cases} \rho = 1 \\ \theta = \arctan(\frac{y}{x}) \\ \Phi = \arctan(\frac{r}{z}) \end{cases} \quad (3)$$

where  $r = \sqrt{x^2 + y^2}$ . Using equations (1) and (2), we can derive the following Jacobian which relates the partial derivatives from the image plane to the unit sphere:

$$J = \begin{bmatrix} \frac{\partial \theta}{\partial u} & \frac{\partial \theta}{\partial v} \\ \frac{\partial \Phi}{\partial u} & \frac{\partial \Phi}{\partial v} \end{bmatrix} = \frac{1}{\det(A)} \begin{bmatrix} \frac{-y - ex}{r^2} & \frac{dy + cx}{r^2} \\ \frac{d\Phi}{dr} \frac{(x - ey)}{r} & \frac{d\Phi}{dr} \frac{(cy - dc)}{r} \end{bmatrix} \quad (4)$$

where  $A = \begin{bmatrix} c & d \\ e & 1 \end{bmatrix}$  and  $\frac{d\Phi}{dr} = \frac{z - r\rho'(r)}{r^2 + z^2}$

Then, we compute the transformation which takes the partial derivatives on the unit sphere from spherical coordinates to rectangular coordinates:

$$S = \begin{bmatrix} -\sin \theta \sin \Phi & \cos \theta \cos \Phi \\ \cos \theta \sin \Phi & \sin \theta \cos \Phi \\ 0 & -\sin \Phi \end{bmatrix} \quad (5)$$

Therefore, we can compute the mapping of an image point  $(u, v)^T$  (considered distorted) and its corresponding optical flow  $(\dot{u}, \dot{v})^T$  to the unit sphere:

$$s = b/\|b\| \quad \dot{s} = SJ \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \quad (6)$$

A possible C implementation is detailed in Code 2

## Code 2: Backprojection of optical flow

```
1 void flow2world(float *flow_x, float *flow_y, float *flow_z,
2               float xp, float yp, float zp, float flow_u,
3               float flow_v, cam_model *cam)
4 {
5     float *pol      = cam->pol;
6     float c         = cam->c;
7     float d         = cam->d;
8     float e         = cam->e;
9     uint8_t length_pol = cam->length_pol;
10    float invdet     = 1/(c-d*e);
11
12    // from cartesian to spherical coordinates
13    float r         = sqrt(SQR(xp) + SQR(yp));
14    float theta     = atan2(yp, xp);
15    float phi       = atan2(r, zp);
16
17    // compute polynomial model derivative
18    float r_i = 1;
19    float dzp = pol[1];
20    for (uint8_t i=2; i < length_pol; i++)
21    {
22        r_i *= r;
23        dzp += i*pol[i]*r_i;
24    }
25    // project optic-flow on unit sphere
26    float d_theta = invdet*(-yp/SQR(r) - e*xp/SQR(r));
27    float d_thetav = invdet*(d*yp/SQR(r) + c*xp/SQR(r));
28    float d_phir = (zp - r*dzp)/(SQR(zp) + SQR(r));
29    float d_phiu = d_phir*invdet*(xp/r - e*yp/r);
30    float d_phiv = d_phir*invdet*(-d*xp/r + c*yp/r);
31    float flow_theta = d_theta*flow_u
32                    + d_thetav*flow_v;
33    float flow_phi = d_phiu*flow_u + d_phiv*flow_v;
34
35    // from spherical to cartesian coordinates
36    *flow_x = -sin(theta)*qsin(phi)*flow_theta
37             + cos(theta)*cos(phi)*flow_phi;
38    *flow_y = cos(theta)*sin(phi)*flow_theta
39             + sin(theta)*cos(phi)*flow_phi;
40    *flow_z = -sin(phi)*flow_phi;
41 }
```

This method is obviously quite computationally expensive. Indeed, further testing on the camera board revealed that the computation of 117 optic-flow vectors accounts for approximately 6ms. This value is to be compared to the refresh rate of our camera which is 200Hz. Thus, the runtime (for one iteration) is more than doubled by the use of the above method.

In an attempt to reduce the computational overhead, we considered an approximate projection of the optic-flow measurement on the unit sphere. Considering the high refresh rate of the camera, any motion will induce a low displacement of the pixels on the camera image (i.e. a small optic-flow measurement). The latter remark allows us to take advantage of the small-angle approximation. More formally, we approximate the actual optic-flow projection  $F$  to the difference between a sampling direction  $P_1$  and a virtual point  $P_2$  that is obtained from the projection of the point  $p_2 = p_1 + f$ , where  $p_1$  is the point on the camera image corresponding to the 3D vector  $P_1$  and  $f$  is the 2D optic-flow:

$$\begin{aligned} P_2 &= g(p_2) = g(p_1 + f) \\ F &\approx P_2 - P_1 \end{aligned} \tag{7}$$

where  $g$  is the spherical projection function described in the previous section and  $F$  is the spherical projection of the optic-flow.

The resulting optic-flow vector is obviously not perpendicular to the sphere. However, building on the same assumption, we can neglect the radial component of the flow and simply use this raw vector for subsequent computations. The C implementation is shown in Code 3.

As for computational gain, running this version of the flow projection dramatically reduces the runtime with less than 2ms to process the 117 optic-flow measurements.

### Code 3: Fast backprojection of optical flow

```
1 void fast_flow2world(float *flow_x, float *flow_y,
2     float *flow_z, float xp1, float yp1,
3     float zp1, float u1, float v1,
4     float flow_u, float flow_v, cam_model *cam)
5 {
6     // add optic-flow to pixel coordinates (u1, v1)
7     float u2 = u1 + flow_u;
8     float v2 = v1 + flow_v;
9
10    // define new coordinates
11    float xp2;
12    float yp2;
13    float zp2;
14
15    // map to 3d scene
16    cam2world(&xp2, &yp2, &zp2, u2, v2, cam);
17
18    // project on unit sphere
19    normalize(&xp2, &yp2, &zp2);
20
21    // compute optic-flow considering small angles
22    *flow_x = xp2 - xp1;
23    *flow_y = yp2 - yp1;
24    *flow_z = zp2 - zp1;
25 }
```

## 2.3 Derotation of Optical Flow

Under the assumption of differential motion, that is small translation and rotation of the camera between each frame, we can approximate the geometrical effects of egomotion on the perceived image motion to a first-order Taylor polynomial. Hence, as the scene is projected on the unit sphere centered on the vantage point, each optical flow measurement can be expressed as a 3D vector tangent to the unit sphere and perpendicular to the viewing direction:

$$f = -w \times d - \frac{v - (v \cdot d)d}{D} \quad (8)$$

where  $d$  is a unit vector describing the viewing direction,  $w$  the angular speed vector,  $v$  the translational velocity vector and  $D$  the distance to the viewed object. The measured optical flow  $f$  can be decoupled and expressed in two parts, namely the rotation-induced and the translation-induced optical flows, respectively  $f_r$  and  $f_t$ .

As we are only interested in the estimation of the angle of attack (i.e. the direction of translation), only the translational part  $f_t$  of the optical flow is required. Hence, we proceed to the removal of the rotational part  $f_r$  of the optical flow through the process of derotation [4]. If necessary, the main axis  $Z$  of the camera can be automatically calibrated using the method proposed in [2], so as to estimate the transformation from the gyroscope frame to the camera frame. Even after calibration, the derotation procedure may introduce additional noise to the optical flow measurement, all the more so that the rotational component is larger than the translational part of the optical flow. This may be exacerbated by the sensor bias compensation and noise reduction only achieved using averaging and low-pass filtering.

However, before proceeding to the derotation of the optical flow, we must first consider scaling the angular rate values. Indeed, once projected on the unit sphere, the optical flow is given in  $[\text{frame}^{-1}]$  whereas the gyroscopes measurements are  $[\text{rad.s}^{-1}]$ . Given the refresh rate of the camera, the conversion is straight forward:

$$w[\text{frame}^{-1}] = 0.005 \cdot w[\text{rad/s}^{-1}] \quad (9)$$

## 2.4 Finding the Focus of Expansion

In order to find the direction of translation, we use the derotated optical flow to constrain the translation direction to lie on a great circle. This great circle is the intersection of a plane - defined by the vector defined by the viewing direction  $d$ , its corresponding optical flow vector  $p_t$  and the center of the sphere  $O$  - with the unit sphere. Hence, each optic-flow vector gives us a constrain on the position of the focus of expansion (FOE), which can be located anywhere on the great circle. Although only two great circles are required to constrain the location of the FOE to two points (and thus to find the axis of motion), several measurements are used so as to recover from possible outliers and noise.

For a robust estimation of the direction of translation, two different approaches are possible. The first one relies on RANSAC (RANDOM SAMple Consensus) and its variants. However, this first method implies several estimations of the location of the FOE based on randomly sampled measurements and may not be suitable for implementation on a microcontroller and fast refresh rate. Another approach is to find the best intersection of all the great circle arising from the optical flow measurements such as the Hough-reminiscent voting method proposed in [3]. Voting has the advantage of performing in constant time under increasing outlier proportions without any loss of accuracy. Its higher robustness and the fact that it does not require an optimization process make this method the most suitable for embedded robotics.

In order to simplify the mathematical representation of the great circles, we rely on their normal vectors. Indeed, since a great circle is the intersection of the sphere with a plane passing through its origin, it can be defined using one of the unit normal vectors. Formally, for a unit vector  $n = (n_x, n_y, n_z)^T$ , a great circle on the unit sphere  $S^2$  is defined as

$$C = S^2 \bigcup \{x | x^T n = 0, x \in \mathbb{R}^3\} \quad (10)$$

This normal vector  $n$  is readily computed using the cross-product,

$$n = \frac{d \times f_t}{\|d \times f_t\|} \quad (11)$$

Thus, we can vote along a great circle, given its normal vector, by computing simple dot products (i.e. 3 multiplications and 2 additions).

In practice, the intersection of two great circles define 2 possible directions of translation. To disambiguate between the two, one may pick one of the two points and calculate the angle between it and the vector  $f_t$ . If the angle is larger than  $\pi/2$  radians, then that point is the focus of expansion. Another possibility is to use an additional sensor (such as a sonar) to choose the correct direction. In order to improve the processing speed on the camera chip (and to avoid ambiguity), we only consider one hemisphere, that is half the number of possible directions of translation. In other words, we only determine the axis of translation rather than its direction and we work on the hemisphere defined by  $z < 0$ .

The voting procedure requires the storage of evenly distributed vectors on the unit sphere. The minimal angle between these vectors define the resolution of the voting procedure. For tractability reasons, we cannot afford to vote on the complete unit sphere in one batch. Hence, we are considering an iterative method which can be divided in two phases. The first one is a coarse voting procedure where we compute a rough estimate of the direction of translation using a few directions sampled from one hemisphere. This estimate is then used in the next phase where we perform voting on a refined spherical grid of higher resolution but reduced spread. The complete voting procedure is detailed in Algorithm 1.

#### Algorithm 1: Hough-reminiscent voting

```

1 Select optic flow measurement  $p_t$  and viewing direction  $d$ 
2 Compute normal vector  $n = \frac{d \times p_t}{\|d \times p_t\|}$ 
3 For  $i = 1$  to  $N_{coarse}$  do
4   If  $x_i^T n = 0$  then
5     Vote for direction  $x_i$ 
6   end if
7 end for
8 Find the bins with maximum votes. This gives the coarse estimate.
9 Rotate the refined bins to center them on coarse estimate
10 For  $j = 1$  to  $N_{refined}$  do
11   If  $x_j^T n = 0$  then
12     Vote for direction  $x_j$ 
13   end if
14 end for
15 Find the bins with maximum votes. This gives the fine estimate

```



The refined voting requires rotating the refined bins, which we computed for the a predefined direction (in our case  $z = (0, 0, -1)$ ). There are three ways to achieve this transformation, namely the rotation matrix, axis-angle and quaternion methods [7]. From a computational point of view, the rotation matrix method is the cheapest one (6 additions and 9 multiplications) whereas the quaternion method is the most expensive (18 additions and 21 multiplications). However, it is easier for us to use the axis-angle representation and then convert it to a rotation matrix so as to perform the transformation on every bins. The latter conversion accounts for 13 additions and 15 multiplications but the subsequent processing will only require 6 additions and 9 multiplications, provided that we can store the entire rotation matrix. A C implementation is given in Code 4.

Obviously, this whole method is sensible to noise which could affect the voting procedure if the accumulator has a finite resolution. In our case, the coarse resolution was set to  $31^\circ$  and the fine resolution was chosen as approximately  $4^\circ$ . The latter value is justified considering the resolution of the camera (which is around 1.5/pixel) and the additional noise induced by derotation. The former value was chosen so as to limit the number of coarse bins. The voting bins were evenly distributed on the unit and generated using the icosahedron method. Table 1 summarizes the voting bins parameters.

| Voting bins parameter |                 |                |                         |
|-----------------------|-----------------|----------------|-------------------------|
| Voting set            | Region          | Number of bins | Resolution ( $^\circ$ ) |
| Coarse                | Hemisphere      | 25             | 31                      |
| Refined               | Coarse estimate | 42             | 4                       |

**Table 1: Voting bins parameters**-The voting bins are evenly distributed on the unit sphere and were generated using icosahedron method.

#### Code 4: Hough-reminiscent voting

```
1 void rotate_bins(voting_bins *bins, float best_x,
2                 float best_y, float best_z,
3                 float *r_dir_x, float *r_dir_y,
4                 float *r_dir_z)
5 {
6     if(best_x==0.0f, best_y==0.0f, best_z==-1.0f){
7         for(uint8_t i=0; i<bins->size; i++){
8             bins->x[i] = r_dir_x[i];
9             bins->y[i] = r_dir_y[i];
10            bins->z[i] = r_dir_z[i];
11        }
12    }
13    else{
14        // store cosine and sine
15        float c = -best_z;
16        float s = maths_fast_sqrt(1 - SQR(c));
17
18        // store axis
19        float u_x = 1.0f*best_y;
20        float u_y = -1.0f * best_x;
21        float u_z = 0.0f;
22        normalize(&u_x, &u_y, &u_z);
23
24        // compute rotation matrix
25        float R[9];
26        aa2mat(R, u_x, u_y, u_z, c, s);
27
28        // compute rotated refined bins
29        for(uint8_t i=0; i<bins->size; i++){
30            bins->x[i] = r_dir_x[i]*R[0]+r_dir_y[i]*R[1]
31                      +r_dir_z[i]*R[2];
32            bins->y[i] = r_dir_x[i]*R[3]+r_dir_y[i]*R[4]
33                      +r_dir_z[i]*R[5];
34            bins->z[i] = r_dir_x[i]*R[6]+r_dir_y[i]*R[7]
35                      +r_dir_z[i]*R[8];
36        }
37    }
38 }
```

## 2.5 Computational cost

## References

- [1] A. Briod, J.-C. Zufferey, and D. Floreano, “A method for ego-motion estimation in micro-hovering platforms flying in very cluttered environments,” *Autonomous Robots*, 2015.
- [2] A. Briod, J.-C. Zufferey, and D. Floreano, “Automatically calibrating the viewing direction of optic-flow sensors,” in *Robotics and Automation (ICRA)*, pp. 3956–3961, 2012.
- [3] J. J. K. Lim, *Egomotion Estimation with Large Field-of-View Vision*. PhD thesis, 2010.
- [4] M. V. Srinivasan, S. Thurrowgood, and D. Soccol, “From Visual Guidance in Flying Insects to Autonomous Aerial Vehicles,” in *Flying Insects and Robots*, pp. 15–28, 2009.
- [5] M. A. Scaramuzza, D. and R. Siegwart, “A flexible technique for accurate omnidirectional camera calibration and structure from motion,” in *Proceedings of IEEE International Conference of Vision Systems*, 2006.
- [6] O. Shakernia, R. Vidal, and S. Sastry, “Omnidirectional egomotion estimation from back-projection flow,” in *Computer Vision and Pattern Recognition Workshop*, pp. 82–82, IEEE, 2003.
- [7] D. Eberly, “Rotation representations and performance issues,” 2002.