Optic flow based mobile robot control

Raphaël Cherney Frédéric Wilhelm

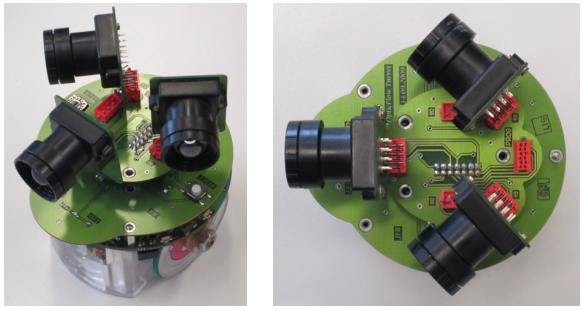
1 Introduction

Like humans, insects use vision to navigate in their environment. However, while humans use stereovision (sense depth through matching of images from left and right eyes), insects use optic flow – the relative movement of the environment expressed in the reference plane of the vision system. Intuitively we know that, when we are moving, objects which appear to move quickly through our visual field are closer to us, while objects that move more slowly are farther away. Using this same idea, insects are able to quickly and robustly traverse cluttered environments [3]. This paper presents an optic flow based mobile robot controller. Using only optic flow measurements, a robot is made to navigate through a complex corridor.

2 Optic flow implementation

2.1 The e-puck platform

The e-puck robot is a small, differential wheeled mobile robot developed at the Swiss Federal Institute of Technology in Lausanne (EPFL). During our tests, the robot was fully autonomous, but it would send and receive data to an off-board computer using a Bluetooth connection through a graphical user interface. The robot has a number of on-board sensors, but for this project we used a special board with three TSL3301 cameras (configured to give a linear 102 pixel image) to control the stepper motor wheels. Figure 1 shows the robot with the 3 cameras: one facing forward, and two at 45 degree angles to each side. The field of view of each camera is 60°, giving a global 150° field of view. We could also have placed the two lateral cameras perpendicularly to the front camera. Such a configuration would increase the global field of view (and avoid overlapping), but would introduce blind spots between the cameras. For the corridor navigation implementation explained below, we preferred to have the lateral cameras pointing slightly to the front, biasing our sensory information for forward motion. In this way, the robot experiences more optic flow from objects it is driving toward.



(a) (b)

Figure 1: The e-puck platform with cameras used for optic flow measurement

2.2 Optic flow calculation

We wanted the optic flow to be calculated in real time using the onboard PIC microcontroller. To do this, we used the Image Interpolation Algorithm (I2A) which is computationally efficient, robust, and gives a linear estimation of speed [1, 2]. The calculation of the optic flow uses two consecutive linear images (taken at Δt apart) to estimate motion. The intensity of pixel n at time t is given by $I(n,t)$. The motion s is estimated by comparing the value from second image $I(n,t+\Delta t)$ to an approximated intensity $\hat{I}(n,t+\Delta t)$, which is simply linear combination of the reference image $I(n,t)$ and space-shifted versions of this image as defined by

$$\hat{I}(n,t+\Delta t) = I(n,t) + s\frac{I(n-1,t) - I(n+1,t)}{2} \tag{1}$$

We can find the motion s by minimizing the mean squared error between the estimated image $\hat{I}(n,t+\Delta t)$ and actual image $I(n,t+\Delta t)$. In our implementation, we compute the mean square error over all of the pixels for each camera. This, in turn, gives us an estimate of the optic flow

$$s = 2\frac{\Sigma_n \left[I(n,t+\Delta t) - I(n,t)\right]\left[I(n-1,t) - I(n+1,t)\right]}{\Sigma_n \left[I(n-1,t) - I(n+1,t)\right]^2} \tag{2}$$

It is evident that the formula (1) is valid only if the image movement is less than one pixel. This imposes a limitation on the time step Δt between the capture of two images. This value is computed for the "worst case" scenario (i.e. when the image movement is maximum) so that all optic flow measurements will

stay within the range -1 to 1. To calculate this value we only consider the rotational optic flow, which is generally bigger than the translational optic flow and which does not depend on the distance of objects. Given that the maximum wheel speed of the robot is 1.0 turns per second, and given the dimension of the robot (wheel diameter of 41 mm and distance of 53 mm between the wheels), we can deduce the maximal rotational speed of the robot:

$$\frac{41 \text{ mm/turn} \cdot 1.0 \text{ turn/s} \cdot 2}{53 \text{ mm}} = 1.55 \text{ rad/s} \quad (3)$$

Given that each camera has a field of view of 60° and 102 pixels, each pixel accounts for $1.027 \cdot 10^{-2}$ rad. Finally, the maximum time step is the time during which the robot turns this angle with the maximum rotation speed. This gives us $\Delta t = 6.6$ ms.

In practice: With the given software used for calibration, the robot is not capable of reaching the maximum rotational speed. Instead, the maximum velocity of each wheel is 60 mm/s, which corresponds to a maximum rotational speed of $\frac{60}{26.5} = 2.27$ rad/s. The resulting time step is then 4.5 ms (it increases when the maximum velocity decreases).

2.3 Calibration

In order to test our optic flow implementation, we measured the optic flow for different rotating speeds. The results for a Δt of 4.5 ms are shown in Figure 2. As expected, we find a linear response for optic flow vs. speed. As the optic flow is measured in pixels, we obtain an measurement of around 1 when the rotational speed is at it's maximum.

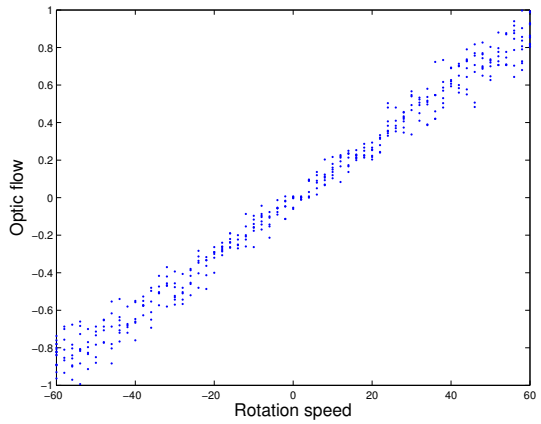


Figure 2: Optic flow vs. rotation speed

2.4 Low-pass filter

The optic flow measurements tend to be quite noisy, as seen in Figure 2. In order to stabilize the readings (and controller), we implemented a simple low-pass filter which replaces the measurement with a weighted sum of the new measurement and the previous value:

$$y(t) = ax(t) + (1-a)y(t-\Delta t), 0 < a < 1 \quad (4)$$

where x the measured optic flow and y the smoothed value. The experimentally determined value of $a = 0.8$ gave us good results.

3 Controller

Our goal for this project was to navigate through a corridor using optic flow measurements to adjust the robot's speed and avoid collisions. We tested a number of different controllers but obtained the best results from a very simple Braitenberg-style controller.

3.1 Direction control

In order to avoid colliding with walls or other obstacles, the robot needs to react to differences in the optic flow measured on each side of the robot. In particular, since objects which are nearer give rise to a larger optic flow, we want to turn away from higher optic flow readings. In order to do this, we simply calculated the difference between the optic flow measurements from the left-facing camera and the right-facing camera. This difference tells us which side (if any) has a greater optic flow. We then adjust the two wheel speeds to turn the robot away from the higher optic flow (add the difference to one wheel and subtract from the other). This effectively causes the robot to move away from the walls and tend toward the middle of corridors.

3.2 Speed control

We improved our controller by adjusting the speed for total amount of optic flow measured by the robot. This caused the robot to slow down as it approached obstacles and travel more "cautiously" through tight spaces. This mimics the behavior observed in actual insects where flying honeybees decrease their speed as a tunnel narrows [3].

3.3 Braitenberg controller

We then decided to change our algorithm slightly and create a simple Braitenberg-style controller (see [4]).

Both wheels start with a forward bias, and each wheel speed is adjusted by subtracting the weighted optic flow value for the camera on the opposite side. In this way, high optic flow on the right side of the robot will cause the left wheel to slow down or reverse, resulting in the robot turning to the left. Similarly, high optic flow on the left side cause the robot to turn right. If there is high optic flow on both sides, the robot will tend to slow down, much like the explicit speed control described in 3.2.

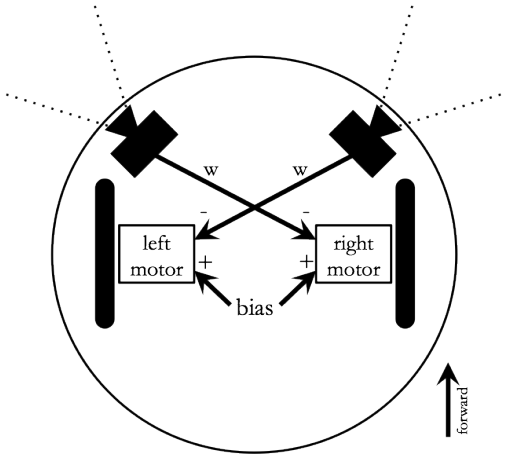


Figure 3: Braitenberg style controller

This simple controller is extremely effective at navigating through the corridor. By adjusting the bias and optic flow weights, we can also make the robot move faster or slower through the course. A common problem with Braitenberg controllers is that they can cause the system to get stuck in corners or dead-ends. This happens when the forward bias and weighted inputs cancel each other out. Fortunately, in the corridor-following problem that we have, this is not really a problem – making a Braitenberg controller especially well suited in this situation.

3.4 Other experiments

We tested variations of each controller and performed additional experiments attempting to incorporate information from the forward-facing camera. We tried splitting the camera into two optic flow regions (one for each side) and using this in our controller. In addition, we attempted to quantify the level of motion and adjust the speed accordingly. Unfortunately, these more advanced controllers resulted in a lower performance. The best results were seen using a simple Braitenberg controller with an additional speed reduction based on the total optic flow.

4 Results

In order to test our optic flow based corridor-following controller, we built a small cardboard coarse for the robot to traverse. The walls were striped to ease the optic flow measurements. We tested the robot at different speeds, and found that (with the proper weights) the robot could reliably avoid the walls and complete the course. Figure 4 shows the robot's path through the corridor (based on frames in Appendix A).

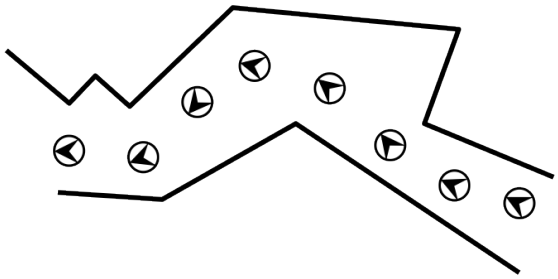


Figure 4: Robot path through corridor using optic flow for navigation

5 Conclusion

Optic flow is an effective, bio-inspired method to navigate in unknown environments. This paper presents a simple optic flow based controller to drive an e-puck mobile robot through a non-trivial corridor. While the controller is reactive and does not plan ahead, it is able to avoid obstacles and traverse the course. The same methods can be used to control more complicated maneuvers, such as landing of unmanned arial vehicles (UAVs) [5] and the techniques can be extrapolated to a wide range of new problems.

References

[1] Srinivasan, M.V. (1994) *An image-interpolation technique for the computation of optic flow and egomotion.* Biological Cybernetics 71, pp. 401-416.

[2] , M. V. (1994). *Generalised gradients versus image interpolation: A critical evaluation of two schemes for measurement of image motion.* Australian Journal of Intelligent Information Processing Systems 1:41–50.

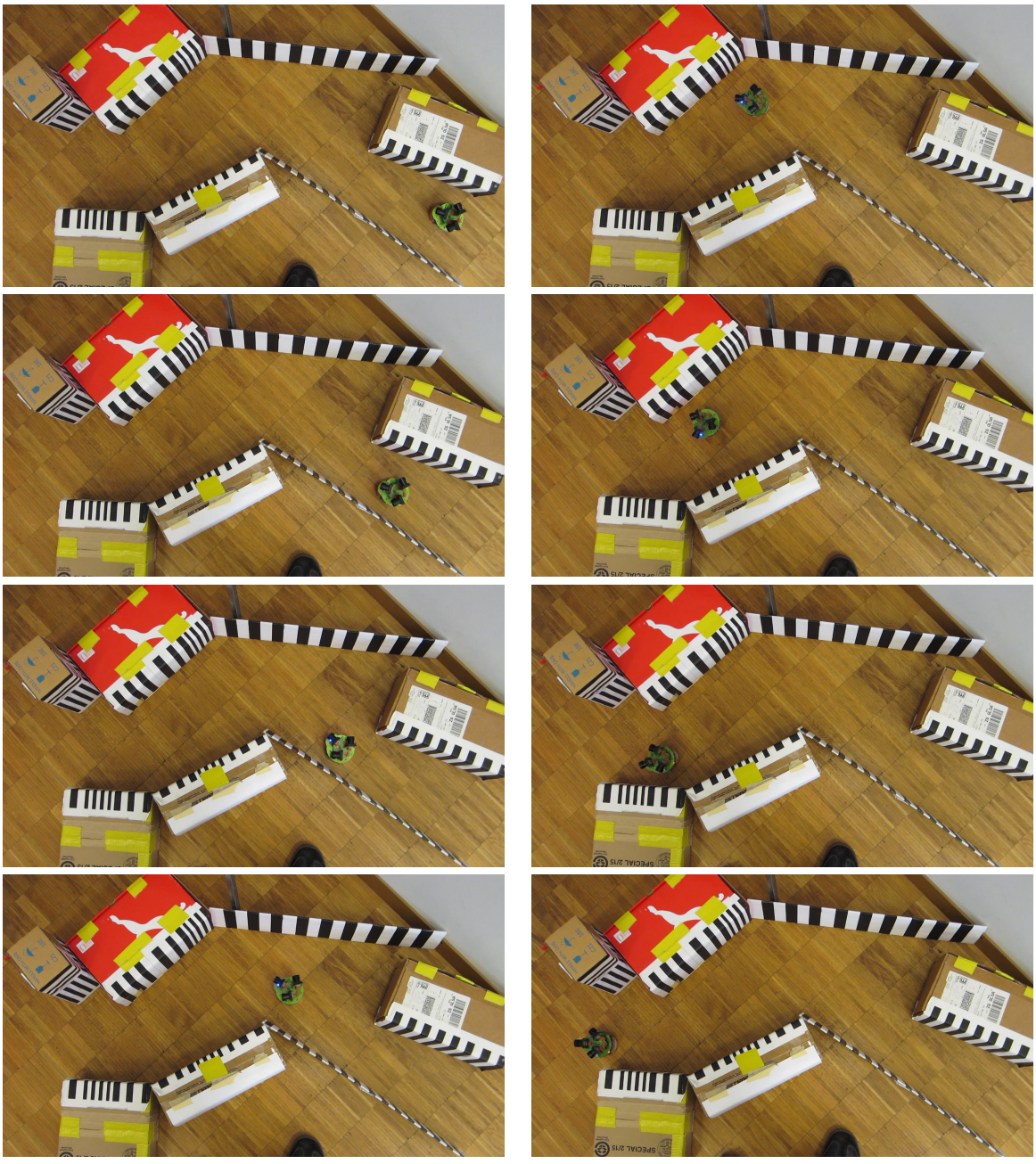
[3] Srinivasan, M.V., Zhang, S.W., Lehrer, M., Collett, T.S. (1996). *Honeybee Navigation en Route to the Goal: Visual Flight Control and Odometry.*

The Journal of Experimental Biology, 199: 237-244.

[4] Braitenberg, V. (1984). *Vehicles: Experiments in synthetic psychology*. Cambridge, MA: MIT Press.

[5] J. S. Chahl, M. V. Srinivasan and S. W. Zhang (2004). *Landing Strategies in Honeybees and Applications to Uninhabited Airborne Vehicles*. The International Journal of Robotics Research 2004 23: 101.

A Results



Corridor navigation using optic flow (2 seconds between frames)

B Code

```
#define REGIONS 3

// Calculate optic flow for one region
signed char optic_flow( unsigned char old_tsl_pixels[TSL_NUM_PIXELS],
                        unsigned char tsl_pixels[TSL_NUM_PIXELS])
{
        unsigned int i;
        signed long top_sum = 0, bottom_sum = 0, tmp;
        signed char s;

        for (i = 1 ; i < TSL_NUM_PIXELS-1 ; i++)
        {
                tmp = ((signed long) old_tsl_pixels[i-1] - (signed long) old_tsl_pixels[i+1]);
                top_sum += ((signed long) tsl_pixels[i] - (signed long) old_tsl_pixels[i]) * tmp;
                bottom_sum += tmp * tmp;
        }

        // Calculate s
        if (bottom_sum != 0)
        {
                tmp = 2 * top_sum / bottom_sum;
                s =     (tmp > 127) ? 127 :
                        (tmp < -127) ? -127 :
                        (signed char) tmp;
        }
        else s = 0;

        return s;
}

// Calculate optic flow for all cameras
void process_image(     unsigned char old_tsl_pixels0[TSL_NUM_PIXELS],
                        unsigned char old_tsl_pixels1[TSL_NUM_PIXELS],
                        unsigned char old_tsl_pixels2[TSL_NUM_PIXELS],
                        unsigned char tsl_pixels0[TSL_NUM_PIXELS],
                        unsigned char tsl_pixels1[TSL_NUM_PIXELS],
                        unsigned char tsl_pixels2[TSL_NUM_PIXELS],
                        unsigned int delta_t, signed char s[REGIONS])
{
        unsigned int i, j;
        signed long     top_sum[REGIONS] = {0, 0, 0},
                        bottom_sum[REGIONS] = {0, 0, 0},
                        tmp[REGIONS] = {0, 0, 0};
        signed char olds[REGIONS] = {0, 0, 0}; // For low-pass filter

        // Calculate s
        s[0] = optic_flow(old_tsl_pixels0, tsl_pixels0);
        s[1] = optic_flow(old_tsl_pixels1, tsl_pixels1);
        s[2] = optic_flow(old_tsl_pixels2, tsl_pixels2);

        // Low pass filter
        float a = 0.8;
        for (j = 0 ; j < REGIONS ; j++)
                s[j] = (signed char) (a * (float) s[j] + (1.0f - a) * (float) olds[j]);

        tpm_send_number(1, 1, s[0]);
        tpm_send_number(2, 1, s[1]);
        tpm_send_number(3, 1, s[2]);

        // Save s
        for (j = 0 ; j < REGIONS ; j++)
                olds[j] = s[j];
}

   ⋮

int main()
{
        // Compute delta_t
        float delta_t = 4.5;
        unsigned int delay = (unsigned int) delta_t * MILLISEC;

   ⋮

        while(1)
        {
```

```
    ⋮
                // Grab images from cameras
                image_capture_time = get_timer4();
                tsl_grab_image_same(tsl_exposition_time, old_tsl_pixels0, old_tsl_pixels1,
                    old_tsl_pixels2);
                while (get_timer4() < image_capture_time + delay); // Wait until delta_t
                tsl_grab_image_same(tsl_exposition_time, tsl_pixels0, tsl_pixels1, tsl_pixels2);

    ⋮
                // Motor command
                // Use the "Control from monitor" checkbox in TP Monitor to control robot manually
                if (tpm_read_number(MODE) & 0b00000001)
                {
                        motorspeed[0] = (signed int)tpm_read_number(MOTOR_LEFT) * 10;
                        motorspeed[1] = (signed int)tpm_read_number(MOTOR_RIGHT) * 10;
                        e_set_speed_left(motorspeed[0]);
                        e_set_speed_right(motorspeed[1]);
                }
                else
                {
                        // Get forward bias from GUI
                        signed int bias = ((signed int) tpm_read_number(4));

                        // Caculate total optic flow and low-pass filter it
                        signed int d_sum = (-(signed int) s[2] + (signed int) s[1]) * 5;
                        sum = (8 * sum + 2 * d_sum)/10;

                        // Get optic flow weight from GUI
                        signed int weight = (signed int) tpm_read_number(5);

                        // Set motor speeds
                        e_set_speed_left(bias*10 - sum*10 + weight*((signed int) s[2])*bias/40);
                        e_set_speed_right(bias*10 - sum*10 - weight*((signed int) s[1])*bias/40);
                }

    ⋮
        }
}
```