Autonomous Micro-Aerial Vehicle Navigation Using a Custom Optic Flow Sensor Ring

Master Project

Raphaël Cherney
Microengineering Section

Fall 2012 – Winter 2013

Self-Organizing Systems Research Group (SSR)
Prof. Radhika Nagpal
Assistant: Karthik Dantu
Harvard University
School of Engineering and Applied Sciences (SEAS)

Laboratory of Intelligent Systems (LIS)
Prof. Dario Floreano
Assistant: Maja Varga
Swiss Federal Institute of Technology (EPFL)
School of Engineering (STI)
Institute of Microengineering (IMT)

EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE LAUSANNE
POLITECNICO FEDERALE DI LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY LAUSANNE

FACULTE SCIENCES ET TECHNIQUES DE L'INGENIEUR

LABORATORY OF INTELLIGENT SYSTEMS (LIS)
CH - 1015 LAUSANNE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

MASTER PROJECT

Title: Autonomous Micro-Aerial Vehicle Navigation Using a Custom Optic Flow Sensor Ring

Student(s): Raphael Cherney (MT)

Professor: Dario Floreano

Assistant 1: Maja Varga **Assistant 2:** Karthik Dantu

Project description:

The RoboBees project[1] is an effort to build a swarm of flapping-wing micro-aerial vehicles to collectively perform tasks such as crop pollination, disaster search, and target tracking. Each RoboBee is projected to weigh half a gram and be about 3 cm in length. Correspondingly, each RoboBee is extremely resource-scarce. However, the swarm is expected to be very large with hundreds of RoboBees. Given such a swarm, one of the main challenges in using it to perform the tasks listed above is coordination. To this end, the RoboBees project continues to research various ways of coordination to overcome the limitation of individual RoboBees and efficiently execute applications using the swarm[2]. Due to weight and energy limitations, it is hard to instrument micro-aerial vehicles with a variety of sensors. Since the RoboBees are currently under development, we use micro-helicopters as proxies for them. The objective of this project is to use a custom ring with eight optic flow sensors to perform ego motion estimation as well as indoor navigation on a micro-helicopter with most of the computation on-board. This will be used as the basis for ongoing research in studying distributed techniques for executing the target applications.

[1] The RoboBees Project, http://robobees.seas.harvard.edu
[2] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, Matt Welsh, "Programming Micro-Aerial Swarms with Karma", In SenSys '11: Proceedings of the 9th International Conference on Embedded Networked Sensor Systems, Seattle, Washington, Nov. 1-4, 2011.

Remarks:

You should present a research plan (Gantt chart) to your first assistant before the end of the second week of the project. An intermediate presentation of your project, containing 10 minutes of presentation and 10 minutes of discussion, will be held on October 31, 2012. The goal of this presentation is to briefly summarize the work done so far and discuss a precise plan for the remaining of the project. Your final report should start by the original project description (this page) followed by a one page summary of your work. This summary (single sided A4), should contain the date, laboratory name, project title and type (semester project or master project) followed by the description of the project and 1 or 2 representative figures. In the report, importance will be given to the description of the experiments and to the obtained results. A preliminary version of your report should be given to your first assistant at the latest 10 days before the final hand-in deadline. 3 copies of your final version, signed and dated, should be brought to the administration of your section before noon January 18, 2013. A 30 minute project defense, including 10 minutes for discussion, will take place between February 4 and 15, 2013. You will be graded based on your results, report, final defense and working style. All documents, including the report (source and pdf), summary page and presentations along with the source of your programs should be handed-in on a CD on the day of the final defense at the latest.

Responsible professor:	Responsible assistant:
Signature: Dario Floreano	Signature: Maja Varga

Lausanne, 12 October 2012

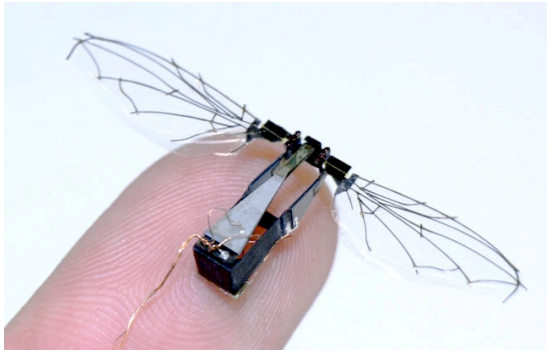
Autonomous Micro-Aerial Vehicle Navigation Using a Custom Optic Flow Sensor Ring

Raphael Cherney, Microengineering Section

Assistant: *Karthik Dantu and Maja Varga*

Professeur: *Radhika Nagpal and Dario Floreano*

The RoboBees project at Harvard University is attempting to push the current, practical limits of microengineering and micro-aerial vehicle (MAV) technology. The project brings together researchers from many different disciplines with the ultimate goal of developing a swarm of autonomous, robotic honey-bees.



RoboBee prototype

Since the RoboBees are currently under development, we use 30 g micro-helicopters as proxies for them. These platforms share similar restrictions on weight, power, and processing. With these devices, we are prototyping and testing strategies for sensing and control for use in larger, distributed swarms. We use a custom sensor ring with our coxial helicopter platform, which consists of 8 miniature cameras distributed in a circle around the body of the MAV. Using specialized vision sensors, we estimate the optic flow around the helicopter. This information can then be used to control the aircraft and ultimately for more complex tasks such as egomotion estimation, mapping, and coordination.

Autonomous MAV with custom sensor ring

We investigated strategies for indoor navigation with fully on-board comuptation. In order to test various sensor configurations and control strategies, we created a 3-dimensional simulation of the MAV and sensor ring using the Webots robotics simulator[1], along with a custom physics plugin.

3-Dimensional simulation

[1] http://www.cyberbotics.com/

Contents

1. Introduction

1.1. Background

Over the past century, we have made major progress in the development of flying vehicles. We have created machines that routinely travel faster than the speed of sound and others that reliably carry millions of passengers around the world. Throughout this process, we have also added increasing levels of autonomy into these systems, with the most obvious example being the autopilot. Today, we have moved well beyond basic autopilots into the realm of unmanned aerial vehicles (UAVs). UAVs are increasingly replacing manned systems and are widely used for a variety of missions ranging from search and rescue to strategic military operations. However impressive these systems may be, they still face many limitations. Most of these systems are heavily reliant on external systems such as the global position system (GPS); most are controlled remotely, requiring a continuous communication link; most have very limited sensing capabilities, leaving them unable to are unable to fly autonomously at low altitudes; and almost none of these aircraft can fly indoors.

An interesting area of research is the development of lightweight, unmanned micro-aerial vehicles (MAVs). These small-scale flying systems typically require less power, have increased maneuverability, are less dangerous, and can reach places that larger aircraft cannot. Many of these systems are even capable of safe flight in cluttered indoor environments. The tradeoff in size, however, means that these vehicles have limited payloads, restricting their sensing and computational capabilities. As we develop smaller and smaller systems, these restrictions become even more difficult. One project that attempts to push the current, practical limits of microengineering and MAV technology is the RoboBees project[1] at Harvard University.

1.2. RoboBees Project

This masters project is part of ongoing research into building a swarm of autonomous, robotic honey-bees. This research is being conducted at several laboratories within Harvard University, including the Microrobotics Laboratory headed by professor Robert Wood and the Self-Organizing Systems Research Group headed by professor Radhika Nagpal. The RoboBees project brings together researchers from different scientific and engineering disciplines to explore the limits of current technology. It is an effort to build a swarm of flapping-wing micro-aerial vehicles to collectively perform tasks such as:

- Crop pollination
- Search and rescue
- Hazardous environment exploration
- Military surveillance
- Weather and climate mapping
- Traffic monitoring

[1] `http://robobees.seas.harvard.edu`

The project is divided into three parts: Body, Brain, and Colony.

Body

This part of the project consists of designing an insect-sized autonomous flapping-wing flying robot. They will leverage existing breakthroughs from professor Wood's laboratory, which achieved their first successful flight of a life-sized robotic fly in 2007. The group will also investigate compact high-energy power sources for sustained, autonomous flight.

Brain

The Brain incorporates the development of the sensors, control, and circuitry to coordinate flight and target identification on the RoboBees. In such a resource constrained environment (limited power and computation), the group is striving to include computationally-efficient control, compact and efficient sensors, and energy-efficient electronic hardware. While not directly replicating the insect nervous system, they are also looking to biology to help inspire the design.

Colony

Inspired by the hive system of honeybees, the Colony group is attempting to develop coordination algorithms to overcome the limitations of individual RoboBees and exploit their numbers to accomplish tasks. This involves research into developing methods that leverage the entire colony – such as parallelism (exploration of large areas), energy efficiency (through information sharing and division of labor), and robustness (since individuals may fail or make errors).

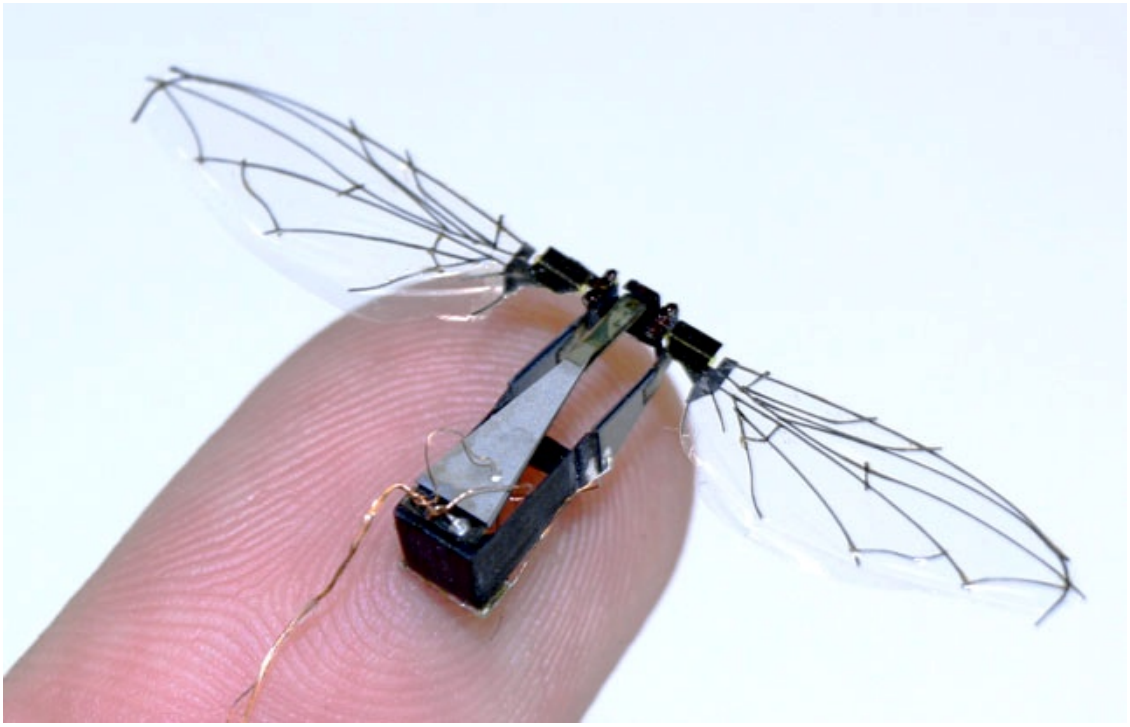
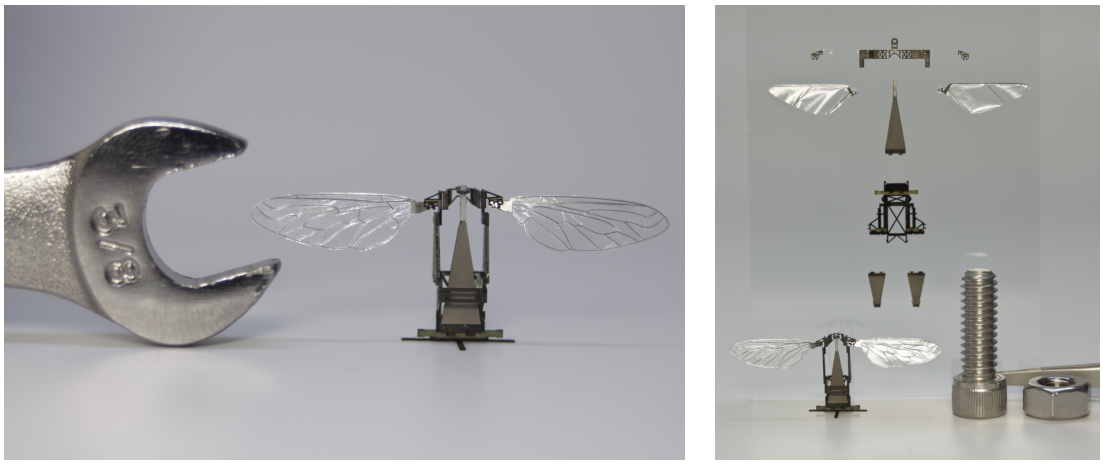


Figure 1: RoboBee prototype from 2009 (Photo by Ben Finio)



(a) (b)

Figure 2: 2011 RoboBee prototype (Photos by Eliza Grinnel)

1.3. Project Goals

Since the RoboBees are currently under development, we use micro-helicopters as proxies for them. These platforms are larger and support a heavier payload. Nevertheless, they share similar restrictions on weight, power, and processing. With these devices, we can begin to prototype and test our strategies for sensing and control for use in larger, distributed swarms. We have developed a specialized sensor ring for use with our coxial helicopter platform. This sensor ring consists of 8 miniature cameras distributed in a circle around the body of the MAV. Using these vision sensors, we can estimate the optic flow around the helicopter. This information can then be used to control the aircraft and ultimately for more complex tasks such as egomotion estimation, mapping, and coordination. The goals of this particular project are to:

- Get familiar with the current micro-helicopter platform
- Explore indoor, autonomous tasks enabled by the sensor ring
- Test candidate strategies in simulation and/or hardware
- Consider different sensor configurations

This work will be integrated into ongoing research in the lab related to navigation and control for resource-limited platforms and distributed robotic systems. Note that the project goals have evolved quite a bit from the beginning of the project due to difficulties with the prototype hardware.

2. State of the Art

The field of UAVs and MAVs is very large, and continually expanding. A very good starting point for vision-based guidance and control is [19], which provides a very good overview of the state of the art for vision-based aerial vehicles through 2009. [21] describes more work on an autonomous lightweight control system for a fixed-wing aircraft based on optic flow. There has also been significant work done at the ETH Zurich on advanced aerial vehicle control and navigation through vision. Some of this work is presented in [25] and [139], but requires much higher computational power than we have available. Similarly, [144] describes a similar corridor following task as we perform but with a far more powerful quadrotor system. [121] is able to perform multi-floor autonomous navigation, but is in another category of system power and complexity.

For our project, we are interested in the limits of autonomous navigation - situations where we have limited and noisy sensing, little computational power, and a low power budget. [148] provides very compelling work done on a 10 g micro-flyer controlled through optic flow (far closer to the projected 0.5 g of the final RoboBee). [119] proposes a simpler MAV autopilot based on optic flow for corridor following. There also is a large body of work from the Australian National University discussing biological systems and mechanisms relevant to our problem [128, 130, 129], and [49] provides a nice overview of transferring some of the biological ideas over to robotics.

3. Hardware

The main hardware for this project is an autonomous 30 gram coaxial helicopter with custom electronics for control and sensing. This includes a specialized sensor ring which includes 8 miniature cameras and onboard processing for optic flow measurement. This section briefly describes these components.

3.1. Coaxial Helicopter

3.1.1. Base Platform

Our MAV platform is based on the commercially-available Blade mCX2 coaxial helicopter (Figure 3). The vehicle is built around a plastic frame and has a length of 20 cm, a height of 12 cm, and a rotor diameter of 19 cm. The rotors are driven by two micro coreless DC motors. The upper rotor is attached to a flybar, or "stabilizer bar." The flybar acts as a passive stabilization mechanism by dampening rotations around the x and y axes. The lower rotor is connected to the swashplate. The swashplate cyclicly adjusts the pitch of the rotor blades in order to control the thrust vector. The swashplate itself is actuated by two micro linear servos integrated on the control board. The micro-helicopter is powered by a single lithium-polymer battery (3.7 V 150 mAh).

(a) Perspective (b) Front (c) Side

Figure 3: Blade mCX2 helicopter (© 2013 Blade)

3.1.2. Helicopter Control Board

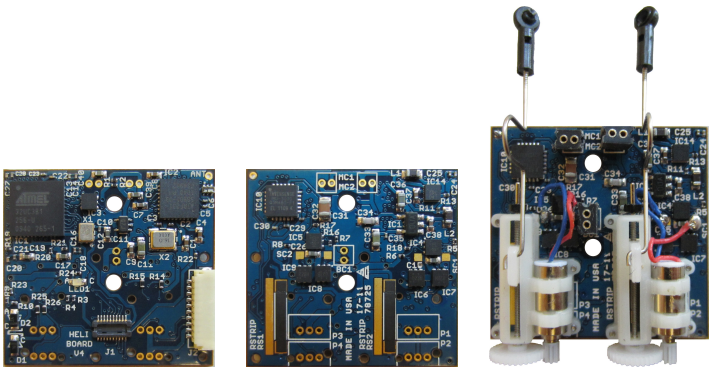
In order to add autonomous capabilities to this helicopter, we swap out the default control board on the base helicopter with a custom control board (Figure 4) designed by Centeye Ltd.[2]. The custom board has the following features:

- 32-bit AVR processor
- 2.4 GHz wireless radio

[2] http://centeye.com/

- 3-axis gyroscope
- Connector for additional sensors
- Voltage regulators
- Battery level sensing
- Two micro linear servomotors
- Drivers for two rotor motors
- LED indicator

The onboard microcontroller, a 32-bit Atmel AT32UC3B, runs at 48 MHz and is programmed in C using Atmel Studio though a custom programming connection. Data can be sent wirelessly from the helicopter to a custom base board connected to a computer over USB. The helicopter control board also supports additional sensors through a special, surface-mount connector. We have tested both an off-board accelerometer and a custom optic flow sensor ring (see 3.3) through this connection. At this point in time, the hardware and firmware is sill in the prototyping stage and work is being done to increase reliability for more extensive testing.



(a) Front (b) Back (c) Fully populated

Figure 4: Helicopter control board (actual size)

3.2. Inertial Sensing

In order to aid with stability, sensing, and control, the MAV includes a set of inertial sensors. These include an onboard gyroscope and optional, external accelerometer.

3.2.1. Gyroscope

Gyroscopes measure angular rates. Our helicopter platform uses a 3-axis, MEMS IMU-3000 gyroscope by InvenSense. This sensor consists of three independent vibratory MEMS rate gyroscopes, which detect rotation about the x, y, and z axes of the sensor using the

Coriolis Effect. The gyroscope includes onboard amplification and filtering of the signal which is then read out by a variable-rate 16-bit analog-to-digital converter (ADC) which can be programmed for several different sensitivities. The gyroscope is mounted directly on the helicopter control board, making the data easier to interpret. Following the typical convention of aerial vehicles, the x-axis faces the nose of the aircraft, the y-axis faces the right side of the vehicle, and the z-axis faces downward. The three rotations of the helicopter are known as roll (ϕ) around the x-axis, pitch (θ) around the y-axis, and yaw (ψ) around the z-axis (Figure 5). The gyroscope measures the corresponding angular rates $p = \dot{\phi}$, $q = \dot{\theta}$, $r = \dot{\psi}$.

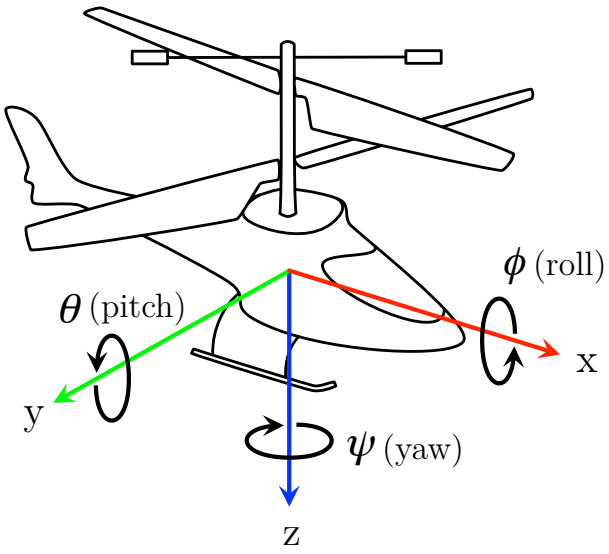


Figure 5: Helicopter axes

It is important to note that gyroscopes have a steady-state error which much be measured and accounted for when collecting data. This can be expressed by the following equation

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = S \cdot \begin{bmatrix} p_{reading} \\ q_{reading} \\ r_{reading} \end{bmatrix} - \begin{bmatrix} p_{offset} \\ q_{offset} \\ r_{offset} \end{bmatrix} \tag{1}$$

where S is the factory calibrated sensitivity of the gyroscope $[^\circ/s]$.

3.2.2. Accelerometer

In order to get an accurate measurement of the MAV attitude, we need more than a simple gyroscope. In order to get additional inertial information, we have a connector which allows us to interface the helicopter control board to the Sparkfun[3] 9 Degrees of Freedom Sensor

[3] `https://www.sparkfun.com/`

Stick (Figure 6). This small sensor board includes an ADXL345 3-axis accelerometer, a HMC5843 3-axis magnetometer, and an ITG-3200 3-axis gyroscope. The sensor stick communicates through I2C with the helicopter control board and can be mounted on the bottom surface of the MAV. This data can then be combined through a complementary or Kalman filter to get a better state estimate.

(a) Front (b) Back

Figure 6: Optional external sensor package (actual size)

3.3. Optic Flow Sensor Ring

A unique aspect of our MAV is the inclusion of a specialized optic flow sensor ring. This sensor includes 8 miniature, logarithmic imaging sensors placed around an (approximately) 10 cm diameter ring. The ring also includes onboard circuitry for the readout and processing of data from the cameras. We are able to continuously calculate the flow around the vehicle at rates in excess of 100 Hz and use this information for navigation and control.

3.3.1. Vision Sensors

The sensor ring uses 8 specialized logarithmic Faraya vision sensors designed by Centeye Ltd. Each monochromatic camera contains a 64×64 pixel array. Each pixel has a logarithmic response to the intensity of the incident light. These chips are designed specifically for embedded applications, sacrificing pixel count for more flexible acquisition, random pixel access, and ease of use. The pixels are shutterless, meaning that they are always adapting to light levels and generating an output. This greatly eases the readout process by removing the precise timing requirements of other imaging technologies such as active pixel sensors (APS) and charged coupled devices (CCD). The chips also feature the ability to bin pixels together for readout, improving the image quality for downsampled images. The pixel values are read out using a 10-bit ADC. Each camera has attached optics which help to focus the light into the pixel array. These optics give the sensors a field-of-view (FOV) of approximately 75° (determined experimentally).

(a) Front (b) Back

Figure 7: Vision sensor (actual size)

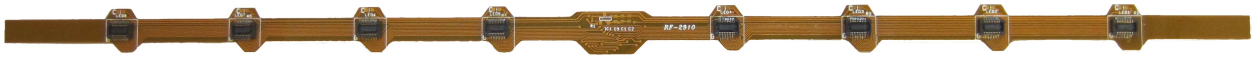
Due to imperfections in the manufacturing process, there are variations between individual pixels. In particular, we consider the fixed pattern noise (FPN) between pixels. We account for this offset by taking a reference image in uniform, dark lighting conditions and saving it in memory. We then subtract this mask from subsequent images we capture. We do, however, assume that response is otherwise the same between pixels (with good results). It is especially important to remove these steady state errors when calculating optic flow; images with steady state errors will appear as texture with no motion, causing the flow values to tend toward zero.

3.3.2. Sensor Strip

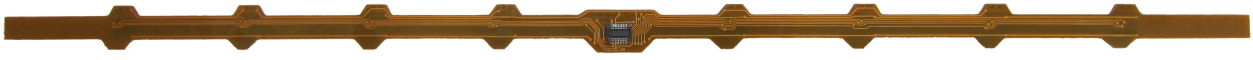
The vision sensors are mounted on a flexible printed circuit board (PCB) with approximately even spacing around the vehicle. This sensor configuration allows the MAV to see in all directions around the vehicle in its xy-plane. The sensor strip is flexible but will tend to hold its shape when connected as a ring. Because it is not perfectly round and the spacing is not even, we cannot rely on the exact positioning or angle of any of the cameras. Given the fact that we are interested in simple and robust behaviors, this should not pose a serious problem for our experiments. Table 1 gives the approximate angle of the camera relative to the center of the ring during one particular test case. Figure 8 shows the strips when laid out flat, and Figure 9 shows the assembled ring with cameras attached.

ID	Even Spacing	Measured
0	-157.5°	-149.2°
1	-112.5°	-110.6°
2	-67.5°	-72.5°
3	-22.5°	-34.0°
4	22.5°	34.8°
5	67.5°	73.6°
6	112.5°	112.3°
7	157.5°	150.8°

Table 1: Camera directions (relative to x-axis)



(a) Front



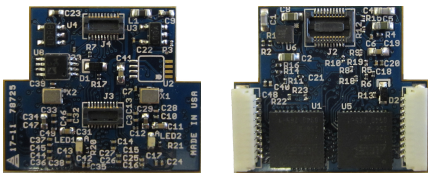
(b) Back

Figure 8: Sensor strip

Figure 9: Optic flow sensor ring (actual size)

3.3.3. Sensor Control Board

The sensor ring is controlled through a 32-bit AT32UC3B AVR microcontroller (the same type as on the helicopter control board). This microcontroller, running at 48 MHz, coordinates the readout of images from the various vision sensors attached to the ring. It also preforms the optic flow calculations and communicates this information to the helicopter control board over I2C.



(a) Front (b) Back

Figure 10: Sensor control board (actual size)

3.4. Assembly

The assembled MAV is completely autonomous with fully onboard power, sensing, and processing. It can fly for several minutes (6+) on a single charge. Figure 11 shows the complete working vehicle. Figure 12 labels the different components.

(a) Perspective

(b) Front

(c) Side

Figure 11: Assembled MAV

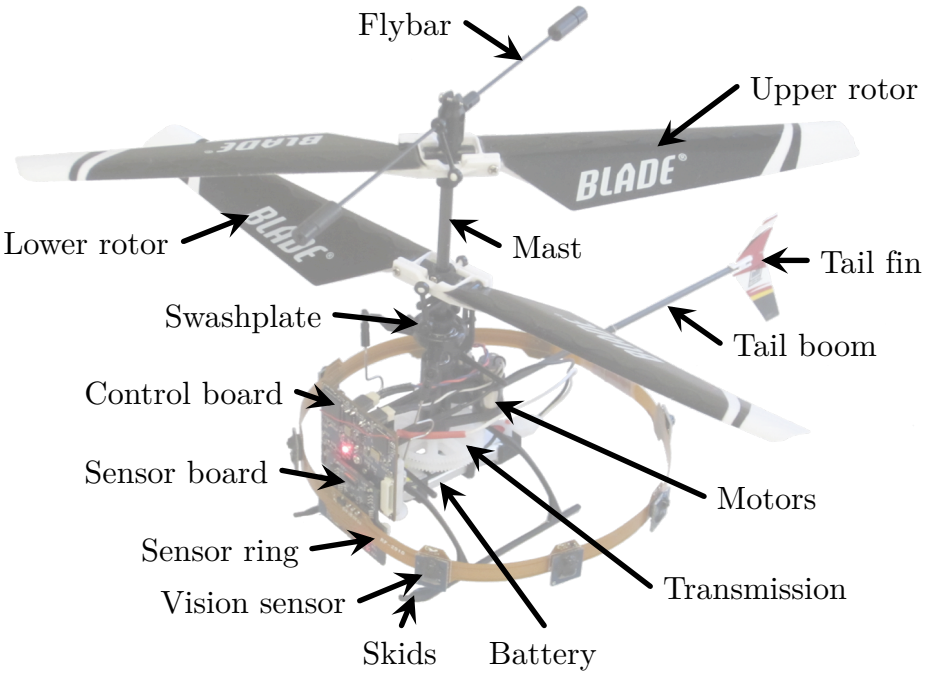


Figure 12: Parts of MAV

3.4.1. Weight Breakdown

As with all flying vehicles, our helicopter has a very tight weight budget. This limitation places similar constraints on the MAV that the RoboBees face. In particular, we want to investigate situations with limited computational and sensing resources. The weight breakdown of the MAV can be seen in Table 2. The MAV totals approximately 30 grams, including all of the onboard sensing. Due to its small size and limited actuation, the helicopter is most suited for indoor flight. The stronger and unpredictable winds in outdoor environments exceed the helicopter's maximum airspeed, overwhelming the onboard stability control.

Component	**Weight [g]**
Main rotor assembly (incl. motors)	18.89
Tail	0.71
Battery	4.25
Helicopter control board	2.05
Sensor ring control board	1.78
Sensor ring (incl. cameras)	2.61
Total	30.29

Table 2: Weight breakdown of MAV

4. Simulation

When working with MAVs, there are a variety of difficulties associated with using actual hardware. These problems include:

- Limited testing environments
- Limited flight time
- Difficulty tracking motion
- Difficulty logging data
- Hardware malfunctions
- Harder to program devices (longer iteration time)

In order to alleviate some of this issues, it is often useful to perform at least some kind of simulation. In our case, the prototype hardware has proved unreliable, making a simulation particularly attractive while we sort out the difficulties. A realistic, 3-dimensional simulation allows us to test different sensor configurations and algorithms without the hassles of reconfiguring and reprogramming several different devices.

4.1. Webots

For our simulation we used the Webots 7 software from Cyberbotics Ltd.[4]. The software incorporates an integrated development environment (IDE), physics engine, and 3-dimentional graphics to create a relatively simple way of implementing 3-dimensional physics-based simulations. Webots is designed as a robotics simulator, and as such has many common actuation and sensing capabilities built in. For example, we can easily add cameras to simulated the visions sensors on our sensor ring. Unfortunately, it is primarily geared toward ground based robots, and requires a custom physics plugin in order to model helicopter physics. The following list of features made Webots stand out as a simulation tool:

- Multi-platform (Linux, Windows, OS X)
- Open Dynamics Engine (ODE) for accurate physics simulation
- 3D visualization
- Sensor and actuator libraries to ease implementation
- Choice of programming languages (C, C++, Java, Python, MATLAB)
- Expandability
- Existing documentation
- Relatively straightforward conversion into hardware

[4] `http://www.cyberbotics.com/`

- Previous experience

In short, using Webots allowed this project and future work to get up and running quickly. By simplifying distribution and easing implementation of new ideas, the model becomes significantly more valuable. A custom made solution may run more quickly, and may ultimately be a worthwhile investment, but a Webots-baed model can be easily distributed and used to prototype experiments. Furthermore, because it is a mature and commercially released piece of software, there are fewer bugs and there exists good support material to get started [35, 36].

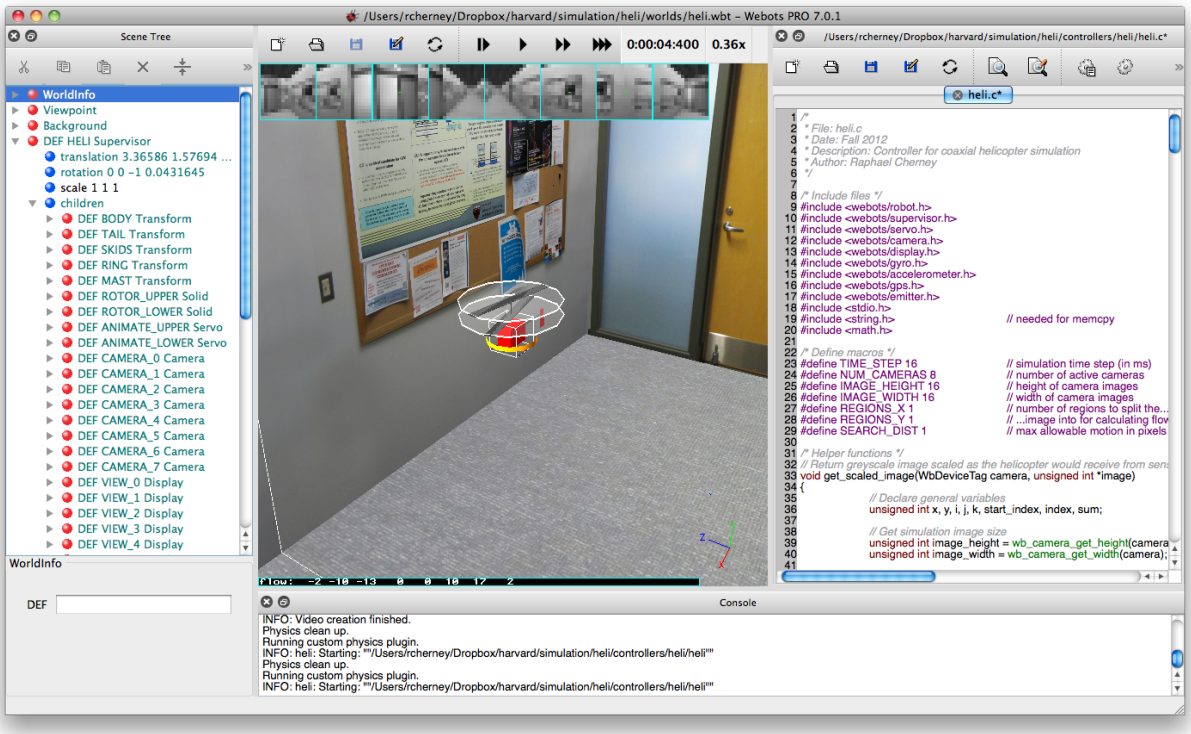


Figure 13: Webots graphical user interface (GUI)

4.2. Structure

There are three major components to our Webots simulation. The first is what is known as a *world* (.wbt) file. In Webots, all simulated objects and their properties are described by a hierarchical tree of specialized Nodes which are based on the Virtual Reality Modeling Language (VRML). These nodes include Cameras, Servos, Transforms, PointLights, and much more. Figure 14 shows the various Nodes that can be used to build up a simulation in Webots. The official Cyberbotics documentation in [35] is particularly useful for designing and using Webots models.

In addition to the world file which defines the simulation environment, we have a *controller* which is a computer program which controls a robot within the world file. These controllers can be written in one of several languages including C, C++, Java, Python, or MATLAB.

The controller can take simulated sensor inputs and make changes to controllable parts of the a Robot (or Supervisor) Node. When structured correctly, these controller programs should port easily to the desired platform.

Finally, we need a custom *physics plugin* to add the forces required for flight. For ground-based simulations, this is typically unnecessary, as the built-in Nodes and physics can simulate a large range of robots from simple two-wheels rovers, to complex humanoid robots.

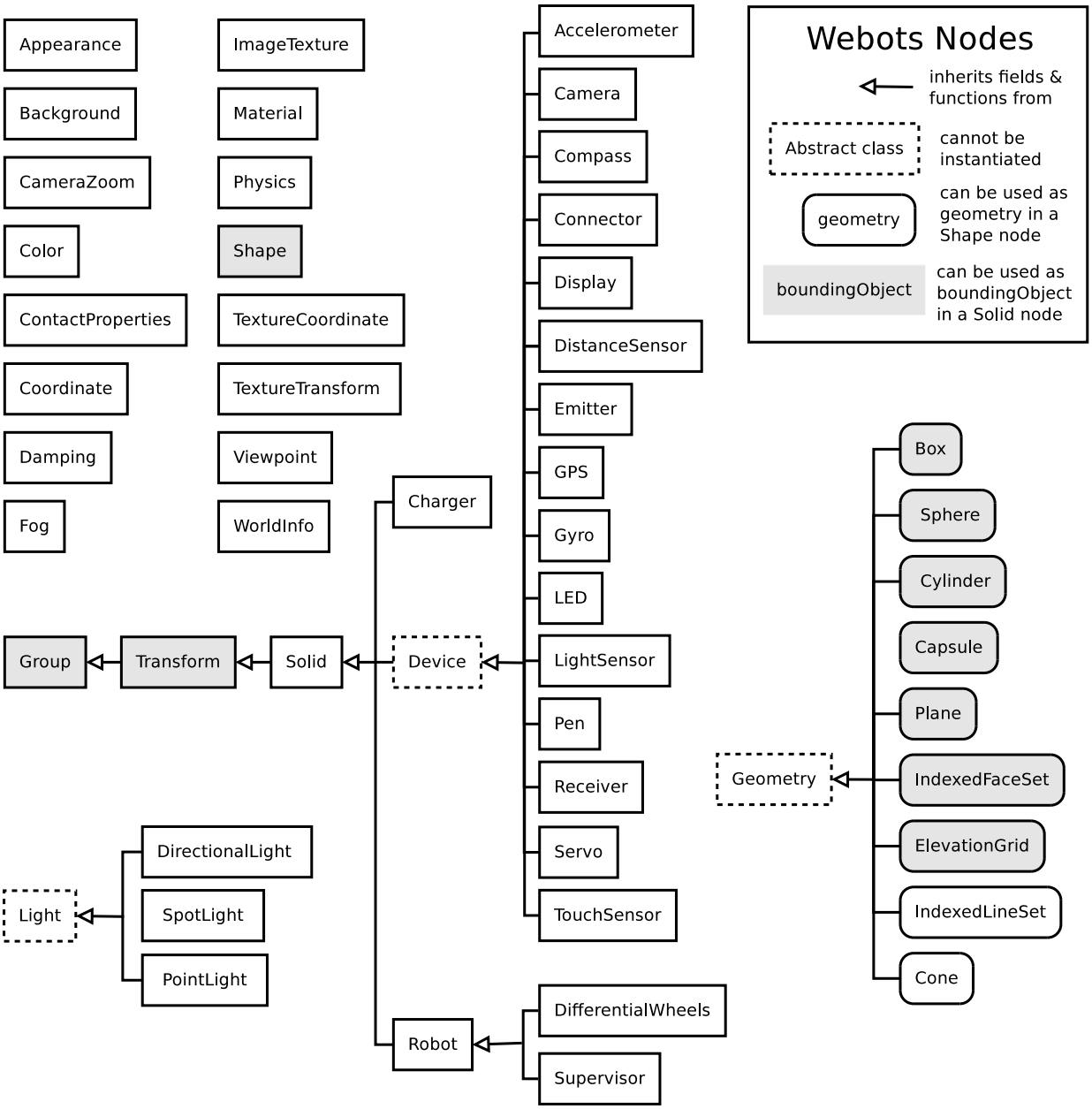


Figure 14: Webots Node chart (from [35])

The files included in our simulation are organized in a very specific way to make them easy open and run on any machine with Webots installed. The entire simulation is contained

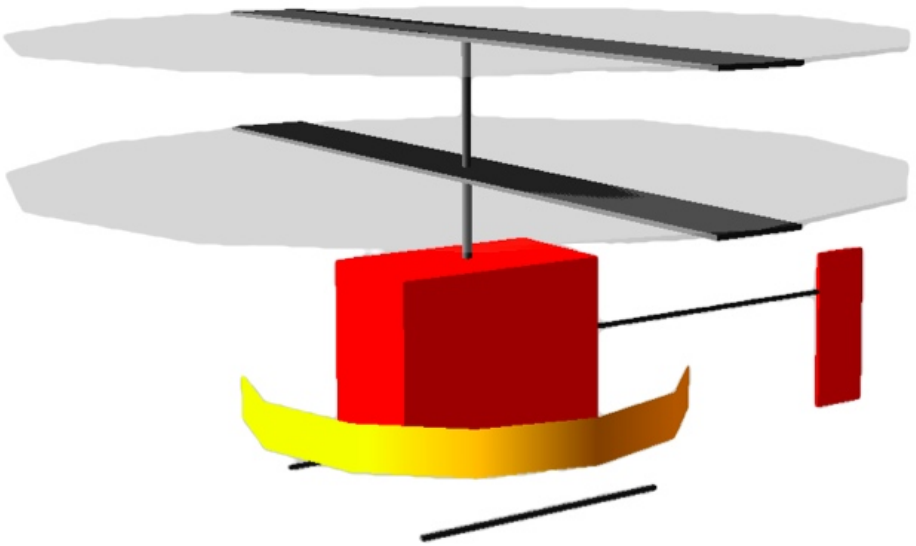
within the `heli` folder as shown in Figure 15. With this file hierarchy, opening the `heli.wbt` world should open Webots and begin the simulation. The controller `heli.c` is written and C and should be compiled after any changes.

```
heli/
    controllers/
        heli/
            heli.c
            Makefile
            log.txt
    plugins/
        physics/
            heli_physics/
                heli_physics.c
                Makefile
    worlds/
        textures/
            ceiling.jpg
            door.jpg
            floor.jpg
            poster.jpg
        heli.wbt
```

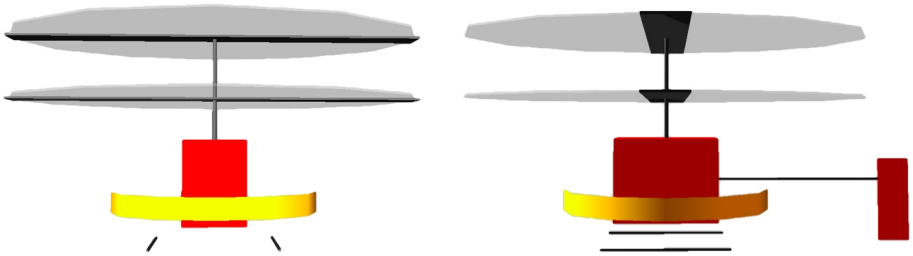
Figure 15: File hierarchy for Webots simulation

4.2.1. Helicopter

We use a simplified, block model of our helicopter platform in our simulation. Figure 16 shows our final Webots model of the MAV. To the extent that it was possible, the properties of the simulated helicopter are based on measurements of the actual platform.



(a) Perspective



(b) Front (c) Side

Figure 16: Helicopter model in Webots

Figure 17 shows the Nodes that are used to model the MAV and the hierarchical relationship between them (children are denoted by indentation). The robot itself is a Supervisor Node, which is simply a Robot that can execute privileged operations such as restarting the simulation, capturing a video, etc. There are a series of Transform and Shape Nodes that make up the visible structure of the helicopter. We assume that the vehicle has a mass of 30 g, and that the center of mass is at the center of the body. There are two Solid Nodes to represent the upper and lower rotors. Thrust forces are applied to the center of these Nodes. The rotors are represented by two semi-transparent cylinders that react to contact with other solid bodies. Two Servo motors animate spinning blades (traveling at a significantly slower angular velocity than the actual blades). The model includes 8 Camera Nodes evenly distributed around the helicopter with the angles given in Table 3 (refer to Figure 18). The images acquired from these cameras can then be displayed in a series of Display Nodes. Finally, we also include a Gyro Node, an Accelerometer Node, and a GPS Node for inertial sensing and tracking. Finally we have Emitter and Receiver Nodes for passing data to our custom physics plugin.

```
HELI Supervisor
    BODY Transform
        BODY Shape
    TAIL Transform
        FIN Transform
            FIN Shape
        BOOM Shape
    SKIDS Transform
        SKID_RIGHT Transform
            SKID Shape
        SKID_LEFT Transform
            SKID Shape
    RING Transform
        RING Shape
    MAST Transfrom
        MAST Shape
    ROTOR_UPPER Solid
        ROTOR Shape
    ROTOR_LOWER Solid
        ROTOR Shape
    ANIMATE_UPPER Servo
        BLADE Shape
    ANIMATE_LOWER Servo
        BLADE Shape
    CAMERA_0 Camera
        CAMERA Shape
    CAMERA_1 Camera
        CAMERA Shape
    CAMERA_2 Camera
        CAMERA Shape
    CAMERA_3 Camera
        CAMERA Shape
    CAMERA_4 Camera
        CAMERA Shape
    CAMERA_5 Camera
        CAMERA Shape
    CAMERA_6 Camera
        CAMERA Shape
    CAMERA_7 Camera
        CAMERA Shape
    VIEW_0 Display
    VIEW_1 Display
    VIEW_2 Display
    VIEW_3 Display
    VIEW_4 Display
    VIEW_5 Display
    VIEW_6 Display
    VIEW_7 Display
    GYRO Gyro
    ACCEL Accelerometer
    TRACKER GPS
    EMITTER Emitter
    RECEIVER Receiver
```

Figure 17: Structure of Webots MAV model

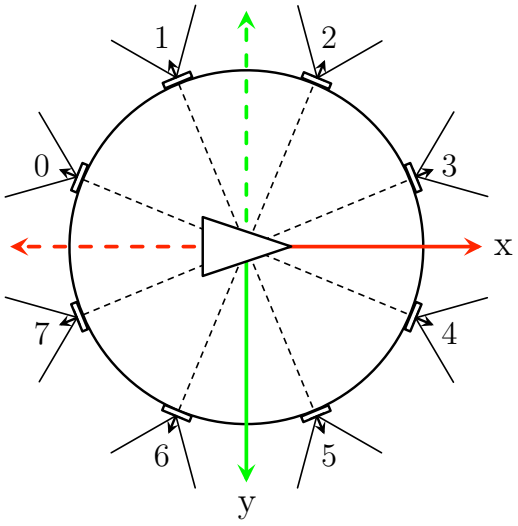


Figure 18: Camera numbering scheme

ID	Angle
0	$-7\pi/8$
1	$-5\pi/8$
2	$-3\pi/8$
3	$-\pi/8$
4	$\pi/8$
5	$3\pi/8$
6	$5\pi/8$
7	$7\pi/8$

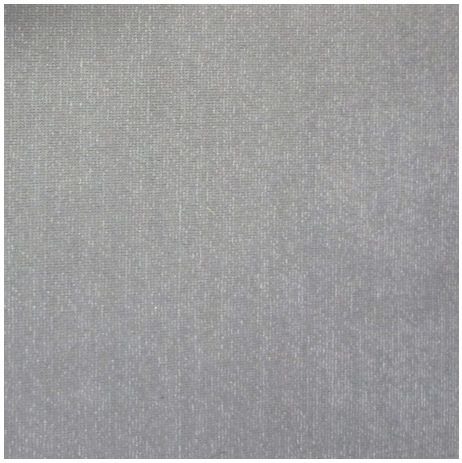
Table 3: Simulated camera directions (relative to x-axis)

4.2.2. Environment

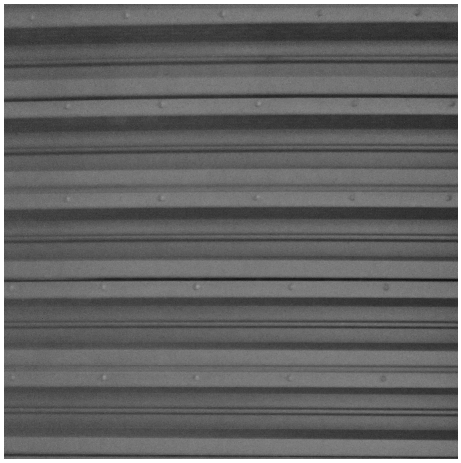
We are interested in flying our robots in real-world, indoor environments. As such, we created a Webots test environment that simulated these conditions. We created a simple floor plan with several hallways, intersections, and an open area. We then applied realistic, custom textures to the walls, floors, and ceilings. The textures are based on images captured from the the area in which we will be testing the MAV (Figure 19). The final map and Webots world can be seen in Figure 20.

(a) Door

(b) Poster

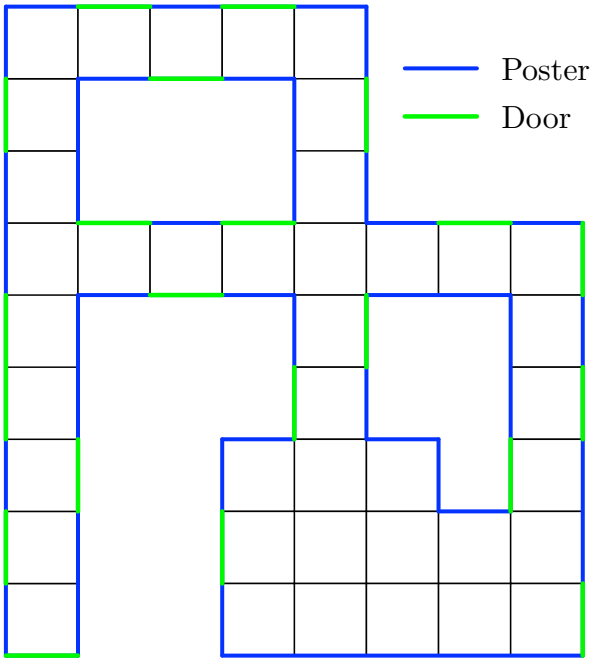


(c) Floor

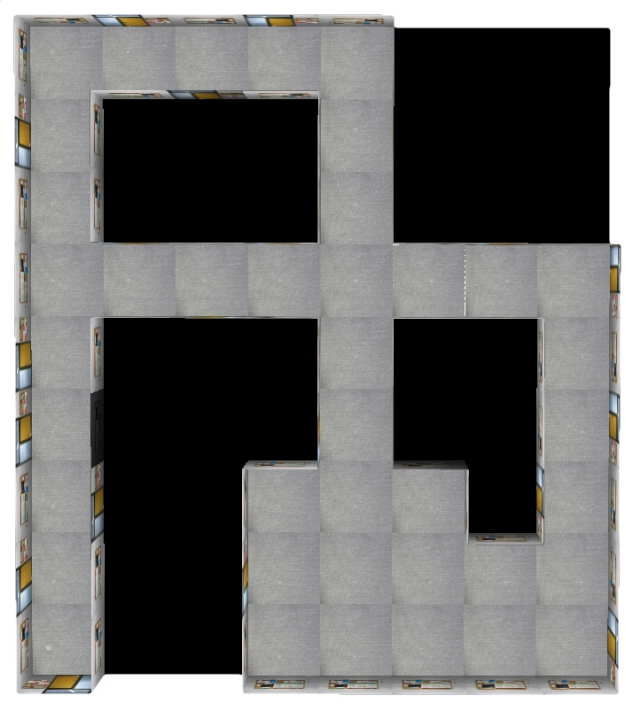


(d) Ceiling

Figure 19: Simulation textures



(a) Map



(b) Webots

Figure 20: Webots simulated indoor environment

4.3. Simplified Physics Model

We use a simplified physics model for simulating the helicopter dynamics. The model is largely based on previous work presented in [111] and [26]. The following sections describe the various forces and torques implemented in our simulation. Figure 21 shows the forces included in our model of the MAV.

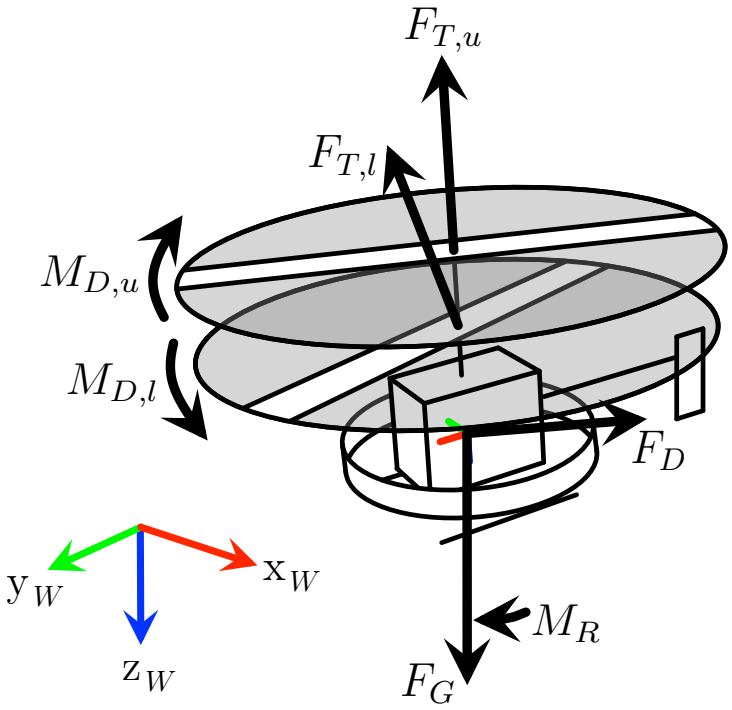


Figure 21: Modeled forces

4.3.1. Assumptions

Before beginning, we make several assumptions about our system to simplify the modeling process. In particular, we assume the following:

- System in near hover condition
- No interaction with ground or other surfaces
- Helicopter and rotors are rigid
- Upper rotor thrust is parallel to the body z-axis (no flybar)
- Servo positions and rotor speeds can be changed instantaneously
- Center of gravity and the body frame origin coincide

These are all reasonable assumptions given the scale of our system, the environment we are considering, and the desired simplicity for our model.

4.3.2. Reference Frames

There are two reference frames in our dynamical model: the inertial or Earth frame (I) and the body-fixed frame (B). The body frame coincides with the center of mass of the MAV and uses the standard arial vehicle conventions (Figure 5) with the x-axis pointing forward, the y-axis to the right, and the z-axis downward toward the ground (though Webots implementation differs slightly). The helicopter has 6 degrees of freedom: motion in the x, y, and z direction and rotation about the three axes (ϕ, θ, and ϕ). If the rotations are written in terms of rotation matrices (R_{z}, R_{y}, R_{x}), the general rotation R can be written as

$$R = R_{\mathrm{z}} R_{\mathrm{y}} R_{\mathrm{x}} \tag{2}$$

where:

$$\begin{aligned} R_{\mathrm{z}}(\psi) &= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ R_{\mathrm{y}}(\theta) &= \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \\ R_{\mathrm{x}}(\phi) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \end{aligned} \tag{3}$$

The three angles giving the three rotation matrices are called Euler angles. In this case we have roll (ϕ) around the x-axis, pitch (θ) around the y-axis, and yaw (ψ) around the z-axis. When modeling these rotations, the order in which they are applied is important. For our modeling, we consider the yaw-pitch-roll convention, giving us

$$\mathrm{R}(\psi,\theta,\phi) = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix} \tag{4}$$

where R is the rotation matrix from the inertial frame to the body frame of the MAV. When modeling in Webots specifically, most of these fundamental rotations can be taken care of automatically. We leverage this flexibility where possible.

4.3.3. Gravity

Gravity causes bodies to be attracted to one another with a force proportional to the product of the two masses and inversely proportional to the square of the distance between them. Mathematically, it is given by the following equation,

$$F_G = G \cdot \frac{m_1 \cdot m_2}{r^2} \tag{5}$$

where F_G is the force due to gravity between the bodies, G is the gravitational constant ($G \approx 6.674 \cdot 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$), m_1 and m_2 are the masses of the two bodies, and r is the

distance between the two center of masses. On the Earth's surface, force of gravity can be simplified to a downward force acting on all objects with a magnitude based on the object's mass. We treat our entire MAV as a single rigid body and apply the force due to gravity at the center of mass of the helicopter. The force is given by,

$$F_G = m \cdot g \tag{6}$$

Where F_G is the applied force, m is the mass of the MAV, and g is the acceleration due to gravity ($g \approx 9.81\ \mathrm{m\,s^{-2}}$). In the inertial frame, this force is represented by the following vector:

$$\vec{F_G} = \begin{bmatrix} 0 \\ 0 \\ m \cdot g \end{bmatrix} \tag{7}$$

This force is already well integrated into the Webots platform, affecting all objects (with Physics node) given either a density or a mass. Our simulated MAV has a mass of 30 g.

4.3.4. Propellor Dynamics

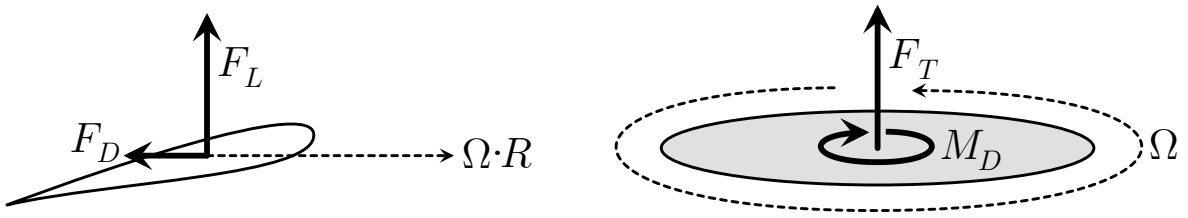
Propellors have complex aerodynamics that have been studied for many years. We are not concerned with the exact intricacies of the physics involved and instead just model the two main forces associated with the moving rotor blades: thrust and rotor drag. As the rotor blade spins with an angular velocity of Ω, the blade will push air downward based on its angle of attack. The force pushing the air downward also acts as lift on the aircraft (F_L) as described by Newton's third law of motion. Because the blades are moving so quickly, we can integrate this force over the cycle and apply a single thrust force F_T at the center of rotation. The magnitude of this force is given by

$$F_T = C_T \cdot \pi \cdot \rho \cdot R^4 \cdot \Omega^2 \tag{8}$$

where C_T is a the trust coefficient which is assumed to be constant, ρ is the air density, and R is the radius of the rotor. As the blade travels through the air, it also experiences a drag force resisting its motion (F_D). As the blade rotates, this force is translated to a torque acting at the center of rotation which we will call the drag moment M_D. The magnitude of the rotor drag torque is given by

$$M_D = C_D \cdot \pi \cdot \rho \cdot R^5 \cdot \Omega^2 \tag{9}$$

where C_D is a constant drag coefficient. Figure 22 shows how these basic forces act on the blade and the forces that we model in our simulation.



(a) Forces on blade

(b) Modeled forces

Figure 22: Propellor aerodynamics

4.3.5. Rotor Thrust

The thrust from the rotors is what lifts the helicopter off of the ground and keeps it in the air. Our MAV has two rotors, an upper rotor rotating counterclockwise and a lower rotor rotating clockwise (from above). The magnitude of the thrust force F_T produced by either rotor is given by equation 8. Because we are assuming a constant air density and rotor size, we can incorporate all of these variables into a single thrust constant C_T. This gives us the following equation for the thrust force of rotor i:

$$F_{T,i} = C_{T,i} \cdot \Omega_i^2 \tag{10}$$

On a most basic level, this force is applied along the z-axis of the MAV. This would give us the following force vector $\vec{F_T}$:

$$\vec{F_T} = F_T \cdot \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -F_T \end{bmatrix} \tag{11}$$

However, the the angle of attack of the rotors – and therefore the lift force – can be actively varied over the course of a rotation. This causes the thrust vector to rotate. The lower rotor uses a swashplate to control this cyclic action. The angle of the swashplate is adjusted by two micro linear servos on the helicopter control board. The swashplate, in turn, causes the thrust force to have an effective tilt of α around the y-axis and β around the x-axis. This allows us to actively control the pitch and roll of the helicopter. Figure 23 shows the different components of the thrust vector created by the rotation, along with several projections. The following set of equations are used to determine the thrust vector components F_{x}, F_{y}, and F_{z} given an initial thrust magnitude, F_T (as determined by equation 10), and the two angles α and β for the rotation of that thrust (relative to the helicopter's reference frame).

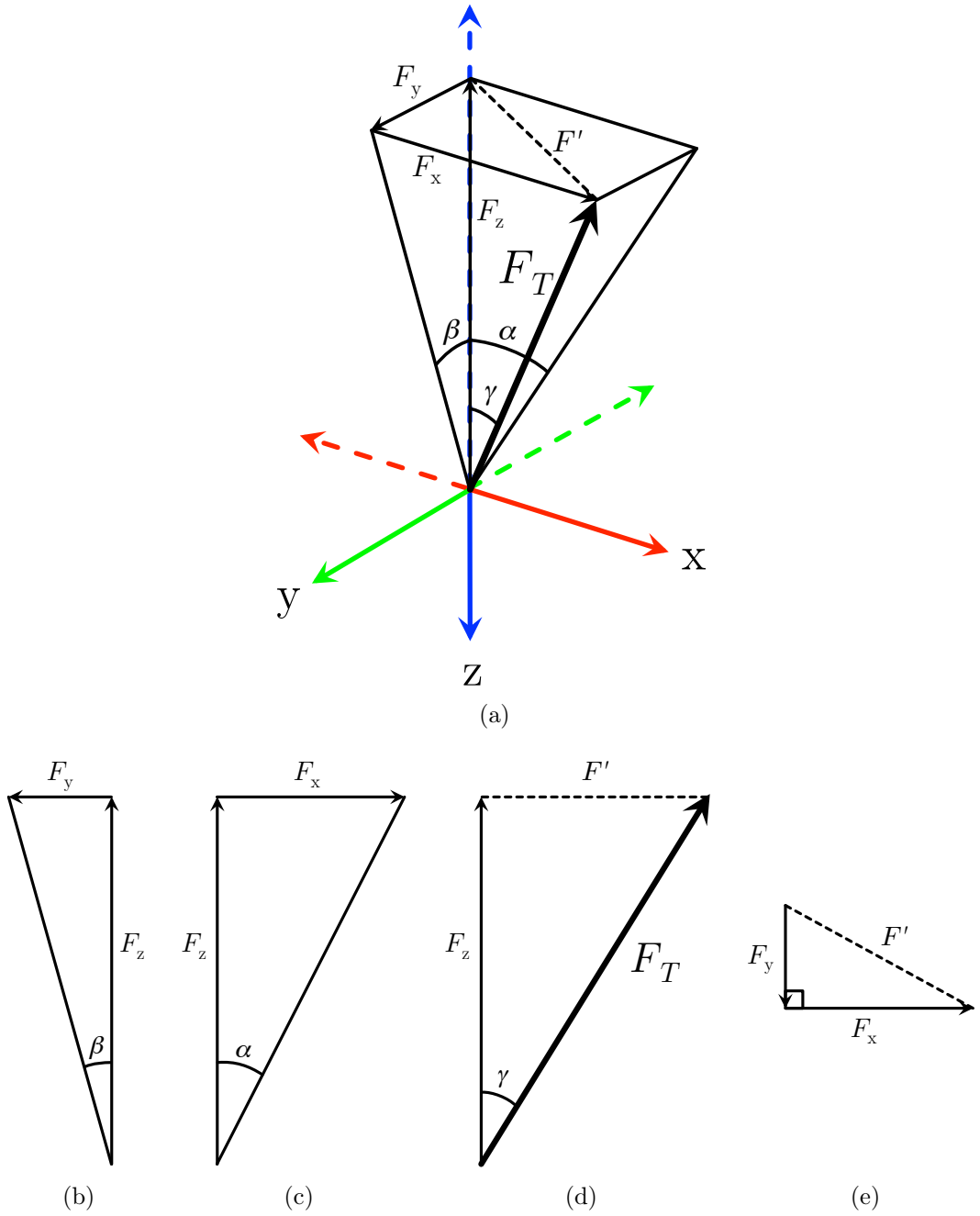


Figure 23: Calculating the components of the thrust vector

$$
\begin{aligned}
\tan^2\gamma &= \frac{(F')^2}{(F_\mathrm{z})^2} \\
\tan^2\gamma &= \frac{(F_\mathrm{x})^2 + (F_\mathrm{y})^2}{(F_\mathrm{z})^2} \\
\tan^2\gamma &= \tan^2\alpha + \tan^2\beta \\
\tan^2\gamma &= \frac{\sin^2\alpha}{\cos^2\alpha} + \frac{\sin^2\beta}{\cos^2\beta} \\
\frac{\sin^2\gamma}{\cos^2\gamma} &= \frac{\sin^2\alpha}{\cos^2\alpha} + \frac{\sin^2\beta}{\cos^2\beta} \\
\frac{1-\cos^2\gamma}{\cos^2\gamma} &= \frac{\sin^2\alpha}{\cos^2\alpha} + \frac{\sin^2\beta}{\cos^2\beta}
\end{aligned} \tag{12}
$$

Simplifying this equation, we get

$$
\cos\gamma = \frac{\cos\alpha\cdot\cos\beta}{\sqrt{1-\sin^2\alpha\cdot\sin^2\beta}} \tag{13}
$$

We can then use this to find the three components of the thrust vector, F_x, F_y, and F_z.

$$
F_\mathrm{z} = -\cos\gamma\cdot F_T = \frac{-\cos\alpha\cdot\cos\beta}{\sqrt{1-\sin^2\alpha\cdot\sin^2\beta}}\cdot F_T \tag{14}
$$

$$
F_\mathrm{x} = \tan\alpha\cdot F_\mathrm{z} = \frac{-\sin\alpha\cdot\cos\beta}{\sqrt{1-\sin^2\alpha\cdot\sin^2\beta}}\cdot F_T \tag{15}
$$

$$
F_\mathrm{y} = -\tan\beta\cdot F_\mathrm{z} = \frac{\cos\alpha\cdot\sin\beta}{\sqrt{1-\sin^2\alpha\cdot\sin^2\beta}}\cdot F_T \tag{16}
$$

Our final thrust vector $\vec{F_{T,l}}$ then becomes

$$
\vec{F_{T,l}} = \begin{bmatrix} F_\mathrm{x} \\ F_\mathrm{y} \\ F_\mathrm{z} \end{bmatrix} \tag{17}
$$

This force is applied to the model at the center of the lower rotor. The physics engine automatically accounts for the moments that this induces on the rigid helicopter model. The value of α and β are determined by a linear relationship to a servo command value sent to the physics plugin from the robot controller. Similarly, the value of Ω_l and Ω_u are linearly related to a velocity command sent by the helicopter controller. In this way, we assume that the servos instantaneously reach their desired set point and the rotors are speed-controlled and can immediately adjust their velocity to incoming commands. More realistic motor limitations should be accounted for in the controller. With the proper testing equipment, we could construct a more realistic relationship between particular command values and real-world forces; however, this is not the goal of our project.

The upper rotor uses a passive flybar to increase stability of the MAV. Due to the high inertia of the stabilizer bar, it offers resistance to any attempt to modify the orientation of

the rotation axis. As a result the stabilizer bar itself and the upper rotor, which are linked, lag behind the roll or pitch movement of the helicopter. A model of this is presented in [26]. Due to the extensive testing required to calibrate and measure this effect, we simply group these passive stabilization elements into a single (optional) restoring force (discussed in 4.3.8). The thrust of the upper rotor ($\vec{F_{T,u}}$) is simply given by equation 11 and applied to the body frame at the center of the upper rotor (along body z-axis).

4.3.6. Rotor Drag

As discussed in 4.3.4, the rotors experience drag as they rotate. This rotor drag creates a torque on the body of the helicopter given by equation 9. We can simplify this torque for our model by keeping the air density and rotor size fixed and combining everything into a a single drag constant C_D. This gives us the following equation for the rotor drag torque

$$M_{D,i} = C_D \cdot \Omega_i^2 \tag{18}$$

This torque is applied to the helicopter based on the direction of rotation of the rotor (since the drag acts opposite to the direction of motion). Because coaxial helicopters have rotors that spin in opposite directions and have the same center axis, the rotor drag from each rotor will tend to cancel the other out. When both rotor velocities are equal, there is no rotor drag torque applied ($M_D = 0$ when $\Omega_l = \Omega_u$). When there is an imbalance in the rotor velocities, the MAV will experience a torque about the body z-axis. This relationship is expressed in the following equation

$$\vec{M_D} = \begin{bmatrix} 0 \\ 0 \\ M_{D,l} - M_{D,u} \end{bmatrix} \tag{19}$$

4.3.7. Fuselage Drag

When the helicopter is in motion, the body will feel a drag force due to air resistance on the fuselage. This force is proportional to the velocity of the body squared. This force can be expressed by the following equation which give the drag force vector $\vec{F_D}$ in the inertial frame,

$$\vec{F_D} = \begin{bmatrix} -C_{F,\mathrm{x}} \cdot u^2 \mathrm{sign}\,(u) \\ -C_{F,\mathrm{y}} \cdot v^2 \mathrm{sign}\,(v) \\ -C_{F,\mathrm{z}} \cdot w^2 \mathrm{sign}\,(w) \end{bmatrix} \tag{20}$$

where C_F is a the drag coefficient of the body in air and u, v, and w are the spacial velocities. For our simple model, we assume that C_F is constant in all directions.

4.3.8. Restoring Moment

With the basic forces we have just discussed, the helicopter will hover and fly, but it lacks the passive stabilization elements of the physical MAV. As a simple first-order solution, we

implement a restoring force which acts to keep the helicopter near-vertical. The restoring torque M_R is linearly related to the angle of the helicopter off the z-axis (see Figure 24).

$$M_{R,\theta} = -C_{R,\theta} \cdot \theta \tag{21}$$

$$M_{R,\phi} = -C_{R,\phi} \cdot \phi \tag{22}$$

For our implementation, we use a constant C_R for both directions of stability. This gives us the following restoring moment to be applied to the body frame,

$$\vec{M_R} = \begin{bmatrix} -C_R \cdot \phi \\ -C_R \cdot \theta \\ 0 \end{bmatrix}$$

The restoring force is applied directly to the MAV body. While this force is not physically correct, it provides a good first-order match to what we observe in the physical platform. By creating this force, we simplify our simulation and control, allowing us to more easily accomplish our goal of testing visually-based control and navigation strategies .

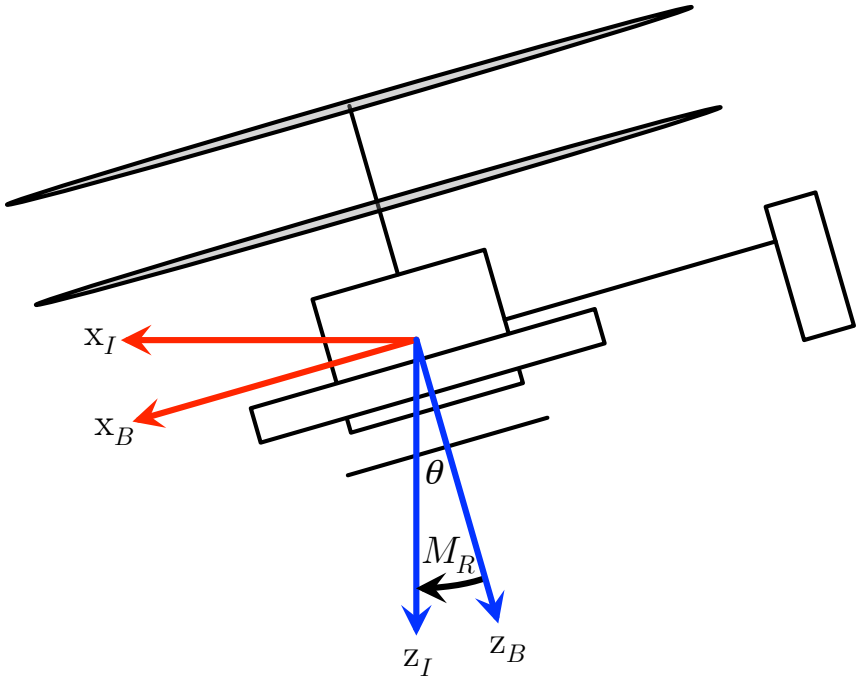


Figure 24: Restoring moment

4.3.9. Other Forces

We purposefully simplified our dynamical model by not including all of the dynamical subsystems and attributes of a coaxial helicopter. In particular, we do not model the rotor reaction torque (caused by changes in the rotational speeds of the motors), the propellor-gyro effect, or rotor flapping torque. We also ignore the dynamic subsystems of the drive

trains, the lower rotor and swashplate, and the the upper rotor and stabilizer bar. These systems require more information and testing on the system in order to model accurately. As we are more interested in quickly being able to test different control and navigation strategies, our current simplified model appears to work well. More complete models are discussed at length in [26] and [17]. One last optional force that can be useful in our model, is adding a Webots Damping Node to the helicopter. This force aids with stability of the simulation and can act as a first-order model of flybar stabilization and/or drag.

4.3.10. Implementation

Using the Webots platform greatly simplifies the process of creating our simulation. Webots already implements gravity and models collisions between objects (given a bounding object). Nevertheless, we require a custom physics plugin in order to apply the various forces required to model flight. The custom physics plugin interacts with the Open Dynamics Engine (ODE) backend through the common API. This process is described in [35] with the ODE API outlined in [123]. We receive four commands from the controller corresponding to the commands for the upper and lower rotor velocities and the pitch and roll commands for rotation of the lower thrust vector. These commands are processed as described in the previous sections to generate a series of forces and moments. We then apply the forces using the commands `dBodyAddForce` (for forces in inertial frame coordinates) and `dBodyAddRelForce` (for adding forces in body frame coordinates). Similarly, torques are added using the commands `dBodyAddTorque` (inertial frame) and `dBodyAddRelTorque` (body frame). The entire physics plugin is included in B.1.

5. Optic Flow

Optic flow, also known as optical flow, is the apparent visual motion of objects, surfaces, and edges in a scene caused by the relative motion between an observer and a scene. We can estimate this motion using series of images taken in close spacial and temporal proximity and extracting information from the changes between them. This information can then be used to gather information about the environment and ultimately incorporated into our control and navigation.

5.1. Algorithm

There are several algorithms for calculating the optic flow based on a series of images. Some of the most common methods include block matching, the Lucas-Kanade method, or other gradient techniques. In our implementation, we use a variant of the image interpolation algorithm (I2A). The algorithm is detailed in [127], with our particular implementation described in [54]. We chose the algorithm because it is simple, fast, and lightweight.

The algorithm calculates the horizontal and vertical flow by comparing the current image with a set of four reference images which are translated from a previous frame. The pixel intensity function at time t_0 and t is $f_0(x, y)$ and $f(x, y)$, respectively where x and y are the image coordinates measured in pixels. The four reference images f_1, f_2, f_3, and f_4 are formed by shifting the image f_0 by reference shifts Δx_{ref} and Δy_{ref} along the horizontal and vertical directions as described in equation 23. These shifts determine the maximum displacement we account for (and therefore sensitivity, as well).

$$\begin{aligned} f_1(x, y) &= f_0(x + \Delta x_{ref}, y) \\ f_2(x, y) &= f_0(x - \Delta x_{ref}, y) \\ f_3(x, y) &= f_0(x, y + \Delta y_{ref}) \\ f_4(x, y) &= f_0(x, y - \Delta y_{ref}) \end{aligned} \tag{23}$$

The algorithm assumes that the image at time t can be linearly interpolated from f_0 and the four reference images. With this assumption, the pixel coordinate translation Δx and Δy can be expressed as in

$$\hat{f} = f_0 + 0.5 \left(\frac{\Delta x}{\Delta x_{ref}} \right) (f_2 - f_1) + 0.5 \left(\frac{\Delta y}{\Delta y_{ref}} \right) (f_4 - f_3) \tag{24}$$

We then just need to solve for the translations Δx and Δy that give us the interpolated image $\hat{f}$ which is closest to the actual image f. We do this by solving the least squares problem in our region of interest which is defined by the window function Ψ (we use a boxcar kernel). The error function which we need to minimize then becomes

$$E = \int \int \Psi \cdot \left(f - \hat{f} \right)^2 dx\, dy \tag{25}$$

We minimize the error by taking the partial derivatives of E with respect to Δx and Δy and setting them equal to zero. This allows us to form the following set of equations:

$$\left(\frac{\Delta x}{\Delta x_{ref}}\right)\int\int \Psi\cdot\left(f_2-f_1\right)^2 dx\,dy+\left(\frac{\Delta y}{\Delta y_{ref}}\right)\int\int \Psi\cdot\left(f_4-f_3\right)\left(f_2-f_1\right) dx\,dy$$
$$=2\int\int \Psi\cdot\left(f-f_0\right)\left(f_2-f_1\right) dx\,dy \quad (26)$$

$$\left(\frac{\Delta x}{\Delta x_{ref}}\right)\int\int \Psi\cdot\left(f_2-f_1\right)\left(f_4-f_3\right) dx\,dy+\left(\frac{\Delta y}{\Delta y_{ref}}\right)\int\int \Psi\cdot\left(f_4-f_3\right)^2 dx\,dy$$
$$=2\int\int \Psi\cdot\left(f-f_0\right)\left(f_4-f_3\right) dx\,dy \quad (27)$$

We can then solve this set of equations for Δx and Δy which, in turn, gives us an estimate of the optic flow ($OF_x \propto \frac{\Delta x}{t-t_0}$ and $OF_y \propto \frac{\Delta y}{t-t_0}$). For our implementation, we directly calculate the following sums over our window:

$$\begin{aligned} A &= \sum\left(f_2-f_1\right)^2 \\ B &= \sum\left(f_4-f_3\right)\left(f_2-f_1\right) \\ C &= \sum\left(f-f_0\right)\left(f_2-f_1\right) \\ D &= \sum\left(f_4-f_3\right)^2 \\ E &= \sum\left(f-f_0\right)\left(f_4-f_3\right) \end{aligned} \quad (28)$$

We can then use these sums to calculate our flow directly by the following equations,

$$\Delta x = 2\cdot\Delta x_{ref}\cdot\frac{CD-BE}{AD-B^2} \quad (29)$$

$$\Delta y = 2\cdot\Delta y_{ref}\cdot\frac{AE-CB}{AD-B^2} \quad (30)$$

This algorithm runs quickly and reliably in both our simulation and on the 32-bit microcontroller in our optic flow sensor ring.

5.2. Translational Flow

There are clear mathematical relationships between the magnitude and direction of the optic flow and the relative motion between bodies. As we intuitively know, when traveling at a constant velocity, objects which are closer appear to move more quickly in our visual field. We also note that optic flow is maximized when the motion is perpendicular to the motion of the observer. This relationship is expressed by equation 31.

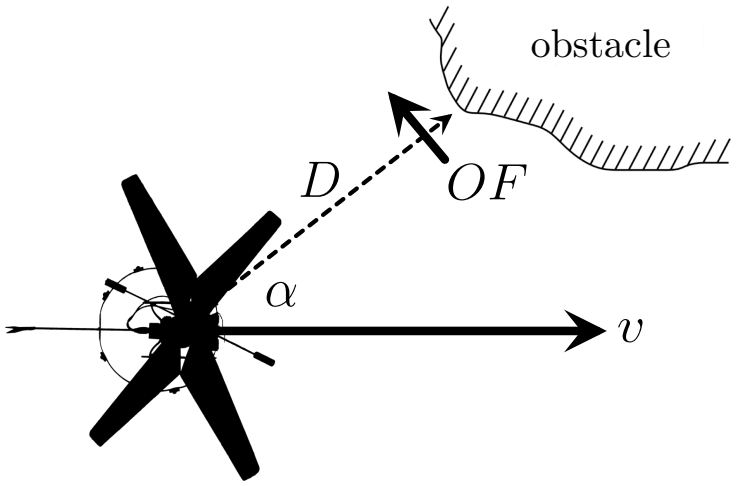


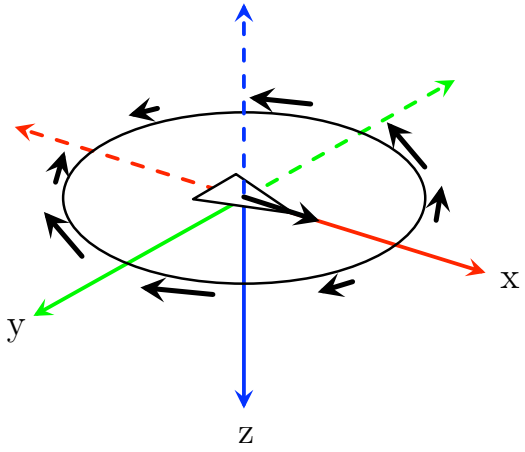
Figure 25: Optic flow during translational flight

$$OF = \frac{v}{D} \cdot \sin \alpha \tag{31}$$

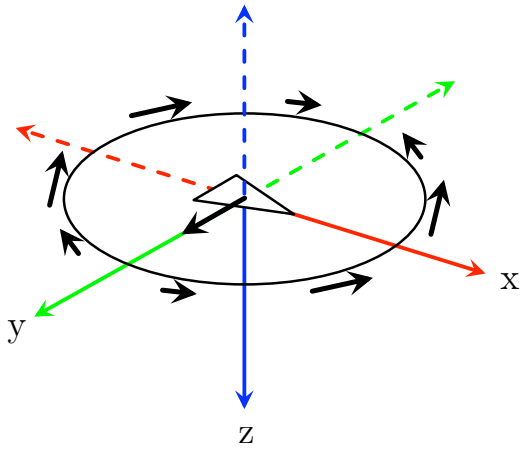
where OF is the measured optic flow, v is the velocity of the object, and α is the angle between the object and direction of travel (Figure 25). We can use this to estimate the distance to a particular object based on the flow

$$D = \frac{v}{OF} \cdot \sin \alpha \tag{32}$$

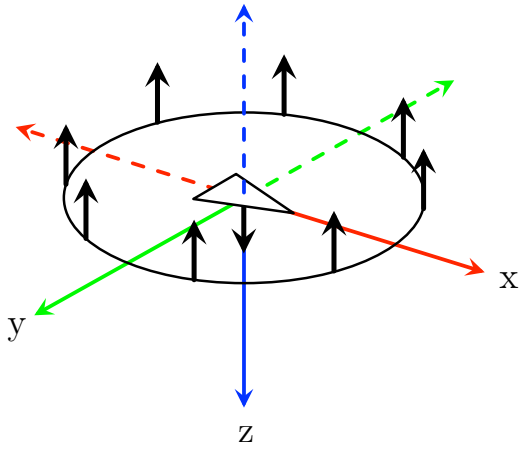
For our MAV and sensor configuration, we know the expected optical flow vectors for each camera given translation along the different body axes (assuming constant distance objects). This is shown in Figure 26. Note that the flow vectors are maximized when the camera direction is perpendicular to the direction of travel. Also note that vectors can be superimposed for motion along more than one axis.



(a) x-translation



(b) y-translation



(c) z-translation

Figure 26: Visualization of optic flow vectors for translations along each axis

5.3. Rotational Flow

The optic flow in a scene has two major, additive components: flow caused by translation ("translational optic flow" discussed in the previous section) and flow cause by rotation ("rotational optic flow"). When the observer is rotated, the entire scene experiences an additive flow opposite the direction of rotation (except along the axis of rotation). This phenomena is shown in Figure 27.

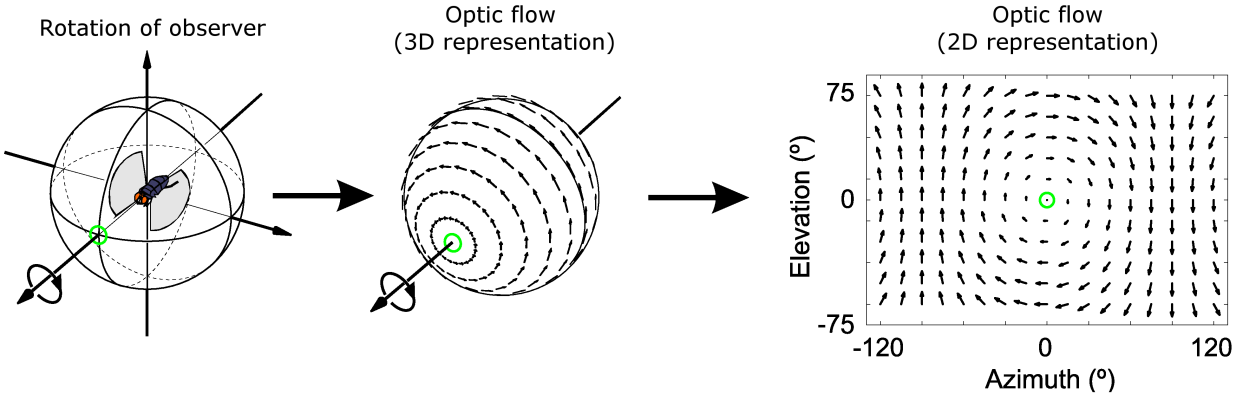
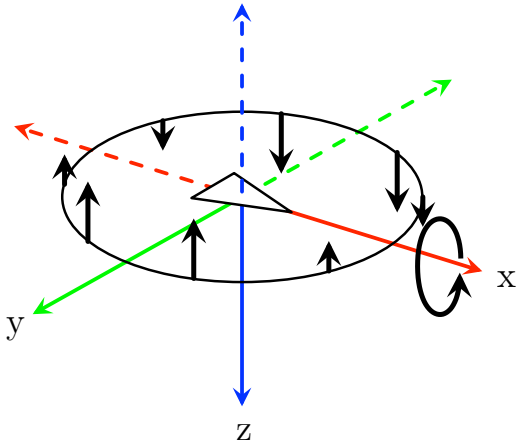
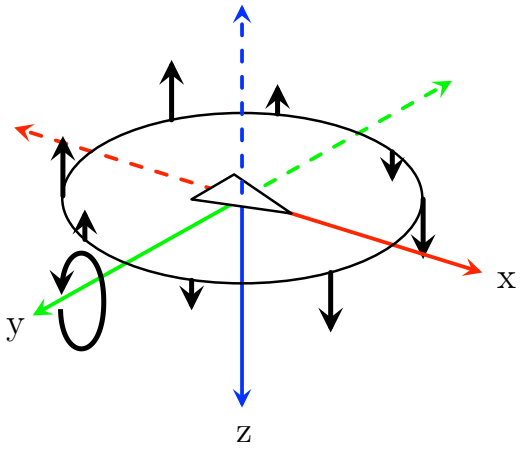


Figure 27: Optic flow experienced by rotating observer (adapted from [69])

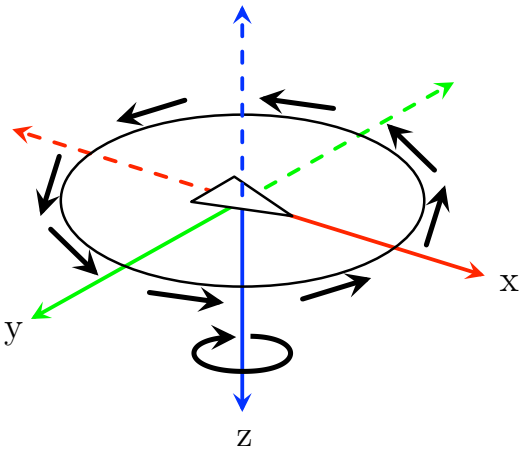
The magnitude of rotational optic flow is related to the rate of rotation (p, q, and r) and the angle to the axis of rotation (with the largest motion orthogonal to the axis of rotation). It does not, however, depend on the distance to the object or translational velocity. In this way, rotational optic flow does not give us any useful information about the scene. In most cases, we simply want to remove this component of our optic flow. Fortunately we have onboard inertial sensors which give us the angular rates for rotations about the x, y, and z-axis. With the optic flow ring on the MAV, rotations about the three body axes cause optic flow readings for each cameras as shown in Figure 28. We can calibrate our sensors and subtract out the components of the optic flow relating to these rotations. Figures 29, 30, and 31 show the results of rotating the simulated MAV along each axis of the body frame (using a single flow vector and 16×16 pixel images). These results match with our expectations for the relationship between angular rate and optic flow.



(a) x-rotation (roll)



(b) y-rotation (pitch)



(c) z-rotation (yaw)

Figure 28: Visualization of optic flow vectors for rotations about each axis

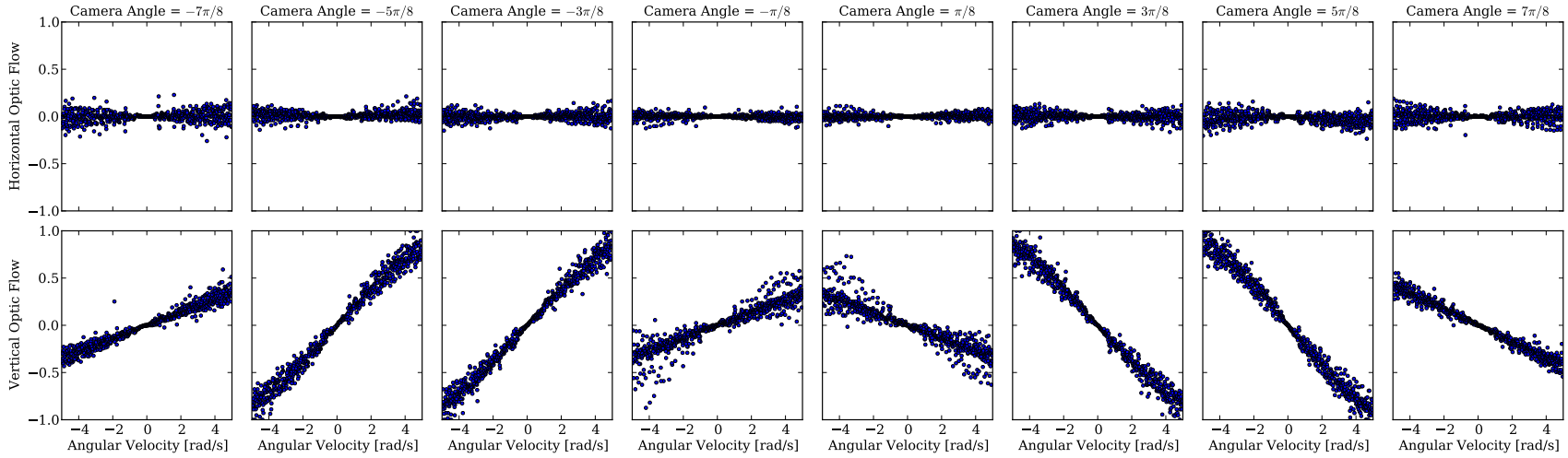


Figure 29: Roll-induced flow

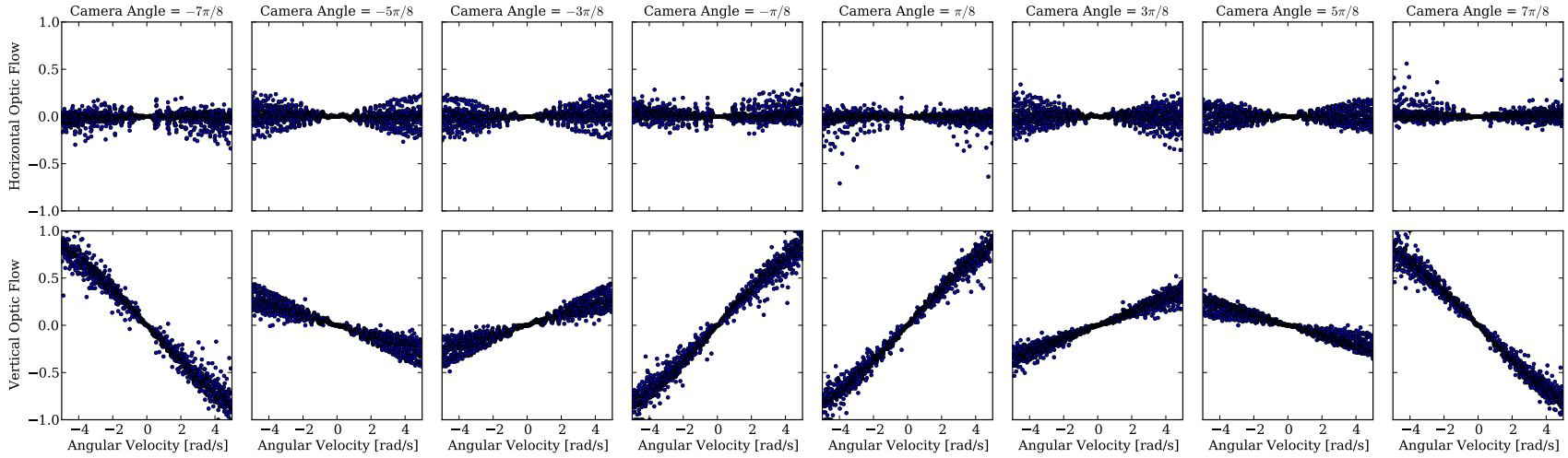


Figure 30: Pitch-induced flow

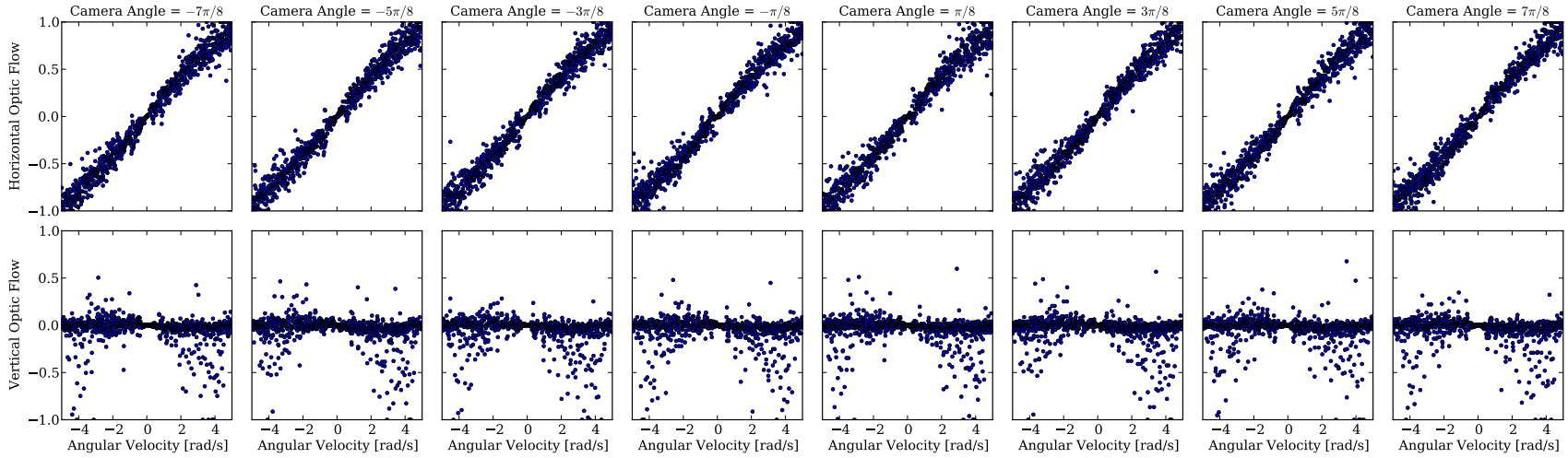


Figure 31: Yaw-induced flow

5.4. Rotation Compensation

In practice, we are most interested in the horizontal flow around MAV. Because of this, we can get away with simply compensating for the rotations about the yaw axis (simplifying our calibration process and implementation). Figure 32 shows experimental data collected with the helicopter showing both the yaw data from the gyroscope and the horizontal optic flow from one of the vision sensors. We are able to get a good match between the two readings, allowing use to de-rotate the flow data. With this adjustment, the flow that we read should be purely translational flow (in the x-direction, at least). Note that it is important for the two pieces of data to have the same timing. Otherwise the de-rotation of the flow data will simply add more noise into the system.

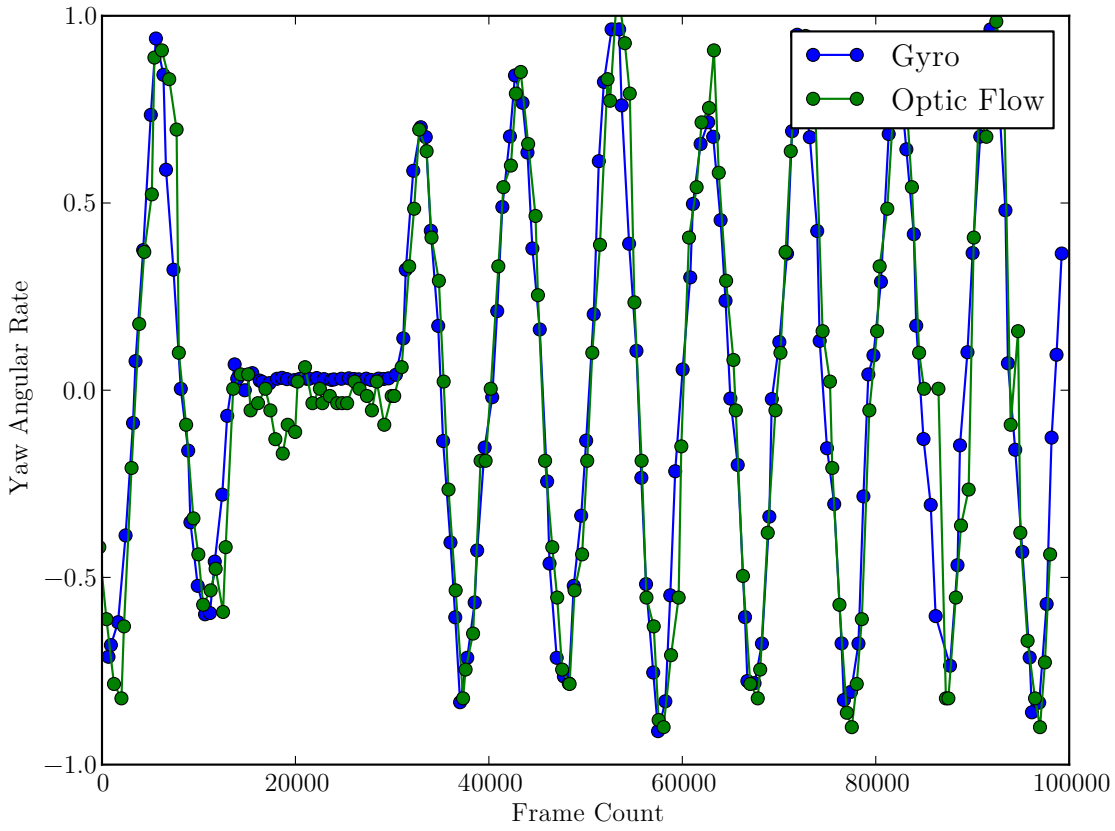


Figure 32: Match between gyroscope and optic flow readings

5.5. Filtering

Optic flow readings can very noisy, especially in areas of low texture. For this reason, it is often useful to filter our incoming optic flow data with a low-pass filter (LPF). In order to get a filtered optic flow estimate ($OF_{filtered}$) we can implement a simple, discrete-time low-pass filter (exponentially-weighted moving average) as shown in equation 33. Note that

we want to apply the filter to the data after any rotation compensation is applied.

$$OF_{filtered}[n] = (1 - \alpha) \cdot OF[n - 1] + \alpha \cdot OF[n] \tag{33}$$

where $0 \leq \alpha \leq 1$ is the smoothing factor, which is equivalent to an RC time constant of

$$RC = \Delta t \left(\frac{1 - \alpha}{\alpha} \right) \tag{34}$$

where Δt is the sampling period. Figure 33 shows the result of a low-pass filter on flow readings from the simulator using ($\alpha = 0.2$)

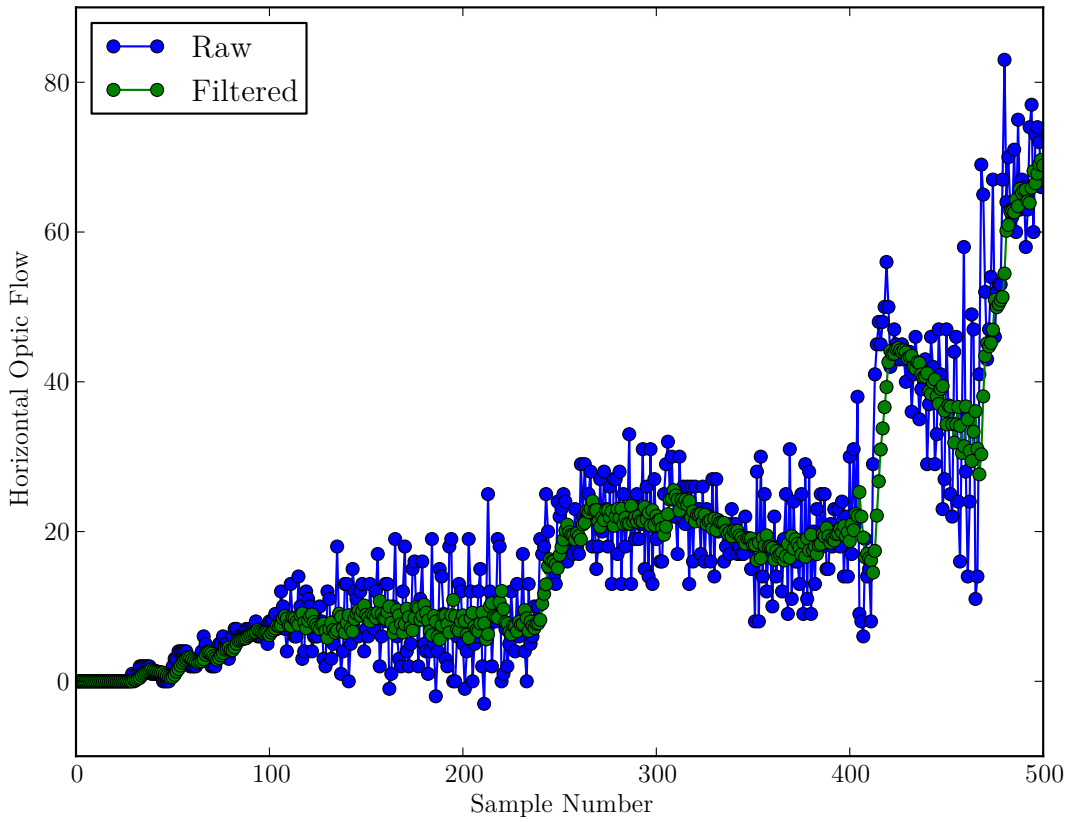


Figure 33: Low-pass filter applied to optic flow readings

6. Sensor Configurations

The current design of the sensor ring has numerous memory and bandwidth constraints. In particular, the ring can only read out approximately 4096 pixels per capture cycle. Given these limitations, there are numerous ways to read out and interpret the data. We considered a small subset of three possible sensor configurations, each with its own advantages and drawbacks. Figure 34 shows these three possibilities, represented with a constant pixel size. Note that the field of view does not actually change between the different configurations.

6.1. Configuration A

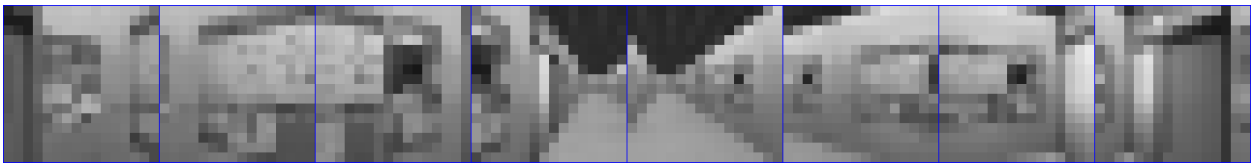


Figure 35: Sensor configuration A

The first configuration grabs 16×16 images from all 8 cameras to get a 360 degree view of the environment around the MAV. While this resolution seems very small, it is more than adequate to calculate the optic flow (especially when using the full 10-bit image depth).

Advantages

- Evenly distributes images around MAV
- Efficient binning of pixels for improved signal
- Can easily condense data into logical vectors
- Easy to implement

Disadvantages

- Low pixel count limits number of flow vectors that can be found with a given camera

Variations

Because there is significant overlap in the images (each image is almost fully reproduced by the neighboring cameras), we can lower the number of images and improve their resolutions. For example, if we only use images from the cameras at $\frac{-7\pi}{8}$, $\frac{-3\pi}{8}$, $\frac{\pi}{8}$, and $\frac{5\pi}{8}$, we will have nearly full coverage around the craft with a higher resolution for each of the images. This could be more useful when attempting other image processing techniques such as block matching which require sharper images. Note, however, it is often useful to have direct bilateral symmetry of the incoming information. Furthermore, there are times when the image overlap is necessary (when using stereo vision techniques, for example).

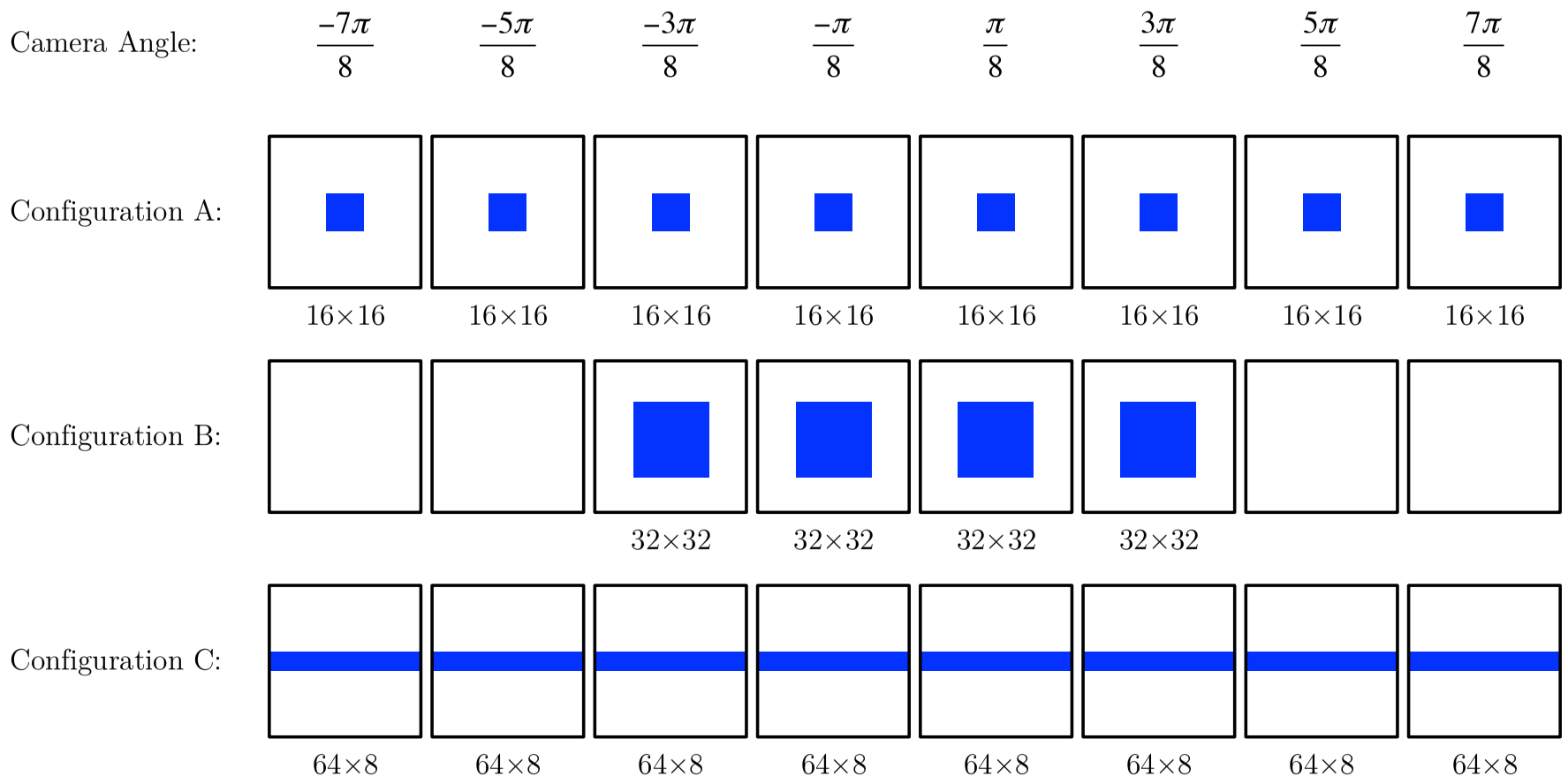


Figure 34: Sensor configurations (constant pixel size)

6.2. Configuration B

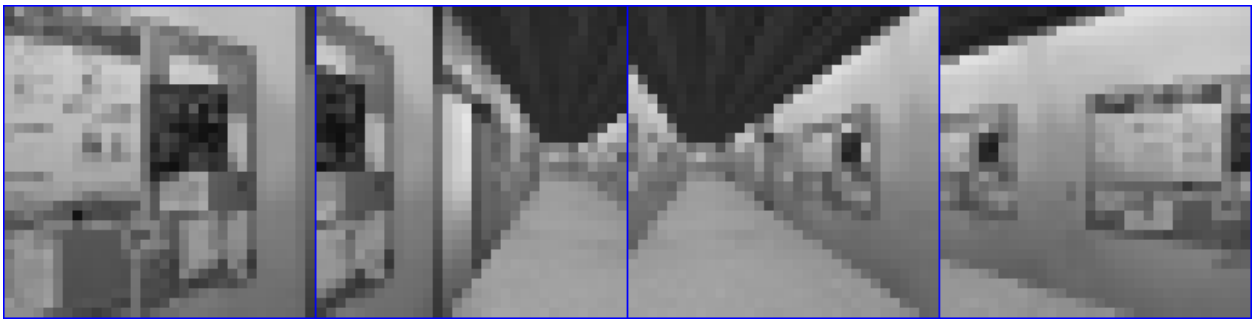


Figure 36: Sensor configuration B

If we chose a particular direction of travel, we can improve the resolution of the cameras facing that direction. In our case, we assume that we will primarily be traveling in the positive x direction of the helicopter body frame. We can then capture improved 32×32 pixel images from the four forward-facing cameras.

Advantages

- Better resolution when traveling "forward"
- Increased sensitivity in the optic flow estimate (due to higher resolution)
- Can support larger number of optic flow vectors per image
- Image overlap provides redundancy

Disadvantages

- No data from behind (cannot track obstacles you have passed)
- Overlapping images means we are not maximizing our total field of view (redundancy may be unnecessary)

6.3. Configuration C

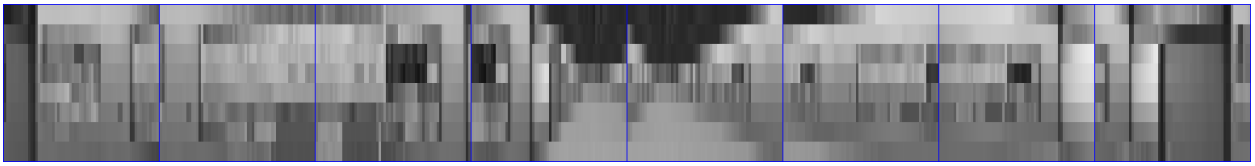


Figure 37: Sensor configuration C

For our experiments, we are more concerned with the horizontal flow than the vertical flow, as we expect much greater translational flow in the xy-plane than along the z-axis. It is

therefore interesting to maximize the horizontal resolution that we read from our cameras. We can do this by capturing 64×8 pixel images. If we implement binning of groups of 8 pixels in a column, we can make use of all of the pixels, giving us a strong signal while using the full horizontal resolution.

Advantages

- Maximizes sensitivity for horizontal flow
- Evenly distributes sensing around MAV (bilaterally symmetric)
- Data is not dependent on travel direction

Disadvantages

- Reduced vertical flow data
- More difficult to implement in hardware

6.4. Comparison

As a simple comparison between the different sensor configurations, we looked at the unfiltered optic flow from a single camera during a common motion sequence. We ran the identical simulation three times, once for each of the three configurations and used the images from camera 5 (see Figure 18) to calculate a single flow vector. The results of this experiment are shown in Figure 38. We note the different sensitivities of the three configurations. In the horizontal flow direction, configuration C is the most sensitive and configuration A is the least sensitive. This matches our expectations, given that the number of horizontal pixels doubles between configuration A and B and again between B and C. Likewise, we note a decreased sensitivity in the vertical direction for configuration C (which only has a height of 8 pixels to estimate flow). We also see the effect of a low-texture environment between seconds 2 and 3 when the camera is traveling along a relatively plain wall. This causes the optic flow estimate to become very noisy. We also note that the flow for configuration C reaches a maximum 4.5 seconds into the simulation. This is one of the issues we face with increased sensitivity (especially when there is a rotational optic flow component we must account for). One way of dealing with this is to dynamically adjust our offset in the I2A algorithm based on previous flow readings. While it is hard to say for certain which of the configurations is most promising, the increased sensitivity and 360° viewing range of configuration C make it particularly attractive for our application.

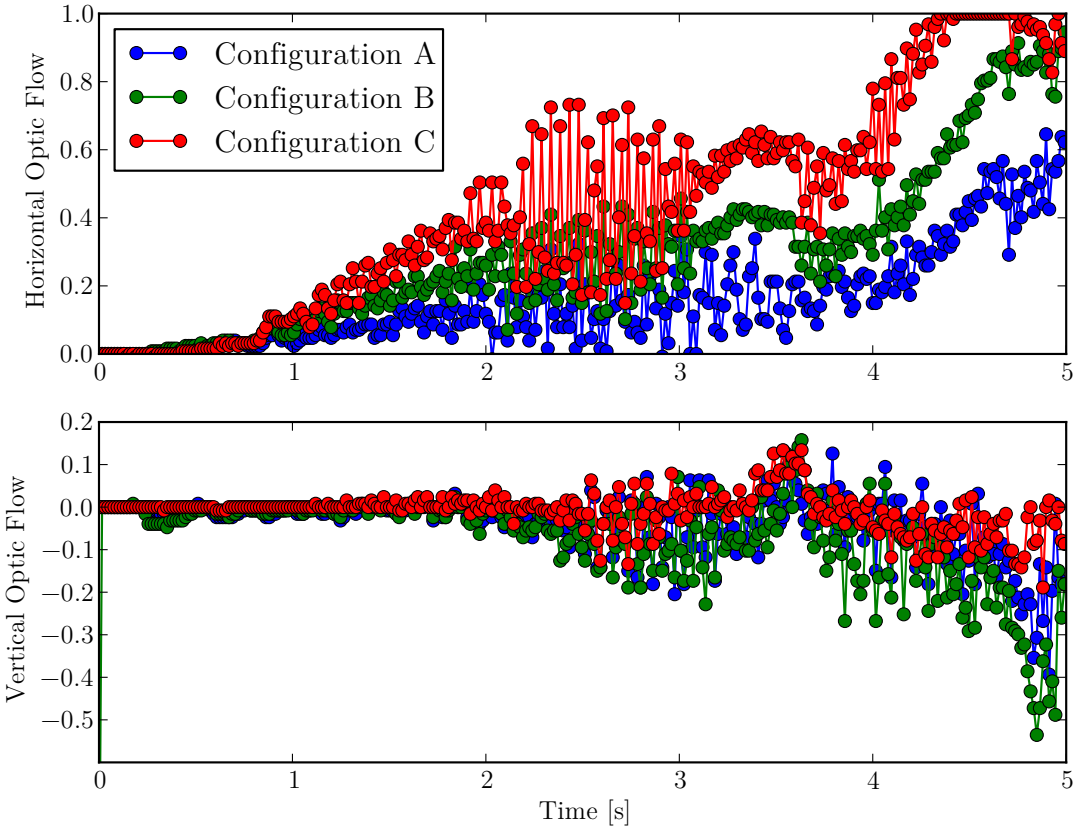


Figure 38: Sensor configuration comparison

7. Autonomous Tasks

We are interested in autonomous indoor flight using our MAV. For our initial testing, we have chosen to separate the control from the more-advanced navigation and mapping tasks. In this way, we can develop and maintain reliable flight, adding advanced capabilities on top of this. To begin, we have identified a set of simple autonomous behaviors to be used as elements of a more advanced, indoor navigational strategy. These behaviors include:

- Corridor following
- Wall following
- Obstacle avoidance
- Hover in place

Ultimately, we hope to use the visual sensing information to guide the controller to areas of interest or specific goals. In fact, these tasks can relatively easily be incorporated into an autonomous exploration or mapping algorithm (when combined with egomotion estimation). For our initial testing, we have focused on the task of corridor following.

7.1. Speed Regulation

We began by evaluating the open-loop control of our helicopter. We find that it is quite stable and will actively avoid excessive yaw or other drift. In order to test how accurately we can control our speed, we ran a series of open loop tests with a Vicon motion capture system. We used a calibrated configuration of 12 high-speed infrared cameras to track the location of reflective markers in 3-dimentional space. We then sent constant pitch commands to the helicopter and tracked its speed. The results can be seen in Figure 39. We ran a similar experiment in our simulation to see how closely the behaviors match. The results of this test are shown in Figure 40. This initial data suggest that, in the future, we should be able to get some estimate of speed simply based on the command being sent.

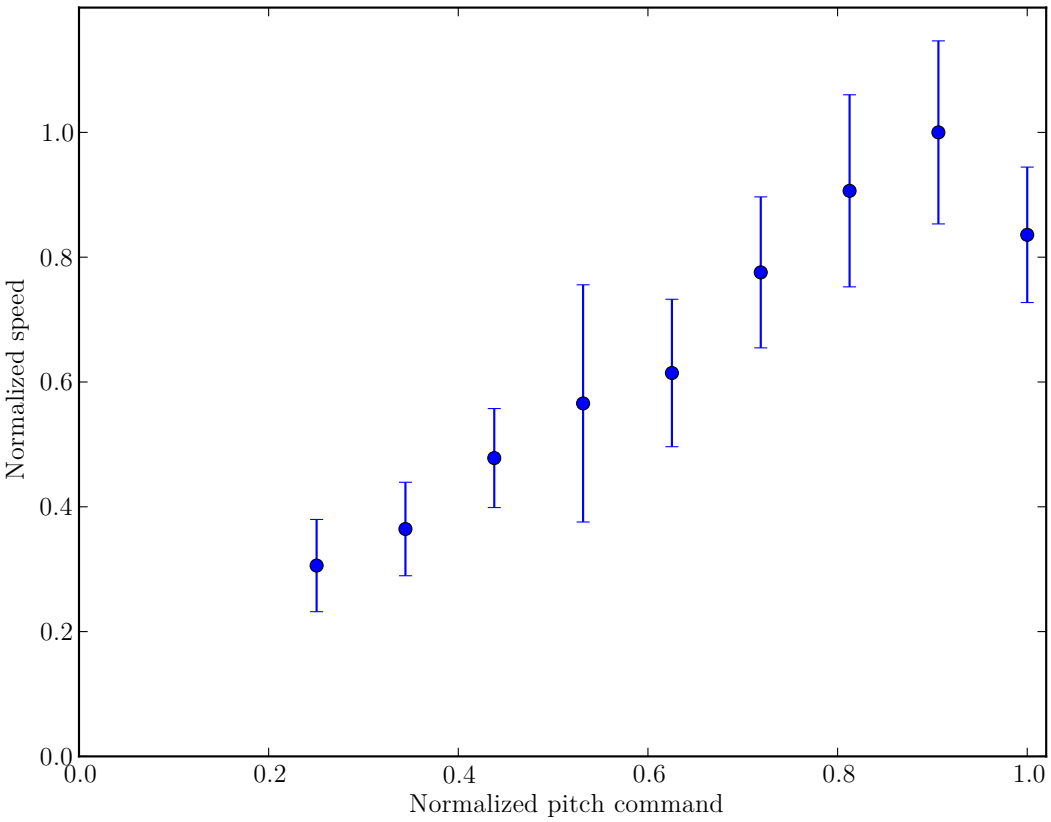


Figure 39: Experimental speed control

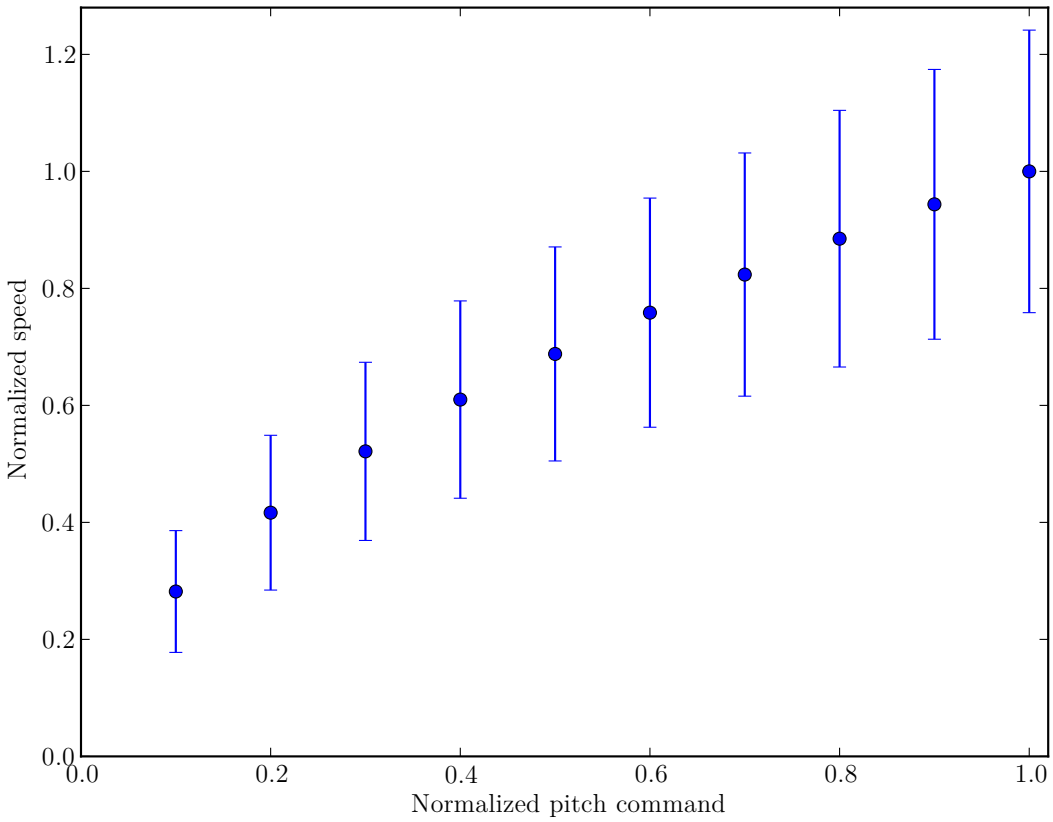


Figure 40: Simulated speed control

7.2. Controller

Our helicopter has four controllable values. These are the speed of rotation of the two rotors (Ω_u and Ω_l) and the tilt angles of the lower thrust vector (α and β) as controlled by two servo motors. We simplify these commands into the four controls listed in Table 4.

δ_t	Thrust control
δ_y	Yaw control
δ_p	Pitch control
δ_r	Roll control

Table 4: Controls used for flight

Thrust entails the common speed of both the upper and lower rotor (which alone will not cause any rotation). The yaw command creates an imbalance between the two rotor commands, creating a torque. The pitch command adjusts force along the positive x direction, controlling speed. Finally, the roll control adjusts the lateral force on the aircraft.

We have begun by testing a very simple, reactive controller that simply takes the optic flow readings from around the ring (OF_i) and combines them through a simple weighted sum (as proposed in [21]). These weights can optionally be adjusted during flight to change the behavior. The basic structure is shown in Figure 41. This same structure, but with different weights can be used to control the various commands. This is expressed in the following equation:

$$\delta_j = \sum_{i=0}^{7} w_{i,j} \cdot OF_i \tag{35}$$

where j identifies the particular command, $w_{i,j}$ are the weights for the command, and OF_i is the optic flow measurement.

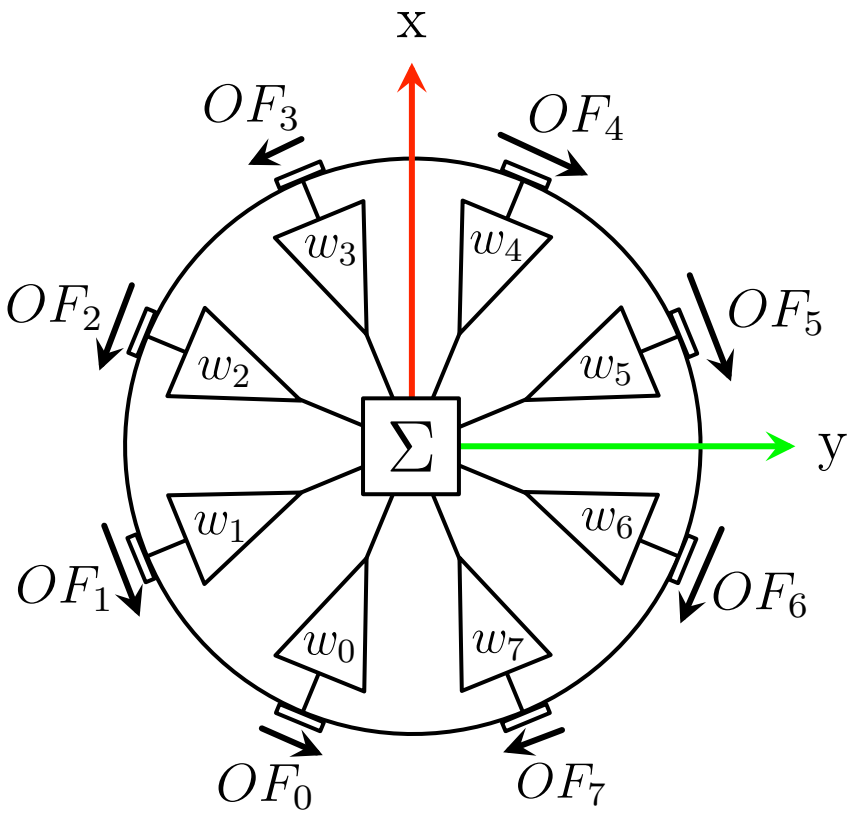


Figure 41: Reactive controller structure

7.3. Corridor Following

As an example behavior, we have started work toward creating a controller for flying through indoor corridors. Much of this work is still in progress (with a large portion of the work going toward the hardware itself). In simulation, we have used the control structure described above to control the roll of the MAV in the simulated environment from Figure 20. We use the idea of optic flow balance when choosing our weights so that the robot will naturally tend to find the center of the corridor. In our tests, the robot begins near a wall with a

constant pitch command and is able to center itself in the hallway (see Figure 42). We have also implemented a similar controller over the yaw in simulation. We are working on implementing this behavior in hardware and hope to have results in the very near future.

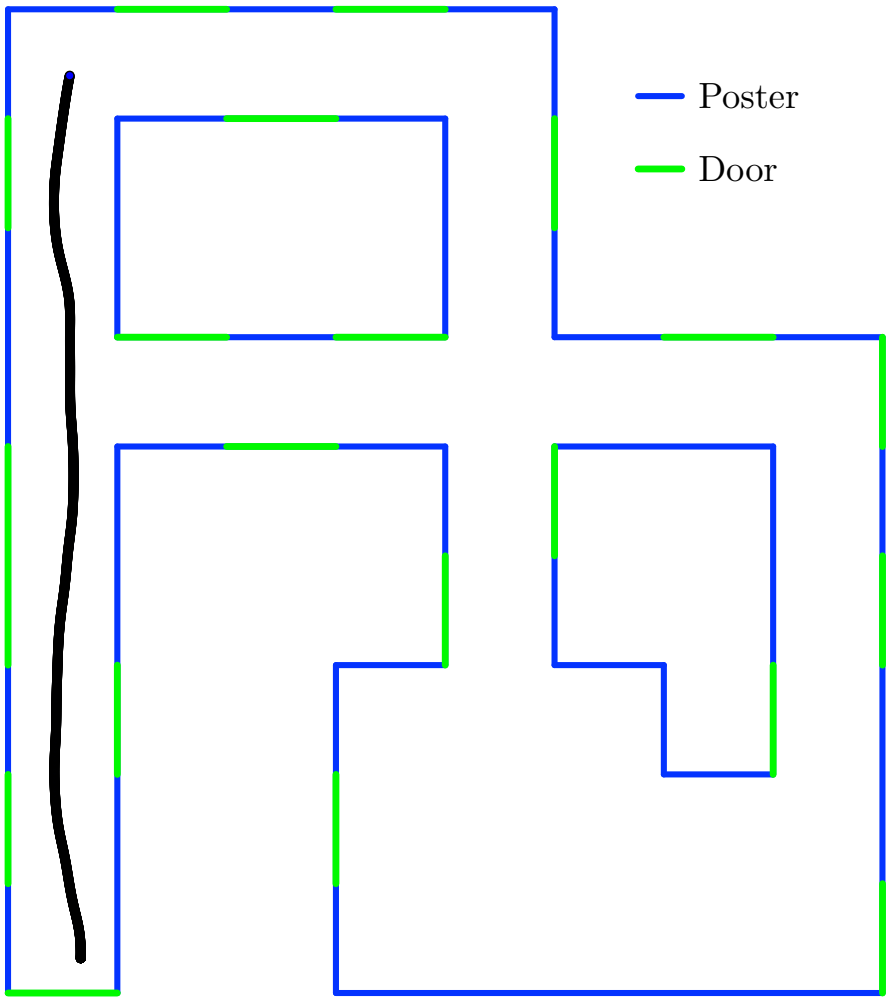


Figure 42: Corridor following flight path

Note that if we simply adjust our speed to keep a constant average flow around the vehicle, we are able to easily implement an intelligent speed regulation. As the corridor gets smaller, the distance on both sides is reduced, causing optic flow values to increase and the controller to reduce the speed.

7.4. Other Behaviors

As mentioned before, we are also looking into ways of implementing other behaviors as well. Wall following is particularly attractive because we always remain near a surface on which we can (in theory) measure the optic flow. The control itself is also rather logical: if we maintain

a constant speed, we can follow the wall by adjusting our direction to keep a constant optical flow (since velocity, flow, and distance are linked).

We are also looking to include a mechanism in our controller to actively avoid obstacles. If the robot determines that it is too close to an obstacle (based on the optic flow), it will aggressively try to avoid collisions. This will likely be implemented much like the saccade of a flying insect.

Because of the layout of our sensors, we should be able to implement a completely vision based hover in place controller as described in [15]. This is useful for keeping the helicopters stationary for tasks like tracking or extended communication.

Using these an other behaviors we hope to implement robust autonomous exploration algorithm, which when combined with egomotion estimation can be used to create maps of the environment. Ultimately, we would like to use several of the platforms simultaneously to autonomously create a map of the environment in a distributed manner.

8. Conclusion and Future Work

In this project, we investigated the autonomous control and navigation of a 30 gram coaxial helicopter platform. We use a custom sensor ring with 8 specialized vision sensors for the calculation of optic flow around the MAV. In addition to programming the hardware itself, we developed a 3-dimensional simulation of the robot in order to more quickly, easily, and reliably test new behaviors and sensor configurations. We then used this platform to began work exploring possible control strategies for the aircraft. This project contributes to the ongoing research in the lab on sensing in control in extremely resource-scarce environments and simple structures for distributed robot systems. We have create a useful tool for the lab, helping with future research in vision-based navigation and control strategies for autonomous MAVs.

There is still a lot of work to be done to create a reliable system of distributed MAVs for indoor environments. In fact, much of the work done during my master project is not actually included in this report. We spent many weeks working on getting the hardware up and running. Unfortunately, we faced several setbacks, causing us to look toward doing some of these tasks in simulation. Moving forward, we would like perform more tests in simulation and port those ideas onto the hardware itself. We hope to accomplish some of these tasks in the very near future.

Acknowledgements

First of all, I would like to thank professor Radhika Nagpal for allowing me the wonderful opportunity to work with her and the Self-Organizing Systems Research Group (SSR) at Harvard University. I want to thank the entire SSR lab for being so welcoming and supportive throughout my stay. I am especially grateful to Karthik Dantu and Richard Moore for being wonderful partners to work with. I owe them immensely for their guidance and ideas throughout this project. I learned a lot and enjoyed our joint struggles to get various pieces of hardware and software working. I would also like to thank Maja Varga and professor Dario Floreano from the Laboratory of Intelligent Systems (LIS) at the Swiss Federal Institute of Technology (EPFL) for supporting my work. Finally, I want to thank my friends and family for their unwavering support throughout all my studies. I would not be where I am without you.

References

[1] Abraham Galton Bachrach. *Autonomous Flight in Unstructured and Unknown Indoor Environments*. PhD thesis, 2009.

[2] Markus Achtelik, Abraham Bachrach, Ruijie He, Samuel Prentice, and Nicholas Roy. Stereo Vision and Laser Odometry for Autonomous Helicopters in GPS-denied Indoor Environments. In *Proceedings of SPIE*, volume 1. SPIE, 2009.

[3] Spencer Ahrens, Daniel Levine, Gregory Andrews, and Jonathan P. How. Vision-Based Guidance and Control of a Hovering Vehicle in Unknown, GPS-denied Environments. In *2009 IEEE International Conference on Robotics and Automation*, pages 2643–2648. IEEE, May 2009.

[4] Nicola Ancona and Tomaso Poggio. Optical Flow from 1D Correlation: Application to a simple Time-To-Crash Detector. pages 209–214, 1993.

[5] Bogdan Arama, Sasan Barissi, and Nasser Houshangi. Control of an Unmanned Coaxial Helicopter Using Hybrid Fuzzy-PID Controllers. In *24th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1064–1068, 2011.

[6] Abraham Bachrach, Ruijie He, and Nicholas Roy. Autonomous Flight in Unknown Indoor Environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, December 2009.

[7] Abraham Bachrach, Samuel Prentice, Ruijie He, Peter Henry, Albert S. Huang, Michael Krainin, Daniel Maturana, Dieter Fox, and Nicholas Roy. Estimation, Planning, and Mapping for Autonomous Flight Using an RGB-D Camera in GPS-denied Environments. *The International Journal of Robotics Research*, 31(11):1320–1343, September 2012.

[8] Abraham Bachrach, Samuel Prentice, Ruijie He, and Nicholas Roy. RANGE - Robust Autonomous Navigation in GPS-denied Environments. *Journal of Field Robotics*, 25(5):646–666, 2011.

[9] Albert-Jan Baerveldt and Robert Klang. A Low-cost and Low-weight Attitude Estimation System for an Autonomous Helicopter. In *IEEE International Conference on Intelligent Engineering Systems*, pages 391–395, 1997.

[10] Emily Baird, Mandyam V. Srinivasan, Shaowu Zhang, Richard Lamont, and Ann Cowling. Visual Control of Flight Speed and Height in the Honeybee. In *From Animals to Animats: 9th International Conference on Simulation of Adaptive Behavior*, pages 40–51, 2006.

[11] Simon Baker and Iain Matthews. Lucas-Kanade 20 Years On: A Unifying Framework (Parts 1-5), 2004.

[12] Peter Barnum, Bo Hu, and Christopher Brown. Exploring the Practical Limits of Optical Flow, 2003.

[13] A. Barrientos, J. Colorado, A. Martinez, and Joao Valente. Rotary-wing MAV Modeling & Control for indoor scenarios. In *2010 IEEE International Conference on Industrial Technology*, pages 1475–1480. IEEE, 2010.

[14] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of Optical Flow Techniques. *International Journal of Comupter Vision*, 12(1):43–77, 1994.

[15] Geoffrey Louis Barrows, James Sean Humbert, Alison Leonard, Craig William Neely, and Travis Michael Young. Vision Based Hover in Place, 2012.

[16] Geoffrey Louis Barrows and Craig William Neely. Translational Optical Flow Sensor, 2012.

[17] Christian Bermes. *Design and dynamic modeling of autonomous coaxial micro helicopters*. PhD thesis, ETH Zürich, 2010.

[18] Fernando Garcia Bermudez and Ronald Fearing. Optical flow on a flapping wing robot. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5027–5032. IEEE, October 2009.

[19] Antoine Beyeler. *Vision-Based Control of Near-Obstacle Flight*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2009.

[20] Antoine Beyeler and Jean-christophe Zufferey Dario. Vision-based control of near-obstacle flight. *Autonomous Robots*, 27(3):201–219, 2009.

[21] Antoine Beyeler and Dario Floreano. optiPilot : control of take-off and landing using optic flow. In *European Micro Air Vehicle Conference and Competition 2009 (EMAV 2009)*, 2009.

[22] Antoine Beyeler, Claudio Mattiussi, Jean-Christophe Zufferey, and Dario Floreano. Vision-based Altitude and Pitch Estimation for Ultra-light Indoor Microflyers. *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, (May):2836–2841, 2006.

[23] Antoine Beyeler, Jean-christophe Zufferey, and Dario Floreano. 3D Vision-based Navigation for Indoor Microflyers. In *2007 IEEE International Conference on Robotics and Automation*, number April, pages 1336–1341, 2007.

[24] Michael Julian Black. *Robust Incremental Optical Flow*. PhD thesis, Yale Yniversity, 1992.

[25] Michael Blosch, Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Vision Based MAV Navigation in Unknown and Unstructured Environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 21–28, 2010.

[26] Edgard Font Calafell. *CoaX Flight Simulator in Webots*. Master's thesis, ETHZ, 2011.

[27] Michael Camilleri, Kenneth Scerri, and Saviour Zammit. Autonomous Flight Control for an RC Helicopter. In *2012 16th IEEE Mediterranean Electrotechnical Conference*, pages 391–394. IEEE, March 2012.

[28] Victor H L Cheng and Moffett Field. Automatic Guidance and Control Laws for Helicopter Obstacle Avoidance. In *Internation Conference on Robotics and Automation*, pages 252–260, 1992.

[29] Marco La Civita. *Integrated Modeling and Robust Control for Full-Envelope Flight of Robotic Helicopters*. PhD thesis, 2002.

[30] T. S. Collett. Insect navigation en route to the goal: multiple strategies for the use of landmarks. *The Journal of Experimental Biology*, 199:227–235, January 1996.

[31] Shane Colton. The Balance Filter: A Simple Solution for Integrating Accelerometer and Gyroscope Measurements for a Balancing Platform, 2007.

[32] David Coombs, Martin Herman, Tsai Hong, and Marilyn Nashman. Real-time obstacle avoidance using central flow divergence and peripheral flow. In *Proceedings of IEEE International Conference on Computer Vision*, pages 276–283. IEEE Comput. Soc. Press, 1995.

[33] Peter Corke. An Inertial and Visual Sensing System for a Small Autonomous Helicopter. *Journal of Robotic Systems*, 21(2):43–51, February 2004.

[34] Anthony Cowley, Camillo J. Taylor, and Ben Southall. Rapid multi-robot exploration with topometric maps. In *2011 IEEE International Conference on Robotics and Automation*, pages 1044–1049. IEEE, May 2011.

[35] Cyberbotics. Webots Reference Manual, 2012.

[36] Cyberbotics. Webots User Guide, 2012.

[37] Hans-Jurgen Dahmen, Mattihas O. Franz, and Holger G. Krapp. Extracting Egomotion from Optic Flow : Limits of Accuracy and Neural Matched Filters. In *Motion Vision - Computational, Neural, and Ecological Constraints*, chapter 3, pages 144–167. 2001.

[38] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. Programming Micro-Aerial Vehicle Swarms with Karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems - SenSys '11*, page 121, New York, New York, USA, 2011. ACM Press.

[39] Anuj Dev, Ben Krose, and Frans Groen. Navigation of a mobile robot on the temporal development of the optic flow. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems, 1997. IROS '97.*, pages 558–563, 1997.

[40] Sotirios Ch. Diamantas, Anastasios Oikonomidis, and Richard M. Crowder. Depth estimation for autonomous robot navigation: A comparative approach. In *2010 IEEE International Conference on Imaging Systems and Techniques*, pages 426–430. IEEE, July 2010.

[41] Michal Karol Dobrzynski, Ramon Pericet-Camara, and Dario Floreano. Vision Tape – A Flexible Compound Vision Sensor for Motion Detection and Proximity Estimation. *IEE Sensors Journal*, 12(5):1131–1139, 2012.

[42] Fred C. Dyer and James L. Could. Honey Bee Navigation: The honey bee's ability to find its way depends on a hierarchy of sophisticated orienation mechanisms. *American Scientist*, 71(6):587–597, 1983.

[43] A. El Hadri and A. Benallegue. Attitude estimation with gyros-bias compensation using low-cost sensors. In *48th IEEE Conference on Decision and Control (CDC) and 28th Chinese Control Conference*, pages 8077–8082. IEEE, December 2009.

[44] Russell Enns and Jennie Si. Helicopter Flight Control Design Using a Learning Control Approach. In *39th IEEE Conference on Decision Control*, pages 1754–1759, 2000.

[45] Mark Euston, Paul Coote, Robert Mahony, Jonghyuk Kim, and Tarek Hamel. A Complementary Filter for Attitude Estimation of a Fixed-Wing UAV. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 340–345. IEEE, September 2008.

[46] A. Fabrice and F. Nicolas. Optic flow sensors for robots: elementary motion detectors based on FPGA. In *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, pages 182–187. IEEE, 2005.

[47] Benjamin M. Finio and Robert J. Wood. Open-loop roll, pitch and yaw torques for a robotic bee. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 113–119. IEEE, October 2012.

[48] Benjamin Michael Finio. *Roll , Pitch and Yaw Torque Control for a Robotic Bee*. PhD thesis, Harvard University, 2012.

[49] N. Franceschini, J. M. Pichon, and C. Blanes. From insect vision to robot vision. *Philosophical Transactions: Biological Sciences*, 337(1281):283–294, 1992.

[50] Nicolas Franceschini. Visual guidance based on optic flow: a biorobotic approach. *Journal of Physiology, Paris*, 98(1-3):281–92, 2004.

[51] Peter Frankhauser, Samir Bouabdallah, Stefan Leutenegger, and Roland Siegwart. Modeling and Decoupling Control of the CoaX Micro Helicopter. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2223–2228. IEEE, September 2011.

[52] Matthias O. Franz and Hanspeter a. Mallot. Biomimetic robot navigation. *Robotics and Autonomous Systems*, 30(1-2):133–153, January 2000.

[53] Sawyer B. Fuller and Richard M. Murray. A hovercraft robot that uses insect-inspired visual autocorrelation for motion control in a corridor. In *2011 IEEE International Conference on Robotics and Biomimetics*, pages 1474–1481. IEEE, December 2011.

[54] M. A. Garratt, A. J. Lambert, and T. Guillemette. FPGA Implementation of an Optic Flow Sensor using the Image Interpolation Algorithm. In *Australasian Conference on Robotics and Automation*, 2009.

[55] A. Gonzalez, R. Mahtani, M. Bejar, and A. Ollero. Control and Stability Analysis of an Autonomous Helicopter. In *World Automation Congress*, pages 399–404, 2004.

[56] Chauncey F. Graetzel, Vasco Medici, Nicola Rohrseitz, Bradley J. Nelson, and Steven N. Fry. The Cyborg Fly: A biorobotic platform to investigate dynamic coupling effects between a fruit fly and a robot. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 14–19. IEEE, September 2008.

[57] William E. Green, Paul Y. Oh, and Geoffrey Barrows. Flying Insect Inspired Vision for Autonomous Aerial Robot Maneuvers in Near-Earth Environments. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, pages 2347–2352. IEEE, 2004.

[58] Giorgio Grisetti. Towards a Navigation System for Autonomous Indoor Flying. In *IEEE International Conference on Robotics and Automation*, number Section III, 2009.

[59] Verena V. Hafner, Ferry Bachmann, Oswald Berthold, Michael Schulz, and Mathias Muller. An autonomous flying robot for testing bio-inspired navigation strategies. In *2010 41st International Symposium on Robotics (ISR)*, pages 1145–1151, 2010.

[60] Tarek Hamel and Robert Mahony. Attitude estimation on SO(3) based on direct inertial measurements. In *IEEE International Conference on Robotics and Automation*, number May, pages 2170–2175. IEEE, 2006.

[61] Robert K. Heffley. Minimum-Complexity Helicopter Simulation Math Model, 1988.

[62] Bruno Herisse, Francois-Xavier Russotto, Tarek Hamel, and Robert Mahony. Hovering flight and vertical landing control of a VTOL Unmanned Aerial Vehicle using Optical Flow. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 801–806. IEEE, September 2008.

[63] Walter Higgins. A Comparison of Complementary and Kalman Filtering. *IEEE Transactions on Aerospace and Electronic Systems*, AES-11(3):321–325, May 1975.

[64] Jonathan How. Aircraft Stability and Control, 2004.

[65] Stefan Hrabar and Gaurav S. Sukhatme. Optimum Camera Angle for Optic Flow-Based Centering Response. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3922–3927. IEEE, October 2006.

[66] Stefan Ellis de Nagy Köves Hrabar. *Vision-Based 3D Navigation for an Autonomous Helicopter*. PhD thesis, University of Southern California, 2006.

[67] Albert S. Huang, Abraham Bachrach, Peter Henry, Michael Krainin, Dieter Fox, and Nicholas Roy. Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera. In *Proceedings of the International Symposium of Robotics Research (ISRR)*, pages 1–16, 2011.

[68] Christoph Hürzeler. Modeling of Coaxial Helicopters, 2011.

[69] Stephen J. Huston and Holger G. Krapp. Visuomotor transformation in the fly gaze stabilization system. *PLoS biology*, 6(7):e173, July 2008.

[70] Myungsoo Jun, Stergios I. Roumeliotis, and Gaurav S. Sukhatme. State Estimation of an Autonomous Helicopter Using Kalman Filtering. In *IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289)*, volume 3, pages 1346–1353. IEEE, 1999.

[71] Bryan Kate, Jason Waterman, Karthik Dantu, and Matt Welsh. Simbeeotic : A Simulator and Testbed for Micro-Aerial Vehicle Swarm Experiments. In *IPSN*, 2012.

[72] Farid Kendoul, Isabelle Fantoni, and Gerald Dherbomez. Three Nested Kalman Filters-Based Algorithm for Real-Time Estimation of Optical Flow , UAV Motion and Obstacles Detection. In *2007 IEEE International Conference on Robotics and Automation*, number April, pages 10–14, 2007.

[73] S. K. Kim and D. M. Tilbury. Mathematical Modeling and Experimental Identification of a Model Helicopter. *AIAA Journal of Guidance, Control, and Dynamics*, pages 1–34, 2000.

[74] W. H. Kirchner and M. V. Srinivasan. Freely Flying Honeybees Use Image Motion to Estimate Object Distance. *Naturwissenschaften*, 76:281–282, 1989.

[75] Jan J. Koenderink. Optic flow. *Vision Research*, 26(1):161–179, 1986.

[76] Jörg Kramer, Rahul Sarpeshkar, and Christof Koch. An Analog VLSI Velocity Sensor. pages 0–3, 1995.

[77] Nikolaos Kyriakoulis, Evangelos Karakasis, Antonios Gasteratos, and Angelos Amanatiadis. Pose Estimation of a Volant Platform with a Monocular Visuo-Inertial System. *2009 IEEE International Workshop on Imaging Systems and Techniques*, pages 421–426, May 2009.

[78] M Anthony Lewis. Visual Navigation in a Robot using Zig-Zag Behavior. *Neural Information Processing Systems*, 1998.

[79] M. Anthony Lewis. Detecting surface features during locomotion using optic flow. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 1, pages 305–310. IEEE, 2002.

[80] Charles Lidstone. *The Gimballed Helicopter Testbed : Design , Build and Validation*. Master's thesis, University of Toronto, 2003.

[81] Mathieu Lihoreau, Nigel E. Raine, Andrew M. Reynolds, Ralph J. Stelzer, Ka S. Lim, Alan D. Smith, Juliet L. Osborne, and Lars Chittka. Radar Tracking and Motion-Sensitive Cameras on Flowers Reveal the Development of Pollinator Multi-Destination Routes over Large Spatial Scales. *PLoS Biology*, 10(9):e1001392, September 2012.

[82] Hyon Lim, Jaemann Park, Daewon Lee, and H. J. Kim. Open-Source Projects on Unmanned Aerial Vehicles: Build Your Own Quadrotor. *IEEE Robotics & Automation Magazine*, (SEPTEMBER 2012):33–45, 2012.

[83] Vincenzo Lippiello, Giuseppe Loianno, and Bruno Siciliano. MAV Indoor Navigation Based on a Closed-Form Solution for Absolute Scale Velocity Estimation using Optical Flow and Inertial Data. In *IEEE Conference on Decision and Control and European Control Conference*, pages 3566–3571. IEEE, December 2011.

[84] Jundong Liu. Lucas-Kanade method. 2012.

[85] Jundong Liu. Optical flow. 2012.

[86] Bruce D. Lucas. *Generalized Image Matching by the Method of Differences*. PhD thesis, Carnegie-Mellon University, 1984.

[87] Bruce D. Lucas and Takeo Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. In *Proceedings of Imaging Understanding Workshop*, volume 130, pages 121–130, 1981.

[88] Will Maddern, Michael Milford, and Gordon Wyeth. Towards Persistent Indoor Appearance-based Localization, Mapping and Navigation using CAT-Graph. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4224–4230, 2012.

[89] Robert Mahony, Felix Schill, Peter Corke, and Yoong Siang Oh. A New Framework for Force Feedback Teleoperation of Robotic Vehicles based on Optical Flow. In *2009 IEEE International Conference on Robotics and Automation*, pages 1079–1085. IEEE, May 2009.

[90] Robert Mahony, Felix Schill, Peter Corke, and Yoong Siang Oh. A new framework for force feedback teleoperation of robotic vehicles based on optical flow. In *2009 IEEE International Conference on Robotics and Automation*, pages 1079–1085. IEEE, May 2009.

[91] Ken Masuya, Tomomichi Sugihara, and Motoji Yamamoto. Design of Complementary Filter for High-fidelity Attitude Estimation based on Sensor Dynamics Compensation with Decoupled Properties. In *IEEE International Conference on Robotics and Automation*, pages 606–611. IEEE, May 2012.

[92] Owen McAree, Jonathan Clarke, and Wen-Hua Chen. Development of an Autonomous Control System for a Small Fixed Pitch Helicopter. In *2nd International Conference on Advanced Computer Control*, pages 13–17. IEEE, 2010.

[93] Randolf Menzel, Robert Brandt, Andreas Gumbert, Bernhard Komischke, and Jan Kunze. Two spatial memories for honeybee navigation. *Proceedings. Biological Sciences / The Royal Society*, 267(1447):961–8, May 2000.

[94] Randolf Menzel, Karl Geiger, Lars Chittka, Jasdan Joerges, Jan Kunze, and Uli Muller. The knowledge base of bee navigation. *The Journal of Experimental Biology*, 199:141–146, January 1996.

[95] Randolf Menzel, Karl Geiger, Jasdan Joerges, Uli Muller, and Lars Chittka. Bees travel novel homeward routes by integrating separately acquired vector memories. *Animal behaviour*, 55:139–152, January 1998.

[96] Michael J. Milford, Felix Schill, Peter Corke, Robert Mahony, and Gordon Wyeth. Aerial SLAM with a Single Camera Using Visual Expectation. In *IEEE International Conference on Robotics and Automation*, pages 2506–2512, 2011.

[97] Michael J Milford and Gordon F Wyeth. Mapping a Suburb With a Single Camera Using a Biologically Inspired SLAM System. *IEEE Transactions on Robotics*, 24(5):1038–1053, 2008.

[98] Byoung-Mun Min, Il-Hyung Lee, Tae-Won Hwang, and Jin-Sung Hong. Unmanned Autonomous Helicopter System Design and Its Flight Test. In *2007 International Conference on Control, Automation and Systems*, pages 2090–2095. IEEE, 2007.

[99] Richard Moore, Saul Thurrowgood, Daniel Bland, Dean Soccol, and Mandyam V. Srinivasan. UAV Altitude and Attitude Stabilisation using a Coaxial Stereo Vision System. In *2010 IEEE International Conference on Robotics and Automation*, pages 29–34. IEEE, May 2010.

[100] Richard J. D. Moore, Saul Thurrowgood, Daniel Bland, Dean Soccol, and Mandyam V. Srinivasan. A Stereo Vision System for UAV Guidance. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3386–3391. IEEE, October 2009.

[101] Laurent Muratet, Stephane Doncieux, Yves Briere, and Jean-Arcady Meyer. A Contribution to Vision-Based Autonomous Helicopter Flight in Urban Environments. *Robotics and Autonomous Systems*, 50:195–209, 2005.

[102] Titus R Neumann and Heinrich H Bülthoff. Behavior-Oriented Vision for Biomimetic Flight Control. In *EPSRC/BBSRC International Workshop on Biologically Inspired Robotics*, volume 203, pages 196–203, 2002.

[103] Jean-D. Nicoud and Zufferey Jean-C. Toward Indoor Flying. In *2002 IEEE/RSJ International Conference on Intelligent Robots and Systems*, number October, pages 787–792, 2002.

[104] Kenzo Nonami, Farid Kendoul, Satoshi Suzuki, Wei Wang, and Daisuke Nakazawa. Fundamental Modeling and Control of Small and Miniature Unmanned Helicopters. In *Autonomous Flying Robots*, chapter 2, pages 33–61. Springer Japan, Tokyo, 2010.

[105] Edwin Olson, Johannes Strom, Ryan Morton, Andrew Richardson, Pradeep Ranganathan, Robert Goeddel, Mihai Bulic, Jacob Crossman, and Bob Marinier. Progress Toward Multi-Robot Reconnaissance and the MAGIC 2010 Competition. *Journal of Field Robotics*, 29(5):762–792, 2012.

[106] Swee King Phang, Jun Jie Ong, Ronald T. C. Yeo, Ben M. Chen, and Tong H. Lee. Autonomous Mini-UAV for Indoor Flight with Embedded On-board Vision Processing as Navigation System. In *2010 IEEE Region 8 International Conference on Computational Technologies in Electrical and Electronics Engineering (SIBIRCON)*, pages 722–727. IEEE, July 2010.

[107] Geoffrey Portelli, Julien Serres, Franck Ruffier, and Nicolas Franceschini. A 3D Insect-Inspired Visual Autopilot for Corridor-Following. In *2008 2nd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 19–26. IEEE, October 2008.

[108] Florian Raudies and Heiko Neumann. An efficient linear method for the estimation of ego-motion from optical flow. 2009.

[109] Jonathan M Roberts, Peter I Corke, and Gregg Buskey. Low-Cost Flight Control System for a Small Autonomous Helicopter. In *Australasian Conference on Robotics and Automation*, number November, pages 27–29, 2002.

[110] Richard Roberts, Christian Potthast, and Frank Dellaert. Learning General Optical Flow Subspaces for Egomotion Estimation and Detection of Motion Anomalies. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 57–64. IEEE, June 2009.

[111] Konrad Rudin. Unmanned Aircraft Design, Modeling and Control : Dynamic modeling of MAVs, 2012.

[112] F. Ruffier, T. Mukai, H. Nakashima, J. Serres, and N. Franceschini. Combining sound and optic fow cues to reach a sound source despite lateral obstacles. In *2008 IEEE/SICE International Symposium on System Integration*, pages 89–93, 2008.

[113] Franck Ruffier and Nicolas Franceschini. Visually guided Micro-Aerial Vehicle : automatic take off , terrain following , landing and wind reaction *. In *2004 IEEE International Conference on Robotics & Automation*, pages 2339–2346, 2004.

[114] Franck Ruffier and Nicolas Franceschini. Aerial robot piloted in steep relief by optic flow sensors. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1266–1273. IEEE, September 2008.

[115] Syuhei Saito, Yue Bao, and Takahito Mochizuki. Autonomous Flight Control for RC Helicopter Using Camera Image. In *SICE Annual Conference 2007*, pages 1536–1539. IEEE, September 2007.

[116] Dario Schafroth, Samir Bouabdallah, Christian Bermes, and Roland Siegwart. From the Test Benches to the First Prototype of the muFly Micro Helicopter. *Journal of Intelligent and Robotic Systems*, 54(1-3):245–260, 2008.

[117] L. Schenato, R.J. Wood, and R.S. Fearing. Biomimetic Sensor Suite for Flight Control of a Micromechanical Flying Insect: Design and Experimental Results. In *2003 IEEE International Conference on Robotics and Automation*, volume 1, pages 1146–1151. IEEE, 2003.

[118] Julien Serres, Franck Ruffier, and Nicolas Franceschini. Two optic flow regulators for speed control and obstacle avoidance *. In *The First IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 750–757, 2006.

[119] Julien Serres, Franck Ruffier, Stephane Viollet, and Nicolas Franceschini. Toward optic flow regulation for wall-following and centring behaviours. *International Journal of Advanced Robotic Systems*, 3(2):1, 2006.

[120] Omid Shakernia, Rene Vidal, and Shankar Sastry. Omnidirectional Egomotion Estimation From Back-projection Flow. In *Conference on Computer Vision and Pattern Recognition*, volume 1, 2003.

[121] Shaojie Shen, Nathan Michael, and Vijay Kumar. Autonomous multi-floor indoor navigation with a computationally constrained MAV. In *2011 IEEE International Conference on Robotics and Automation*, pages 20–25. IEEE, May 2011.

[122] H. Shim, T. J. Koo, F. Hoffmann, and S. Sastry. A Comprehensive Study of Control Design for an Autonomous Helicopter. In *37th IEEE Conference on Decision & Control*, number December, pages 3653–3658, 1998.

[123] Russell Smith. Open Dynamics Engine (ODE) Manual.

[124] Steven W. Smith. Recursive Filters. In *The Scientist and Engineer's Guide to Digital Signal Processing*, pages 319–332. 1997.

[125] Dean Soccol, Saul Thurrowgood, and Mandyam Srinivasan. A Vision System for Optic-flow-based Guidance of UAVs, 2006.

[126] Sai Prashanth Soundararaj, Arvind K. Sujeeth, and Ashutosh Saxena. Autonomous Indoor Helicopter Flight using a Single Onboard Camera. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5307–5314. IEEE, October 2009.

[127] M. V. Srinivasan. An image-interpolation technique for the computation of optic flow and egomotion. *Biological Cybernetics*, 71:401–415, 1994.

[128] M. V. Srinivasan, S. W. Zhang, M. Lehrer, and T. S. Collett. Honeybee navigation en route to the goal: visual flight control and odometry. *The Journal of Experimental Biology*, 199:237–44, January 1996.

[129] Mandyam V. Srinivasan. Honeybees as a Model for the Study of Visually Guided Flight , Navigation , and Biologically Inspired Robotics. *Physical Review*, 91:413–460, 2011.

[130] Mandyam V. Srinivasan, Saul Thurrowgood, and Dean Soccol. From Visual Guidance in Flying Insects to Autonomous Aerial Vehicles. In Dario Floreano, Jean-Christophe Zufferey, Mandyam V. Srinivasan, and Charlie Ellington, editors, *Flying Insects and Robots*, pages 15–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[131] Mandyan V. Srinivasan, Shaowu Zhang, Monika Altwein, and Jurgen Tautz. Honeybee navigation: nature and calibration of the "odometer". *Science*, 287(5454):851–853, February 2000.

[132] Irem Stratmann and Erik Solda. Omnidirectional Vision and Inertial Clues for Robot Navigation. *Journal of Robotic Systems*, 21(1):33–39, January 2004.

[133] Toshikazu Tanaka, Daisuke Sasaki, Kentaro Matsumiya, Yuichiro Morikuni, and Kiyotaka Kato. Autonomous Flight Control for a Small RC Helicopter: A Measurement System with an EKF and a Fuzzy Control Via GA-Based Learning. In *SICE-ICASE Internation Joint Conference*, pages 1279–1284, 2006.

[134] An-Ting Tsaot, Yi-Ping Hung, Chiou-Shann Fuh, and Yong-Sheng Chen. Ego-Motion Estimation Using Optical Flow Fields Observed from Multiple Cameras. pages 457–462, 1997.

[135] B. Uragun. Energy Efficiency for Unmanned Aerial Vehicles. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, pages 316–320. IEEE, December 2011.

[136] Carlos M. Vélez S. and Andrés Agudelo. Simulation and multirate control of a mini-helicopter robot. In *5th WSEAS International Conference on Waveles Analysis and Multirate Systems*, volume 2005, pages 17–23, 2005.

[137] K. Weber, S. Venkatesh, and M. V. Srinivasan. Insect Inspired Behaviours for the Autonomous Control of Mobile Robots. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume 1, pages 156–160, 1996.

[138] Stephan Weiss, Markus W. Achtelik, Simon Lynen, Margarita Chli, and Roland Siegwart. Real-time Onboard Visual-Inertial State Estimation and Self-Calibration of MAVs in Unknown Environments. volume 231855, 2012.

[139] Stephan M. Weiss. *Vision Based Navigation for Micro Helicopters*. PhD thesis, 2012.

[140] J. Zeil, N. Boeddeker, and J. M. Hemmi. Visually Guided Behavior. In *Encyclopedia of Neuroscience*, volume 1, pages 369–380. 2009.

[141] S. Zein-Sabatto. Intelligent flight controllers for helicopter control. In *Proceedings of International Conference on Neural Networks (ICNN'97)*, volume 2, pages 617–621. IEEE, 1997.

[142] Shaowu Zhang, Sebastian Schwarz, Mario Pahl, Hong Zhu, and Juergen Tautz. Honeybee memory: A honeybee knows what to do and when. *The Journal of experimental biology*, 209(22):4420–8, November 2006.

[143] Min Zhu, Yunjian Ge, Shangfeng Huang, and Wenbing Chen. A Control System of The Miniature Helicopter With Stereo-Vision and Time Delay Predictor. In *2007 International Conference on Information Acquisition*, pages 608–613. IEEE, July 2007.

[144] Simon Zingg, Davide Scaramuzza, Stephan Weiss, and Roland Siegwart. MAV Navigation through Indoor Corridors Using Optical Flow. In *2010 IEEE International Conference on Robotics and Automation*, pages 3361–3368. IEEE, May 2010.

[145] Jean-Christophe Zufferey, Antoine Beyeler, and Dario Floreano. Vision-based Navigation fi-om Wheels to Wings. In *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, number October, pages 2968–2973, 2003.

[146] Jean-Christophe Zufferey, Antoine Beyeler, and Dario Floreano. Autonomous flight at low altitude using light sensors and little computational power. *International Journal of Micro Air Vehicles*, 2(2):107–117, June 2010.

[147] Jean-Christophe Zufferey, Antoine Beyeler, and Dario Floreano. Autonomous flight at low altitude with vision-based collision avoidance and GPS-based path following. In *2010 IEEE International Conference on Robotics and Automation*, pages 3329–3334. IEEE, May 2010.

[148] Jean-Christophe Zufferey and Dario Floreano. Fly-inspired visual steering of an ultralight indoor aircraft. *IEEE Transactions on Robotics*, 22(1):137–146, February 2006.

[149] Jean-Christophe Zufferey, Alexis Guanella, Antoine Beyeler, and Dario Floreano. Flying over the reality gap: From simulated to real indoor airships. *Autonomous Robots*, 21(3):243–254, September 2006.

[150] Jean-Christophe Zufferey, Adam Klaptocz, Antoine Beyeler, Jean-Daniel Nicoud, and Dario Floreano. A 10-gram vision-based flying robot, January 2007.

A. Notation

x, y, z	spacial direction (right hand convention)
x, y, z	spacial position [m]
u, v, w	velocity in x, y, z direction (respectively) [m/s]
ϕ, θ, ψ	roll, pitch, and yaw angle (respectively) [rad]
p, q, r	roll, pitch, and yaw rate (respectively) [rad/s]
α, β	lower rotor tilting angle about body x and y axis (respectively) [rad]
Ω	rotational speed of rotor [rad/s]
F	force [N] or [kg·m/s^2]
M	moment [N·m]
B	body reference frame
I	inertial reference frame

B. Simulation Code

The organization of the Webots simulation is discussed in 4.2. The following sections contain the custom physics plugin (`heli_physics.c`) which models the forces of flight; a basic helicopter controller (`heli.c`) that runs on the simulated MAV; and the Webots world file (`heli.wbt`), which describes the simulation environment and all of the properties of each object. Note that we have removed most of the environment from the world file in order to save space (it is essentially just a repetition of the same part).

B.1. Custom Physics Plugin

```
1  /*
2   * File: my_physics.c
3   * Date: Fall 2012
4   * Description: Webots physics plugin for simplified coaxial
        helicopter model
5   * Author: Raphael Cherney
6   * Modifications:
7   */
8
9  #include <ode/ode.h>
10 #include <plugins/physics.h>
11 #include <stdio.h>
12 #include <math.h>
13
14 #define THRUST_CONSTANT 1.0
15 #define ROTOR_DRAG_CONSTANT 0.001
16 #define FUSELAGE_DRAG_CONSTANT 0.02
17 #define SERVO_CONSTANT  0.001
18 #define RESTORE_CONSTANT 0.0001
19
20 /*
21  * Note: This plugin will become operational only after it
        was compiled and associated with the current world (.wbt)
        .
22  * To associate this plugin with the world follow these
        steps:
23  *  1. In the Scene Tree, expand the "WorldInfo" node and
        select its "physics" field
24  *  2. Then hit the [...] button at the bottom of the Scene
        Tree
25  *  3. In the list choose the name of this plugin (same as
        this file without the extention)
26  *  4. Then save the .wbt by hitting the "Save" button in
        the toolbar of the 3D view
27  *  5. Then revert the simulation: the plugin should now
        load and execute with the current simulation
28  */
29
30 // Bodies used in physics plugin
31 static dBodyID body;
32 static dBodyID rotor_lower;
33 static dBodyID rotor_upper;
34
```

```
35 void webots_physics_init(dWorldID world, dSpaceID space,
   dJointGroupID contactJointGroup)
36 {
37     /*
38      * Get ODE object from the .wbt model, e.g.
39      *   dBodyID body1 = dWebotsGetBodyFromDEF("MY_ROBOT");
40      *   dBodyID body2 = dWebotsGetBodyFromDEF("MY_SERVO");
41      *   dGeomID geom2 = dWebotsGetGeomFromDEF("MY_SERVO");
42      * If an object is not found in the .wbt world, the
        function returns NULL.
43      * Your code should correcly handle the NULL cases
        because otherwise a segmentation fault will crash
        Webots.
44      *
45      * This function is also often used to add joints to the
         simulation, e.g.
46      *   dJointID joint = dJointCreateBall(world, 0);
47      *   dJointAttach(joint, body1, body2);
48      *   ...
49      */
50
51     body = dWebotsGetBodyFromDEF("HELI");
52     rotor_lower = dWebotsGetBodyFromDEF("ROTOR_LOWER");
53     rotor_upper = dWebotsGetBodyFromDEF("ROTOR_UPPER");
54
55     dWebotsConsolePrintf("Running␣custom␣physics␣plugin.\n")
       ;
56 }
57
58 void webots_physics_step()
59 {
60     /*
61      * Do here what needs to be done at every time step, e.g
        . add forces to bodies
62      *   dBodyAddForce(body1, f[0], f[1], f[2]);
63      *   ...
64      */
65
66     int size;
67     float *commands = dWebotsReceive(&size);
68
69     // Handle NULL cases
70     if(body == NULL || rotor_lower == NULL || rotor_upper ==
        NULL) return;
71
```

```
72      // Read commands
73      float rotor_lower_command = commands[0];
74      float rotor_upper_command = commands[1];
75      float servo_pitch_command = commands[2];
76      float servo_roll_command = commands[3];
77
78      // Get the Euler angles
79      const double* rotation = dBodyGetRotation(body);
80      float pitch, roll, yaw;
81      if (rotation[4] > 0.999)
82      {
83          yaw = 0;
84          pitch = M_PI/2;
85          roll = atan2(rotation[2], rotation[10]);
86          dWebotsConsolePrintf("Euler␣angles␣singularity:␣
                NORTH␣POLE\n");
87      }
88      else if (rotation[4] < -0.999)
89      {
90          yaw = 0;
91          pitch = -M_PI/2;
92          roll = atan2(rotation[2], rotation[10]);
93          dWebotsConsolePrintf("Euler␣angles␣singularity:␣
                SOUTH␣POLE\n");
94      }
95      else
96      {
97          yaw = atan2(-rotation[8], rotation[0]);
98          pitch = atan2(rotation[4], sqrt(rotation[5]*rotation
                [5] + rotation[6]*rotation[6]));
99          roll = atan2(-rotation[6], rotation[5]);
100     }
101     //dWebotsConsolePrintf("pitch: %f\nroll: %f\nyaw: %f\n\n
            ", pitch, roll, yaw);
102
103     // Get velocities (in inertial frame)
104     const double* velocity = dBodyGetLinearVel(body);
105     float u = velocity[0];
106     float v = velocity[2];
107     float w = velocity[1];
108
109     // For velocities in the body frame, use the following
110     //float u = velocity[0]*(cos(yaw)*cos(pitch))+ velocity
            [2]*(sin(yaw)*cos(pitch))+ velocity[1]*(-sin(pitch));
```

```
111     //float v = velocity[0]*(cos(yaw)*sin(pitch)*sin(roll)-
           sin(yaw)*cos(roll))+ velocity[2]*(sin(yaw)*sin(pitch)*
           sin(roll)+cos(yaw)*cos(roll))+ velocity[1]*(cos(pitch)
           *sin(roll));
112     //float w = velocity[0]*(cos(yaw)*sin(pitch)*cos(roll)+
           sin(yaw)*sin(roll))+ velocity[2]*(sin(yaw)*sin(pitch)*
           cos(roll)-cos(yaw)*sin(roll))+ velocity[1]*(cos(pitch)
           *cos(roll));
113     //dWebotsConsolePrintf("u: %f\nv: %f\nw: %f\n\n", u, v,
           w);
114
115     // Thrust force is proportional to rotor angular
           velocity squared
116     float thrust_lower = pow(rotor_lower_command, 2) *
           THRUST_CONSTANT;
117     float thrust_upper = pow(rotor_upper_command, 2) *
           THRUST_CONSTANT;
118
119     // Lower thrust vector is rotated by the pitch and roll
           servos (via the swashplate)
120     float angle_pitch = servo_pitch_command * SERVO_CONSTANT
           ;   // in radians
121     float angle_roll = servo_roll_command * SERVO_CONSTANT;
                // in radians
122
123     // Calculate components of thrust
124     float thrust_x = thrust_lower * (-sin(angle_pitch) * cos
           (angle_roll)) / sqrt(1 - pow(sin(angle_pitch), 2) *
           pow(sin(angle_roll), 2));
125     float thrust_y = thrust_lower * (cos(angle_pitch) * sin(
           angle_roll)) / sqrt(1 - pow(sin(angle_pitch), 2) * pow
           (sin(angle_roll), 2));
126     float thrust_z = thrust_lower * (-cos(angle_pitch) * cos
           (angle_roll)) / sqrt(1 - pow(sin(angle_pitch), 2) *
           pow(sin(angle_roll), 2));
127     //dWebotsConsolePrintf("thrust_x: %f\nthrust_y: %f\
           nthrust_z: %f\n\n", thrust_x, thrust_y, thrust_z);
128
129     // Fuselage drag is proportional to the linear velocity
           squared
130     float drag_x, drag_y, drag_z;
131     if (u > 0)
132     {
133         drag_x = u * u * -FUSELAGE_DRAG_CONSTANT;
134     }
```

```
135     else
136     {
137         drag_x = u * u * FUSELAGE_DRAG_CONSTANT;
138     }
139     if (v > 0)
140     {
141         drag_y = v * v * -FUSELAGE_DRAG_CONSTANT;
142     }
143     else
144     {
145         drag_y = v * v * FUSELAGE_DRAG_CONSTANT;
146     }
147     if (w > 0)
148     {
149         drag_z = w * w * -FUSELAGE_DRAG_CONSTANT;
150     }
151     else
152     {
153         drag_z = w * w * FUSELAGE_DRAG_CONSTANT;
154     }
155     //dWebotsConsolePrintf("drag_x: %f\ndrag_y: %f\ndrag_z:
          %f\n\n", drag_x, drag_y, drag_z);
156
157     // Drag moment is proportional to rotor angular velocity
          squared
158     float drag_torque = (pow(rotor_lower_command, 2) - pow(
          rotor_upper_command, 2)) * ROTOR_DRAG_CONSTANT;
159
160     // Calculate restoring moment
161     float restore_pitch = pitch * -RESTORE_CONSTANT;
162     float restore_roll = roll * -RESTORE_CONSTANT;
163     //dWebotsConsolePrintf("restore roll: %f\nrestore pitch:
          %f\n\n", restore_roll, restore_pitch);
164
165     // Apply forces
166     dBodyAddRelForce(rotor_upper, 0, thrust_upper, 0);
                    // upper rotor thrust
167     dBodyAddRelForce(rotor_lower, thrust_x, -thrust_z,
          thrust_y);       // lower rotor thrust
168     dBodyAddForce(body, drag_x, drag_z, drag_y);
                    // fuselage drag
169     dBodyAddRelTorque(body, 0, drag_torque, 0);
                    // rotor torque imbalance
170     dBodyAddRelTorque(body, restore_roll, 0, restore_pitch);
                    // restoring moment
```

```
171 }
172
173 void webots_physics_draw()
174 {
175     /*
176      * This function can optionally be used to add OpenGL
           graphics to the 3D view, e.g.
177      *   // setup draw style
178      *   glDisable(GL_LIGHTING);
179      *   glLineWidth(2);
180      *
181      *   // draw a yellow line
182      *   glBegin(GL_LINES);
183      *   glColor3f(1, 1, 0);
184      *   glVertex3f(0, 0, 0);
185      *   glVertex3f(0, 1, 0);
186      *   glEnd();
187      */
188 }
189
190 int webots_physics_collide(dGeomID g1, dGeomID g2)
191 {
192     /*
193      * This function needs to be implemented if you want to
           overide Webots collision detection.
194      * It must return 1 if the collision was handled and 0
           otherwise.
195      * Note that contact joints should be added to the
           contactJointGroup, e.g.
196      *   n = dCollide(g1, g2, MAX_CONTACTS, &contact[0].geom
           , sizeof(dContact));
197      *   ...
198      *   dJointCreateContact(world, contactJointGroup, &
           contact[i])
199      *   dJointAttach(contactJoint, body1, body2);
200      *   ...
201      */
202
203     return 0;
204 }
205
206 void webots_physics_cleanup()
207 {
208     /*
```

```
209       * Here you need to free any memory you allocated in
              above, close files, etc.
210       * You do not need to free any ODE object, they will be
              freed by Webots.
211       */
212
213      dWebotsConsolePrintf("Physics clean up.\n");
214  }
```

B.2. Helicopter Controller

```
1  /*
2   * File: heli.c
3   * Date: Fall 2012
4   * Description: Controller for coaxial helicopter simulation
5   * Author: Raphael Cherney
6   */
7
8  /* Include files */
9  #include <webots/robot.h>
10 #include <webots/supervisor.h>
11 #include <webots/servo.h>
12 #include <webots/camera.h>
13 #include <webots/display.h>
14 #include <webots/gyro.h>
15 #include <webots/accelerometer.h>
16 #include <webots/gps.h>
17 #include <webots/emitter.h>
18 #include <stdio.h>
19 #include <string.h>     // needed for memcpy
20 #include <math.h>
21
22 /* Define macros */
23 #define TIME_STEP 16          // simulation time step (in ms)
24 #define NUM_CAMERAS 8         // number of active cameras
25 #define IMAGE_HEIGHT 16       // height of camera images
26 #define IMAGE_WIDTH 16        // width of camera images
27 #define REGIONS_X 1           // number of regions to split
      the...
28 #define REGIONS_Y 1           // ...image into for calculating
       flow
29 #define SEARCH_DIST 1         // max allowable motion in
      pixels
30
31 /* Helper functions */
32 // Return greyscale image scaled as the helicopter would
      receive from sensor ring
33 void get_scaled_image(WbDeviceTag camera, unsigned int *
      image)
34 {
35     // Declare general variables
36     unsigned int x, y, i, j, k, start_index, index, sum;
37
38     // Get simulation image size
```

```
39      unsigned int image_height = wb_camera_get_height(camera)
           ;
40      unsigned int image_width = wb_camera_get_width(camera);
41
42      // Get image from camera
43      const unsigned char *raw_image = wb_camera_get_image(
           camera);
44
45      // Generate greyscale image
46      unsigned int greyscale_image[image_width * image_height
           ];
47      i = 0;
48      for (y=0; y<image_height; y++)
49      {
50          for (x=0; x<image_width; x++)
51          {
52              greyscale_image[i] = ((unsigned int)
                   wb_camera_image_get_red(raw_image, image_width
                   , x, y) +
53                                    (unsigned int)
                                       wb_camera_image_get_green
                                       (raw_image, image_width,
                                        x, y) +
54                                    (unsigned int)
                                       wb_camera_image_get_blue
                                       (raw_image, image_width,
                                        x, y)) / 3.0;
55              i++;
56          }
57      }
58
59      // Scale the image (averaging over window produces a
           better result than the default subsampling)
60      unsigned int *output_image = image;
61      unsigned int window_height = image_height / IMAGE_HEIGHT
           ;
62      unsigned int window_width = image_width / IMAGE_WIDTH;
63      i = 0;
64      for (y=0; y<IMAGE_HEIGHT; y++)
65      {
66          for (x=0; x<IMAGE_WIDTH; x++)
67          {
68              start_index = (y * window_height) * image_width
                   +
```

```
69                              (x * window_width);
                                                // starting index for
                                   window sum
70              sum = 0;
71              for (k=0; k<window_height; k++)
72              {
73                  for (j=0; j<window_width; j++)
74                  {
75                      index = start_index + j + image_width*k;
76                      sum += greyscale_image[index];
77                  }
78              }
79              output_image[i] = sum / (window_width *
                   window_height);
80              i++;
81          }
82      }
83  }
84
85  // Calculate optic flow using I2A algorithm
86  signed char *calculate_flow(unsigned int *previous_image,
       // previous image data
87                              unsigned int *current_image,
                                   // most recent image data
88                              unsigned int search_dist,
                                   // distance in which to assume
                                    match exists
89                              unsigned int image_size[2],
                                   // image size = [height, width
                                   ]
90                              unsigned int num_regions[2],
                                   // number of [rows, columns]
                                   to split image into
91                              signed char *optic_flow)
                                   // flow array to write data to
92  {
93      // Declare variables
94      unsigned int region_x, region_y, x, y, start_index,
           index, i;
95      unsigned int region_height = image_size[0] / num_regions
           [0];
96      unsigned int region_width = image_size[1] / num_regions
           [1];
97      signed long long f4f3, f2f1, fcf0;
                    // differences for I2A
```

```
98      signed long long A, B, C, D, E;
                // sums for I2A
99      float delta_x, delta_y, denominator;
                // flow estimate
100
101     i = 0;       // initialize index for optic flow array
102     // For each region of the image...
103     for (region_y=0; region_y<num_regions[0]; region_y++)
104     {
105         for (region_x=0; region_x<num_regions[1]; region_x
                ++)
106         {
107             start_index = (region_y * region_height) *
                    image_size[1] +
108                           (region_x * region_width);
                                    // starting index for
                                region
109
110             // Calculate the sums in the region
111             A = B = C = D = E = 0;
112             for (y=search_dist; y<region_height-search_dist;
                    y++)
113             {
114                 for (x=search_dist; x<region_width-
                        search_dist; x++)
115                 {
116                     index = start_index + x + y * image_size
                            [1];
117
118                     f4f3 = (signed long long) previous_image
                            [index - search_dist * image_size[1]]
                            - // (f4 - f3)
119                          (signed long long) previous_image
                                [index + search_dist *
                                image_size[1]];
120                     f2f1 = (signed long long) previous_image
                            [index - search_dist] -
                                // (f2 - f1)
121                          (signed long long) previous_image
                                [index + search_dist];
122                     fcf0 = (signed long long) current_image[
                            index] -
                                // (fc - f0)
123                          (signed long long) previous_image
                                [index];
```

```
124
125                     A += f2f1 * f2f1;
126                     B += f4f3 * f2f1;
127                     C += fcf0 * f2f1;
128                     D += f4f3 * f4f3;
129                     E += fcf0 * f4f3;
130                 }
131             }
132
133             // Solve for motion
134             denominator = (float) (A * D - B * B) /
135                           (2.0f * (float) search_dist *
                                 127.0f);  // including a scaling
                                  by 127 for char conversion
136             if (denominator != 0)
137             {
138                 delta_x = (float) (C * D - B * E) /
                       denominator;
139                 delta_y = (float) (A * E - C * B) /
                       denominator;
140             }
141             else
142             {
143                 delta_x = 0;
144                 delta_y = 0;
145             }
146             if (delta_x > 127) delta_x = 127;
147             else if (delta_x < -127) delta_x = -127;
148             if (delta_y > 127) delta_y = 127;
149             else if (delta_y < -127) delta_y = -127;
150
151             optic_flow[i] = (signed char) delta_x;
152             optic_flow[i+1] = (signed char) delta_y;
153             i = i + 2;
154         }
155     }
156
157     return optic_flow;
158 }
159
160 void display_text(WbDeviceTag tag, char text[64])
161 {
162     wb_display_set_color(tag, 0x000000);
163     wb_display_fill_rectangle(tag, 0, 0,
          wb_display_get_width(tag), wb_display_get_height(tag))
```

```
        ;
164     wb_display_set_color(tag, 0xFFFFFF);
165     wb_display_draw_text(tag, text, 0, 0);
166 }
167
168 /*
169  * Main program
170  * The arguments of the main function can be specified by
        the
171  * "controllerArgs" field of the Robot node
172  */
173 int main(int argc, char **argv)
174 {
175     // Declare variables
176     static float rotor_lower_command = 0.384;
177     static float rotor_upper_command = 0.384;
178     static float servo_pitch_command = 0.0;
179     static float servo_roll_command = 0.0;
180     float commands[4];
181     static unsigned long long time = 0;
182     int i, j, x, y;
183     char text[64];
184
185     /* Necessary to initialize webots */
186     wb_robot_init();
187
188     /*
189     * You should declare here WbDeviceTag variables for
          storing
190     * robot devices like this:
191     *  WbDeviceTag my_sensor = wb_robot_get_device("
          my_sensor");
192     *  WbDeviceTag my_actuator = wb_robot_get_device("
          my_actuator");
193     */
194     // Cameras
195     unsigned int current_image[NUM_CAMERAS][IMAGE_HEIGHT *
          IMAGE_WIDTH];
196     unsigned int previous_image[NUM_CAMERAS][IMAGE_HEIGHT *
          IMAGE_WIDTH];
197     signed char optic_flow[NUM_CAMERAS][2 * REGIONS_Y *
          REGIONS_X];
198     WbDeviceTag camera[NUM_CAMERAS];
199     const char *camera_name[NUM_CAMERAS] = {"camera_0", "
          camera_1", "camera_2", "camera_3", "camera_4", "
```

```
      camera_5", "camera_6", "camera_7"};
200   WbDeviceTag display[NUM_CAMERAS];
201   const char *display_name[NUM_CAMERAS] = {"view_0", "
        view_1", "view_2", "view_3", "view_4", "view_5", "
        view_6", "view_7"};
202   for (i=0; i<NUM_CAMERAS; i++)
203   {
204       camera[i] = wb_robot_get_device(camera_name[i]);
205       wb_camera_enable(camera[i], TIME_STEP);
206       display[i] = wb_robot_get_device(display_name[i]);
207   }
208
209   // Gyro
210   WbDeviceTag gyro = wb_robot_get_device("gyro");
211   wb_gyro_enable(gyro, TIME_STEP);
212
213   // GPS tracker
214   WbDeviceTag gps = wb_robot_get_device("gps");
215   wb_gps_enable(gps, TIME_STEP);
216
217   // Emitter
218   WbDeviceTag emitter = wb_robot_get_device("emitter");
219
220   // Rotors
221   WbDeviceTag rotor_bottom = wb_robot_get_device("
        rotor_lower");
222   wb_servo_set_position(rotor_bottom, INFINITY);
223   wb_servo_set_velocity(rotor_bottom, 10);  // 1 rotation
        per second
224   WbDeviceTag rotor_top = wb_robot_get_device("rotor_upper
        ");
225   wb_servo_set_position(rotor_top, INFINITY);
226   wb_servo_set_velocity(rotor_top, 10);  // 1 rotation per
         second
227
228   // Info display
229   WbDeviceTag info = wb_robot_get_device("info");
230   display_text(info, "Running␣simulation...");
231
232   // Output file
233   FILE *log_file;
234   log_file = fopen("log.txt", "w");
235
236   /* Main loop */
237   do
```

```
238     {
239         /*
240          * Read the sensors :
241          * Enter here functions to read sensor data, like:
242          *  double val = wb_distance_sensor_get_value(
                 my_sensor);
243          */
244         // Read from gyro
245         const double *gyro_data = wb_gyro_get_values(gyro);
246         //float derotate[NUM_CAMERAS][2];
247 
248         // Read from gps
249         const double *gps_data = wb_gps_get_values(gps);
250 
251         // Read from cameras
252         for (i=0; i<NUM_CAMERAS; i++)
253         {
254             // Store the previous image
255             memcpy(previous_image[i], current_image[i],
                    sizeof(current_image[i]));
256             // Get new image from camera[i]
257             get_scaled_image(camera[i], current_image[i]);
258 
259             static unsigned int search_dist = SEARCH_DIST;
260             static unsigned int image_size[2] = {
                    IMAGE_HEIGHT, IMAGE_WIDTH};
261             static unsigned int num_regions[2] = {REGIONS_Y,
                     REGIONS_X};
262             // Calculate optic flow using the new image data
263             calculate_flow(previous_image[i], current_image[
                    i], search_dist, image_size, num_regions,
                    optic_flow[i]);
264         }
265 
266         // Display scaled images
267         for (i=0; i<NUM_CAMERAS; i++)
268         {
269             unsigned int display_height =
                    wb_display_get_height(display[i]);
270             unsigned int display_width =
                    wb_display_get_width(display[i]);
271             unsigned int pixel_height = display_height/
                    IMAGE_HEIGHT;
272             unsigned int pixel_width = display_width/
                    IMAGE_WIDTH;
```

```
273
274             j = 0;
275             for (y=0; y<display_height; y+=pixel_height)
276             {
277                 for (x=0; x<display_width; x+=pixel_width)
278                 {
279                     int color = ((unsigned char)
                           current_image[i][j] << 16) |
280                                   ((unsigned char)
                                      current_image[i][j] << 8)
                                       |
281                                   ((unsigned char)
                                      current_image[i][j]);
282                     wb_display_set_color(display[i], color);
283                     wb_display_fill_rectangle(display[i], x,
                           y, pixel_width, pixel_height);
284                     j++;
285                 }
286             }
287
288             // Display flow vectors
289             int region_height = display_height/REGIONS_Y;
290             int region_width = display_width/REGIONS_X;
291
292             j = 0;
293             for (y=region_height/2; y<display_height; y+=
                   region_height)
294             {
295                 for (x=region_width/2; x<display_width; x+=
                       region_width)
296                 {
297                     wb_display_set_color(display[i], 0
                           x00FF00);     // green
298                     wb_display_draw_line(display[i], x, y, x
                           +optic_flow[i][j]*region_width/256, y+
                           optic_flow[i][j+1]*region_height/256);
299                     j = j + 2;
300                 }
301             }
302         }
303
304         // De-rotate horizontal flow vectors
305         float derotated_flow[NUM_CAMERAS];
306         const float flow_constant = 29.77;
307         for (i=0; i<NUM_CAMERAS; i++)
```

```
308         {
309             derotated_flow[i] = (float) optic_flow[i][0] - (
                    gyro_data[1] * flow_constant);
310         }
311
312         // Low-pass filter
313         static float filtered_flow[NUM_CAMERAS] =
                {0,0,0,0,0,0,0,0};
314         const float alpha = 0.2;
315         for (i=0; i<NUM_CAMERAS; i++)
316         {
317             if (!(filtered_flow[i] != filtered_flow[i]))
                    // check that the value is not NaN
318             {
319                 filtered_flow[i] = (1 - alpha) *
                        filtered_flow[i] + alpha * derotated_flow[
                        i];
320             }
321             else
322             {
323                 filtered_flow[i] = 0;
324             }
325         }
326
327         /* Process sensor data here */
328
329         // Control values
330         static float throttle = 0.3835;
331         static float yaw = 0.0;
332         static float pitch = -0.6;
333         static float roll = 0.0;
334
335         // Throttle control
336         if (gps_data[1] > 1.7)
337         {
338             throttle = 0.382;
339         }
340         else if (gps_data[1] < 1.3)
341         {
342             throttle = 0.385;
343         }
344
345         // Weights
346         const float yaw_weights[NUM_CAMERAS] =
                {.001,.001,-.005,-.01,-.01,-.005,.001,.001};
```

```
347         const float roll_weights[NUM_CAMERAS] =
               {-.01,-.01,-.01,-.01,-.01,-.01,-.01,-.01};
348
349         // Linear weighted controller
350         yaw = 0;
351         roll = 0;
352         for (i=0; i<NUM_CAMERAS; i++)
353         {
354             if (!(filtered_flow[i] != filtered_flow[i]))
                   // check that the value is not NaN
355             {
356                 //yaw += yaw_weights[i] * filtered_flow[i];
357                 roll += roll_weights[i] * filtered_flow[i];
358             }
359         }
360
361         // Display commmands
362         sprintf(text, "yaw:%f␣roll:%f", yaw, roll);
363         display_text(info, text);
364
365         // Set commands
366         rotor_lower_command = throttle - yaw;
367         rotor_upper_command = throttle + yaw;
368         servo_pitch_command = pitch;
369         servo_roll_command = roll;
370
371         // Send commands
372         commands[0] = rotor_lower_command;
373         commands[1] = rotor_upper_command;
374         commands[2] = servo_pitch_command;
375         commands[3] = servo_roll_command;
376         wb_emitter_send(emitter, commands, sizeof(commands))
               ;
377
378         // Save gps data
379         fprintf(log_file, "%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
               gps_data[0], gps_data[1], gps_data[2], commands
               [0], commands[1], commands[2], commands[3]);
380
381         /*
382          * Perform a simulation step
383          * and leave the loop when the simulation is over
384          */
385         time += TIME_STEP;
386         if (time > 20000)
```

```
387             {
388                 //break;
389                 continue;
390             }
391         }
392         while (wb_robot_step(TIME_STEP) != -1);
393 
394         /* Enter here exit cleanup code */
395 
396         /* Necessary to cleanup webots stuff */
397         wb_robot_cleanup();
398 
399         fclose(log_file);
400 
401         return 0;
402 }
```

B.3. Webots World

```
#VRML_SIM V6.0 utf8
WorldInfo {
  info [
    "Description"
    "Author: Raphael Cherney <raphael.cherney@epfl.ch>"
    "Date: Fall 2012-13"
  ]
  physics "heli_physics"
  basicTimeStep 16
  displayRefresh 1
}
Viewpoint {
  orientation -0.0915324 -0.964368 -0.248225 2.46213
  position 0.368689 2.50813 0.154523
  follow "supervisor"
}
Background {
  skyColor [
    0.8 0.8 0.8
  ]
}
DEF HELI Supervisor {
  translation 1.5 1.5 1.5
  children [
    DEF BODY Transform {
      children [
        DEF BODY Shape {
          appearance Appearance {
            material Material {
              diffuseColor 1 0 0
            }
          }
          geometry Box {
            size 0.05 0.04 0.03
          }
        }
      ]
    }
    DEF TAIL Transform {
      translation -0.065 0 0
      children [
        DEF FIN Transform {
          translation -0.0475 -0.01 0
```

```
44            children [
45              DEF FIN Shape {
46                appearance Appearance {
47                  material Material {
48                    diffuseColor 1 0 0
49                  }
50                }
51                geometry Box {
52                  size 0.015 0.04 0.001
53                }
54              }
55            ]
56          }
57          DEF BOOM Shape {
58            appearance Appearance {
59              material Material {
60                diffuseColor 0.09 0.09 0.09
61              }
62            }
63            geometry Box {
64              size 0.08 0.001 0.001
65            }
66          }
67        ]
68      }
69      DEF SKIDS Transform {
70        translation 0 -0.03 0
71        children [
72          DEF SKID_RIGHT Transform {
73            translation 0 0 0.03
74            children [
75              DEF SKID Shape {
76                appearance Appearance {
77                  material Material {
78                    diffuseColor 0.09 0.09 0.09
79                  }
80                }
81                geometry Box {
82                  size 0.06 0.001 0.001
83                }
84              }
85            ]
86          }
87          DEF SKID_LEFT Transform {
88            translation 0 0 -0.03
```

```
89            children [
90              DEF SKID Shape {
91                appearance Appearance {
92                  material Material {
93                    diffuseColor 0.09 0.09 0.09
94                  }
95                }
96                geometry Box {
97                  size 0.06 0.001 0.001
98                }
99              }
100           ]
101         }
102       ]
103     }
104     DEF RING Transform {
105       translation 0 -0.01 0
106       children [
107         DEF RING Shape {
108           appearance Appearance {
109             material Material {
110               diffuseColor 0.846586 0.425132 0.001755
111             }
112           }
113           geometry Cylinder {
114             bottom FALSE
115             height 0.01
116             radius 0.05
117             top FALSE
118           }
119         }
120       ]
121     }
122     DEF MAST Transform {
123       translation 0 0.045 0
124       children [
125         DEF MAST Shape {
126           appearance Appearance {
127             material Material {
128               diffuseColor 0.09 0.09 0.09
129             }
130           }
131           geometry Cylinder {
132             height 0.05
133             radius 0.001
```

```
        }
      }
    ]
  }
  DEF ROTOR_UPPER Solid {
    translation 0 0.07 0
    children [
      DEF ROTOR Shape {
        appearance Appearance {
          material Material {
            transparency 0.8
          }
        }
        geometry Cylinder {
          height 0.001
          radius 0.1
        }
      }
    ]
    boundingObject USE ROTOR
    physics Physics {
      density -1
      mass 1e-05
    }
  }
  DEF ROTOR_LOWER Solid {
    translation 0 0.04 0
    children [
      DEF ROTOR Shape {
        appearance Appearance {
          material Material {
            transparency 0.8
          }
        }
        geometry Cylinder {
          height 0.001
          radius 0.1
        }
      }
    ]
    boundingObject USE ROTOR
    physics Physics {
      density -1
      mass 1e-05
    }
```

```
    }
    DEF ANIMATE_UPPER Servo {
      translation 0 0.07 0
      children [
        DEF BLADE Shape {
          appearance Appearance {
            material Material {
              diffuseColor 0.09 0.09 0.09
            }
          }
          geometry Box {
            size 0.02 0.001 0.2
          }
        }
      ]
      name "rotor_upper"
    }
    DEF ANIMATE_LOWER Servo {
      translation 0 0.04 0
      rotation 0 -1 0 0
      children [
        DEF BLADE Shape {
          appearance Appearance {
            material Material {
              diffuseColor 0.09 0.09 0.09
            }
          }
          geometry Box {
            size 0.02 0.001 0.2
          }
        }
      ]
      name "rotor_lower"
    }
    DEF CAMERA_0 Camera {
      translation -0.046 -0.01 -0.019
      rotation 0 1 0 1.1781
      name "camera_0"
      fieldOfView 1.309
      windowPosition 0.0841924 0.0586081
      pixelSize 0
    }
    DEF CAMERA_1 Camera {
      translation -0.019 -0.01 -0.046
      rotation 0 1 0 0.3927
```

```
      name "camera_1"
      fieldOfView 1.309
      windowPosition 0.203539 0.0586081
      pixelSize 0
    }
    DEF CAMERA_2 Camera {
      translation 0.019 -0.01 -0.046
      rotation 0 1 0 -0.3927
      name "camera_2"
      fieldOfView 1.309
      windowPosition 0.322232 0.0586081
      pixelSize 0
    }
    DEF CAMERA_3 Camera {
      translation 0.046 -0.01 -0.019
      rotation 0 1 0 -1.1754
      name "camera_3"
      fieldOfView 1.309
      windowPosition 0.441 0.0586081
      pixelSize 0
    }
    DEF CAMERA_4 Camera {
      translation 0.046 -0.01 0.019
      rotation 0 1 0 -1.9635
      name "camera_4"
      fieldOfView 1.309
      windowPosition 0.56275 0.0586081
      pixelSize 0
    }
    DEF CAMERA_5 Camera {
      translation 0.019 -0.01 0.046
      rotation 0 1 0 -2.7489
      name "camera_5"
      fieldOfView 1.309
      windowPosition 0.682682 0.0586081
      pixelSize 0
    }
    DEF CAMERA_6 Camera {
      translation -0.019 -0.01 0.046
      rotation 0 1 0 -3.40339
      name "camera_6"
      fieldOfView 1.309
      windowPosition 0.800983 0.0586081
      pixelSize 0
    }
```

```
    DEF CAMERA_7 Camera {
      translation -0.046 -0.01 0.019
      rotation 0 1 0 -4.18879
      name "camera_7"
      fieldOfView 1.309
      windowPosition 0.920915 0.0586081
      pixelSize 0
    }
    DEF VIEW_0 Display {
      name "view_0"
      windowPosition 0.06 0
    }
    DEF VIEW_1 Display {
      name "view_1"
      windowPosition 0.17 0
    }
    DEF VIEW_2 Display {
      name "view_2"
      windowPosition 0.28 0
    }
    DEF VIEW_3 Display {
      name "view_3"
      windowPosition 0.39 0
    }
    DEF VIEW_4 Display {
      name "view_4"
      windowPosition 0.5 0
    }
    DEF VIEW_5 Display {
      name "view_5"
      windowPosition 0.61 0
    }
    DEF VIEW_6 Display {
      name "view_6"
      windowPosition 0.72 0
    }
    DEF VIEW_7 Display {
      name "view_7"
      windowPosition 0.83 0
    }
    DEF INFO Display {
      name "info"
      width 512
      height 8
      windowPosition 0 1
```

```
    }
    DEF GYRO Gyro {
    }
    DEF ACCEL Accelerometer {
    }
    DEF TRACKER GPS {
    }
    DEF EMITTER Emitter {
    }
    DEF RECEIVER Receiver {
    }
  ]
  boundingObject Box {
    size 0.06 0.06 0.06
  }
  physics Physics {
    density -1
    mass 0.03
    damping Damping {
      linear 0
    }
  }
  controller "heli"
}
```