

# Flight Simulation of a 3 gram Autonomous Glider

*Jon Perry Entwistle  
Ronald S. Fearing*

Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2006-80

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-80.html>

May 24, 2006



Copyright © 2006, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

#### Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0412541. The author also gratefully acknowledges DARPA for support under fund #FA8650-05-C-7138.

---

# **Flight Simulation of a 3 Gram Autonomous Glider**

by Jon Perry Entwistle

---

## **Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

### **Committee:**

---

Prof. Ron Fearing  
Research Advisor

---

Date

\* \* \* \* \*

---

Prof. Seth Sanders  
Second Reader

---

Date

# Contents

|  |           |
|--|-----------|
| <b>Simulator Design</b> .....                    | <b>2</b>  |
| 1.1    Aerodynamics .....                        | 3         |
| 1.2    Body Dynamics.....                        | 7         |
| 1.3    Sensors .....                             | 11        |
| 1.3.1    Ocelli .....                            | 11        |
| 1.3.2    Optical Flow Sensor Module .....        | 13        |
| <b>System Integration and Flight Tests</b> ..... | <b>20</b> |
| 2.1    Open Loop Control.....                    | 20        |
| 2.1.1    Passive Test Results.....               | 21        |
| 2.1.2    Timed Response Results .....            | 24        |
| 2.1.3    Triggered Response Test Results .....   | 26        |
| 2.2    Closed Loop Control.....                  | 27        |
| 2.2.1    Ocelli Control.....                     | 27        |
| 2.2.2    Optical Flow Control .....              | 27        |
| <b>Analysis and Conclusions</b> .....            | <b>33</b> |
| 3.1    Sensor Concerns .....                     | 33        |
| 3.2    Platform Concerns .....                   | 35        |
| 3.3    Conclusions and Future Work .....         | 36        |
| <b>Bibliography</b> .....                        | <b>37</b> |
| <b>Appendix</b> .....                            | <b>A1</b> |

## **Abstract**

Flight Simulation of a 3 Gram Autonomous Glider

by

Jon Perry Entwistle<sup>\*</sup>

Master of Science in Electrical Engineering

University of California, Berkeley

Prof. Ronald Fearing, Chair

The Microglider project at UC Berkeley aims at designing an autonomous, palm-sized glider to serve as a test-bed for minimum sensor flight control. In order to quickly test and refine the flight control strategies of the microglider, a software tool is presented here which simulates the microglider in flight. The simulated glider is tested using ocelli and optical flow sensors, and a discrete motion control strategy is designed and refined.

---

<sup>\*</sup> This material is based upon work supported by the National Science Foundation under Grant No. IIS-0412541. The author also gratefully acknowledges DARPA for support under fund #FA8650-05-C-7138.

# Chapter 1

## Simulator Design

The glider simulator is implemented directly in Matlab, and consists of 3 modules: aerodynamics, body dynamics, and sensors. These three primary modules are structured as shown in Figure 2. The basic structure of the simulator is that of a continuous loop. This loop is circled once per timestep, and at the end of the loop the glider state is updated to reflect the changes that occur during that loop. Both the aerodynamics and body dynamics modules operate at a frequency of 1 kHz, while the sensor module operates at 100 Hz. All three of the primary modules require the glider state, and each will be described in detail in subsequent sections.

The state of the glider refers to the glider's translational position and velocity, as well as the glider's rotational direction and angular velocity. The translational position and velocity are stored as 3 element vectors, corresponding to X, Y, and Z directions in the frame of reference of the environment (the inertial reference). The translational position and velocity at any given time are referred to as  $S(t)$  and  $Sdot(t)$ , respectively. The rotational direction is in a 4 element vector  $q(t)$ , a quaternion that captures the 3 degrees of

freedom of rotation. Finally, the angular velocity is stored using a 3 element vector  $w(t)$ , which refers to the X, Y, and Z directions with respect to the glider's frame of reference.

$$S = \begin{bmatrix} x\_pos \\ y\_pos \\ z\_pos \end{bmatrix} \quad \dot{S} = \begin{bmatrix} vx \\ vy \\ vz \end{bmatrix} \quad q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \quad w = \begin{bmatrix} wx \\ wy \\ wz \end{bmatrix}$$

Figure 1: Glider state variables

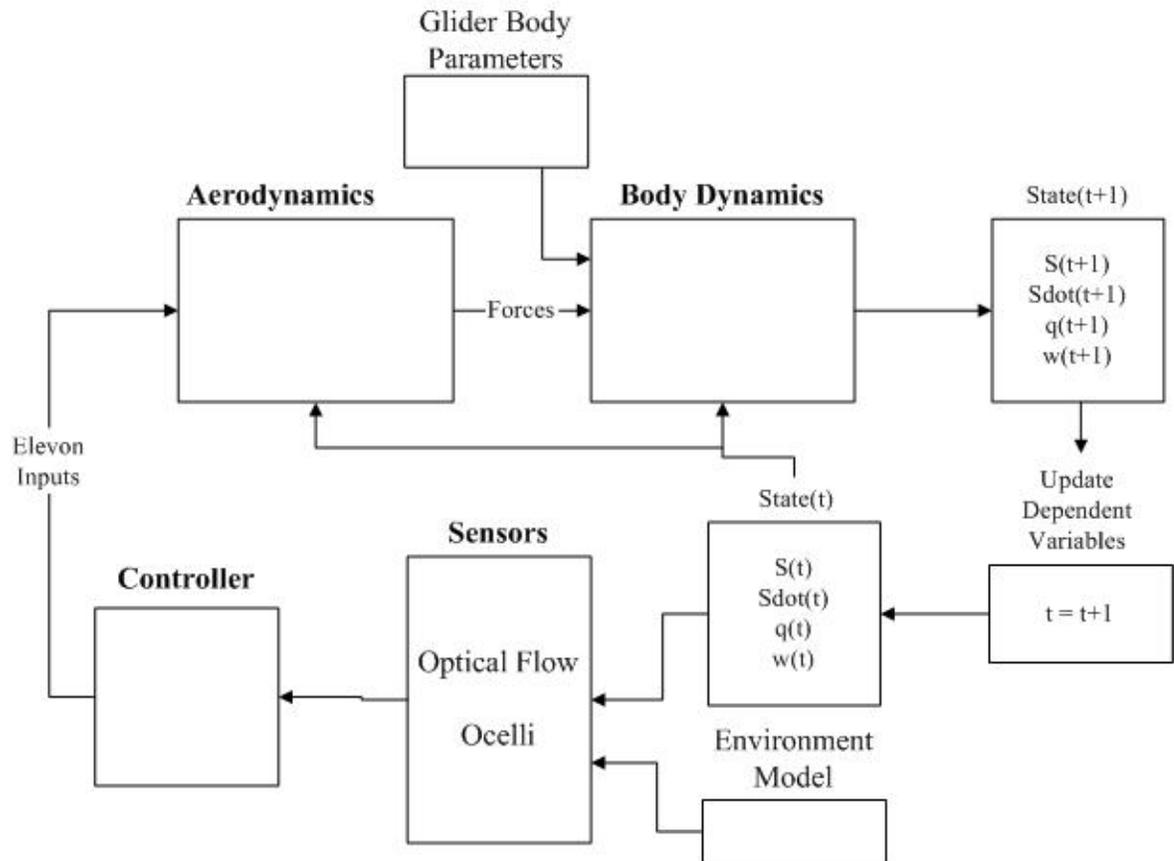


Figure 2: Complete simulator model

## 1.1 Aerodynamics

The aerodynamics section takes the glider position and velocity, and calculates the forces acting on the glider due to aerodynamic effects. The aerodynamics section of the

simulator was designed using measured glider parameters and basic aerodynamic theory (primary references include [1], [2], [3]). The aerodynamic theory is then implemented with the glider parameters to allow the extension of measured glider responses to arbitrary conditions in simulation.

The basic aerodynamic theory states that flight at nearly constant low Reynolds numbers is characterized by known changes in lift and drag forces with changes in wind speed. Measuring the lift and drag forces at a known wind speed allows one to compute lift and drag coefficients. These coefficients can then be used to calculate wind and drag forces at arbitrary wind speeds. Lift force  $L$ , lift coefficient  $C_L$ , and wind speed  $V$ , are related by the following equation

$$L = 0.5 * C_L * \rho * A * V^2$$

where  $\rho$  is the density of air and  $A$  is the wing area. This equation can calculate drag by replacing lift force  $L$  with drag force  $D$ , and lift coefficient  $C_L$  with drag coefficient  $C_D$ . These relationships are used to perform the force calculations in simulation.

From the aerodynamic theory one can see that the necessary glider parameters are lift and drag coefficients. These lift and drag coefficients can be calculated from lift and drag forces measured using wind tunnel tests. The sources of the lift and drag forces are the wings, the tail, and the body.

On the glider, the largest sources of lift and drag are the wings. To measure these forces, a single wing was mounted on a two-axis force sensor and placed in a wind tunnel with a fluid velocity of 4 m/s. The wing was then rotated through angles of attack from  $30^\circ$  to  $-30^\circ$ , and the lift and drag forces were measured at each angle. Figure 3 shows the results of these measurements. As expected, within a certain range the lift forces vary roughly linearly versus angle of attack. The drag forces increase more dramatically as one deviates from zero angle of attack.

The second source of lift and drag is the tail section. Unlike the wings, the resulting tail forces vary on the state of the tail, that is, the position of the two elevons. In order to measure this variation, a tail section was mounted in the wind tunnel in much the

same way as the wing. Figure 4 shows the resulting forces generated by varying the deflection of the elevon.

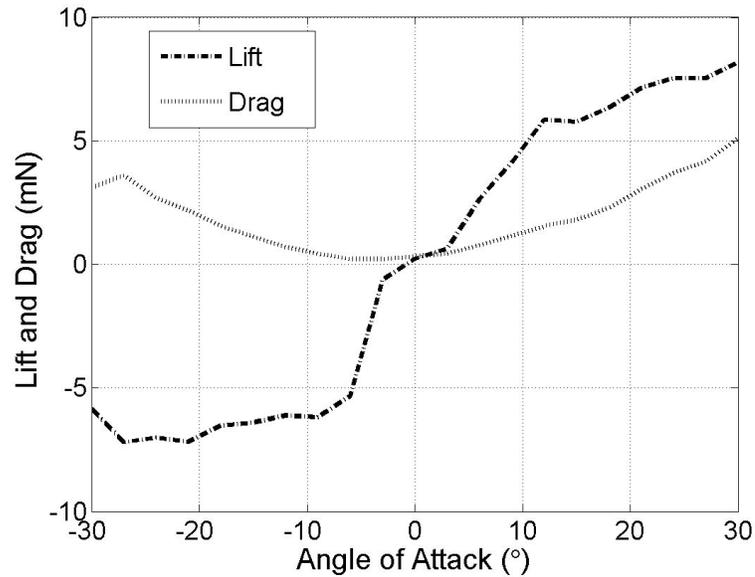


Figure 3: Lift and drag forces of wing-body as a function of angle of attack in 4m/s wind tunnel

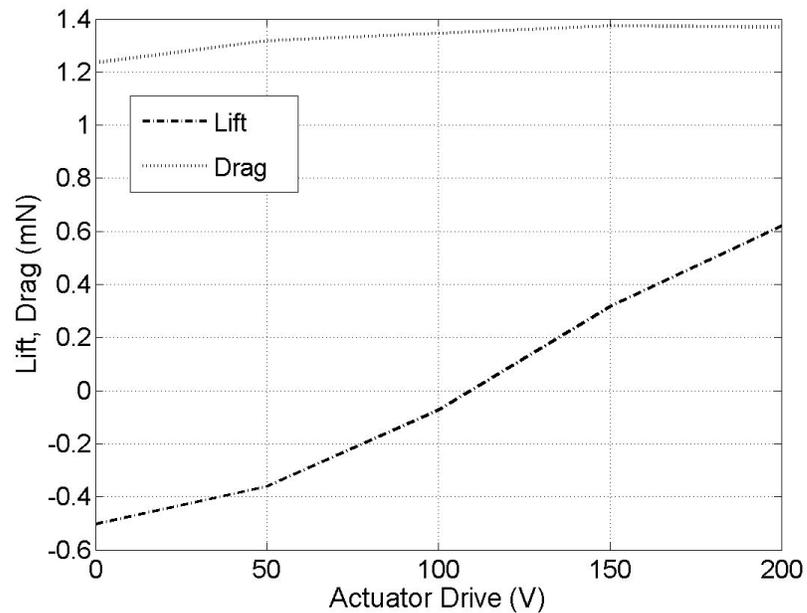


Figure 4: Elevon lift and drag versus applied voltage in 4m/s wind tunnel

Finally, the body of the glider is another source of lift and drag. However, since the body area is much smaller than either the tail or the wings, the lift produced by the body is negligible, and is ignored. The drag produced by the body is similarly ignored.

With the lift and drag forces measured for arbitrary angles of attack and elevon deflection, these forces are converted into lift and drag coefficients, as described above.

Taking the aerodynamic theory and computed glider parameters, the forces acting on the glider are computed by the simulator at each time step. These forces are the output of the aerodynamic section. Figure 5 shows the total aerodynamics module.

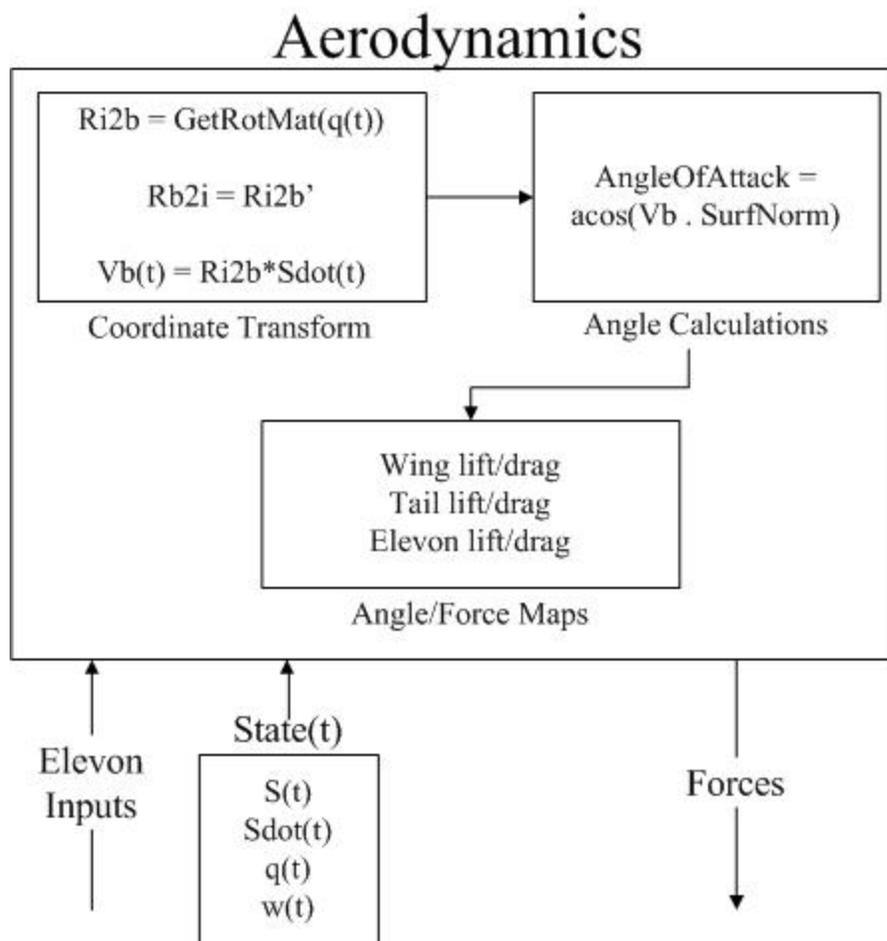


Figure 5: Aerodynamics module

From the glider rotational state  $q(t)$ , two rotation matrices are created that convert from inertial to body coordinates, and from body to inertial coordinates ( $Ri2b$  and  $Rb2i$ , respectively), as shown in this equation:

$$Ri2b = \left( q_4^2 - q_{1:3} \bullet q_{1:3} \right) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + 2q_{1:3} q_{1:3}' - 2q_4 \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix}$$

These matrices are then used to convert the net fluid flow velocity magnitude and direction from inertial to body coordinates ( $Vb(t)$ ). This velocity is used to calculate the angle of attack of the wing and tail section. The angles of attack are fed directly into equations described above that calculate the resulting lift and drag forces based on the angle of attack and the flow velocity. Finally, these forces, as well as the glider state, are fed as inputs into the body dynamics module.

## 1.2 Body Dynamics

The body dynamics section is the second main module of the glider simulator. This section takes the forces computed by the aerodynamic section, as well as the current state of the glider, and calculates the resulting accelerations, as well as the next state. The body dynamics section is made up of standard fixed-body kinematics equations, as well as measured and calculated glider parameters. Plugging these parameters into the kinematics equations allows the software to simulate the effects of forces on the glider. In addition to updating the state of the glider based on these forces, this section also calculates and applies rotational frictional forces based on glider movement.

The kinematics equations used in this section are all based on fixed-body assumptions. At each time step, the glider is assumed to be a fixed body, in that none of the relative positions of the glider components changes. This is done because the glider does not change shape significantly at normal flight speeds, and the resulting equations are much simpler.

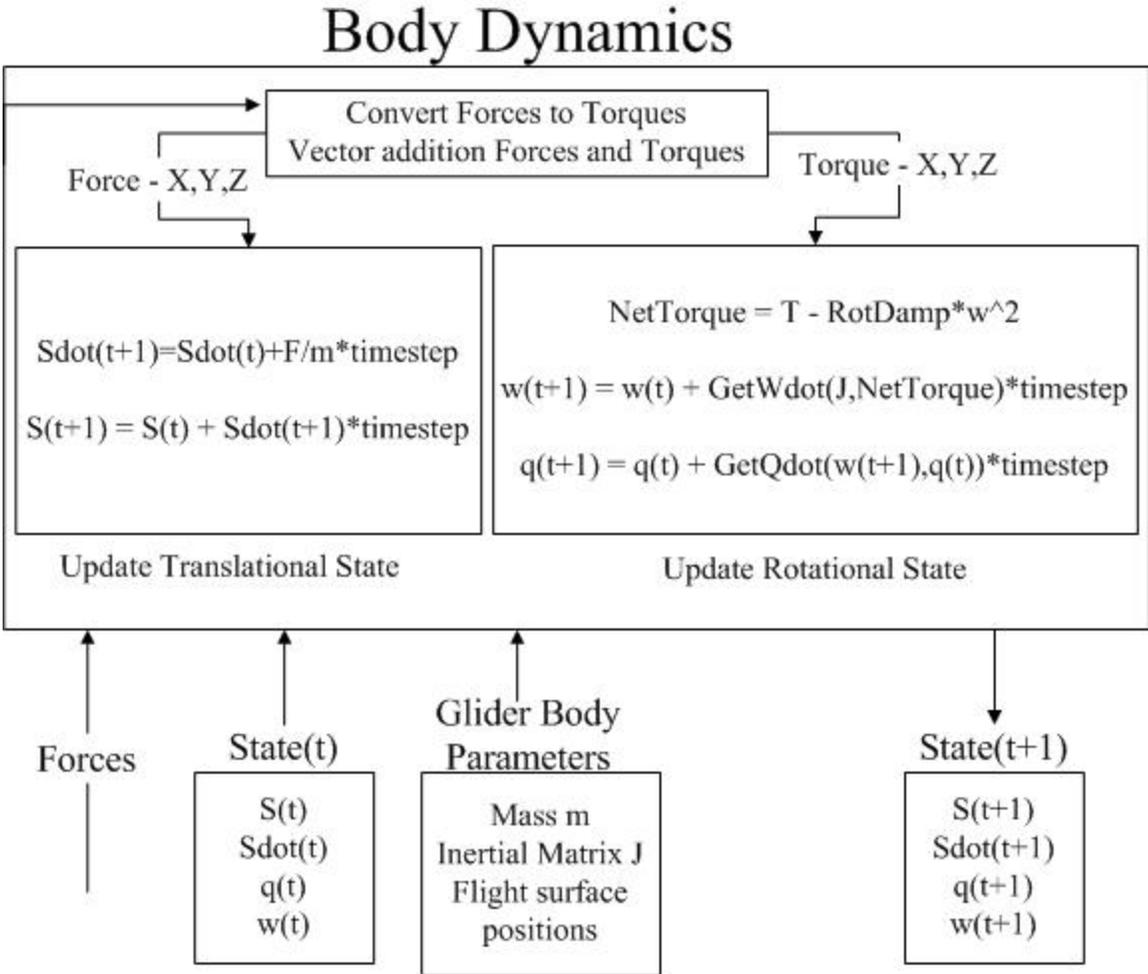


Figure 6: Body Dynamics module

The actual kinematics equations are based on Newton’s second law, both in translation and rotation. These equations give a translational and rotational acceleration based on the position, direction, and magnitude of the forces acting on a body, as well as the distribution of mass of the body.

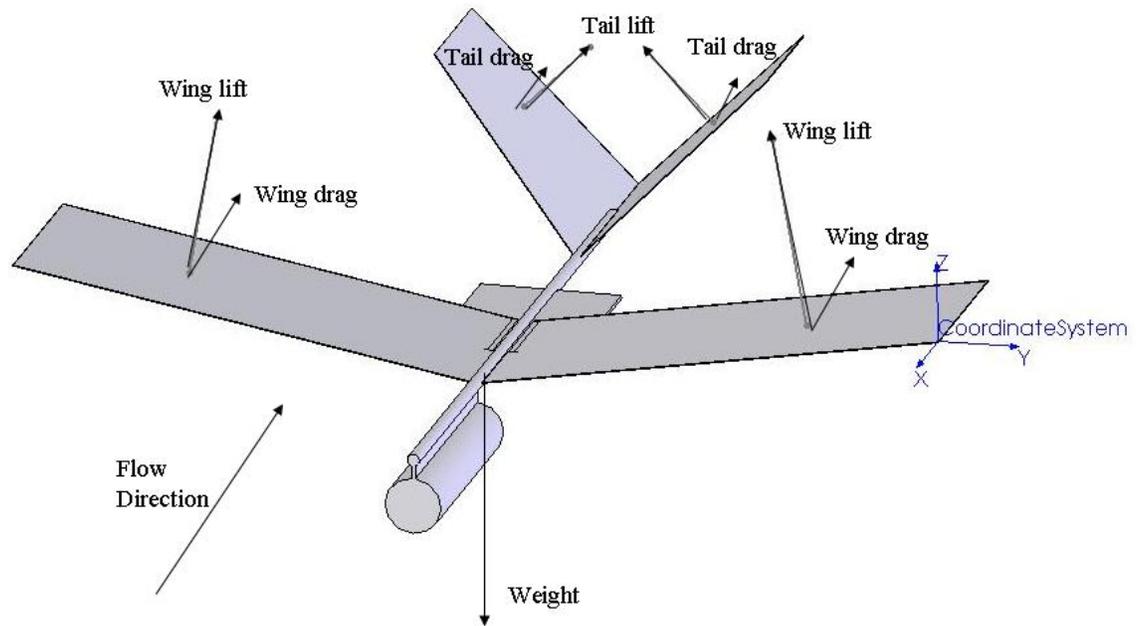


Figure 7: Solidworks model of glider, used to calculate glider inertial matrix. Force vectors show position and direction of calculated forces

The necessary glider parameters are glider mass and the distribution of that mass. The glider mass is measured directly using a scale. The mass distribution, on the other hand, was measured using a SolidWorks model of the glider, as shown in Figure 7. After constructing a model of the glider in SolidWorks, SolidWorks returns the mass distribution, in the form of an inertial matrix, as well as the position of the center of mass. This inertial matrix is scaled appropriately to the mass of the glider, and is plugged directly into the kinematics equations. The resulting inertial matrix is as follows

$$J = \begin{bmatrix} 328.3 & -0.9366 & -162.1 \\ -0.9366 & 1008.3 & 0.2283 \\ -162.1 & 0.2283 & 1238.7 \end{bmatrix} g * mm^2$$

as measured from the center of mass and in line with the glider body coordinates.

The kinematics equations and glider parameters allow the simulator to calculate the accelerations based on the forces acting on the glider. These accelerations are applied to the glider state in two methods. For translation, Newtonian approximation is used directly

in the  $x$ -,  $y$ -, and  $z$ -directions. For glider rotations, the glider state is stored and updated using Newtonian approximation on quaternions.

Translational accelerations are easily handled by multiplying the acceleration by the time step to get a change in velocity. This change is then added to the current velocity to compute the new glider velocity. Multiplying the velocity by the time step then gives the change in position. Adding this change to the current position gives the new position of the glider. Since position, velocity, and acceleration are all separable 3-dimensional quantities, this Newtonian integration scheme gives very good results. The appropriate equations for updating translational state are found in Figure 6.

Glider rotation is not handled the same way as translation because rotational position is not a 3-dimensional quantity, since angles cannot be independently separated. Instead, rotational state occupies  $SO(3)$ . Because of this, applying Newtonian integration directly to the three angles of rotational position results in errors adding up incrementally, instead of canceling out over time as in translation. This can lead to such strange effects as rotating bodies increasing their angular momentum over time, without applied forces. Therefore, glider rotational state is stored using quaternions.

Quaternions allow a much better method of updating glider state. The angular position of the glider is stored using a 4 element vector, and the angular velocity is stored using a 3 element vector. At each time step, the torque on the glider is converted to a change in angular velocity by means of the `GetWdot` function, which implements this equation:

$$\dot{w} = J^{-1}(T - w \times Jw)$$

The resulting angular acceleration is multiplied by the time step to give a change in angular velocity. This value is then added to the current angular velocity to give a new angular velocity. From this new angular velocity, the angular position is updated using the `GetQdot` function:

$$\begin{aligned}\dot{q}_{1:3} &= 0.5 * q_4 * w - w \times q_{1:3} \\ \dot{q}_4 &= -0.5 * w^T q_{1:3}\end{aligned}$$

The resulting undamped body dynamics yields a system in which the angular momentum of the system does not increase or change significantly over time. This was tested by rotating a random object in simulation and calculating the angular momentum using the following equation.

$$angular\_momentum = \sqrt{w'JJw}$$

Rotational damping forces are also calculated and applied in the body dynamics section. These are added to the torque just before the glider state is updated. The damping forces always resist the current rotational motion, and are calculated as proportional to the square of the current angular velocity, times a damping matrix RotDamp. The appropriate values of RotDamp are very difficult to measure. Therefore, these parameters were adjusted by comparing the simulator output to actual flight tests. This will be explored further in Chapter 2.

## 1.3 Sensors

The final module is the sensor module, which takes in the state of the glider as well as the simulated environment and outputs the expected sensor measurements. The two primary sensors simulated are the ocelli sensor and the optical flow sensor. This reflects the sensors currently implemented on the microglider. Unlike the glider aerodynamics and body dynamics modules, which are calculated at a frequency of 1 kHz, the sensor module is updated at a frequency of 100 Hz.

### 1.3.1 Ocelli

On the glider, ocelli are simple sensors that measure light levels and output a continuous voltage, proportional to the intensity of the light. The response of the ocelli is nearly instantaneous, and on the glider, their use is only restricted by the sample rate of the glider microprocessor. Because the ocelli is a real-time sensor with no memory element or hysteresis, its response can be calculated easily at any given time in the simulator.

The ocelli sensor is simulated by applying the measured ocelli response parameters and programming them into the simulator. The two key parameters are a map of the light entering the ocelli as a function of light direction, distance, and intensity, and a map of the output voltage as a function of light input. However, in order to simplify the simulator, these two functions were combined, and the resulting function gives a map of output voltage as a function of light direction, distance, and intensity.

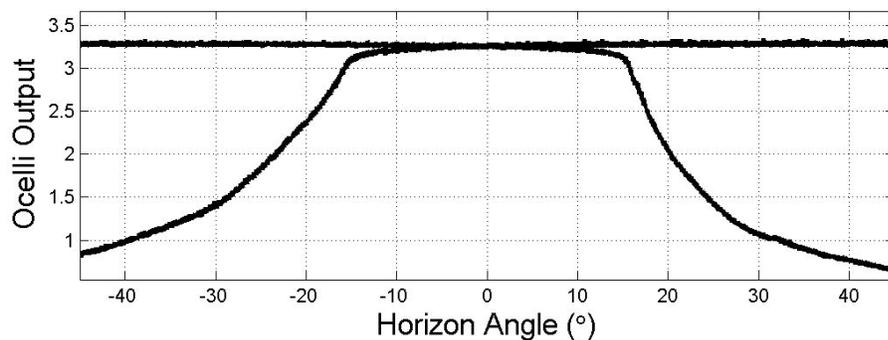


Figure 8: Output in volts for two ocelli at  $90^\circ$  to each other versus angle to light source (horizon angle)

In order to obtain this function map, a combination of direct measurements and basic theory were used. The three degrees of freedom on the input were light direction, distance, and intensity. The response based on changes of light direction was measured by measuring the output voltage of the sensor when a light source was rotated about 180 degrees with respect to the sensor, the maximum angle that the sensor can respond to. The result of this measurement can be seen in Figure 8. As one can see, the ocelli selected are highly directional. The response based on light distance was calculated based on the fact that light from a point source decreases in intensity as a square of the distance to the source. Knowing that ocelli are designed to be roughly linear with respect to intensity, this fact allows one to combine distance and source intensity and calculate the resulting output voltage.

Since direction is independent of intensity, these function maps were combined to form a single function that calculated the ocelli output based on any light source at any position. The ocelli sensor module takes this function and inputs the required angles and

intensities, based on angle calculations of the simulator rotational state versus the light position. Thus the ocelli sensor module takes in the state of the glider, the positions of the light sources, as well as the positions of the ocelli on the glider, and outputs the voltage registered at each ocelli.

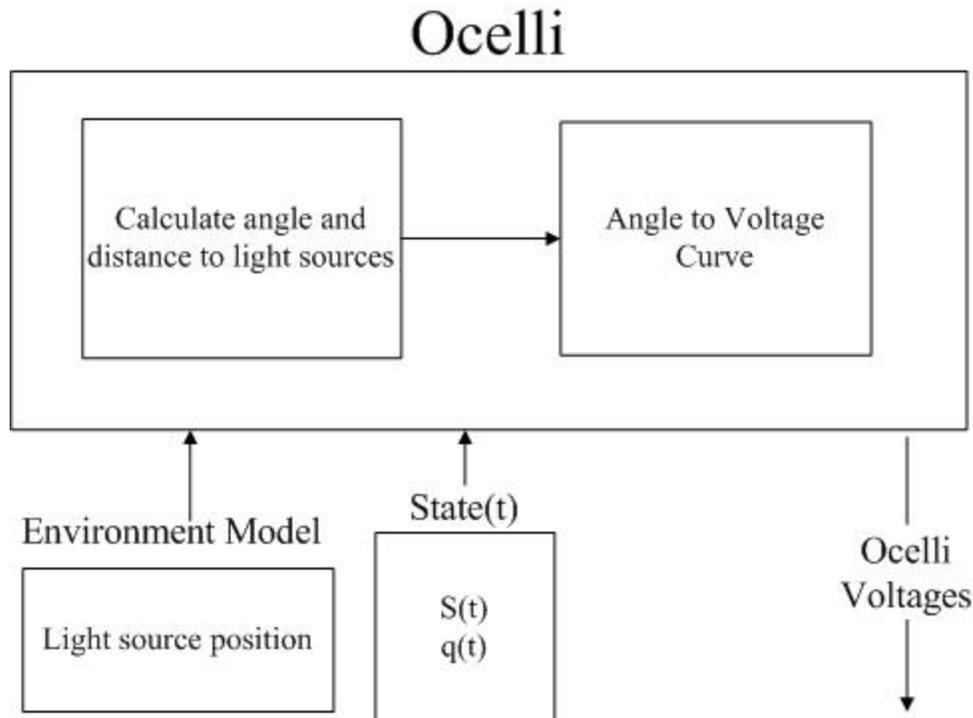


Figure 9: Ocelli sensor module

### 1.3.2 Optical Flow Sensor Module

The second and last sensor in the sensor model is the optical flow sensor module. Optical flow is the measurement of velocity of objects across a visual field of view. The closer an object is to the plane, the faster it moves across the visual field. The result is a vector map of optical flow. Figure 10 shows the sample optical flow from an airplane approaching some rocks. The as the plane moves forward, the approaching rocks cause large vectors in the forward field of view.

Optical flow measurement allows the glider to 'see' objects and obstacles in the environment, with much lower computational requirements. This means that the glider can

process the sensor output in real-time, and react to and avoid obstacles in the environment. This module is implemented in Matlab in two segments, rendering and image-processing. The entire module breakdown can be seen in Figure 11.

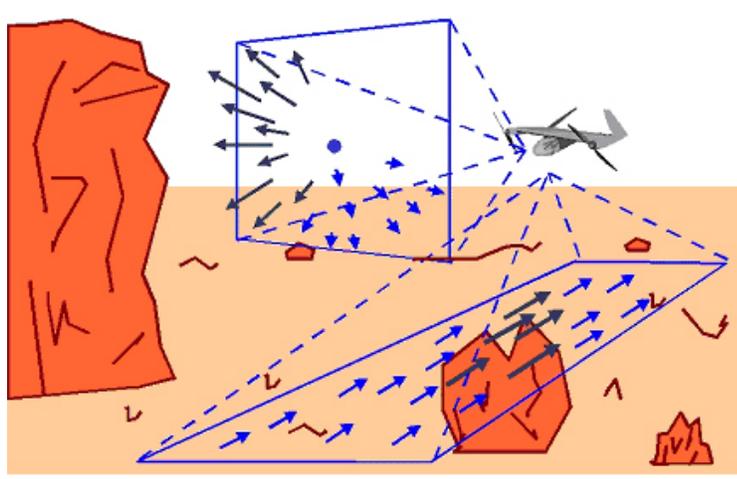


Figure 10: Optic flow as seen by an airplane approaching a set of obstacles [4]

Rendering consists of taking the glider state, as well as the environment, and simulating the image plane as seen by the optical flow sensor on the glider. In simulation, the environment is made up of 3-dimensional blocks covered in a randomized grid pattern. A grid pattern was chosen for its high degree of optical flow. The position and dimensions of each block, as well as the size of the grid can be adjusted arbitrarily, and the colors of each grid square are randomly assigned. Each block is rendered using a function called `MakeBlock`, and a sample rendered scene is shown in Figure 12.

Using Matlab's built in 'camera' functions, a camera position is defined by the translational position of the glider, and the camera direction is defined by the rotational position. At each timestep, the simulator displays the scene view seen by the glider, and that view is fed into the image-processing module, using the `GetFrame` command. While Matlab requires the image plane to actually be displayed on screen, a potential time waster, this method was seen as much easier and faster than writing custom ray-tracing code to calculate the correct image map.

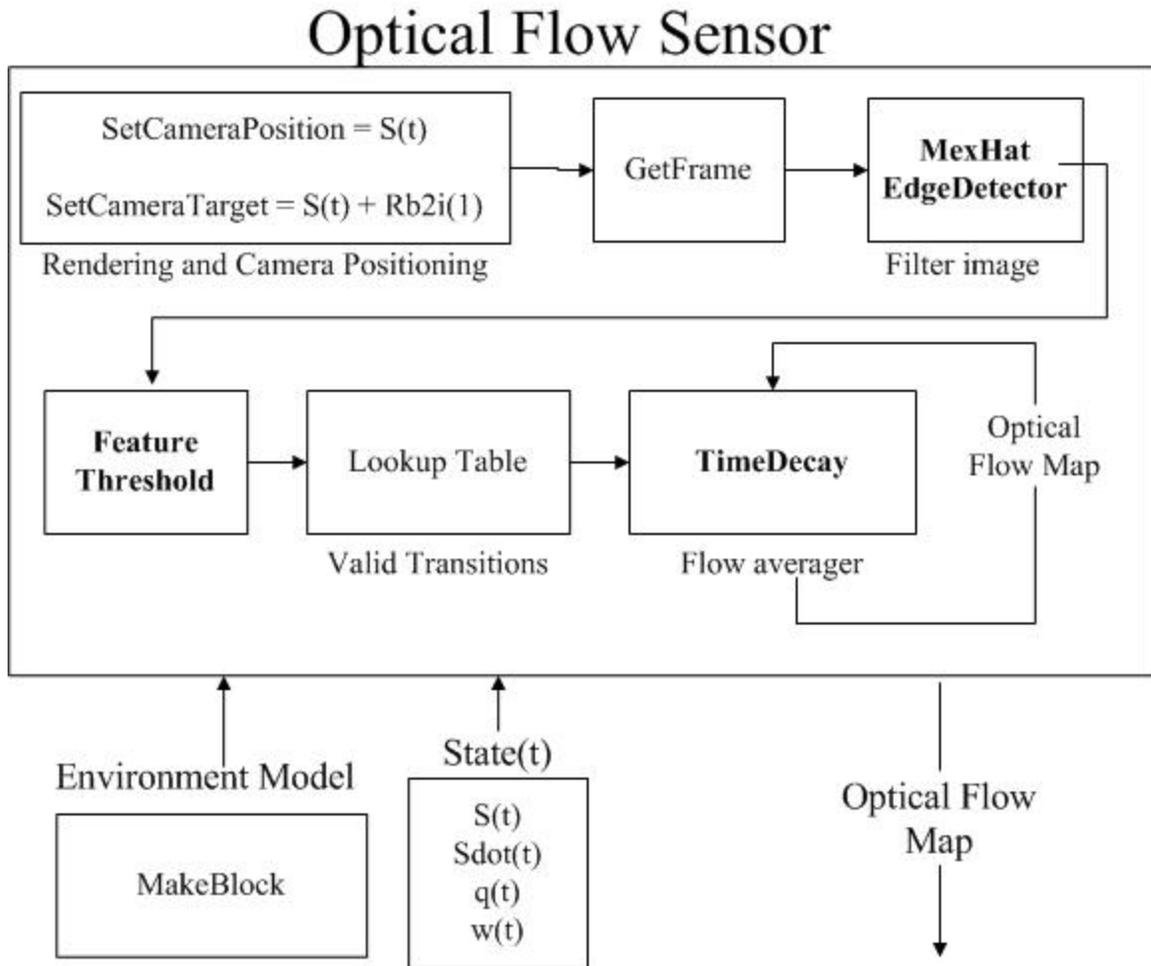


Figure 11: Optical Flow Module. Shown in bold are the three parameters that can be independently adjusted to change the module behavior - the filter type, the FeatureThreshold, and the TimeDecay of the flow averager.

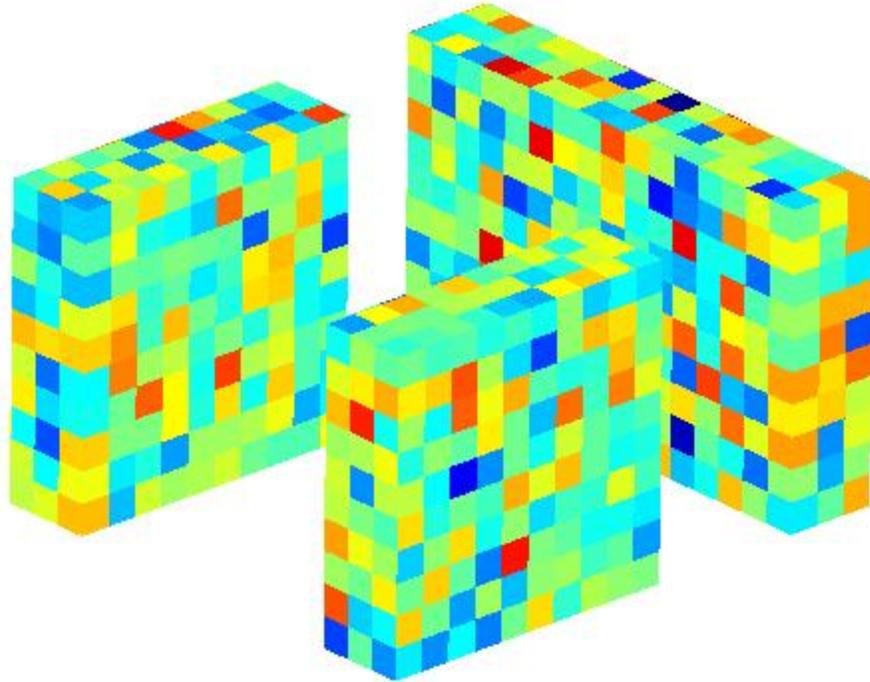


Figure 12: Sample rendered 3D scene

The image-processing module is the section that simulates the output of the optical flow sensor on the glider. This module is based on the chosen sensor, a custom-built optical flow sensor built by Dr. Geoffrey Barrows of Centeye. However, since the sensor itself is proprietary, the sensor module was designed based on the public patents of Dr. Barrows' work. The primary references for this module are [5] and [6].

The optical flow sensor is based on feature-tracking. The sensor reads in light levels, and applies a filter across the visual plane to detect features. The filters used can be set to be such filters as a saddle filter, a Mexican-hat filter, or an edge filter, as shown in Figure 13. Once the visual plane is filtered, each pixel of the output is compared to a threshold value, designed to separate a valid feature from the visual noise. Features whose value is above the threshold are set to 1, and features below the threshold are set to zero. This results in a feature-map for each input frame.

$$A: \frac{1}{4} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

$$B: [1 \quad -1 \quad -1 \quad 1]/4$$

$$C: [1 \quad -1]/2$$

Figure 13: Sample feature detection filters. A – Saddle filter.  
B – Line filter. C – Edge filter

Each frame is processed using this filter-threshold technique. Given the feature-maps of two consecutive frames, the sensor now searches for valid transitions. These transitions are based on the Horridge algorithm [7]. A 'valid' transition is one in which a feature clearly moves from one pixel to an adjacent pixel from one frame to the next. The purpose in looking for valid transitions is to eliminate cases in which it is unclear in what direction features have moved. For example, if a single feature appears to split in two directions, then the difficulty in possibly distinguishing which direction is the true direction is outweighed by the need for a fast, simple algorithm. In this case, the feature transition is ignored, and no optical flow is recorded.

Valid transitions are separated from invalid transitions by means of a Boolean function, or lookup table, as shown in Table 1. This table shows the calculated optical flow for a set of three pixels from two consecutive frames, based on the principle of valid transitions. For example, if the first frame shows 010 and the second frame shows 001, it appears that the transition that began in the middle of the frame has moved to the right edge. The output of (0,1) means that zero flow occurred at the left pixels, and a rightward flow occurred at the right pixels. In simulation, the feature maps are separated at 3 pixel increments, and the values in the lookup table are recorded as the optical flow from one frame to the next.

| $\begin{matrix} 2^{\text{nd}} \\ \backslash \\ 1^{\text{st}} \end{matrix}$ | 000 | 001  | 011   | 111  | 101 | 100    | 110    | 010   |
|--|-----|------|-------|------|-----|--------|--------|-------|
| 000  | 0,0 | 0,0  | 0,0   | 0,0  | 0,0 | 0,0    | 0,0    | 0,0   |
| 001  | 0,0 | 0,0  | 0,-.5 | 0,-1 | 0,0 | 0,0    | -.5,-1 | 0,-1  |
| 011  | 0,0 | 0,.5 | 0,0   | -1,0 | 0,0 | -1,-.5 | -1,-1  | 0,-.5 |
| 111  | 0,0 | 0,1  | 1,0   | 0,0  | 0,0 | -1,0   | 0,-1   | 0,0   |
| 101  | 0,0 | 0,0  | 0,0   | 0,0  | 0,0 | 0,0    | 0,0    | 0,0   |
| 100  | 0,0 | 0,0  | 1,.5  | 1,0  | 0,0 | 0,0    | .5,0   | 1,0   |
| 110  | 0,0 | .5,1 | 1,1   | 0,1  | 0,0 | -.5,0  | 0,0    | .5,0  |
| 010  | 0,0 | 0,1  | 0,.5  | 0,0  | 0,0 | -1,0   | -.5,0  | 0,0   |

Table 1: Look-up table for optical flow output due to successive feature frame triplets. 0 represents no flow, positive number mean rightward flow, negative numbers mean leftward flow

Unfortunately, since all valid transitions involve a feature either moving one pixel in one frame or staying at the same location from frame to frame, only optical flow direction is calculated, and magnitude is very poorly calculated. Therefore, to estimate the magnitude of optical flow, the valid transitions are averaged over time, using a low pass filter implemented with a first-order leaky capacitor model.

Each pixel of the visual field is thought to be a capacitor, with a voltage across it corresponding to the magnitude and direction, positive or negative, of optical flow. When no new optical flow is recorded, this capacitor is in a closed circuit with a resistor, resulting in an exponential decay of voltage and therefore, optical flow. However, each time that new optical flow is recorded at that location in the visual field, a constant voltage source is added in series with the circuit, resulting in the voltage across the capacitor increasing.

The result of this setup is that if optical flow is never recorded at a given pixel, the voltage across the capacitor is zero, correctly showing zero optical flow. If at every timestep optical flow is recorded at that pixel, then the voltage across the capacitor will

reach the voltage of the battery, interpreted to be an optical flow of one or 0.5 pixels per frame, depending on the value in the lookup table.

In the case of optical flow of less than one pixel per frame, one can expect that the occurrences of a valid transition across a pixel to occur proportionally to the optical flow. This means that the battery will sometimes be charging the capacitor, and sometimes the capacitor will be discharging. However, as the circuit acts as a low pass filter, the average voltage across the capacitor will be equal to the actual optical flow.

In the physical sensor, these calculations are done with actual resistors and capacitors. This setup is mimicked in simulation using basic circuit theory.

The output of these low-pass filters is the output of the optical flow sensor module.

With this basic structure, the sensor can be adjusted in 3 key ways: the chosen feature detection filter, the feature threshold, and the time constant of the low-pass circuit. These changes are highlighted in Figure 11. The effects of each of these three parameters are explored further in section 2.2.2.

## Chapter 2

# System Integration and Flight Tests

With the passive glider simulated, the simulator was then used for controlled flight tests. The glider was tested and controlled with both open and closed loop methods.

### 2.1 Open Loop Control

Open loop methods were used for the following three purposes:

- To **validate** the simulator
- To improve the basic **glider design**
- To design **turning strategies** in preparation for closed loop methods

These methods were accomplished both in simulation and with the use of the flight tests with the physical glider, recorded via a high-speed camera at 125 frames per second.

Simulator validation consisted of calculating and adjusting the simulator parameters to make simulated flights match the results from physical flight tests. These parameters include the body rotational and translational drag and damping coefficients, as well as the tail section lift parameters. These parameters were not known exactly because of their inherent difficulty in measurement. By adjusting these parameters in simulation,

the simulator could be tuned to match the glider's behavior with respect to glide slope, speed, stability, and turning radius.

With the simulated glider performing as expected, the simulation was then used to improve glider performance. By adjusting parameters in simulation such as wing size and position, weight, center of mass, and flight surface positions, the simulator was able to compute the expected flight of an arbitrary glider, allowing the glider to be optimized much faster than with traditional flight tests.

Finally, open loop methods were used to prepare turning strategies in preparation for closed loop control. As will be discussed later, many of the closed loop control methods to be tested required stable, open loop turns that can be linked together. Designing these turning strategies meant determining appropriate elevon deflection over time to initiate, continue, and complete a turn.

There were three types of open loop tests, each fulfilling one or more of the three purposes outlined above.

- **Passive Tests** – validation and glider design
- **Timed Response Tests** – validation, glider design, and turning strategies
- **Triggered Response Tests** - validation

Passive tests consisted of launching the glider at various initial conditions, and noting the transient response with fixed elevon deflection. Timed response tests consisted of launching the glider, and then applying known elevon deflections over a period of time. The final type of tests, triggered response tests, consisted of a known elevon deflection occurring based on a sensor trigger, in this case an ocelli voltage threshold being exceeded.

### **2.1.1 Passive Test Results**

Passive tests of zero elevon deflection established the baseline of simulator and glider performance. The first key condition of the microglider is stability. Since the glider is very minimally controlled, it must be passively stable in flight. This means that the pitch angle must stabilize to a fixed glide slope, and that the roll angle should be stable

around zero degrees. This proved to be a difficult first step, both in simulation and in reality.

Using multiple flight tests, the microglider was eventually designed to be passively stable. This allowed measurement of glide slope and glide velocity. For a glider of mass of 1.632 grams, the flight tests revealed a glide slope of 2.96 to 1 ( $18.67^\circ$ ), at a velocity of 5.126 m/s. Taking these measurements, the remaining unknown parameters were adjusted until the simulator had the same glide slope and velocity at a given weight. In addition, the settling time of the simulator pitch angle was verified to be within the same order of magnitude to the settling time of the microglider in flight. Figure 13 shows the microglider reaching its stable glide slope in simulation after a horizontal launch.

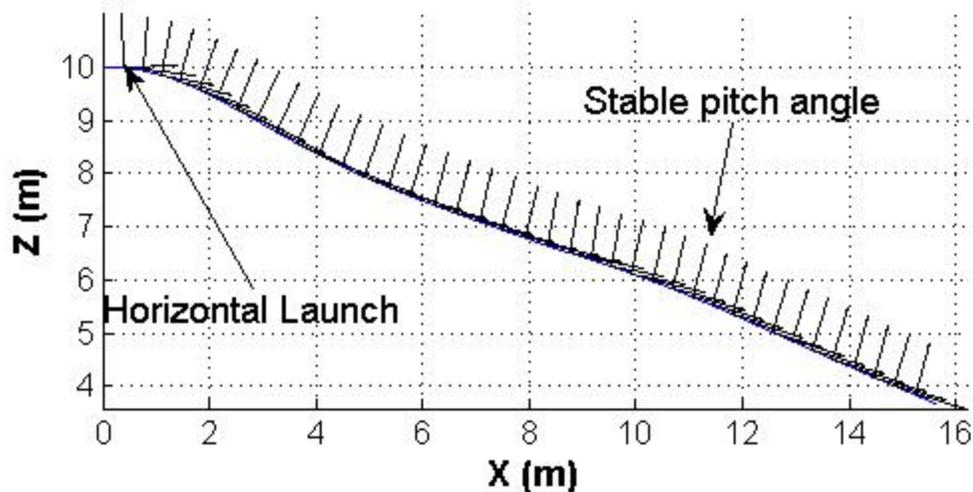


Figure 13: Glider pitch stabilization from horizontal launch to steady state glide angle of  $23.1^\circ$  and velocity of 4.19m/s. Displayed is the glider coordinate system every tenth of a second.

Passive tests were then used to improve these same measurements on the glider. The position of the center of mass proved to be of critical importance in simulation, as was shown in later flight tests. The closer the center of mass was to the nose of the glider, the steeper the glide slope became, causing the glider to nose-dive. However, bringing the center of mass towards the rear of the glider caused the glider to become increasingly unstable. This tradeoff was established early in simulation, and became a critical design

parameter on the glider. In the end, the position of the center of mass was chosen to be 8mm ahead of the wing center chord.

Further tests were run to see if the glider could be optimized for speed. By adjusting the weight and position of the center of mass, as well as the position and size of the wings and tail, the simulated glider ground speed was increased to 13 m/s, based on a glide slope of 2:1 and a velocity of 14.5 m/s. Figure 14 shows the glider glide slope and velocity as a function of mass and wing size, as well as the resulting ground speed. As can be seen on the figure, increasing the glider speed to this level requires a much heavier glider, with significantly smaller wings. The resulting glider would be very unstable, and most likely uncontrollable. However, these estimates did serve to help define potential glider uses.

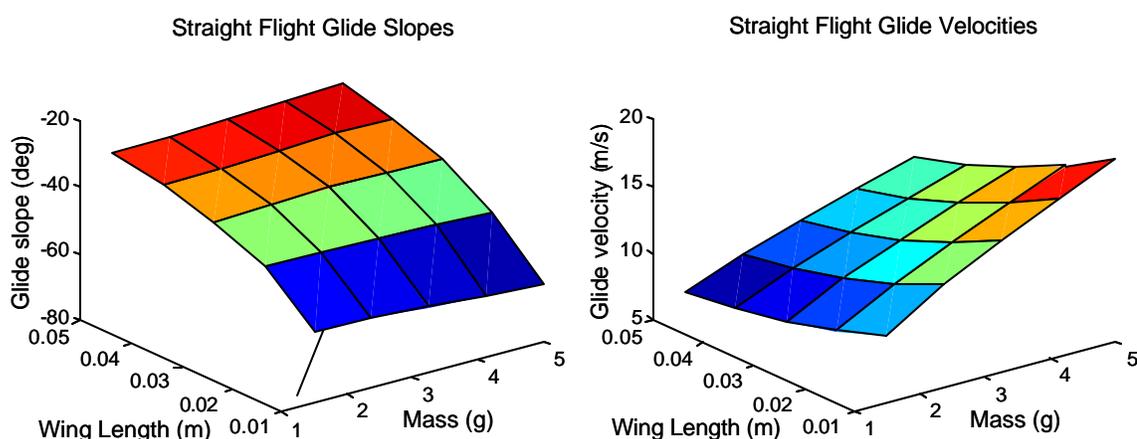


Figure 14: Calculated glide slope and glide velocity for various glider configurations

Passive flight tests also verified the roll stability of the glider. Due to a positive wing dihedral of 10 degrees, the glider was laterally stable. For example, a positive roll of the glider to the right resulted in the right wing having a larger effective angle of attack, and therefore more lift and drag, while the left wing had less lift and less drag. This resulted in a negative yaw to the right, as well as a resulting negative roll moment to the left that negated the original roll. While this stability was much less pronounced than the

pitch stability, it still meant that the glider would not continue to roll uncontrollably following a small perturbation.

### 2.1.2 Timed Response Results

Timed response tests were used for all three purposes of the open loop tests: validation of the glider during turns, improvement the glider turning performance, as well as optimization of the turning strategies. Noting the glider behavior during turns allowed the adjustment of the simulator given the same inputs. The primary parameter adjusted during this phase of the validation was the rotational damping coefficients, which are very difficult to measure on the glider itself. This process was primarily trial and error, but resulted in parameters that appeared to be reasonable, based on flight tests. The resulting rotational damping matrix is shown here, and is applied as shown in Figure 6. Since this term represents the percentage decrease in rotational velocity over a time step, it is dimensionless.

$$RotDamp \equiv \begin{bmatrix} 15 & 0 & 0 \\ 0 & 15 & 0 \\ 0 & 0 & 10 \end{bmatrix} \times 10^{-3}$$

Using timed response tests also allowed measurement and optimization of glider turning performance. The glider turn radius was tested for various wing sizes and weights, to get a sense of the effects of these adjustments on performance.

Designing turns was also a primary purpose of the timed response tests. Early on, it was discovered that for certain glider configurations, a steady, step-input elevon deflection in one direction would cause the glider to go unstable, flipping over or pitching wildly. Since this resulted in unpredictable behavior, it was decided that turns must be limited to keep the glider in its region of stability. One type of turn that kept the glider stable was called a 'turn primitive.'

Turn primitives are turns that begin and end in a horizontal, stable position, with only the yaw angle different. By ending in the same stable state as they began, except

rotated in the yaw direction, these turn primitives can be done one after another without causing the glider to go unstable.



Figure 15: Turn primitive. From left to right 1. Roll right; 2. Pitch up; 3. Roll left and stabilize, ready for new primitive

Each turn primitive consisted of three phases, the roll entry, the pitch up, and the roll out, as shown in Figure 15. The roll entry has one elevon high and one elevon low, creating a difference in elevon lifts which causes the glider to roll. The pitch up phase has both elevons high, decreasing the lift generated by the elevons and causing the glider to pitch while rolled and continue the turn. The final phase is the roll out, which has the glider reverse the elevon positions of the roll entry and causes the glider to rotate back to stable, zero roll position. Since the roll stability is relatively weaker than the pitch stability, the timings of these three phases is very important to ensure that the glider reaches a nearly level state as quickly as possible after each turn.

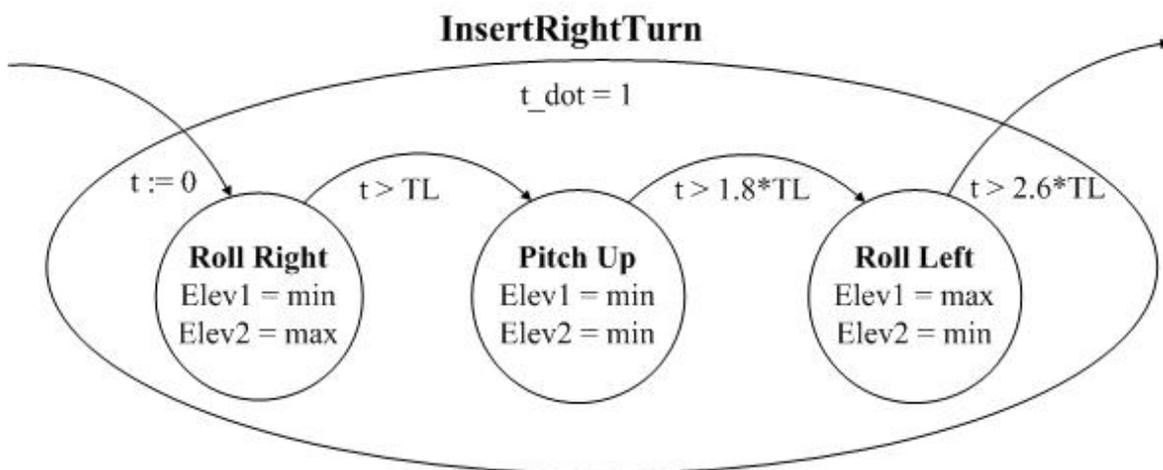


Figure 16: Turn primitive state machine for right turn. A left turn is enabled by switching elevons 1 and 2.

Defining the length of the initial roll as the Turn Length (TL), the final optimum turn primitive becomes a roll of 1 TL, a pitch of  $0.8 \cdot TL$ , and a roll out of  $0.8 \cdot TL$ . The Turn Length can be varied from 0 to a half a second to enable turns of up to nearly 90 degrees. Figure 17 shows the turn angle for various values of TL.

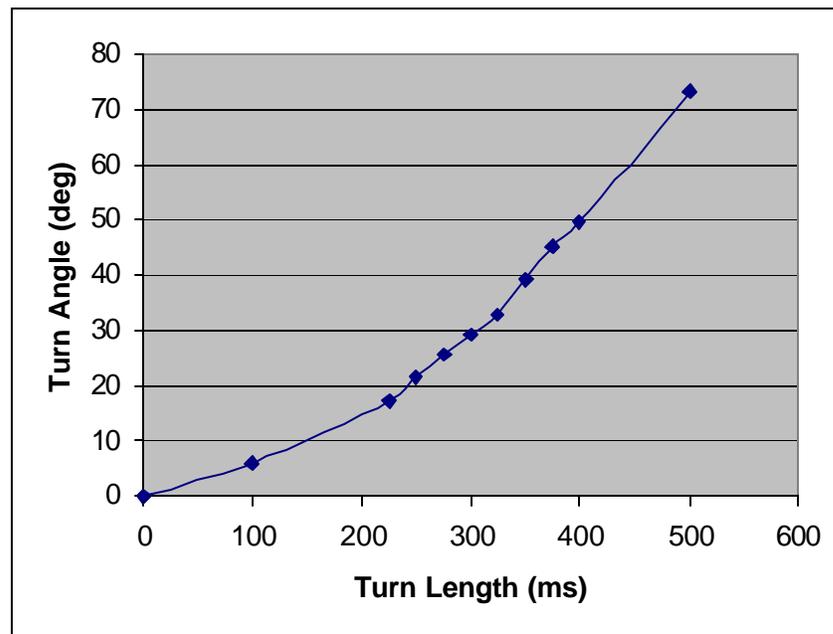


Figure 17: Turn Primitive angle vs. turn length

### 2.1.3 Triggered Response Test Results

The use of triggered response open loop tests was minimal, and was limited to determining the response of the glider moving past a fixed light source. These tests were the first to use the ocelli sensor module, and helped to validate its correct implementation in the simulator. A key result of these tests was to see that the angle of the glider versus the light source was very important to ensure that the ocelli would be able to correctly detect the source. For example, if the source was too low, the ocelli may not see the source at all.

## **2.2 Closed Loop Control**

Closed loop control tests consisted of flight control based on feedback from the ocelli or optical flow sensor. Each of these sensors gives very different information with different levels of accuracy, as well as for different purposes.

### **2.2.1 Ocelli Control**

The original purpose of the ocelli on the glider was to serve as a test sensor for target tracking. A fixed light source was placed in the environment, and the glider was to turn progressively closer to the light source. These tests were a natural extension of the triggered response tests, and served primarily as further validation of the glider and sensor suite.

### **2.2.2 Optical Flow Control**

The purpose of the optical flow sensor on the glider is to enable obstacle avoidance. If the glider is approaching an obstacle, the optical flow will increase in the field of view, signaling a potential collision. The glider can then react to that obstacle by turning using a programmed turn primitive. This is a very similar method to the one that many insects use during flight. A fly, for example, perceives the world visually through optical flow, and is able to avoid hitting obstacles by turning sharply away from walls (saccading).

The state machine controlling this behavior is shown in Figure 18. Normally the glider remains in the state Scan For Obstacles. In this state, optical flow measurements are taken at 100 Hz and compared to the optical flow threshold. When the optical flow exceeds this threshold, the optical flow is turned off, and a turn primitive is initiated in the opposite direction of the flow. Once the turn primitive is finished, the glider reaches the Pause state, where it remains for a small period of time while the passively stable dynamics of the glider return the glider to straight and level flight. Once this is achieved, the glider returns to scanning for obstacles.

The optical flow sensors are turned off during and slightly after the turning phase in the same way optical flow is turned off during the flight of a fly. While turning, optic flow due to rotation greatly exceeds the translational optical flow. However, because the sensor has a limited viewing plane, the distinction between rotational and translational optical flow is nearly impossible to identify. Therefore, while turning the optical flow sensor becomes swamped by the rotational flow, and further obstacle detection is not possible.

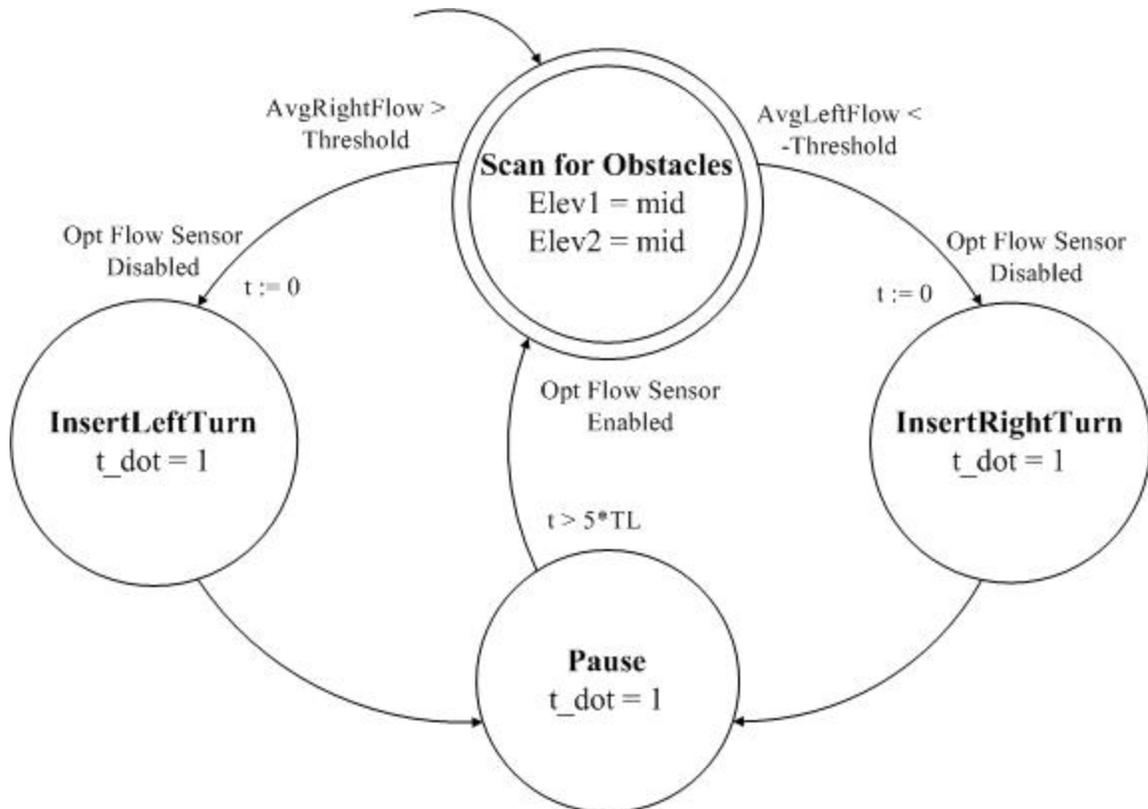


Figure 18: Hybrid obstacle avoidance control strategy using optical flow

Given this control strategy, the simulated environment for the optical flow tests was a virtual hallway, with very large, textured walls. The hallway setup permitted the glider to attempt multiple obstacle collisions with the wall as the glider attempted to fly through the hallway. In addition, the distance that the glider flew down the hallway was an easy, consistent metric that allowed for flight test comparisons. The longer the flight, the better the glider had succeeded in avoiding the walls, and the better the settings had been.

With the glider in this virtual hallway, a number of key parameters existed for optimizing the optical flow based flight. These are

- Feature threshold
- Optical flow threshold
- Turn length

Each combination of these parameters was tested in the virtual hallway setup. Since consistency was an important concern, each flight test was run 3 times, with the random hallway texture as the only variation. In addition, the width of the hallway was also varied, in order to test the glider's capability in various environments. Figures 19 and 20 show a successful flight path down the tunnel, where the glider managed to detect and avoid the hallway walls.

Optimal values for each of these parameters was first approximated through repeated testing. Once a reasonable range for each parameter was found, the repeated trials were performed. Table 2 shows the data points that were tested. Finally, after each of the resulting 180 variations was tested through 3 flight tests, the distances traveled on those tests was averaged.

| Parameter              | Number of variations | Data points                            |
|------------------------|----------------------|--|
| Feature threshold      | 3                    | 0.25, 0.3, 0.35                        |
| Optical flow threshold | 5                    | 6, 7, 8, 9, 10<br>(*10 <sup>-3</sup> ) |
| Turn length (ms)       | 3                    | 250, 300, 350                          |
| Tunnel width (m)       | 4                    | 8, 10, 12, 14                          |

Table 2: Parameter variations tested

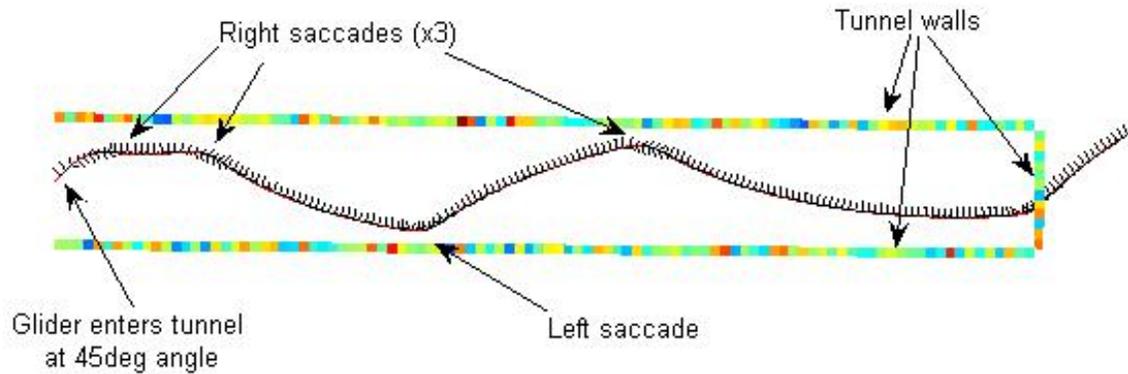


Figure 19: Aerial view of Glider flight path. Shown is coordinate system of glider every 10<sup>th</sup> of a second.

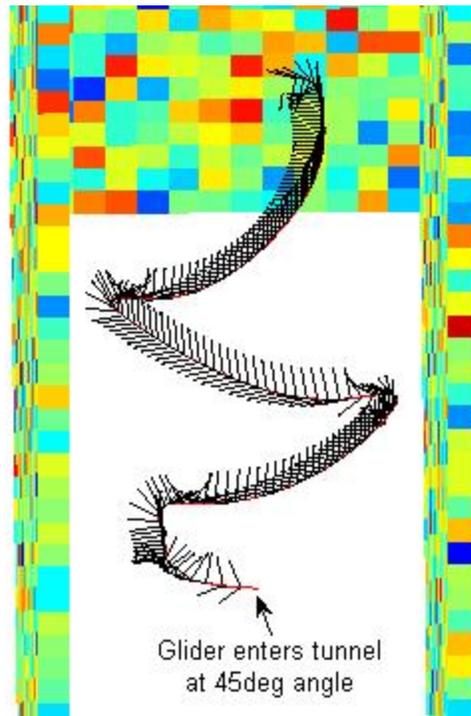


Figure 20: Top-down rear view of Glider flight path. Glider is flying away from camera towards back wall.

Based on these tests, trends were very difficult to quantify. The best results are shown in Table 3. These results suggest that a large turn length, coupled with relatively high thresholds yields the best results. Figures 21 through 23 show the range of tunnel distances traveled versus the various parameter combinations. As can be seen in the flight

test results, the glider system appears to be highly dependent on the correct parameter settings, and the resulting distances are highly variable.

| Average Tunnel Distance (3 trials) | Tunnel Width (m) | Turn Length (ms) | Optical Flow Threshold ( $\times 10^{-3}$ ) | Feature Threshold |
|------------------------------------|------------------|------------------|---|-------------------|
| 79.64                              | 12               | <b>350</b>       | 10  | <b>0.3</b>        |
| 78.65                              | 12               | 300              | 9   | <b>0.3</b>        |
| 71.27                              | 10               | <b>350</b>       | 8   | <b>0.3</b>        |
| 65.88                              | 14               | 300              | 8   | 0.35              |
| 48.74                              | 8                | <b>350</b>       | 10  | 0.35              |

Table 3: Parameter settings for longest 4 test flights down 100 m tunnel of various widths, as well as longest flight for narrow tunnel (8 m)

However, with correct parameter settings, consistently long flights are possible. Figures 19 and 20 show a glider flight with a turn length of 350ms, an optical flow threshold of  $9 \times 10^{-3}$ , and a feature threshold of 0.3. As can be seen in the figures, the glider successfully detects and avoids the walls 4 times before reaching the end of the tunnel.

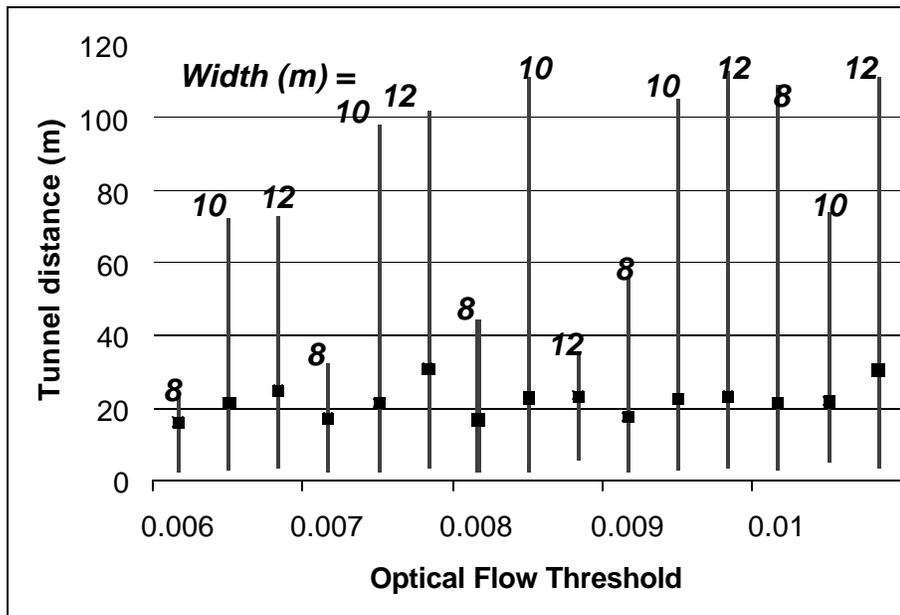


Figure 21: Range and average tunnel distances vs optical flow threshold for various tunnel widths ( $n = 27$ )

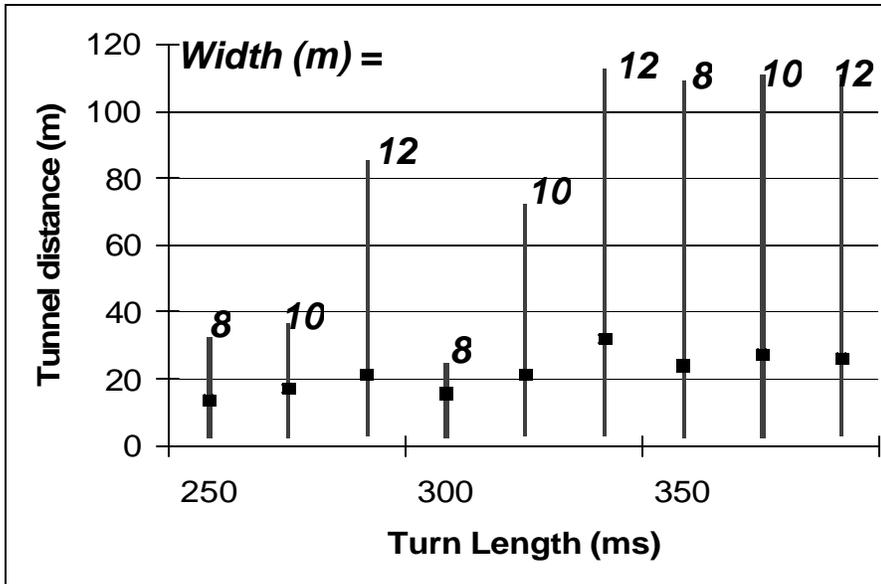


Figure 22: Range and average tunnel distances vs turn length for various tunnel widths (n = 45)

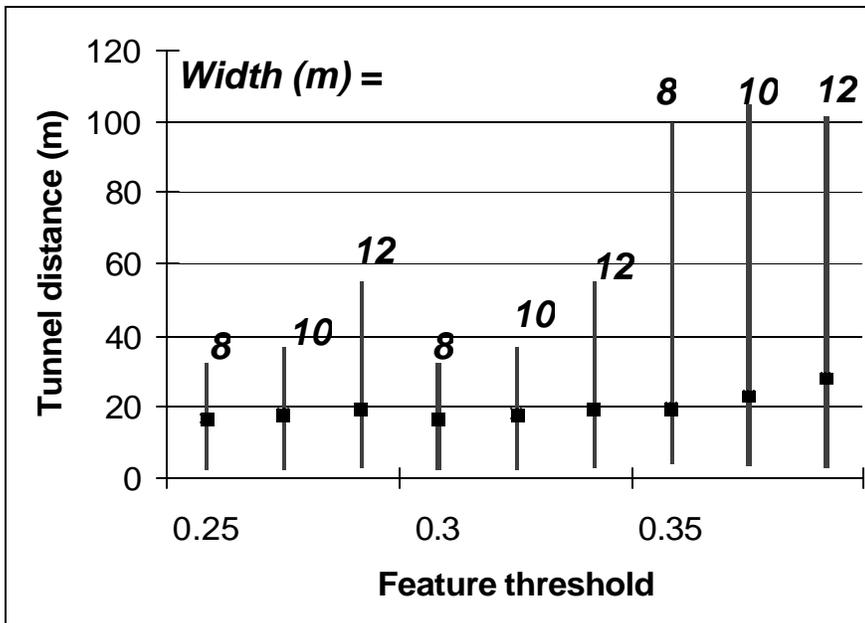


Figure 23: Range and average tunnel distances vs feature threshold for various tunnel widths (n = 45)

## **Chapter 3**

# **Analysis and Conclusions**

Presented here is a software tool that simulates the microglider in flight. Unfortunately, the discrete motion control strategy developed in this report has not yielded robust results. Even with small parameter changes, the flight distance traveled down a virtual tunnel can vary wildly. While some parameter combinations yield fairly good results, it is unclear what the parameter dependence is. This means that while designing a real glider, the entire parameter space may need to be searched in order to find an appropriate control strategy. In identifying the causes of these poor results, two areas must be addressed: the sensors and the flight platform.

### **3.1 Sensor Concerns**

One key result of this study is the importance of accurate sensing modules. The optical flow sensor simulated in software proved to be not nearly robust enough in its measurements in order to obtain reliable environment information. The sensor was highly dependent on environment conditions, required very accurate tuning, and in ideal

conditions was only able to give a very crude approximation of the horizontal and vertical optical flows.

First, the sensor itself was highly dependent on the environment conditions. During the flight tests, each parameter variation was tested 3 times, with the only difference being the coloration and texture of the tunnel walls. Although this change represents a small one considering the variety that a real autonomous glider would encounter, it made a significant difference in the performance of the sensor.

This dependence on texture is likely due to the method by which the optical flow sensor detects optical flow. All optical flow sensors are dependent on the texture of a scene to be high enough that motion can be detected. The most common way to measure the texture of a scene is by measuring the 2-dimensional gradient of the projected image of the scene. For example, a wall painted one solid color would be very difficult to detect by most optical flow sensors, and would be said to have low texture, since the gradient of the projected scene would be low. However, once the texture is above a certain threshold, the hope is that the optical flow would be measured correctly. Unfortunately, the optical flow sensor utilized in this study was not only limited by very low texture, but was also unable to detect very high texture.

The optical flow sensors made use of feature detectors, which looked for unique areas in the projected scene that were highly responsive to a given feature-detection filter. These filters only looked at 2 to 4 different points in the scene, making them effectively second or fourth-order filters. Because the filters were such low order, they were severely band-limited in their response to different textures. Thus, a texture that was too high or low would not be accurately detected by the filters.

Because of the dependence on environmental texture, the parameter tuning of the optical flow sensor became very important. With any given parameter setting, the glider was highly responsive at a given distance away from a wall. This meant that if the glider somehow ended up closer to the wall than that given distance, the optical flow due to expansion would out-weigh the optical flow due to the wall approaching at an angle. This often caused the glider to turn directly into a wall, instead of turning away to avoid it.

Finally, with regard to the sensors, the output of the sensors was very poor, even in ideal conditions. Since the sensors relied on a few features to estimate optical flow across the whole image plane, much of the rest of the image plane had little or no direct optical flow information. Thus, instead of having a measured estimate of the optical flow for every point in the image plane, many of the gaps had to be assumed. This meant that extracting accurate data from the optical flow such as environmental models or glider states was next to impossible.

## 3.2 Platform Concerns

With regard to the glider platform itself, the glider's abilities are severely limited. The microglider has a relatively large turning radius, lacks an accessible source of accurate state information, and has a large lag time associated with the use of turn primitives.

The microglider flies at over 5 meters per second, and requires a 4 meter turning radius to execute a 90 degree turn. This large turning radius and high speed mean that the smallest tunnel that could be feasibly navigated is 8 meters wide. This limitation was compensated for by only testing the glider in flight arenas that were greater than 8 meters in width.

A more difficult limitation of the flight platform was the lack of state information. Since state information was not able to be derived from the optical flow sensors, the control strategy had to rely on the passive stability of the glider. Although this allowed for some level of control, it severely limited the possible turn strategies.

Because the passive stability of the glider was the only reliable means of stabilization, turn primitives had to be used. After each turn was executed, a long settling time was required to ensure that the glider would return to a stable state. This led to a very slow response of the glider. Because the optical flow sensor was so dependent on texture and distance to the wall, this slow response time could allow the glider to pass through the optimal detection distance while the glider was still turning. This would mean that after the turn completed, the glider would be too close to the wall to accurately detect it, and a collision would result.

### **3.3 Conclusions and Future Work**

While certain parameter settings yielded good results, the artificial nature of the flight arena leads one to believe that these results would not be obtainable in a real setting. Future work in this area must concentrate on increasing sensor bandwidth, glider maneuverability, or some combination of both. For example, honeybees are able to navigate very effectively with optical flow and a small amount of state-feedback, due in large part to the high amount of maneuverability (180 degree turns in a few wing beats). On the other hand, traditional airplanes are much more limited in maneuverability, but must rely on very accurate state feedback as well as a reliable environment model. A third possibility is to increase the integration of sensors and flight maneuvers. By making up for sensor limitations with platform strengths, and vice-versa, a system may be created that does not rely on either very good sensors or flight platform. This is an area much inspired by nature, and that warrants further study.

# Bibliography

- [1] M. J. Abzug, and E. E. Larrabee, Airplane Stability and Control. 2<sup>nd</sup> ed., New York, NY: Cambridge University Press, 2002.
- [2] A. E. Bryson, Control of Spacecraft and Aircraft. Princeton, NJ: Princeton University Press, 1994.
- [3] B. Etkin, Dynamics of Flight: Stability and Control. 2<sup>nd</sup> ed., NY: John Wiley & Sons, 1982.
- [4] R.J. Wood, S. Avadhanula, E. Steltz, M. Seeman, J. Entwistle, A. Bachrach, G. Barrows, S. Sanders, R.S. Fearing. "Design, Fabrication and Initial Results of a 2g Autonomous Glider." presented at 31<sup>st</sup> IEEE Industrial Electronics Conf. Raleigh, NC, 2005.
- [5] G. Barrows. "Optic Flow Sensor with Fused Elementary Motion Detector Outputs." US Patent 6,384,905 B1, 7 May 2002.
- [6] G. Barrows. "Feature Tracking Linear Optic Flow Sensor." US Patent 6,020,953, 1 Feb. 2000.
- [7] G.A. Horridge. "A Template Theory to Relate Visual Processing to Digital Circuitry." Proceedings of the Royal Society of London. Series B, Biological Sciences, vol. 239, no. 1294, pp. 17-33, 22 Feb. 1990.

# Appendix A – Matlab Code

|   |    |
|---|----|
| function [F, UOptFlow, VOptFlow, left, right,end_state] =<br>flight_sim13(opt_flow_threshold, tunnel_width, turn_length,<br>feature_threshold); ..... | 2  |
| function [] = loadparameters13(); .....   | 11 |
| function [wing_drag] = get_wing_drag(alpha1, vel, wing_length); .....   | 15 |
| function [wing_lift] = get_wing_lift(alpha1, vel, wing_length) .....  | 15 |
| function FT = get_forces_torques(Rb2i, pos, dir, mag); .....  | 16 |
| function [v1, v2] = insert_turn(right0_left1, turn_length, max_voltage,<br>min_voltage); .....  | 16 |
| function [q] = get_q(R); .....  | 16 |
| function [R] = get_rot_mat(q); .....  | 16 |
| function [q_dot] = get_q_dot(q, w); .....   | 17 |
| function [w_dot] = get_w_dot(J, w, torq).....   | 17 |
| function R = rot(phi,theta,psi) .....   | 17 |
| function [skewed_mat] = skew(vect); .....   | 17 |
| function quiver3vect(pos,dir,S,plotstring); .....   | 17 |
| function [pd_value] = get_pd_value(glider_pos, glider_q, pd_pos,<br>pd_rot_mat, source_pos, source_range, source_on_off); .....                       | 18 |
| function [pd_voltage] = get_pd_voltage(yaw_angle, pitch_angle,<br>pd_alpha); .....  | 18 |
| function [] = make_block(corner1, corner2); .....   | 18 |

```

function [F, UOptFlow, VOptFlow, left, right,end_state] = flight_sim13(opt_flow_threshold,
tunnel_width, turn_length, feature_threshold);
% Flight simulator version 12.0
% 3 axis, 6 DOF glider simulator.
% Optical flow code integrated.
% Jon P Entwistle

loadparameters13();
load savefile;
current_fig =(gcf); % Save current figure to return after running flight_sim
figure(99); %Optical flow view is always plotted in figure 99
clf reset;

hold on;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make tunnel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% make_block([0,tunnel_width,20],[100,tunnel_width+1,150]);
% make_block([0,-tunnel_width,20],[100,-tunnel_width-1,150]);
% make_block([100,-tunnel_width,20],[101,tunnel_width,150]);
% make_block([0,-tunnel_width,20],[100,tunnel_width,19]);
axis equal;
grid off;
hold off;
axis off;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sensor settings
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
camproj perspective
camva(45);

R_sensor = eye(3); %Sensor points straight ahead
%R_sensor = rot(0,0,-pi/2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rotational initial conditions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
q(:,1) = get_q(rot(0,0*-pi/4,-pi/4)); % Initialize the gliders direction

q(:,1)= [0.0009; 0.2952; -0.0023; 0.9554]; %q vector for 45deg entry into
tunnel
q(:,1) = q(:,1)/norm(q(:,1));

w(:,1) = [0;0;0];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Light source placement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
source_pos = [60; 0; 0];
source_range = [100;100];
source_on_off = ones(2,last);
source_on_off(2,1:last/2) = 0;

```

```

source_on_off(2,last/2+1:last) = 0;
source_on_off(1,1:last/4) = 0;
pd_pos = [0, wing_length*cos(wing1_dihedral),
wing_length*sin(wing1_dihedral);...
0, -wing_length*cos(wing2_dihedral), wing_length*sin(wing2_dihedral)]';
pd_values = zeros(2,last);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plotting functions - 1=on, 0=off. Only one should be used at at time.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
testPlotForces = 0;
testPlotTraj = 1;
testPlotNRG = 0;
testPlotFinalForces = 0;
testAnimateTraj =0;
testPlotPDValues = 0; % Does not work
testPlotDihedral = 0;

for t = 1:last-1 % Main for-loop that iterates once per timestep

%   if (t == 2000)
%       state_dot(:,t)
%       q(:,t)
%   end % Used to calculate post-transient state.
%
%   if (abs(state(Y,t)) > tunnel_width)
%       state(:,t)
%       break;
%   end

% if (t == 1000)
%   j = t;
%   [voltage1(j:j+floor(2.6*turn_length)-
1),voltage2(j:j+floor(2.6*turn_length)-1)] = ...
%       insert_turn(0, turn_length, max_voltage,
min_voltage);
% end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Energy calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PE(t) = state(Z,t)*g*glider_mass;
KE(t) = .5*glider_mass*(norm(state_dot(X:Z,t)))^2 + w(:,t)'*J*w(:,t)/2;
mom(t) = norm(state_dot(X:Z,t))*glider_mass + sqrt(w(:,t)'*J*J*w(:,t));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate dependent state variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
v = state_dot(X:Z,t);
Ri2b = get_rot_mat(q(:,t));%Ri2b is the rotation matrix to convert from
% inertial coordinates to body coordinates
Rb2i(:, :,t) = Ri2b'; % Rb2i is the rotation matrix to convert from body
% coordinates to inertial coordinates
vb = Ri2b*v; % Velocity vector in body coordinates
glide_slope(t) = atan2(v(Z), norm(v(X:Y)))*rad2deg;
if (norm(vb) ~= 0)

```

```

        vb1 = vb/norm(vb);
else
    vb1 = vb; % normalized velocity direction in body coordinates
end

state2(ALPHA1,t) = (acos(dot([0;0;1],vb1)) - pi/2)*rad2deg;
state2(GAMMA1,t) = atan2(vb(Y),vb(X));
state2(VEL,t) = norm(v);

state3(W1_ALPHA,t) = (acos(dot(wing1_surf_norm,vb1)) - pi/2)*rad2deg;
state3(W2_ALPHA,t) = (acos(dot(wing2_surf_norm,vb1)) - pi/2)*rad2deg;
state3(T1_ALPHA,t) = (acos(dot(tail1_surf_norm,vb1)) - pi/2)*rad2deg;
state3(T2_ALPHA,t) = (acos(dot(tail2_surf_norm,vb1)) - pi/2)*rad2deg;

elev1_angle(t) = get_elevon_angle(voltage1(t));
elev2_angle(t) = get_elevon_angle(voltage2(t));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Onboard processing - occurs at onboard_frequency Hz
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   if (mod(t*timestep,1/onboard_frequency) == 0)
%
%       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       % Calculate photodiode sensor values
%       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       % pd_values(1,t) = get_pd_value(state(X:Z,t), q(:,t), [0;0;0], ...
%       %         rot(0,0,pi/3), source_pos, source_range, source_on_off(:,t));
%       % pd_values(2,t) = get_pd_value(state(X:Z,t), q(:,t), [0;0;0], ...
%       %         rot(0,0,-pi/3),source_pos, source_range, source_on_off(:,t));
%
%       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       % Compute the optic flow seen by the sensor
%       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       campos(state(X:Z,t)')
%       camtarget(state(X:Z,t)'+Rb2i(1:3,1,t)')
%       camup(Rb2i(1:3,3,t)');
%
%       sensor_t = ceil(t*onboard_frequency*timestep);
%       F = getframe(gcf);
%       %[full_frame,map] = frame2im(F(sensor_t));
%       full_frame = F.cdata;
%
%       if (sensor_t ~= 1)
%           frame1 = frame2;
%       end
%       %full_frame2 = full_frame(:, :, 1);
%       [frame_h, frame_w, blah] = size(full_frame);
%
%       if (frame_h ~= 420 || frame_w ~= 560)
%           [frame_h, frame_w]
%           return;
%       end
%
%

```

```

%         frame2 = full_frame(14:28:406, 16:33:544,1)/255;
%         %frame2 = full_frame(1:ceil(frame_h/15):frame_h,
1:ceil(frame_w/17):frame_w, 1)/255;
%         if (sensor_t ~= 1)
%
%             % Filter images and set values below feature_threshold to zero,
all else to
%             % 1
%             Ufeatures1 = (convn(frame1, mex_hat, 'valid') >=
feature_threshold);
%             % Vfeatures1 = (convn(frame1, mex_hat', 'valid') >=
feature_threshold);
%             Ufeatures2 = (convn(frame2, mex_hat, 'valid') >=
feature_threshold);
%             % Vfeatures2 = (convn(frame2, mex_hat', 'valid') >=
feature_threshold);
%             %size(Ufeatures1) = 15,14
%             %size(Vfeatures1) = 12,17
%
%             % Check for valid transitions
%             UTransitions = UTransitions*0;
%             VTransitions = VTransitions*0;
%             for i = 1:size(Ufeatures1,1)
%                 for j = 1:2:size(Ufeatures1,2)-2
%                     x = 4*Ufeatures1(i,j) + 2*Ufeatures1(i,j+1) +
Ufeatures1(i,j+2)+1;
%                     y = 4*Ufeatures2(i,j) + 2*Ufeatures2(i,j+1) +
Ufeatures2(i,j+2)+1;
%                     UTransitions(i,j+1) = optical_flow_lookup(x,y*2-1);
%                     UTransitions(i,j+2) = optical_flow_lookup(x,y*2);
%                 end
%             end
%
%             for i = 1:size(Vfeatures1,2)
%                 for j = 1:2:size(Vfeatures1,1)-2
%                     x = 4*Vfeatures1(j,i) + 2*Vfeatures1(j+1,i) +
Vfeatures1(j+2,i)+1;
%                     y = 4*Vfeatures2(j,i) + 2*Vfeatures2(j+1,i) +
Vfeatures2(j+2,i)+1;
%                     VTransitions(j+1,i) = optical_flow_lookup(x,y*2-1);
%                     VTransitions(j+2,i) = optical_flow_lookup(x,y*2);
%                 end
%             end
%
%             % RC Circuit equivalent - optical flow is equal to voltage
across a
%             % capacitor in an RC circuit. With no transitions voltage
decays
%             % exponentially. Each transition corresponds to a voltage of
(+/-)1 placed
%             % across the circuit for 1 timestep.
%             UOptFlow(:, :, sensor_t) = standard_decay*UOptFlow(:, :, sensor_t-
1) + switch_closed*UTransitions;
%             % VOptFlow(:, :, sensor_t) = standard_decay*VOptFlow(:, :, sensor_t-
1) + switch_closed*VTransitions;
%
%             width = size(UOptFlow,2);

```

```

%
%         left_flow(sensor_t) =
sum(sum(UOptFlow(:,1:ceil(width/3),sensor_t)))/(ceil(width/3)*size(UOptFlow,1)
));
%         right_flow(sensor_t) =
sum(sum(UOptFlow(:,floor(2*width/3):width,sensor_t)))/((width-
floor(2*width/3)+1)*size(UOptFlow,1));
%
%         if (sensor_t >=4)
%             left(sensor_t) = (left_flow(sensor_t) + left_flow(sensor_t-
1) + left_flow(sensor_t-2)+left_flow(sensor_t-3))/4;
%             right(sensor_t) = (right_flow(sensor_t) +
right_flow(sensor_t-1) + right_flow(sensor_t-2)+right_flow(sensor_t-3))/4;
%
%             if ((left(sensor_t) < -opt_flow_threshold) &&
turn_right(t) == 0);
%                 state(X:Z,t)
%                 left(sensor_t)
%
%                 long_turn = turn_length;
%                 j = t;
%                 [voltage1(j:j+floor(2.6*long_turn)-
1),voltage2(j:j+floor(2.6*long_turn)-1)] = ...
%                 insert_turn(0, long_turn, max_voltage,
min_voltage);
%
%                 turn_right(t:t+5*long_turn) = 1;
%
%             end
%             if ((right(sensor_t) > opt_flow_threshold) &&
turn_right(t) == 0);
%                 state(X:Z,t)
%                 right(sensor_t)
%
%                 long_turn = turn_length;
%                 j = t;
%                 [voltage1(j:j+floor(2.6*long_turn)-
1),voltage2(j:j+floor(2.6*long_turn)-1)] = ...
%                 insert_turn(1, long_turn, max_voltage,
min_voltage);
%
%                 turn_right(t:t+5*long_turn) = -1;
%
%             end
%         end
%     end %If (sensor_t ~=1)
% end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Calculate forces
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

    wing_drag = get_wing_drag(state2(ALPHA1,t), state2(VEL,t), wing_length);
    tail_drag = get_wing_drag(state2(ALPHA1,t), state2(VEL,t), 42.5*10^-
3)*tail_area/(wing_width*42.5*10^-3);

```

```

    wing1_lift = get_wing_lift(state3(W1_ALPHA,t),state2(VEL,t),
wing_length);
    wing2_lift = get_wing_lift(state3(W2_ALPHA,t),state2(VEL,t),
wing_length);

    elev1_force = ((state2(ALPHA1,t) + elev1_angle(t))*0.2625 + 0.3125)...
    *state2(VEL,t)^2/16;
    elev2_force = ((state2(ALPHA1,t) + elev2_angle(t))*0.2625 + 0.3125)...
    *state2(VEL,t)^2/16;

    taill1_lift = get_wing_lift(state3(T1_ALPHA,t), state2(VEL,t),
wing_length)...
    *tail_area/wing_area/2;
    taill2_lift = get_wing_lift(state3(T2_ALPHA,t), state2(VEL,t),
wing_length)...
    *tail_area/wing_area/2;

%     elev1_force = elev1_force*.8;
%     elev2_force = elev2_force*.8;
%     tail_drag = tail_drag*.6;
%     taill1_lift = taill1_lift*.6;
%     taill2_lift = taill2_lift*.6;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Vector sum of forces
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    FT(:,t) = get_forces_torques(Rb2i(:, :, t), wing1_pos-c_o_m,
cross(cross(vb1, ...
            wing1_surf_norm),vb1), wing1_lift) ...
    +get_forces_torques(Rb2i(:, :, t), wing2_pos-c_o_m,
cross(cross(vb1, ...
            wing2_surf_norm),vb1), wing2_lift) ...
    +get_forces_torques(Rb2i(:, :, t), taill1_pos-c_o_m,
cross(cross(vb1, ...
            taill1_surf_norm),vb1), taill1_lift) ...
    +get_forces_torques(Rb2i(:, :, t), taill2_pos-c_o_m,
cross(cross(vb1, ...
            taill2_surf_norm),vb1), taill2_lift) ...
    +get_forces_torques(Rb2i(:, :, t), taill1_pos-c_o_m, -vb1,
tail_drag)...
    +get_forces_torques(Rb2i(:, :, t), taill2_pos-c_o_m, -vb1,
tail_drag)...
    +get_forces_torques(Rb2i(:, :, t), wing1_pos-c_o_m, -vb1,
wing_drag)...
    +get_forces_torques(Rb2i(:, :, t), wing2_pos-c_o_m, -vb1,
wing_drag)...
    +get_forces_torques(Rb2i(:, :, t), elev1_pos-c_o_m,
cross(cross(vb1, ...
            elev1_surf_norm),vb1), elev1_force) ...
    +get_forces_torques(Rb2i(:, :, t), elev2_pos-c_o_m,
cross(cross(vb1, ...
            elev2_surf_norm),vb1), elev2_force);
    %+get_forces_torques(Rb2i(:, :, t), elev1_pos, -vb1,
elev1_drag) ...
    %+get_forces_torques(Rb2i(:, :, t), elev2_pos, -vb1, elev1_drag);

```

```

FT(:,t) = FT(:,t) + [0;0;-g*glider_mass;0; 0; 0];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Update translational states
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
state_dot(X:Z,t+1) = state_dot(X:Z,t) + timestep*FT(X:Z,t)/glider_mass;
state(X:Z,t+1) = state(X:Z,t) + state_dot(X:Z,t+1)*timestep;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Update rotational states
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
net_torq = FT(TX:TZ,t) - r_damp*(abs(w(:,t)).*w(:,t));
w(:,t+1) = w(:,t)+timestep*get_w_dot(J,w(:,t),net_torq);
q(:,t+1) = q(:,t)+timestep*get_q_dot(q(:,t),w(:,t+1));
q(:,t+1) = q(:,t+1)/norm(q(:,t+1)); % Ensure numerical accuracy

if (state(Z,t+1) <= 0)
    state(:,t+1)
    t
    break;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Debug/Plot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% if ((testPlotTraj ==1) && (mod(t,30) == 0))
%     subplot(2,2,1);
%     quiver3(ones(3,1)*state(X,t),ones(3,1)*state(Y,t),ones(3,1)*...
%         state(Z,t),Rb2i(1,1:3,t),Rb2i(2,1:3,t),Rb2i(3,1:3,t),0,'k');
%     axis equal;
%     hold on;
% end

if ((testPlotForces == 1) && mod(t,100) == 0)
    quiver3vect(wing1_pos-c_o_m,cross(cross(vb1, wing1_surf_norm),vb1),...
        wing1_lift*.01+.0001,'k');
    hold on;
    quiver3vect(wing2_pos-c_o_m,cross(cross(vb1, wing2_surf_norm),vb1),...
        wing2_lift*.01+.0001,'k');
    quiver3vect(wing1_pos-c_o_m,-vb1,wing_drag*.01+.0001,'k');
    quiver3vect(wing2_pos-c_o_m,-vb1,wing_drag*.01+.0001,'k');
    quiver3vect(tail1_pos-c_o_m,cross(cross(vb1, tail1_surf_norm),vb1),...
        tail1_lift*.01+.0001,'b');
    quiver3vect(tail2_pos-c_o_m,cross(cross(vb1, tail2_surf_norm),vb1),...
        tail2_lift*.01+.0001,'b');
    quiver3vect(tail1_pos-c_o_m,-vb1,tail_drag*.01+.0001,'b');
    quiver3vect(tail2_pos-c_o_m,-vb1,tail_drag*.01+.0001,'b');
    quiver3vect(elev1_pos-c_o_m,cross(cross(vb1, elev1_surf_norm),vb1),...
        elev1_force*.01+.0001,'r');
    quiver3vect(elev2_pos-c_o_m,cross(cross(vb1, elev2_surf_norm),vb1),...
        elev2_force*.01+.0001,'r');
    quiver3vect([0;0;0],Ri2b*[0;0;-1],g*glider_mass*.01+.0001,'g');
    quiver3vect([0;0;0],Ri2b*FT(X:Z,t)/norm(FT(X:Z,t)),...
        norm(FT(X:Z,t)).01+.0001,'c');
%     quiver3vect([0;0;0],FT(TX:TZ,t)/norm(FT(TX:TZ,t)),...

```

```

%           norm(FT(TX:TZ,t))* .01+.0001);
axis([-0.15 0.15 -0.15 0.15 -0.15 0.15]);
F(t/100) = getframe;
clf;
end

end % Main for loop

end_state = state(:,t);

if (testPlotDihedral == 1)
    plot(state3(W1_ALPHA,:) - state3(W2_ALPHA,:));
    hold on;
end

if (testPlotForces == 1)
    movie(F,5,10);
end

if (testAnimateTraj == 1)
    quiver3(ones(3,1)*state(X,1),ones(3,1)*state(Y,1),ones(3,1)*...
            state(Z,1),Rb2i(1,1:3,1),Rb2i(2,1:3,1),Rb2i(3,1:3,1),0,'k');
    axis equal;
    hold on;
    plot3(state(X,1:t),state(Y,1:t),state(Z,1:t));
    plot3(source_pos(1,:),source_pos(2,:),source_pos(3,),'k. ');
    quiver3(ones(3,1)*state(X,t),ones(3,1)*state(Y,t),ones(3,1)*...
            state(Z,t),Rb2i(1,1:3,t),Rb2i(2,1:3,t),Rb2i(3,1:3,t),0,'k');
    axis equal;
    for i = 2:t-1
        if (mod(i,floor(1/(framerate*timestep))) == 0)

            quiver3(ones(3,1)*state(X,i),ones(3,1)*state(Y,i),ones(3,1)*...
                    state(Z,i),Rb2i(1,1:3,i),Rb2i(2,1:3,i),Rb2i(3,1:3,i),0,'k');
            for j = 1:size(source_on_off,1)
                if (source_on_off(j,i) ~= 0)

plot3(source_pos(1,j),source_pos(2,j),source_pos(3,j),'gv');
                    else

plot3(source_pos(1,j),source_pos(2,j),source_pos(3,j),'rd');
                    end
                end
                F(i*framerate*timestep) = getframe;
            end
        end
        movie(F,5,framerate)
    end

if (testPlotTraj == 1)
    %subplot(3,2,1);
    figure(100);
    start = 1;
    plot3(state(X,start:t),state(Y,start:t),state(Z,start:t));

```

```

hold on;
plot3(state(X,1:t),state(Y,1:t),state(Z,1:t),'r');
%   for i = start:t
%       if (mod(i,floor(1/(framerate*timestep))) == 0)
%           quiver3(ones(3,1)*state(X,i),ones(3,1)*state(Y,i),ones(3,1)*...
%
state(Z,i),Rb2i(1,1:3,i),Rb2i(2,1:3,i),Rb2i(3,1:3,i),0,'k');
%       end
%   end
xlabel('x'); ylabel('y'); zlabel('z');
axis equal;
%   make_block([0,tunnel_width,20],[100,tunnel_width+1,150]);
%   make_block([0,-tunnel_width,20],[100,-tunnel_width-1,150]);
%   make_block([3,1,90],[4,4,105]);
% hold on;
% make_block([0,1,90],[2,4,105]);
% make_block([0,-1,90],[2,-4,105]);
% axis equal;
%grid off;
% hold off;
%axis off;

%   subplot(3,2,2);
%   plot((start:t)*timestep, state2(VEL,start:t)); title('Velocity');
%   subplot(3,2,3);
%   plot((start:t)*timestep, state2(ALPHA1,start:t)); title('Alpha');
%   subplot(3,2,4);
%   plot((start:t)*timestep, glide_slope(start:t)); title('Glide slope');
%   subplot(3,2,5);
%   plot((start:t)*timestep, w(1,start:t)); title('wx');
%   subplot(3,2,6);
%   plot((start:t)*timestep, w(3,start:t)); title('wz');
%subplot(3,2,4);
%plot((start:t)*timestep, w(2,start:t)); title('wy');
end

if (testPlotNRG == 1)
plot(timestep:timestep:t*timestep, KE(1:t)+PE(1:t));
title('Total energy of the system (KE + PE)');
xlabel('Time (s)');
hold on;
end

if (testPlotPDValues == 1)
subplot(2,1,1);
start = 1;
plot3(state(X,start:t),state(Y,start:t),state(Z,start:t));
hold on;
plot3(state(X,4000:6600),state(Y,4000:6600),state(Z,4000:6600),'r');
for i = start:t
    if (mod(i,floor(1/(framerate*timestep))) == 0)
        quiver3(ones(3,1)*state(X,i),ones(3,1)*state(Y,i),ones(3,1)*...
            state(Z,i),Rb2i(1,1:3,i),Rb2i(2,1:3,i),Rb2i(3,1:3,i),0,'k');
    end
end
end
plot3(source_pos(1),source_pos(2),source_pos(3),'p');

```

```

xlabel('x'); ylabel('y'); zlabel('z');
axis equal;
subplot(4,1,3);
plot(timestep*(1:t),pd_values(:,1:t));
subplot(4,1,4);
plot(timestep*(1:t),pd_values(1,1:t) - pd_values(2,1:t));
end

if (testPlotFinalForces == 1)
    quiver3vect(wing1_pos-c_o_m,cross(vb1, wing1_surf_norm),...
        wing1_lift*.01+.0001);
    hold on;
    quiver3vect(wing2_pos-c_o_m,cross(vb1, wing2_surf_norm),...
        wing2_lift*.01+.0001);
    quiver3vect(wing1_pos-c_o_m,-vb,wing_drag*.01+.0001);
    quiver3vect(wing2_pos-c_o_m,-vb,wing_drag*.01+.0001);
    quiver3vect(taill1_pos-c_o_m,cross(vb1, taill1_surf_norm),...
        taill1_lift*.01+.0001);
    quiver3vect(tail2_pos-c_o_m,cross(vb1, tail2_surf_norm),...
        tail2_lift*.01+.0001);
    quiver3vect(taill1_pos-c_o_m,-vb,tail_drag*.01+.0001);
    quiver3vect(tail2_pos-c_o_m,-vb,tail_drag*.01+.0001);
    quiver3vect(elev1_pos-c_o_m,cross(vb1, elev1_surf_norm),...
        elev1_force*.01+.0001);
    quiver3vect(elev2_pos-c_o_m,cross(vb1, elev2_surf_norm),...
        elev2_force*.01+.0001);
    quiver3vect([0;0;0],Ri2b*[0;0;-1],g*glider_mass*.01+.0001);
    %quiver3vect([0;0;0],Ri2b*FT(X:Z,t)/norm(FT(X:Z,t)),...
    %    norm(FT(X:Z,t))*0.01+.0001);
    quiver3vect([0;0;0],FT(TX:TZ,t)/norm(FT(TX:TZ,t)),...
        norm(FT(TX:TZ,t))*0.01+.0001);
    axis([-0.15 0.15 -0.15 0.15 -0.15 0.15]);
end

Description = 'flight_test';
%save flight_test2; % Save all workspace data to file

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Output end state
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
glide_s = glide_slope(t);
velocity = state2(VEL,t);
q(:,t)
state_dot(:,t)

figure(current_fig); %Restore original figure to active

function [] = loadparameters13();
% Function creates savefile, which contains the necessary data structures
% and parameters to run flight_sim13.
% [] = loadparameters13();
% Jon P. Entwistle, UC Berkeley

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Various parameters and constants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
end_time = 6; % length of simulation in seconds (start time is always 0)
timestep = .001; %in seconds
last = end_time/timestep;
framerate = 10; % frames per second
g = 9.81; % m/s^2
rad2deg = 180/pi;
deg2rad = pi/180;
simulationfrequency = 1/timestep; % this variable from example1.m
onboard_frequency = 100; %frequency in Hz of sensors

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%State variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% state, state_dot, w, and FT indices
X = 1;
Y = 2;
Z = 3;
TX = 4; % Torque in X direction
TY = 5;
TZ = 6;

% state2 indices
ALPHA1 = 1;
GAMMA1 = 2;
VEL = 3;

state = zeros(3, last); % Translational position in inertial coordinates
state_dot = zeros(3, last); % Translational velocities
q = zeros(4,last); % Angular position (in quaternians)
w = zeros(3,last); % Angular velocities
Rb2i = zeros(3,3,last); %Convert from body to inertial coordinates

state2 = zeros(3, last); % Alternative state descriptions

W1_ALPHA = 1;
W2_ALPHA = 2;
T1_ALPHA = 3;
T2_ALPHA = 4;
state3 = zeros(4, last);

FT = zeros(6,last); %Forces (1-3) and torques (4-6)

elev1_angle = zeros(1,last); % State of Y-pos elevon
elev2_angle = zeros(1,last); % State of Y-neg elevon

max_voltage = 200; % in volts - Max voltage causes pitch down
min_voltage = 200*2/(3*3.3 + 2); % Min voltage causes pitch up

voltage1 = (max_voltage+min_voltage)/2*ones(1,last); % Input to elev1
voltage2 = (max_voltage+min_voltage)/2*ones(1,last); % Input to elev2

glide_slope = zeros(1,last);

```

```

turn_lright = zeros(1, last); % Turns off optical flow during turn (-1 left,
1 right)
PE = zeros(1, last); % Potential energy
KE = PE; % Kinetic energy
mom = PE; % Momentum

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Glider parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
glider_mass = 1.969;%3 % in grams
J = glider_mass/1.969*[328.3 -.9366 -162.1;-.9366 1008.3 .2283;-162.1 .2283
1238.7]*10^-6;
%Inertial matrix aligned with body coordinates as measured from the COM
%based on solidworks model
% g/m^2
c_o_m = [8.2814*10^-3;0;-1.5255*10^-3]; % Position of center of mass based on
solidworks model - meters
%c_o_m = [1*10^-3;0;-1.5255*10^-3];

r_damp = [.0015 0 0;0 .0015 0;0 0 .0010]; % Rotational damping coefficients
% These are pure estimates based on expected
% flight paths.

%r_damp = zeros(3);

% Position and orientation of glider surfaces. Positive X is at front of
% glider, positive Z is up from glider (opposite direction of gravity when
% glider flying level). Positive Y direction is Z x X. Origin of
% coordinates is located directly on center shaft inbetween two wings.
% This defines the 'body coordinates'.
% All distances are in meters.
wing1_dihedral = 10*deg2rad;
wing2_dihedral = 10*deg2rad;
wing1_angle = 5*deg2rad;
wing2_angle = 5*deg2rad;
tail1_dihedral = pi/4;
tail2_dihedral = pi/4;
wing_length = 42.5*10^-3;

tail_width = 23*10^-3;
wing2tail = (31+2/3*15)*10^-3;
tail2elev = (10/2+1/3*15)*10^-3;

% wing/tail 1 refers to port side. wing/tail 2 refers to starboard side.
% Position of center of lift of wing
wing1_pos = [0;wing_length/2*cos(wing1_dihedral);
wing_length/2*sin(wing1_dihedral)];
% surface normal to wing
wing1_surf_norm = rot(wing1_dihedral, 0, 0)*rot(0, -wing1_angle, 0)*[0;0;1];
wing2_pos = [0;-wing_length/2*cos(wing2_dihedral);
wing_length/2*sin(wing2_dihedral)];
wing2_surf_norm = rot(-wing2_dihedral, 0, 0)*rot(0, -wing2_angle, 0)*[0;0;1];

```

```

tail1_pos = [-
wing2tail;tail_width*1/3*cos(tail1_dihedral);tail_width*1/3*sin(tail1_dihedra
l)];
tail1_surf_norm = rot(tail1_dihedral, 0, 0)*[0;0;1];
tail2_pos = [-wing2tail;-
tail_width*1/3*cos(tail2_dihedral);tail_width*1/3*sin(tail2_dihedral)];
tail2_surf_norm = rot(-tail2_dihedral, 0, 0)*[0;0;1];
elev1_pos = [-wing2tail-tail2elev; tail_width*1/2*cos(tail1_dihedral);
tail_width*1/2*sin(tail1_dihedral)];
elev2_pos = [-wing2tail-tail2elev; -tail_width*1/2*cos(tail2_dihedral);
tail_width*1/2*sin(tail2_dihedral)];
elev1_surf_norm = tail1_surf_norm;
elev2_surf_norm = tail2_surf_norm;

wing_width = 12*10^-3; % Needs to be remeasured
wing_area = wing_length*wing_width;
tail_area = .017*.039 + .010*(.039+.008)/2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

state(Z,1) =100;
state_dot(X,1) = 3;
state_dot(Y,1) = 3;
state_dot(Z,1) = 0;%-0.5;

state_dot(X,1) = 5.7;
state_dot(Y,1) = 5.7;
state_dot(Z,1) = -5.3;%-0.5;

state_dot(X,1) = 6.5507;
state_dot(Y,1) = 0;
state_dot(Z,1) = -4.3972;%-0.5;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Optical Flow MG1 Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Filter definitions
mex_hat = [1 -1 -1 1]/4;
mex_hat = [-1 -1 1 1]/4;
saddle = [1 -1;-1 1]/4;

%feature_threshold = .3;

optical_flow_lookup = ...
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;...
0 0 0 0 0 -1 0 -.5 0 0 0 0 -.5 -1 0 -1;...
0 0 0 1 0 0 0 .5 -1 0 0 0 -.5 0 0 0;...
0 0 0 .5 0 -.5 0 0 -1 -.5 0 0 -1 -1 -1 0;...
0 0 0 0 1 0 1 .5 0 0 0 0 .5 0 1 0;...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;...
0 0 .5 .1 .5 0 1 1 -.5 0 0 0 0 0 0 1;...

```

```

0 0 0 1 0 0 1 0 -1 0 0 0 0 -1 0 0];

% optical_flow_lookup = ...
% [0 0 0 -1 -1 1 -1 0 1 0 1 -1 0 1 0 0];...
% 0 1 0 0 -1 0 -1 -1 1 1 1 0 0 0 0 -1;...
% 1 -1 1 0 0 0 0 1 0 -1 0 0 -1 0 -1 1;...
% 1 0 1 1 0 -1 0 0 0 0 0 1 -1 -1 -1 0;...
% -1 0 -1 -1 0 1 0 0 0 0 0 -1 1 1 1 0;...
% -1 1 -1 0 0 0 0 -1 0 1 0 0 1 0 1 -1;...
% 0 -1 0 0 1 0 1 1 -1 -1 -1 0 0 0 0 1;...
% 0 0 0 1 1 -1 1 0 -1 0 -1 1 0 -1 0 0];

time_constant = 2;
sensor_timestep = 1;
switch_closed = (1-exp(-sensor_timestep/time_constant)); % Add this value if
valid transition occurs
standard_decay = exp(-sensor_timestep/time_constant);
    % Multiply this value at each iteration to account for transition decay

UOptFlow = zeros(15,14,10); %ONLY INITIALIZE ONCE, NOT AT EACH TIMESTEP
VOptFlow = zeros(12,17,10);
UTransitions = 0*ones(15,14);
VTransitions = 0*ones(12,17);
% UOptFlow = zeros(15,16,10); %ONLY INITIALIZE ONCE, NOT AT EACH TIMESTEP
% VOptFlow = zeros(14,17,10);
% UTransitions = 0*ones(15,16);
% VTransitions = 0*ones(14,17);

save savefile;

function [wing_drag]=get_wing_drag(alpha1, vel, wing_length);
% Wing drag in mN

wing_drag = (4.5*(1-cosd(alpha1))/(1-cosd(30))+.5) *
vel^2/16*wing_length/(42.5*10^-3);
% Over a small range, drag on a flat surface is proportional to the cos of
% the angle of attack, times vel^2 (measured wind speed 4m/s). This
% matches the measured drag well. Original wings were 42.5mm long, wing
% drag proportional to length.

function [wing_lift]=get_wing_lift(alpha1, vel, wing_length)
% Wing lift in mN

% Piecewise curve fit with measured data for 4m/s wind speed
if (alpha1 > 12)
    wing_lift = (4.5 + 1/9 * alpha1) * vel^2/16;
elseif (alpha1 <= 12 & alpha1 >= -12)
    wing_lift = .5*(alpha1) * vel^2/16;
elseif (alpha1 > 30)
    wing_lift = .5*(30) * vel^2/16;
else
    wing_lift = (-6) * vel^2/16;
end %if

```

```

% Original wings were 42.5mm long, wing lift proportional to length
wing_lift = wing_lift*wing_length/(42.5*10^-3);

function FT = get_forces_torques(Rb2i, pos, dir, mag);
%Function receives the position, direction, and magnitude of a force in
%body coordinates as well as the inverse of the rotational matrix for the
body
%coordinates, and returns the net translational force in inertial
%coordinates and the net torques in body coordinates. Position is versus
%the center of mass of the glider.
% FT = get_forces_torques(Rb2i, pos, dir, mag);
% Jon P. Entwistle, UC Berkeley

F = Rb2i*dir*mag/norm(dir);
T = mag*cross(pos, dir/norm(dir));
FT = [F;T];

function [v1, v2] = insert_turn(right0_left1, turn_length, max_voltage, min_voltage);
% Function inserts a saccade turn starting at the first elements of
% voltage1 and voltage2. Turn_length - 1x turn, 1x pitch up, .6x reverse
% turn. right0_left1 == 0 for right turn, == 1 for left turn.

v1 = zeros(1, floor(2.6*turn_length));
v2 = zeros(1, floor(2.6*turn_length));

lam1 = right0_left1; lam2 = 1-lam1; % Turn
v1(1:turn_length) = lam1*max_voltage + (1 - lam1)*min_voltage;
v2(1:turn_length) = lam2*max_voltage + (1 - lam2)*min_voltage;

lam1 = 0; lam2 = lam1; % Pitch up
v1(turn_length+1:2*turn_length) = lam1*max_voltage + ...
(1 - lam1)*min_voltage;
v2(turn_length+1:2*turn_length) = lam2*max_voltage + ...
(1 - lam2)*min_voltage;

lam1 = 1 - right0_left1; lam2 = 1-lam1; % Reverse turn
v1(1.8*turn_length+1:floor(2.6*turn_length)) = ...
lam1*max_voltage + (1 - lam1)*min_voltage;
v2(1.8*turn_length+1:floor(2.6*turn_length)) = ...
lam2*max_voltage + (1 - lam2)*min_voltage;

function [q] = get_q(R);
% Function takes rotation matrix R and returns quaternion state.
% [q] = get_q(R);
% Jon P. Entwistle, UC Berkeley

q = zeros(4,1);

q(4) = sqrt(1 + trace(R))/2;
q(1:3) = 1/(4*q(4))*[R(2,3) - R(3,2); R(3,1) - R(1,3); R(1,2) - R(2,1)];

function [R] = get_rot_mat(q);
% Function takes quaternion state q and returns the rotation matrix R

```

```

% corresponding to that quaternion.
% [R] = get_rot_mat(q);
% Jon P. Entwistle, UC Berkeley

R = (q(4)^2 - q(1:3)'*q(1:3))*eye(3) + 2*q(1:3)*q(1:3)' -
2*q(4)*skew(q(1:3));

function [q_dot]=get_q_dot(q, w);
% Function takes current quaternion state q and current angular velocity w
% and returns q_dot, the time derivative of q.
% [q_dot] = get_q_dot(q, w);
% Jon P. Entwistle, UC Berkeley

q_dot = q;

q_dot(1:3) = .5 * (q(4)*w - cross(w, q(1:3)));
q_dot(4) = -.5*w'*q(1:3);

function [w_dot]=get_w_dot(J, w, torq)
% Function takes in inertial matrix J, current angular velocities w, and
% current torque torq. Function returns w_dot, the time derivative of w
% for the next time step.
% [w_dot] = get_w_dot(J, w, torq)
% Jon P. Entwistle, UC Berkeley

w_dot = J^-1*(torq - cross(w, J*w));

function R = rot(phi,theta,psi)
% Returns inertial to body rotation matrix for a set of Euler angles

R = [cos(psi)*cos(theta)  -sin(psi)*cos(phi)+cos(psi)*sin(theta)*sin(phi)
sin(psi)*sin(phi)+cos(psi)*sin(theta)*cos(phi)  ;
      sin(psi)*cos(theta)  cos(psi)*cos(phi)+sin(psi)*sin(theta)*sin(phi) -
cos(psi)*sin(phi)+sin(psi)*sin(theta)*cos(phi)  ;
      -sin(theta)          cos(theta)*sin(phi)
cos(theta)*cos(phi)          ];

function [skewed_mat]=skew(vect);
% Returns skew matrix of a 3 element vector

skewed_mat = [0 -vect(3) vect(2); vect(3) 0 -vect(1); -vect(2) vect(1) 0];

function quiver3vect(pos,dir,S,plotstring);
% Used to easily plot 3 or 4 vectors

if nargin ==3, quiver3(pos(1),pos(2), pos(3), dir(1),dir(2), dir(3),S); end
if nargin ==4
    quiver3(pos(1),pos(2), pos(3), dir(1),dir(2), dir(3),S,plotstring);
end

```

```

function [pd_value] = get_pd_value(glider_pos, glider_q, pd_pos, pd_rot_mat, source_pos, source_range,
source_on_off);
% Function takes current glider position glider_pos, current glider
% orientation glider_q, and an array of positions of photodiodes in body
% coordinates pd_pos. Function also takes an array of source positions
% source_pos, an array of source ranges (in m) source_range, and a bit
% array describing whether source is on or off (1 = on, 0 = off)
% source_on_off. Function returns an array giving the output of each
% photodiode given in pd_pos. Function assumes omnidirectional sources as
% well as omnidirectional sensors.
% [pd_values] = get_pd_values(glider_pos, glider_q, pd_pos, source_pos,
% source_range, source_on_off);
% Jon P. Entwistle, UC Berkeley

Ri2b = get_rot_mat(glider_q)
pd_pos_i = Ri2b'*pd_pos + glider_pos
    % photodiode position in absolute inertial coordinates

pd_source_dir = (source_pos - pd_pos_i)/norm(source_pos - pd_pos_i)
    % Direction from pd to source in inertial coordinates
pd_rot_mat_i = pd_rot_mat'*Ri2b' % Rotation matrix from inertial to pd
coordinates
source_dir_pd = pd_rot_mat_i*pd_source_dir % source direction in pd
coordinates

yaw_angle = atan2(source_dir_pd(2),source_dir_pd(1))
pitch_angle = atan2(source_dir_pd(3), sqrt(source_dir_pd(1)^2 +
source_dir_pd(2)^2))
pd_alpha = acos(dot([1;0;0], source_dir_pd))

pd_value = get_pd_voltage(yaw_angle, pitch_angle, pd_alpha);

function [pd_voltage] = get_pd_voltage(yaw_angle, pitch_angle, pd_alpha);
% Returns photodiode voltage based on angle of light source

yaw_angle2 = abs(yaw_angle);

if (yaw_angle2 <= pi/3)
    pd_voltage = 3.25;
elseif (yaw_angle2 >= pi/2)
    pd_voltage = .75;
else
    pd_voltage = .0028*(yaw_angle2*180/pi)^2 - .5*(yaw_angle2*180/pi) +
23.15;
end

function [] = make_block(corner1, corner2);

feature_size = 1;

x1=corner1(1);y1=corner1(2);z1=corner1(3);
x2=corner2(1);y2=corner2(2);z2=corner2(3);

```

```

% randn*ones([feature_size, feature_size, feature_size]);

%[X,Y,Z] = meshgrid(x1:.1*sign(x2-x1):x2, y1:.1*sign(y2-y1):y2,
z1:.1*sign(z2-z1):z2);
[X,Y,Z] = meshgrid(x1:sign(x2-x1):x2, y1:sign(y2-y1):y2, z1:sign(z2-z1):z2);

mat=zeros(size(X));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create random texture
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:feature_size:size(X,1)-feature_size
    for j = 1:feature_size:size(X,2)-feature_size
        for k = 1:feature_size:size(X,3)-feature_size
            block2 = randn*ones([feature_size, feature_size, feature_size]);
            mat((i+i+feature_size-1),j:(j+feature_size-1),k:(k+feature_size-
1)) = block2;
        end
    end
end

for i = size(X,1):-feature_size:feature_size
    for j = size(X,2):-feature_size:feature_size
        for k = size(X,3):-feature_size:feature_size
            block2 = randn*ones([feature_size, feature_size, feature_size]);
            mat((i-feature_size+1):i,(j-feature_size+1):j,(k-
feature_size+1):k) = block2;
        end
    end
end

w = mat;%smooth3(mat);
block = slice(X,Y,Z,w,[x1,x2],[y1,y2],[z1,z2]);
set(block, 'EdgeColor', 'none')%,'FaceColor','interp')

```