

An Attitude Controller for Small Scale Rockets

by
Florian Kehl

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

**Master of Science
in Nanosciences**

at the University of Basel, Switzerland

October 2009

Advisors:

Prof. Kristofer S. J. Pister
University of California, Berkeley
United States of America

Prof. Ernst Meyer
University of Basel
Switzerland



Abstract: Current trends in miniaturization indicate the reduction in size of all electronic devices, and satellites are no exception. Drawing from progress in wireless sensor network technology, microsystems can potentially be used in such space-based networks. This project considers the deployment of small-scale satellites into orbital trajectories. In particular, a low mass, low cost attitude controller for a minirocket is designed and tested. State estimation, open loop flight control, and closed loop feedback actuation are all demonstrated on this platform, leading the way for a small scale system to deliver a 10g payload into low Earth orbit.

Contents

1	Introduction	5
2	Background	6
2.1	Reaching Low Earth Orbit	6
2.1.1	Energy and Δv budget	6
2.1.2	Guidance	7
2.2	Requirements for Minirocketry	7
2.3	Model Rocketry	8
2.3.1	Basic Principle	8
2.3.2	Engines	9
2.4	Inertial Sensors	10
2.4.1	Micromachined Accelerometers	10
2.4.2	Micromachined Gyroscopes	10
2.5	Coordinate System Transformation	11
2.5.1	Euler Angles	12
2.5.2	Quaternions	12
3	Materials and Methods	14
3.1	Rocket	14
3.2	Sensors	14
3.3	Controller	15
3.4	Actuators	16
4	Experimental Technique	19
4.1	Sensor Calibration	19
4.1.1	Accelerometer	19
4.1.2	Gyroscope	20
4.2	Uncontrolled Flight	20
4.3	Open-Loop Control	21
4.3.1	Ground-based Experiments	21
4.3.2	In-flight Experiments	22
4.4	Closed-Loop Feedback Control	22
4.4.1	Ground-based Experiments	23
4.4.2	In-flight Experiments	23
5	Results and Discussion	24
5.1	Sensor Calibration	24

5.2	Uncontrolled Flight	25
5.3	Open-Loop Control	27
5.3.1	Ground-based Experiments	27
5.3.2	In-flight Experiments	27
5.4	Closed-Loop Feedback Control	28
5.4.1	Ground-based Experiments	28
5.4.2	In-flight Experiments	30
Conclusion		31
Acknowledgment		32
Appendix		35
A:	Altitude Prediction	35
B:	Python Source Code	35

1 Introduction

Wireless Sensor Networks (WSNs) are of increasing interest in both academia and industry. Distributed sensing, processing, and communication have far reaching applications; indeed WSNs have found their way to all varieties of settings around the world, from oceans to plains to mountains to urban environments [2]. There is considerable ongoing research employing WSNs for their distributed nature, as well as research into WSNs themselves in areas such as reliability and robustness, communication theory, and distributed algorithms.

An area not yet infiltrated by WSNs though is space – orbits above Earth’s atmosphere and beyond are still dominated by large one-off spacecraft. Only recently has there been analysis of potential deployments in space (such as in [3], [4]). Such satellite systems would be able to address fundamental WSN research in the absence of notable interference from ground based sources and physical obstacles, as well as conduct atmospheric and astronomical research. As the availability and functionality of electronics go up and the cost goes down, the required hardware becomes smaller, cheaper, and more accessible. While previous small satellite research has focused on systems on the order of kilograms, sensor nodes have shrunk to where a 10 gram system is sufficiently powerful for many purposes.

This thesis addresses the issue of deploying such sensor nodes into a low Earth orbit (LEO) for applications in space-based WSNs. In particular, this thesis begins to examine a small scale rocket-based solution for delivering a 10 gram payload to a desired orbital trajectory. The additional difficulty in miniaturizing a satellite deployment system is offset by the drastically lower cost and risk factors compared to current options.

Ultimately, a launch vehicle (LV) should be of comparable scale and cost to the payload mote being deployed. A full launch solution will also require careful rocket and propellant design; this paper mainly focuses on miniaturizing a control system to be used to guide the LV into a desired trajectory. The hardware developed here is applicable as a final stage in orbital insertion – a rocket is designed and built using low cost, off-the-shelf components to estimate and control attitude alone. A high altitude balloon launch, additional minirocket stages, or piggybacking off of a large scale rocket can be used to get such a system to an approximate trajectory first.

An overview of rocket systems and the difficulties in their miniaturization is presented in chapter 2. The specific hardware designed in this work is described in chapter 3, with an explanation of the experimental setup in chapter 4 and some testing results and their discussion in section 5. The final chapter offers some conclusions and avenues for future research.

2 Background

2.1 Reaching Low Earth Orbit

2.1.1 Energy and Δv budget

The effort needed to execute any orbital maneuver is commonly described by the scalar Δv or delta-v, referencing a change in velocity. In order for a LV to reach LEO from rest on the surface of the Earth, the required Δv_{leo} is composed as follows [5]:

$$\Delta v_{leo} = v_o + \Delta v_d + \Delta v_g + \Delta v_c + \Delta v_{atm} - v_{rot}. \quad (2.1)$$

The first term of equation (2.1), v_o , is the speed needed to keep the payload in orbit around the Earth. The remaining terms are mainly dependent on the trajectory of choice to reach this particular orbit. For a circular orbit,

$$v_o = \sqrt{\frac{G \cdot M_e}{(R_e + h)}}, \quad (2.2)$$

where G is the gravitational constant, M_e is the mass and R_e is the radius of the Earth, and h is the orbit altitude. Equation (2.2) indicates that v_o for a satellite in a LEO at $200km$ altitude is approximately $7.8km/s$. The additional Δv terms are Δv_d due to air drag, Δv_g needed to climb against Earth's gravitational potential, Δv_c in losses when effecting a desired trajectory, and Δv_{atm} for lower engine performance during the ascent in the atmosphere. These losses sum up to an additional $1.5 - 2km/s$, despite the direct Δv boost caused by the Earth rotation $v_{rot} = 463m/s$ at the equator [5]. All told, a Δv_{leo} of $9.5 - 10 km/s$ is required to reach LEO for a ground launched LV.

Calculating the Δv generated by a LV's propulsion system requires a time history of its instantaneous thrust ($|F|$) and mass (m):

$$\Delta v = \int \frac{|F|}{m} dt, \quad (2.3)$$

where the integral is carried out over the duration of the maneuver in question. This can be evaluated out for a specific rocket design to yield the ideal rocket equation

$$\Delta v = I_{sp} \cdot g \cdot \ln \left(\frac{m_i}{m_f} \right). \quad (2.4)$$

The specific impulse I_{sp} is an intrinsic property of the fuel and $g = 9.81m/s^2$ is the acceleration due to gravity. Clearly, the Δv of the LV depends strongly on the rocket's mass ratio between the initial take-off mass m_i including the mass of the fuel and the final mass m_f after burnout consisting of only the payload and structural mass. Multistage rockets are evaluated by summing the Δv 's of each burn calculated independently.

2.1.2 Guidance

Active control is necessary to guide a payload along a trajectory to reach LEO. There is no ballistic path to LEO (*“what goes up must come down”*), so the LV has to perform an orbital insertion maneuver at some point. Even if the rocket system piggybacks off of larger scale carrier, the delivery method is often incapable of varying initial launch elevation and position, leaving it to the LV to correct for and fly the proper trajectory [6].

To guide the LV along the specified trajectory, a controller must determine the position and attitude of the rocket to subsequently counteract any deviation from the preset flight path. Ideally, the full six degree-of-freedom (6DOF) position would be known, identifying both its location and orientation in space. Typically the controller employs an inertial measurement unit (IMU) augmented with additional sensors to generate an accurate estimate of the 6DOF state of the LV from inertial rate measurements and external observations.

From the state estimate, corrections need to be applied to maintain a desired flight profile. Given that most of the flight occurs in extremely thin atmosphere, aerodynamic surfaces such as fins would be useless for the LV’s attitude control. Therefore, more sophisticated systems as a gimbaled nozzle for thrust vectoring need to be considered.

This work addresses the guidance subsystem requirement of a minirocket launch vehicle.

2.2 Requirements for Minirocketry

A rocket experiences a drag force

$$F_d = \frac{1}{2}\rho C_d v^2 A, \quad (2.5)$$

where ρ is the density of the surrounding air, C_d is the drag coefficient, v is the velocity of the rocket relative to the air, and A is the rocket’s cross sectional area. This results in a penalty

$$\Delta v_d = \int \frac{F_d}{m} dt, \quad (2.6)$$

that scales up with decreasing length.

With this increased Δv requirement, the rocket equation (2.4) indicates that the structural mass of a small scale LV must be kept as low as possible. Miniaturization of electronics and advancements in structural materials have enabled the miniaturization of satellite payloads and associated LV, and the primary design consideration for such a system is keeping a low mass.

The goal of course is to be able to launch a 10 gram satellite into orbit from the surface of the Earth. However, due to the low overall mass of the mini-rocket, the payload delivery system can itself piggyback on other vehicles to reduce the Δv requirement. The minirocket can be used for orbital trajectory insertion, with a large scale rocket or an ultra high altitude balloon (UHAB) carrying the rocket system to the upper atmosphere.

The concept of launching rockets from a balloon is not a new idea and has been performed extensively during the 1950s by J.A. Van Allen for upper atmospheric research [7]. Deployment of a 690kg payload to a peak altitude of 49.4km by a UHAB has been demonstrated [8]. Similarly, conventional rocket LVs often have spare payload capacity which can be used to deliver much smaller systems [9].

There are a number of energy advantages for air-based launches:

- Starting powered flight at 50km altitude or above 98.5% of the atmosphere, the LV experiences less than 3% of the drag force compared to a ground launched vehicle, tremendously decreasing Δv_d [6]. Therefore, increased drag due to decreased length (Eq. 2.6) is a minor concern.
- Δv_g and Δv_c are also lowered since the LV has to fight Earth's gravity and follow a given trajectory for a shorter amount of time. Additionally, the LV doesn't need to compensate for wind gusts, a concern at lower altitudes.
- The engine operates at peak performance when exhausting into near vacuum, directly increasing thrust F due to an increased ratio between combustion chamber and ambient pressures [10].

To keep costs, complexity and structural mass at a minimum, a solid propellant seems to be favorable for a small-scale LV. The need for pipes, valves, tanks, and insulation in liquid propellant engines would contribute to a high overall structural mass. The main disadvantages of solid-fuel propellant are the lower specific impulse I_{sp} and the lack of active throttling, though the latter can be overcome by intelligent propellant grain design, allowing specific thrust-time characteristics (which is outside the scope of this thesis).

2.3 Model Rocketry

2.3.1 Basic Principle

Model rockets are small scale rockets and are generally built out of paper, plastic, wood and other light weight material. Single-use, off-the-shelf engines are used to launch the rocket to altitudes usually around $100 - 500\text{m}$. The flight can be divided in the following phases:

1. Ignition: The motor is ignited by an electrical match.
2. Thrusting Phase: The burning propellant is delivering full thrust, accelerating the rocket to maximal velocity.
3. Coasting Phase: After the engine burnout, the rocket coasts for several seconds while decelerating till it reaches peak altitude.
4. Recovery System Deployment: At the apogee, the recovery system is deployed by an ejection charge.

5. Descent: The rocket slowly descends to the ground by hanging on the recovery system, e.g. a parachute.

To meet these tasks, model rocket engines are subject to a specific design which will be explained in the following subsection.

2.3.2 Engines

Disposable black powder model rocket engines from *Estes*TM were used in this report. The engine essentially consists of a cardboard casing, a clay nozzle and propellant (Fig. 2.1a). The propellant can be divided into three fractions: first, the black powder, providing thrust for liftoff and acceleration; second, a slowly burning delay component allowing the rocket to decelerate during the coasting phase; third, an explosive ejection charge for the deployment of the recovery system (Fig. 2.1b).

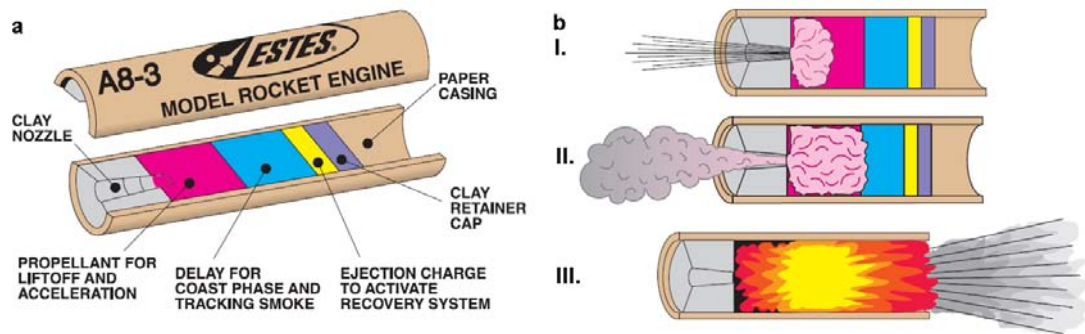


Figure 2.1: a) Longitudinal section of an *Estes* model rocket engine. b) Burning phases: I. thrusting for liftoff and acceleration, II. smoking delay charge for tracking and decelerating the rocket, III. ejection charge for the deployment of the recovery system. (*Estes-Cox Corp.*, from <http://www.estesrockets.com>)

Depending on the size and the mass of the rocket, desired velocity and peak altitude or on the experiment in general, different engines can be selected. The model rocket motors mainly differ in the delivered total impulse and the duration of the delay charge. Figure 2.2 shows a characteristic thrust curve for an *Estes*TM E9 engine with a total impulse of $28.5Ns$ and a peak-thrust in the very beginning for the liftoff of the rocket. The peak results from the spherical core-burning in the beginning with a maximal reaction surface, followed by the end-burning with constant burning surface and hence steady thrust. Each model rocket engine type has a letter-number-number code: the letter indicates the total impulse in Newton-seconds delivered by the engine (Total impulse doubles with every subsequent letter, so a “D” engine produces twice the power of a “C” engine), the first number shows the average thrust in Newtons and the second number the duration of the delay charge in seconds.

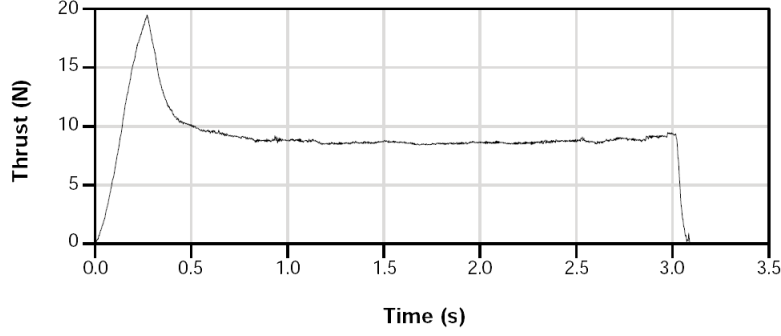


Figure 2.2: Thrust curve of a *EstesTM E9* model rocket engine.

2.4 Inertial Sensors

Microelectromechanical systems (MEMS) are miniaturized, micro scale devices, often integrating sensors, actuators and electronics on one chip. Nowadays, MEMS are a very versatile group of devices and standard components in medical, automotive and home electronics due to low-cost batch fabrication. The main application of MEMS are inertial sensors. Inertial sensors are divided into accelerometers and gyroscopes, whereas the former measure linear acceleration and the latter measure angular velocity about one or several axes [11].

2.4.1 Micromachined Accelerometers

There have been several different types of micromachined accelerometers reported in literature and available commercially, but the vast majority use a suspended proof mass as mechanical sensing system (Fig. 2.3). The proof mass is attached to a reference frame and will be deflected by any inertial force due to acceleration and its moment of inertia. The displacement of the proof mass is measured either by capacitive, piezoresistive, piezoelectric or tunneling current to determine the magnitude of the acceleration.

2.4.2 Micromachined Gyroscopes

Any moving mass in a rotating reference frame is subject to the Coriolis force \mathbf{F}_c , which is given by the mass m of the moving object and the cross product of the angular velocity $\mathbf{\Omega}$ of the rotation and the radial velocity \mathbf{v} of the object by:

$$\mathbf{F}_c = 2m\mathbf{\Omega} \times \mathbf{v} \quad (2.7)$$

Most MEMS gyroscopes rely on actively driven, oscillating structures to measure angular rate. In the presence of rotation and therefore due to the Coriolis force, the oscillating proof mass couples energy from the primary, excited vibration mode in a secondary mode, perpendicular to the former. The measured amplitude of this secondary oscillation is proportional to the angular velocity $\mathbf{\Omega}$. A schematic model is shown in Fig. 2.4a: the

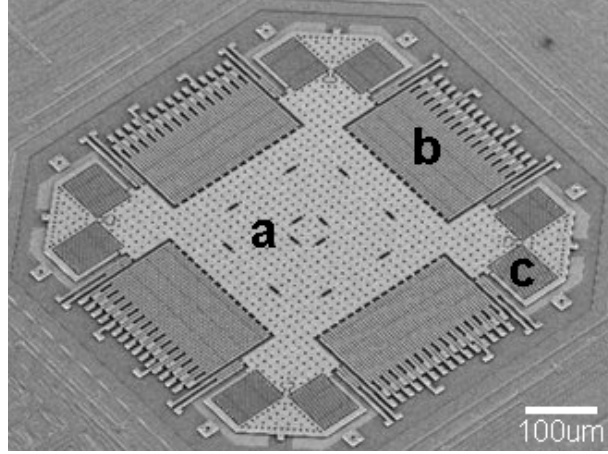


Figure 2.3: SEM micrograph of a MEMS dual-axis accelerometer: a proof mass (a) is suspended by springs (c) and thereby able to move in both in-plane directions. Comb capacitors (b) measure the displacement of the proof mass due to in-plane accelerations. (Picture: Analog Devices, Inc.)

suspended proof mass is excited to oscillate along the x-axis with a given amplitude and frequency. Rotation about the out-of-plane z-axis induces a secondary oscillation along the y-axis which can be measured with the same techniques as above-mentioned.

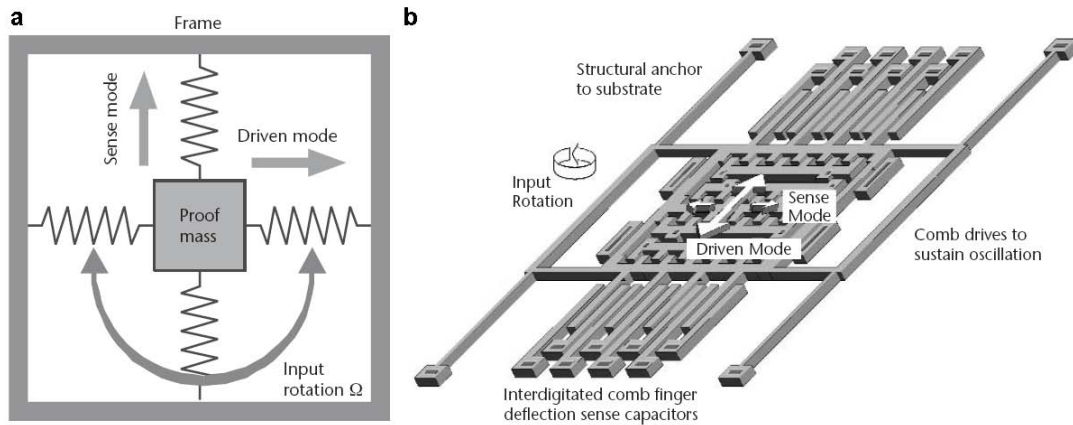


Figure 2.4: a) Schematic model of an angular rate MEMS gyro. b) Surface-micromachined gyroscope. (After: [11].)

2.5 Coordinate System Transformation

A LV's attitude information is essential in order to transform vector measurements, e.g. accelerations, from the reference frame of the rocket into the observer's inertial earth

frame coordinate system. By fusing data from a three-axis accelerometer and a three-axis gyroscope, the full 6DOF state estimate can be resolved. An objects attitude is defined by the transformation between the body-fixed reference frame to the inertial frame. This transformation can be described using different coordinate parametrizations. The most common ones are Euler angles and quaternions.

2.5.1 Euler Angles

Leonhard Euler stated that any orientation of an object in 3-dimensional space can be described by a maximum of three subsequent rotations, defined by the Euler angles Ψ , Θ and Φ (yaw, pitch and roll), which is equivalent with saying that the rotational matrix M_{rot} can be decomposed into a product of three independent rotations:

$$\begin{aligned}
 M_{rot} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix} \begin{bmatrix} \cos\Theta & 0 & -\sin\Theta \\ 0 & 1 & 0 \\ \sin\Theta & 0 & \cos\Theta \end{bmatrix} \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ -\sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos\Theta\cos\Psi & \cos\Theta\sin\Psi & -\sin\Theta \\ \sin\Phi\sin\Theta\cos\Psi - \cos\Phi\sin\Psi & \sin\Phi\sin\Theta\sin\Psi + \cos\Phi\cos\Psi & \sin\Phi\cos\Theta \\ \cos\Phi\sin\Theta\cos\Psi + \sin\Phi\sin\Psi & \cos\Phi\sin\Theta\sin\Psi - \sin\Phi\cos\Psi & \cos\Phi\cos\Theta \end{bmatrix}
 \end{aligned} \tag{2.8}$$

For real-world applications, Euler angles are unfavorable since for some rotations the angles become undefined, often referred as the “gimbal lock” problem [13]. To overcome this problem of singularities and decrease the computational burden due to the numerous sine and cosine functions, quaternions where used in the following for the rocket’s state estimation.

2.5.2 Quaternions

Quaternions are hypercomplex, 4-dimensional numbers, conceived by Sir William R. Hamilton in 1843. Quaternions are of big interest and use in both theoretical and applied mathematics and are often used to describe 3-dimensional rotations in an elegant manner. Since a detailed quaternion discussion is out of the scope of this report, a brief introduction into the concept of using quaternions for attitude estimation is described in the following [13].

Quaternions q are composed of a three-dimensional vector part and a scalar part and can be expressed as

$$q = q_0 + \mathbf{q} = q_0 + iq_1 + jq_2 + kq_3 \tag{2.9}$$

where i , j and k are unit vectors in the hyper-dimensional plane. Like 2-dimensional complex numbers, quaternions obey similar mathematical rules:

$$\begin{aligned}
 i^2 &= j^2 = k^2 = ijk = -1 \\
 ij &= -ji = k \\
 jk &= -kj = i \\
 ki &= -ik = j
 \end{aligned} \tag{2.10}$$

Using quaternions to describe rotations, a quaternion vector can be defined which represents a rotation about a unit vector $\mathbf{e}=(e_x, e_y, e_z)$ through an angle θ . This 4-dimensional vector can be written in the following format:

$$q = \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{pmatrix} = \begin{pmatrix} \cos(\theta/2) \\ \sin(\theta/2) e_x \\ \sin(\theta/2) e_y \\ \sin(\theta/2) e_z \end{pmatrix} \quad (2.11)$$

The rotation of a vector \mathbf{v} that corresponds to multiplication by a rotation matrix M_{rot}

$$\mathbf{v}' = M_{rot} \mathbf{v} \quad (2.12)$$

can be accomplished using quaternion algebra as

$$\mathbf{v}' = q^* \mathbf{v} q \quad (2.13)$$

where q^* is the complex conjugate of q . If the attitude of the body, here the LV, is constantly changing, the quaternion rates need to be related to the body angular rates:

$$\frac{d}{dt} q = \dot{q} = \frac{1}{2} Q \boldsymbol{\omega} \quad (2.14)$$

with the rotation matrix Q

$$Q = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{bmatrix} \quad (2.15)$$

and the attitude rate vector $\boldsymbol{\omega}$, measured by the IMU gyroscopes in the reference frame. Expanding equation 2.14 gives:

$$\begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix} = \frac{1}{2} \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{bmatrix} \begin{pmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \quad (2.16)$$

Integration over time of equation 2.16 and normalizing yields to a new quaternion. This new quaternion contains the actual attitude information and with equation 2.13, the reference frame accelerations, measured by the IMU accelerometers, can be transformed in accelerations in the inertial frame. Double integration of the latter gives the actual position of the LV.

3 Materials and Methods

3.1 Rocket

In order to test the guidance hardware designed for small scale LVs, a model rocket based test system was developed. Miniaturization of this rocket wasn't attempted, as the focus was on the guidance subsystem comprising sensors and actuators.

Off-the-shelf model rocket components were used for the basic rocket structure, namely cardboard tubes, polystyrene and balsa wood. The rocket contained a parachute for recovery and held disposable off-the-shelf solid-fuel engines. Depending on the experimental setup, different engines with characteristic performances could be mounted. The rocket was designed to carry the sensors and controller in its body, and incorporated actuators for active control. A camera was also mounted for in-flight video recording for post-flight analysis.

The dimension of the final rocket, shown in figure 3.1, was $1.25m$ with a diameter of $0.06m$ and overall mass of $0.57kg$.



Figure 3.1: Final rocket: 1) gimbaled nozzle, 2) parachute bay, 3) camera, 4) IMU, motor and payload section, 5) antenna.

3.2 Sensors

An on-board inertial measurement unit (IMU) was used to measure the body referenced 6DOF inertial rates. As above-mentioned (Sec. 2.4), a MEMS accelerometer measured 3 axis linear motion while MEMS gyros measure the 3 axis angular rates. These sensors can be integrated to calculate the 6DOF position (Sec. 2.5).

An on-board off-the-shelf camera (*FlyCamOne*² from *ACMETM*) enabled visual comparison with the measured data and provided additional information on different flight phases, e.g. parachute deployment.

3.3 Controller

The flight controller for the rocket was a custom designed circuit board for use in small robotic applications and was developed by Ankur Mehta from the UC Berkeley beforehand. The Guidance and Inertial Navigation Assistant (GINA) board shown in figure 3.2a comprises the MEMS inertial sensors, a microcontroller, a 2.4GHz wireless radio, and headers to a daughter card (Fig. 3.2b) to drive the actuators. We used a MSP430 microprocessor from Texas Instruments, an ADXL345 three-axis digital accelerometer and an ADXRS612 yaw rate gyro from Analog Devices, an IDG-1004 dual-axis gyro from InvenSense and an AT86RF231 low power 2.4GHz radio transceiver from ATMEL. The 2g system is the size of a US quarter at half the mass.

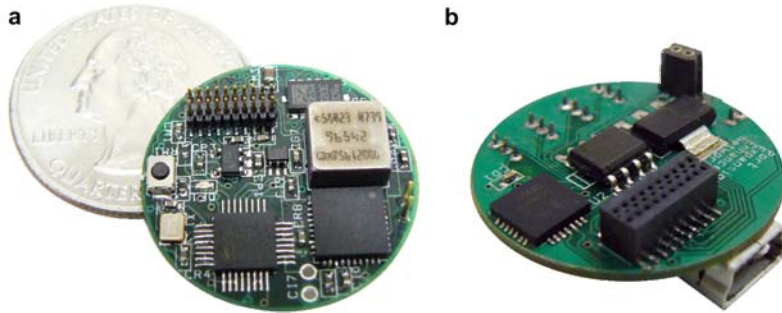


Figure 3.2: a) The 2g GINA controller board incorporates inertial sensing, processing, communications, and actuation. MEMS accelerometers and gyros, a 2.4 GHz radio, and a connector to the actuator driver board are visible; the processor is on the backside of the board. b) The GINA daughter board to drive the actuators.

Though the microcontroller is capable of implementing control laws itself, for ease of development the system was set up to use a laptop as a command station. The microcontroller polled the sensors and transmitted the data wirelessly to a base station connected via serial port to the laptop, which processed the data to generate control outputs. The control signals were sent back over the wireless link to the GINA microcontroller, which then drove the actuators via the daughter board.

The software running on the base station was written in Python and essentially received, processed and logged the flight data. Since orbital trajectories are more robust to altitude errors than they are to attitude errors and there's no easy way to throttle a solid-fuel rocket engine, the focus of the controller was on attitude control. The base station received acceleration and angular rates from accelerometers and the gyros on the GINA board and integrated them into a position and attitude estimate. This estimate was demonstrated to track the actual orientation of the rocket quite closely over several minutes, and so this state was fed into various feedback loops to control the rocket's roll, pitch, and yaw over the duration of a flight. An integrated PID controller tried to keep the rocket in a given attitude and therefore calculated the control output commands.

These control signals were relayed to the GINA board to set the motor speeds and servo positions. For debugging, open-loop experiments or just for safety reasons, the software allowed manual joystick commands to control the actuators. Additionally, an autocalibration routine minimized drift due to temperature dependent offset errors. A flowchart of the program is shown in figure 3.3. The entire source code is attached in appendix B.

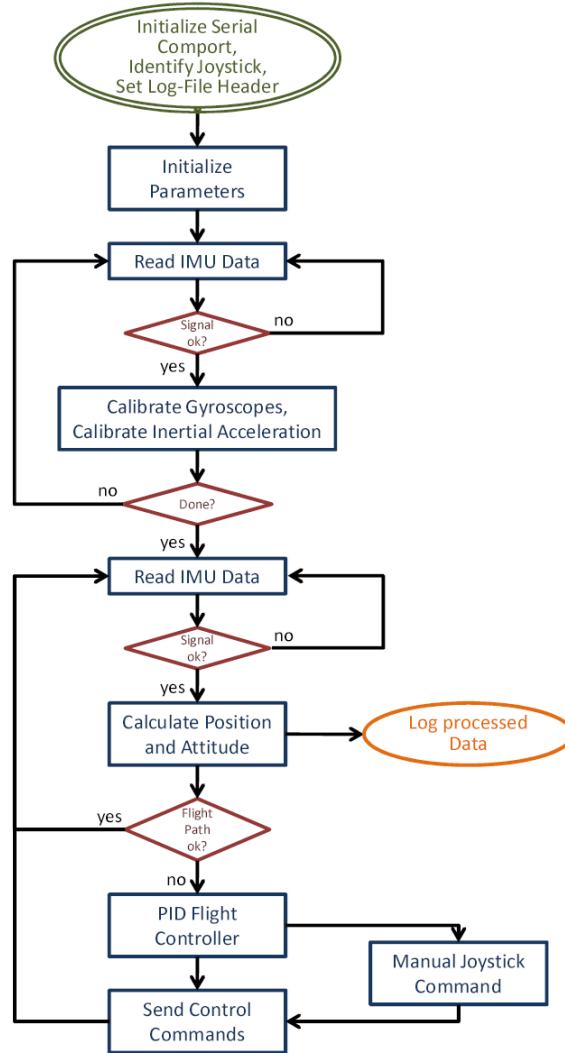


Figure 3.3: Flowchart of the base station software.

3.4 Actuators

To control the longitudinal (Fig. 3.4a), or roll axis, the rocket body (1) contained two concentrically mounted discs (3), driven by two counter-rotating brushless DC motors (2). Controlled acceleration and deceleration of these discs was used to counteract external

torques on the rocket's roll axis by compensating angular momentum.

For yaw and pitch control, a gimbaled nozzle was developed to vector the thrust along both axes (Fig. 3.4b). An inner engine mount tube (4) was gimbaled on a spherical bearing (7), driven by two high-torque servos (5,6). Controlling the position of the servos steered the rocket engine to point in any direction within a $\pm 4.5^\circ$ cone. Since thrust plume and ejection charge are expelled in opposite directions of the engine, the system had to be designed in such a way that the engine mount tube was unhindered on either opening. Due to the gimbaled nozzle architecture, all the accelerating and decelerating forces were absorbed by the bearing's constraining rings and hence there was no force acting on the servo motors. A screwable aluminum engine retainer facilitated exchanging the disposable rocket engines (Fig. 3.5).

For the recovery system deployment, the rocket needed to be separable to open the parachute bay. Since the servo motors were located in the rocket's tail and the controller in its nose cone, we had to develop robust but low friction contacts to assure both parachute ejection and servo function. The solution was copper foil contacts (Fig. 3.6) at the tubes joint for unhindered separation. An ejection baffle, consisting of two plywood boards with misaligned drill-holes absorbed hot sparks from the ejection charge to protect the parachute.

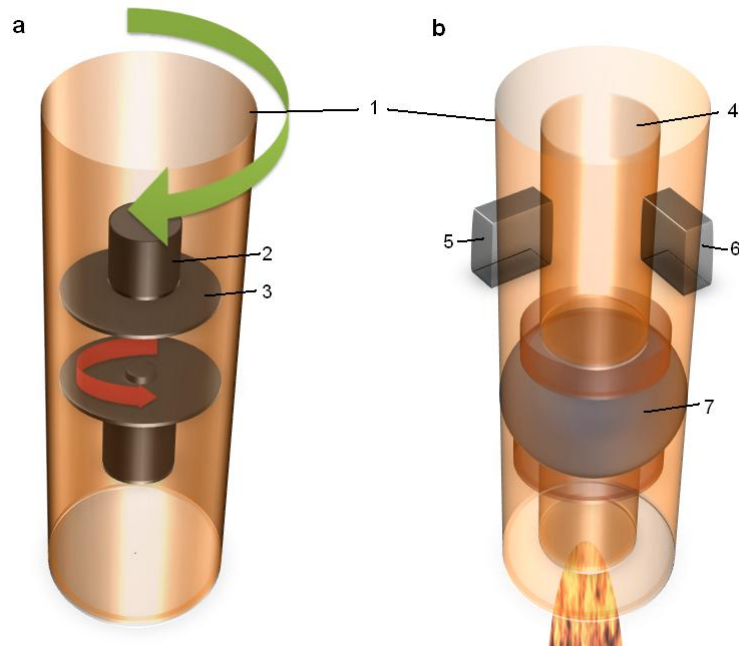


Figure 3.4: Control principles: a) spinning discs for roll, b) gimbaled nozzle for pitch and yaw.



Figure 3.5: Partially manufactured gimbaled nozzle: inner engine mount tube with spherical bearing and engine retainer.



Figure 3.6: Copper foil contacts at the inside of the lower tube and at the outside of the upper tube (not shown) allowed unhindered separation. The ejection baffle at the end of the parachute bay protected the parachute from hot sparks.

4 Experimental Technique

4.1 Sensor Calibration

To get real mechanical quantities out of the IMU sensor readings, both the accelerometers and the gyros had to be calibrated. More precisely, one had to find the offset d and scaling factors c to convert the sensor output x_{sens} in a meaningful physical value x_{real} for all 6DOF, represented by

$$x_{real} = cx_{sens} + d \quad (4.1)$$

4.1.1 Accelerometer

As a reference for the accelerometer calibration, we used the Earth gravitational acceleration, which is defined as $1g$ or $9.81ms^{-2}$. Arbitrarily rotating the IMU while recording the accelerometer output $\mathbf{a}_{sens} = (a_x^{sens}, a_y^{sens}, a_z^{sens})$ led to a raw dataset visualized in figure 4.1 by an off-centered ellipsoid, caused by the different accelerometer readings along the three spatial axes.

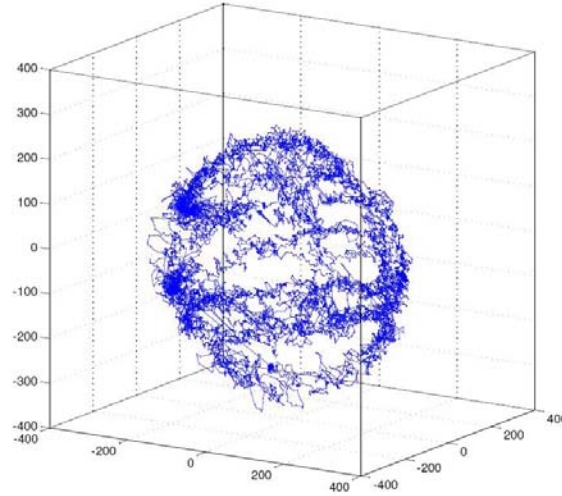


Figure 4.1: Visualization of accelerometer raw data by arbitrary rotations.

Metaphorically speaking, the real acceleration $\mathbf{a}_{real} = (a_x^{real}, a_y^{real}, a_z^{real})$ could be obtained by transforming the ellipsoid in a sphere with a radius of $1g$. We know that

$$|\mathbf{a}_{real}| = \sqrt{(c_x a_x^{sens} + d_x)^2 + (c_y a_y^{sens} + d_y)^2 + (c_z a_z^{sens} + d_z)^2} = 1g \quad (4.2)$$

and hence for the whole dataset with m measurements

$$\begin{bmatrix} (a_{x,1}^{sens})^2 & a_{x,1}^{sens} & (a_{y,1}^{sens})^2 & a_{y,1}^{sens} & (a_{z,1}^{sens})^2 & a_{z,1}^{sens} & 1 \\ (a_{x,2}^{sens})^2 & a_{x,2}^{sens} & (a_{y,2}^{sens})^2 & a_{y,2}^{sens} & (a_{z,2}^{sens})^2 & a_{z,2}^{sens} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (a_{x,m}^{sens})^2 & a_{x,m}^{sens} & (a_{y,m}^{sens})^2 & a_{y,m}^{sens} & (a_{z,m}^{sens})^2 & a_{z,m}^{sens} & 1 \end{bmatrix} \begin{pmatrix} c_x^2 \\ 2c_x d_x \\ c_y^2 \\ 2c_y d_y \\ c_z^2 \\ 2c_z d_z \\ d_x^2 + d_y^2 + d_z^2 \end{pmatrix} = (1g)^2 \quad (4.3)$$

Numerical approximation of equation 4.3 in MATLAB returned the scaling factors c_{acc} and offsets d_{acc} for the accelerometers.

4.1.2 Gyroscope

For the gyroscope calibration, the IMU was put on a turn table with known angular rate. The device was spun in several different orientations at altered but known angular velocities ω_{real} . Similar to equation 4.2, we knew

$$|\omega_{real}| = \left| \begin{pmatrix} \omega_x^{real} \\ \omega_y^{real} \\ \omega_z^{real} \end{pmatrix} \right| = \left| \begin{pmatrix} c_x \omega_x^{sens} + d_x \\ c_y \omega_y^{sens} + d_y \\ c_z \omega_z^{sens} + d_z \end{pmatrix} \right| \quad (4.4)$$

and for all measurements m at different angular velocities ω_{real}

$$\begin{bmatrix} (\omega_{x,1}^{sens})^2 & \omega_{x,1}^{sens} & (\omega_{y,1}^{sens})^2 & \omega_{y,1}^{sens} & (\omega_{z,1}^{sens})^2 & \omega_{z,1}^{sens} & 1 \\ (\omega_{x,2}^{sens})^2 & \omega_{x,2}^{sens} & (\omega_{y,2}^{sens})^2 & \omega_{y,2}^{sens} & (\omega_{z,2}^{sens})^2 & \omega_{z,2}^{sens} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (\omega_{x,m}^{sens})^2 & \omega_{x,m}^{sens} & (\omega_{y,m}^{sens})^2 & \omega_{y,m}^{sens} & (\omega_{z,m}^{sens})^2 & \omega_{z,m}^{sens} & 1 \end{bmatrix} \begin{pmatrix} c_x^2 \\ 2c_x d_x \\ c_y^2 \\ 2c_y d_y \\ c_z^2 \\ 2c_z d_z \\ d_x^2 + d_y^2 + d_z^2 \end{pmatrix} = \begin{pmatrix} |\omega_1^{real}| \\ |\omega_2^{real}| \\ \vdots \\ |\omega_m^{real}| \end{pmatrix} \quad (4.5)$$

Again, numerical approximation of equation 4.5 in MATLAB returned the scaling factors c_{gyr} and offsets d_{gyr} for the gyroscopes.

4.2 Uncontrolled Flight

The body-referenced angular rates measured by the IMU can be integrated into a full earth-referenced 6DOF state estimate (Subsection 2.5). Using those measurements, the position of a sample uncontrolled rocket flight can be calculated and compared with theoretical predictions. The predictions account for aerodynamic drag and gravity and are based on thrust-time data sheets from the National Association of Rocketry NAR for off-the-shelf model rocket engines [12] given the dimensions of the rocket and its time

dependent mass. For the latter, we assumed the expelled mass to be proportional to the thrust.



Figure 4.2: Uncontrolled rocket at take-off.

4.3 Open-Loop Control

The goal of open-loop controlled experiments was to directly influence the rocket's attitude. Knowing the effect of the actuators on the attitude was critical to finally program a closed-loop feedback controller.

4.3.1 Ground-based Experiments

Due to safety reasons and better data validation, extensive ground-based experiments regarding the rocket actuation were performed before the actual controlled launch.

To test the roll actuation, the rocket was suspended on a thread, allowing free rotation along the longitudinal axis. The effect of accelerating and decelerating the spinning discs of the roll actuator (Fig. 3.4a) could be estimated by measuring the angular velocity of the rocket.

For pitch and yaw, the rocket was pinned at its center of gravity and was able to rotate along one of the axes (Fig. 4.3). By vectoring the thrust by an angle ε , the effect of gimballing the nozzle on the pitch and yaw rates could be measured. Since the rocket is rotation-symmetric, this experiment was performed just along one axis. Additionally, theoretical predictions based on thrusting angle ε , the rocket's moment of inertia and the engines thrust characteristics were compared with the measured results. For these experiments we chose *EstesTM C6-0* and *D12-0* engines. The zero indicates that these

engines are booster engines with no ejection charge which could have caused damage to the rocket and no parachute deployment was necessary.



Figure 4.3: To measure the effect of vectoring the thrust by an angle ε , the rocket was pinned at its center of gravity allowing free rotation along one axis.

4.3.2 In-flight Experiments

After extensive ground-based testing, in-flight open-loop experiments have been performed.

By controlled acceleration and deceleration of the roll actuator motors during the ascent, we measured the effect on the roll rate. Controlled by manual joystick commands, we tried to rotate the rocket along its longitudinal axis in both directions.

For the actual open loop controlled flight with the vectored thrust, the rocket was programmed to sinusoidally swing the nozzle in the pitch axis back and forth at a 3 Hz frequency, starting the wiggle when the rocket reached an altitude of 7m. The goal was to directly control the rocket's pitch and hence its trajectory by gimbaling the nozzle. For this experiment, the rocket was powered by a *EstesTM E9-4* engine with a 3s burning time for additional time for thrust vectoring.

4.4 Closed-Loop Feedback Control

To keep a rocket on a desired trajectory, closed loop feedback control is necessary. Since the rocket presented in this thesis steers by vectoring its thrust in a particular direction, holding the roll axis constant is important to prevent a corkscrew like flight path. Depending on the deviation from the initial roll angle at the launch pad, a PID controller

drives the rotation speed of the two discs, compensating any external torque, e.g. wind gusts or engine nonuniformities.

4.4.1 Ground-based Experiments

For testing and simulation purposes, the rocket was again suspended by attaching a thread to its nose cone for unhindered longitudinal rotation, in an analogous manner as described in subsection 4.3.1. Once the PID gains were manually tuned, two sets of experiments were conducted: first, a fan blowing asymmetrically on the rocket fins induced a constant torque; second, a table tennis ball hitting one of the fins generated a torque impulse. In both cases, the PID controller tried to hold the rocket in its initial orientation.

4.4.2 In-flight Experiments

For the closed-loop feedback controlled flights, the PID controller was programmed to maintain the rocket's initial orientation. For this purpose, we chose the same PID gains validated during the ground-based experiments (Subsection 4.4.1). A less sophisticated P controller for pitch and yaw tried to keep the rocket in an upright position by gimbaling the nozzle.

5 Results and Discussion

5.1 Sensor Calibration

Correct sensor calibration and subsequent coordinate system transformation was validated by manually moving and rotating the IMU and track its position and orientation. A 3D plot of a such a motion captured trajectory can be seen in figure 5.1, representing *bsac* or *Berkeley Sensor & Actuator Center*. The letters were written in the air by hand with the IMU while tracking its position. Obviously, the trajectory of interest would be the flight path of the rocket during a launch.

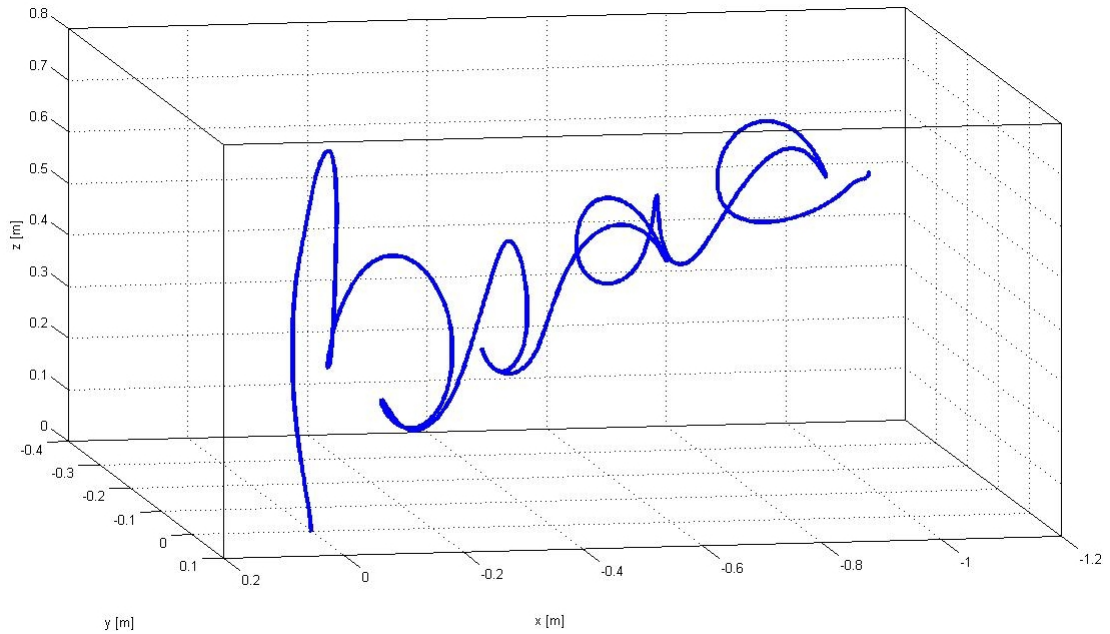


Figure 5.1: Spatial representation of the position over time tracking of the IMU, demonstrated by manually writing *bsac* in the air.

For debugging and verification of the correct attitude interpretation, the Python code (Section 3.3) was extended by an OpenGL graphical output, representing a digital 3D model of the small scale rocket. Any change in orientation of the rocket was simultaneously displayed by the digital model on the screen (Fig. 5.2).

One of the major problems for state estimation with an IMU is drift, since the outputs of these sensors suffer from additive Gaussian noise as well as zero bias drift. The additive

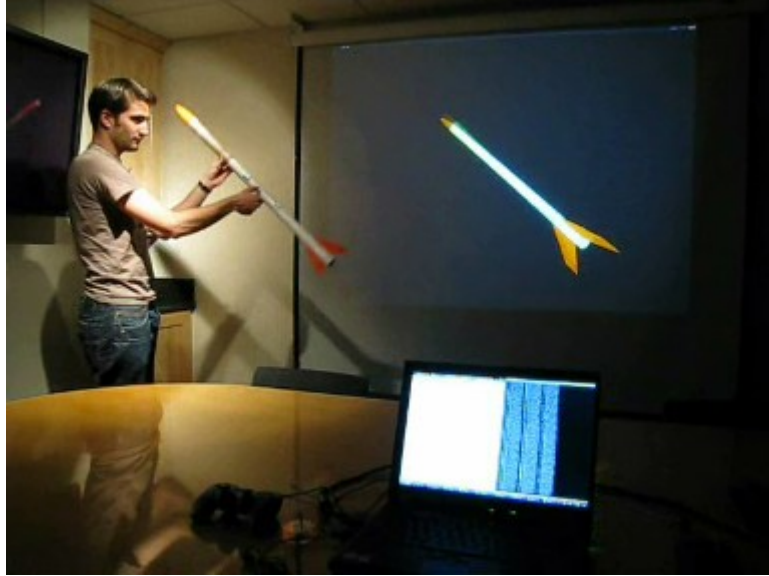


Figure 5.2: Real-time attitude estimation represented by a digital 3D model.

noise often integrates out, but the random offset in the rates integrates over time into nontrivial errors. Determining attitude from angular rates requires one integration, and so diverges from true rather slowly; position however is the double integral of acceleration and can accumulate errors rather quickly. To compensate for these errors, additional sensors like GPS, magnetic field or optical sensors would be necessary. Though, on the time scale of our model rocket flights (duration ≈ 10 s, depending on the engine), drift was a minor issue.

5.2 Uncontrolled Flight

A typical measured profile of an uncontrolled flight can be seen in figure 5.3 (here with an *EstesTM E9-4* engine). The blue dots indicate the inertial acceleration of the rocket, combining all accelerometer and gyroscope readings. The green and the red line represent the rocket's velocity and altitude respectively, obtained by integration and double-integration of the acceleration. The theoretical predictions are plotted as dashed lines in black.

Looking at the acceleration, we can clearly see the characteristic thrust-time behavior of the engine (fig. 2.2), with a initial peak thrust due to the engines core burning for the lift-off boost, a subsequent steady burning followed by the engines burnout and deceleration of the rocket during the coasting phase. Predicted and measured data match perfectly until the engine's burnout. A significant shorter burning time of the engine than predicted by the NAR (due to manufacturing tolerances) led therefore to the deviations in velocity and altitude after this event. In this particular flight, wireless connection to the base station was lost around $T+2.5$ s and again shortly before reaching the apogee for a short period of time due to unknown reason, but occurred during a few launches.

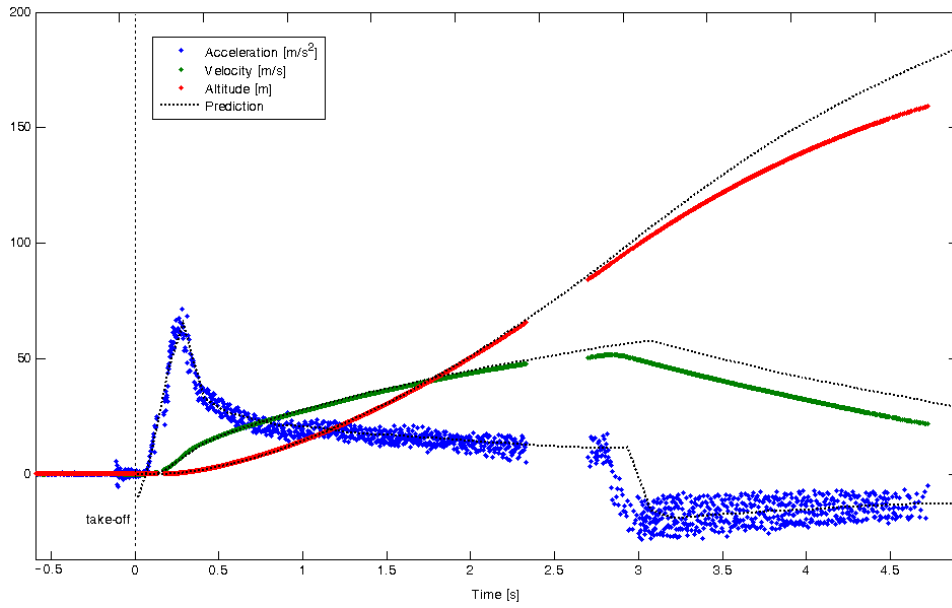


Figure 5.3: Measured versus predicted flight profile. Integration of the inertial acceleration (blue) led to velocity (green) and altitude (red). Theoretical predictions are in black.

Tracking the rocket with the antenna decreased the chance of losing data packets.

The on-board camera provided interesting additional data on attitude but also helped to assign data features to certain events of a flight. As mentioned above, model rocket engines differ in performance, hence it was helpful to distinguish the exact time of an event like the burn-out time or parachute deployment (Fig. 5.4).



Figure 5.4: Still pictures of a rocket launch: a) thrusting phase, b) coasting at apogee, c) parachute deployment.

5.3 Open-Loop Control

5.3.1 Ground-based Experiments

As expected, acceleration of the roll actuator's discs induced torque on the rocket's longitudinal axis, hence we were able to spin the rocket in both directions. The major drawback of this system is its tendency for saturation: once the motors are running at full speed, no more torque can be generated by the actuator.

To verify the effect on pitch and yaw rates, the gimballed nozzle was commanded to vector the engine to maximal deflection at $\varepsilon = 4.5^\circ$. During peak thrust of an *EstesTM C6-0* we measured an angular acceleration of approximately 6 rad/s^2 , which matched well with the expected behavior, as seen in figure 5.5a. In Fig. 5.5b, the thrust vector ε was switched from one endpoint of $+4.5^\circ$ to the opposite endpoint at -4.5° at the time indicated by the vertical dotted line. After about a 0.2s delay from the servo response time, the nozzle vectored its thrust in the opposite direction, decelerating the pitching of the rocket, causing it to stop and reverse its rotation.

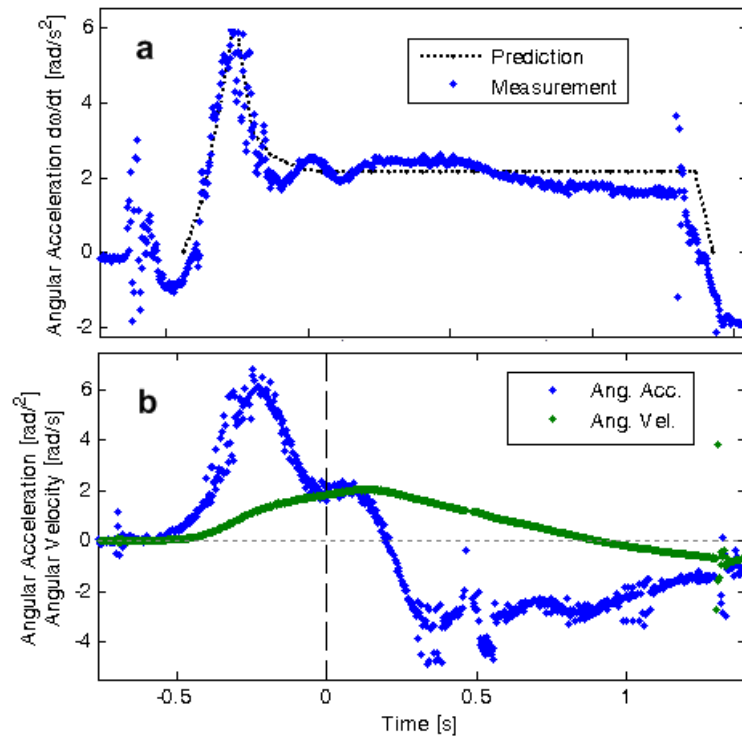


Figure 5.5: Angular acceleration depending on the thrusting vector ε .

5.3.2 In-flight Experiments

A rocket was flown with the roll commands driven with joystick control. The resulting roll angle is shown in figure 5.6. The rocket's roll rate is clearly influenced by the

(unfortunately unrecorded) disc acceleration and deceleration: without any actuation, a steady, mainly velocity dependent curve would be observed; the highly irregular peaks indicate external control. The initial smooth roll of the rocket upon lift-off is due to induced torque either by the engine or aerodynamics. One big advantage of this system is its independency of thrust, being still capable to control during the coasting phase.

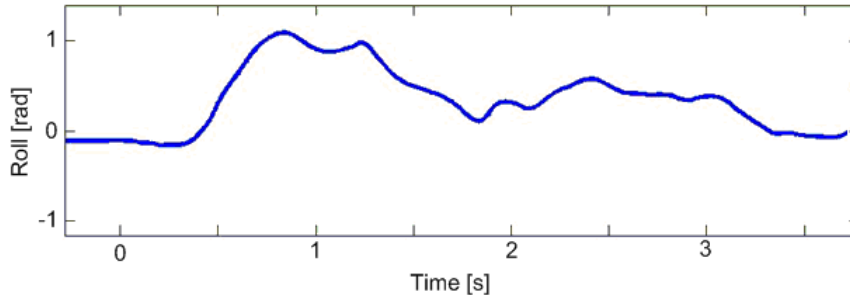


Figure 5.6: Roll angle during flight: the highly irregular peaks indicate external control.

In a subsequent flight, the thrust was vectored as abovementioned. Around $T+1s$, when the controller's state estimate indicated a 7m altitude, the nozzle started its sinusoidal movement which led to the same wave like pitch movement, revealed by its angular velocity (Fig. 5.7b). As seen before, the rocket's response was time shifted by around 0.2s due to the servo's response time. The initial pitching of the rocket is visible on all previous launches, and occurs due to unstable low-velocity behavior. As the relative speed of the rocket increases, the fins add stability and the swinging disappears. An evidently wavelike trajectory can be observed in the rocket's smoke trail (Fig. 5.7a), visually supporting the measured data.

5.4 Closed-Loop Feedback Control

5.4.1 Ground-based Experiments

Analogous to the open-loop case, the roll actuator tends to saturate, hence the actuator can only compensate for a certain amount of angular momentum, limited by the motor speed and the combined moment of inertia of the rotor and disc. The discs act just as a reservoir for angular momentum but are not capable to get rid of it. Knowing expected torques on the rocket body, appropriate disc and motor selection can help to overcome this problem, but might also increase the overall mass. Therefore, application of a constant torque will finally end in rotation, but it can be delayed by a significant amount of time as shown in figure 5.8 compared to the uncontrolled case.

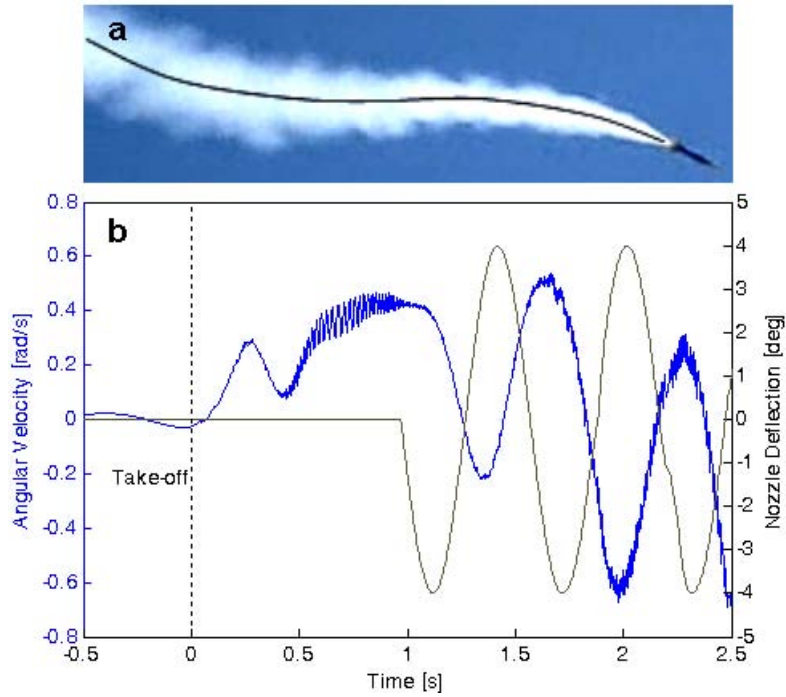


Figure 5.7: a) The sinusoidal smoke trail is generated by thrust vectoring, b) Measured angular velocity demonstrates the rocket's response to pitch control. The oscillation immediately after the take-off is due to unstable flight at low velocities.

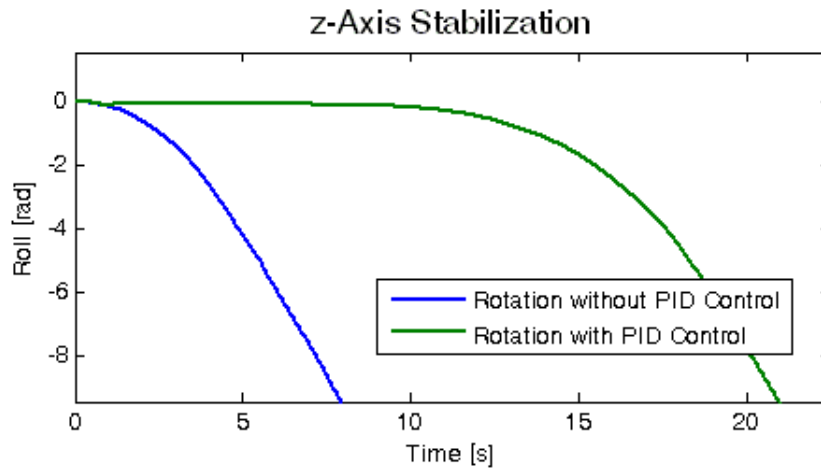


Figure 5.8: Effect of spinning discs on constant torque: In the uncontrolled case (blue line), the rocket started to roll immediately, whereas in the controlled event (green line) the PID controller kept the rocket in position for a significant amount of time, but finally ended in rotation.

In the case of an abrupt event like the table tennis ball hitting a fin (simulating a short duration gust or the like) the system reacts fast to stop the rotation. In the uncontrolled case (Fig. 5.9a), the rocket keeps turning steadily after the impact, slightly decelerated by the air resistance of the fins. By contrast, in the PID controlled scenario, the rotation stops suddenly and the roll controller drives the rocket back towards its initial position, as seen in figure 5.9b.

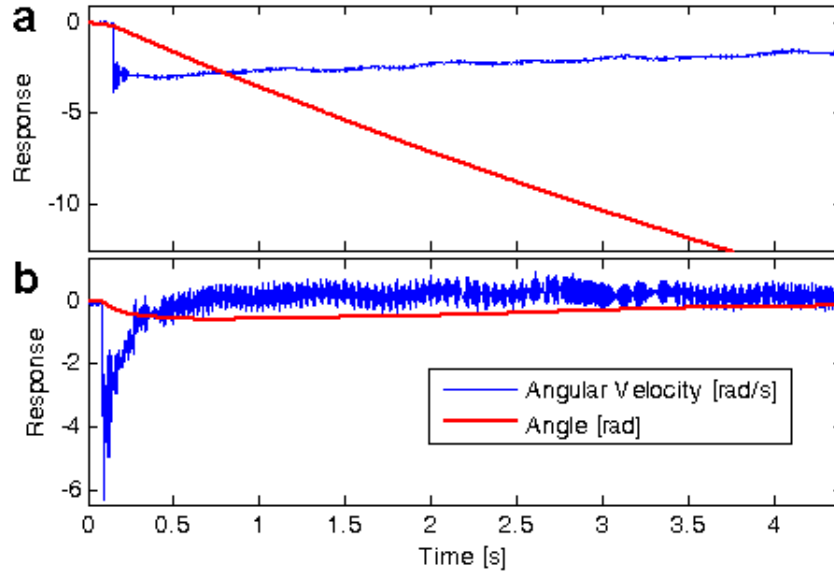


Figure 5.9: Impulse response of spinning discs: a) uncontrolled and b) PID controlled case.

5.4.2 In-flight Experiments

Initial experiments on PID controlled test flights have been performed, and some stabilizing effects were observable visually during the ascent, both by eye and by the on-board camera, but no evident features could be extracted out of the data. More robust experimental conditions will be devised to evaluate the controlled rocket behavior. In fact, for all of these experiments, widely varying launch conditions (e.g. wind, engine performance, and body condition) makes detailed analysis difficult.

Conclusion

In this thesis, an extremely low cost, off the shelf rocket system was demonstrated with the capability for attitude controlled flight. The very small 2g GINA controller was accurately able to estimate a portion of the state required for trajectory control and command actuators to control that state. Due to the short time of flight, sensor drift was negligible. For longer flight periods, unavoidable in the case of orbital launches, additional sensors would be necessary to compensate for these errors. Though beyond the scope of this thesis, sensors which directly measure position and attitude can be filtered together with the measured rates to generate a more accurate state estimate. Typically, magnetometers, GPS, and cameras complement inertial sensors for localization of robotic systems [14], [15]. In the case of minirockets, however, weight is at a premium, and so a minimum of additional sensors are desired. A camera looking at defined features such as the curvature of the earth or celestial bodies can resolve a full 6DOF state estimate, so it may be a good addition to the sensor suite.

On the hardware side, a more sophisticated system design with advanced materials like carbon fibers or similar should be used to decrease the structural mass. Once the rocket is capable of full attitude control, interesting experiments without stabilizing fins could be addressed, while decreasing aerodynamic influences. Obviously, since not the goal of this thesis, the reached velocities and altitudes are far from being sufficient for getting into LEO. Therefore and because attitude control alone is insufficient for orbital insertion, superior, thrust controlled engines need to be developed.

Acknowledgment

I want to thank my advisors Prof. Kris Pister and Prof. Ernst Meyer and the whole group in 471 Cory for their support. Special thanks to Ankur Mehta for providing the GINA controller and the associated help. In the end I want to thank Prof. Peter Seitz for making this all possible.

Bibliography

- [1] Kehl, F., and Mehta, A. (2009). "An Attitude Controller for Small Scale Rockets", submitted to: *ICRA10 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska May 3 - 8, 2010.
- [2] Romer, K., and Mattern, F. (2004). "The design space of wireless sensor networks", *IEEE Wireless Communications* **11** (6), pp. 54-61.
- [3] Hitt, D. L., Zakrzewski, C. M., and Thomas, M. A. (2001). "Mems-based satellite micropropulsion via catalyzed hydrogen peroxide decomposition", *Smart Materials and Structures* **10** (6), 1163-1175.
- [4] Vladimirova, T. et al. (2007). "Characterising wireless sensor motes for space applications", *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pp. 43-50.
- [5] Sarigul-Klijn, N., et al. (2004), "Air Launching Earth-to-Orbit Vehicles: Delta V gains from Launch Conditions and Vehicle Aerodynamics", *AIAA Paper* 2004-872.
- [6] Gizinski, S.J. and Wanagas, J.D. (1992), "Small Satellite Delivery Using a Balloon-Based Launch System", *AIAA Paper* 92-1845
- [7] Van Allen, J. A., 1959. Balloon-Launched Rockets for High-Altitude Research. In Newell, H. *Sounding Rockets*, Ch. 9.
- [8] Rainwater, E.L. and Smith, M.S. (2004), "Ultra high altitude balloons for medium-to-large payloads", *Advances in Space Research* **33** (10), pp. 1648-1652.
- [9] Toorian, A., Diaz, K., and Lee, S. (2008). "The cubesat approach to space access", *IEEE Aerospace Conference*, March 2008, pp. 1-14.
- [10] Nakka, R.A., 1984. *Solid Propellant Rocket Motor Design and Testing*. University of Manitoba.
- [11] Beeby, S. (2004). *MEMS Mechanical Sensors*, London: Artech House, Inc.
- [12] National Association of Rocketry. 2002. *NAR Certified Motors* [Online] (Updated 03 June 2009). Available at: <http://www.nar.org/SandT/NARenglist.shtml> [Accessed 15 Sept 2009].
- [13] Charlton, M. C. (2003). *A sounding rocket attitude determination algorithm suitable for implementation using low cost sensors*. Ph.D. Fairbanks: University of Alaska Fairbanks.

- [14] A. M. Eldridge, A. M. (2006). *Improved State Estimation for Miniature Air Vehicles*, Master's thesis: Brigham Young University.
- [15] Gemeiner, P., Einramhof, P., and Vincze, M. (2007). "Simultaneous Motion and Structure Estimation by Fusion of Inertial and Vision Data", *The International Journal of Robotics Research* **26** (6), pp. 591–605.

Appendix

A: Altitude Prediction

The numerical altitude predictions account for aerodynamic drag and gravity and are based on thrust-time data sheets from the National Association of Rocketry NAR for off-the-shelf model rocket engines [12] given the dimensions of the rocket and its time dependent mass. For the latter, we assumed the expelled mass to be proportional to the thrust. In the one dimensional case, the time dependent acceleration $a(t)$ acting on the rocket can be written as

$$a(t) = \frac{T(t) - m(t) \cdot g - \frac{1}{2}C_d\rho Av(t)^2}{m(t)}$$

where $T(t)$ is the time dependent thrust, $m(t)$ the time dependent mass, g the gravitational acceleration, C_d the drag coefficient, ρ the air density, A the exposed surface area and $v(t)$ the time dependent velocity.

B: Python Source Code

```
'''
AuTalk 1.0
-----
Based on MoteWin_Talk0.8 and AuTalk0.4.
'''

import serial
import numpy
import sys
import time
import math
import pythoncom, pyHook
import pygame
from pygame import locals
from quaternion import eul2quat, quatinv, quatrotate, quaternorm, quatnormalize, quat_v_rotate_test
from math import sin, cos, tan, pi
from numpy import array, dot, transpose, identity

def bin(x): # converts decimal to 16 bit binary sting
    return ''.join(x & (1 << i) and '1' or '0' for i in
        range(15,-1,-1))

def motespeak(LEDg, LEDo, mot1, mot0, srv1, srv2, srv3): # generates serial output string

    # check inputs and correct input if necessary
    if LEDg <= 0:
        LEDg = 0
    else:
        LEDg = 1

    if LEDo <= 0:
        LEDo = 0
    else:
        LEDo = 1
```

```

if mot1 <= 0:
    mot1 = 0
if mot1 >= 800:
    mot1 = 800

if mot0 <= 0:
    mot0 = 0
if mot0 >= 800:
    mot0 = 800

if srv1 <= -1000:
    srv1 = -1000
if srv1 >= 1000:
    srv1 = 1000

if srv2 <= -1000:
    srv2 = -1000
if srv2 >= 1000:
    srv2 = 1000

if srv3 <= -1000:
    srv3 = -1000
if srv3 >= 1000:
    srv3 = 1000

# set LED status
LEDg = LEDg
LEDo = LEDo

# mot 1
mot1 = mot1
mot1_b = bin(mot1)
mot1_bh = mot1_b[:8]
mot1_dh = int(mot1_bh, 2)
mot1_bl = mot1_b[8:]
mot1_dl = int(mot1_bl, 2)

# mot 0
mot0 = mot0
mot0_b = bin(mot0)
mot0_bh = mot0_b[:8]
mot0_dh = int(mot0_bh, 2)
mot0_bl = mot0_b[8:]
mot0_dl = int(mot0_bl, 2)

# servo 1
svr1 = srv1
svr1_b = bin(svr1)
svr1_bh = svr1_b[:8]
svr1_dh = int(svr1_bh, 2)
svr1_bl = svr1_b[8:]
svr1_dl = int(svr1_bl, 2)

# servo 2
svr2 = srv2
svr2_b = bin(svr2)
svr2_bh = svr2_b[:8]
svr2_dh = int(svr2_bh, 2)
svr2_bl = svr2_b[8:]
svr2_dl = int(svr2_bl, 2)

# servo 3
srv3 = srv3
srv3 = (srv3<<4)+((1-LEDg)<<1)+(1-LEDo)
srv3_b = bin(srv3)
srv3_bh = srv3_b[:8]
srv3_dh = int(srv3_bh, 2)
srv3_bl = srv3_b[8:]
srv3_dl = int(srv3_bl, 2)

# calculate checksum
chksum = mot1_dh + mot1_dl + mot0_dh + mot0_dl + svr1_dh + svr1_dl\
    + svr2_dh + svr2_dl + srv3_dh + srv3_dl
chksum_b = bin(~chksum)
chksum_b8 = chksum_b[8:]
chksum_d8 = int(chksum_b8, 2)

# build output string
ser_str = (chr(0x80) + chr(0x80) + chr(0x80) + chr(0x0a) + \
    chr(mot1_dl) + chr(mot1_dh) + chr(mot0_dl) + chr(mot0_dh) + chr(svr1_dl)\
    + chr(svr1_dh) + chr(svr2_dl) + chr(svr2_dh) + chr(svr3_dl) + chr(srv3_dh)\
    + chr(chksum_d8) + chr(0xa5))

```

```

        return ser_str

# define communication variables
s, n, m, numstar, numstar_intro, num_error, \
    drop_pkg, pkg = 0, 0, 0, 0, 0, 0, 0, ''

# define calc variables
XL_x, XL_y, XL_z = 0, 0, 0
GY_x, GY_y, GY_z = 0, 0, 0
Acc_b = array((0,0,0))*1.0
Acc_e = array((0,0,0))*1.0
Vx, Vy, Vz = 0, 0, 0
Sx, Sy, Sz = 0, 0, 0
precalib = 0
gyr_off_x, gyr_off_y, gyr_off_z = 0, 0, 0
w_x, w_y, w_z = 0, 0, 0
temp1, temp2 = 0, 0
count, runs, pkg_trnsm = 0, 0, 0
t2 = 0
t2_cal = 0
AccEcalib = 0

# number of calibration cycles (gyr, AccE)
num_calib = 1000
num_AccEcalib = 1500

# quaternion calc
omega_rot = array(((0,0,0,0),(0,0,0,0),(0,0,0,0),(0,0,0,0)))*1.0
quat = array((1,0,0,0))*1.0
quat_dot = array((0,0,0,0))*1.0
euler = array((0,0,0))*1.0
EulM = array(((0,0,0),(0,0,0),(0,0,0)))*1.0

# accelerometer scaling factors
cx, cy, cz = 0.00376798, 0.00372142, 0.00368486

# accelerometer offset factors
dx, dy, dz = 0.03493920, 0.04247200, 0.20666636

# gyro scaling factors
gx, gy, gz = 0.003308, 0.003238, 0.003093

# earth frame Acceleration
Acc_e_x, Acc_e_y, Acc_e_z = 0, 0, 0

# Body frame Angles
AngB_x, AngB_y, AngB_z = 0, 0, 0

# PID Control Parameter Setting
AngB_zPrevE = 0
AngB_zCorrI = 0
AngB_zSet = 0
Kp_z, Ki_z, Kd_z = 3, 0.5, 1
mot_break = 0

# Gimbaled Nozzle
wavecnt_x = 0
wavecnt_y = 0
center_x = -350
center_y = -180
nz_enable = 0
nz_count = 0

# set motor/servo inputs to 0
mot1_in = 0
mot0_in = 0
srv1_in = 0
srv2_in = 0 + center_x
srv3_in = 0 + center_y

# identify joystick
pygame.init()
pygame.joystick.init() # main joystick device system
try:
    j = pygame.joystick.Joystick(0) # create a joystick instance
    j.init() # init instance
    print 'Enabled joystick: ' + j.get_name()
except pygame.error:
    print 'no joystick found.'

# set serial COM-Port
ser = serial.Serial(6, baudrate=921600) #sets COM No. and Baudrate
filename = raw_input('Please enter filename: ') + '.txt' #define filename
print "Writing to file: %s" % filename

```



```

while gyr_count<100: # search for 3 stars in a row
    s = ser.read(1)
    if s == '*':
        numstar = numstar + 1
    else:
        numstar = 0
    pkg = pkg + s
    gyr_count = gyr_count + 1
    if numstar == 3: # 3 stars => start capture and process data
        pkg = pkg[:-3] # cut off the 3 stars
        if len(pkg) == 30: #checks if package has the expected size
            dat = numpy.fromstring(pkg[2:22], 'int16')
            rssi = numpy.fromstring(pkg[26], 'uint8')
            if rssi < 20: # if rssi signal to weak: drop packet
                drop_pkg = drop_pkg + 1
                break
        else: #gyro offset calculation
            gyr_off_x = gx*dat[5] + gyr_off_x
            gyr_off_y = gy*dat[6] + gyr_off_y
            gyr_off_z = gz*dat[7] + gyr_off_z
            precalib = precalib + 1
            if precalib == num_calib - 1:

                calib_time = str(calib_time)
                print '***Gyro Calibration Done***\n'
                print 'Gyro Calibration Duration: ', calib_time[0:5], 'seconds\n'
                print '***Acceleration Calibration Starting***'

                # if motors are running unmeant, stop them before launch
                mot_stop = 0
                while mot_stop < 100:
                    srv3_in = 0 + center_y
                    srv3_in = int(srv3_in)
                    srv2_in = 0 + center_x
                    srv2_in = int(srv2_in)
                    mot1_in = 0
                    mot1_in = int(mot1_in)
                    mot0_in = 0
                    mot0_in = int(mot0_in)

                    # send inputs to serial output
                    ser_out = motespeak(1, 0, mot1_in, mot0_in, srv1_in, srv2_in, srv3_in)
                    ser.write(ser_out)
                    mot_stop = mot_stop + 1

        else: # if package size != 30, package is discarded
            pkg = ''
            break

# start earth frame calibration (calculate offset)
while AccEcalib < num_AccEcalib:
    AccEcalib_count = 0
    calib2_time = time.clock()
    while AccEcalib_count<100: # search for 3 stars in a row
        s = ser.read(1)
        if s == '*':
            numstar = numstar + 1
        else:
            numstar = 0
        pkg = pkg + s
        AccEcalib_count = AccEcalib_count + 1
        if numstar == 3: # 3 stars => start capture and process data
            pkg = pkg[:-3] # cut off the 3 stars
            if len(pkg) == 30: #checks if package has the expected size
                dat = numpy.fromstring(pkg[2:22], 'int16')
                rssi = numpy.fromstring(pkg[26], 'uint8')
                if rssi < 20: # if rssi signal to weak: drop packet
                    drop_pkg = drop_pkg + 1
                    break
            else: #Earth frame acceleration offset calculation

                #reset time
                t_cal = time.clock()

                #time step dt
                dt_cal = t_cal - t2_cal
                t2_cal = t_cal

                # calculate angular rate w
                GY_x, GY_y, GY_z = dat[5], dat[6], dat[7]
                w_x = gx*GY_x - gyr_off_x/precalib
                w_y = -(gy*GY_y - gyr_off_y/precalib)
                w_z = gz*GY_z - gyr_off_z/precalib

```

```

# calculate body frame angle
AngB_x = AngB_x + w_x*dt_cal
AngB_y = AngB_y + w_y*dt_cal
AngB_z = AngB_z + w_z*dt_cal

# calculate body frame acceleration
#(x and y interchanged due to coordinate system set.)
XL_x, XL_y, XL_z = dat[2], dat[3], dat[4]
Acc_b_x = cy*XL_y + dy
Acc_b_y = cx*XL_x + dx
Acc_b_z = cz*XL_z + dz
Acc_b[0] = Acc_b_x
Acc_b[1] = Acc_b_y
Acc_b[2] = Acc_b_z

#rotate quaternion
q0 = quat[0]
q1 = quat[1]
q2 = quat[2]
q3 = quat[3]
quat_dot[0] = -0.5*(q1*w_x+q2*w_y+q3*w_z)+(1-quatnorm(quat))*q0
quat_dot[1] = 0.5*(q0*w_x+q2*w_z-q3*w_y)+(1-quatnorm(quat))*q1
quat_dot[2] = 0.5*(q0*w_y+q3*w_x-q1*w_z)+(1-quatnorm(quat))*q2
quat_dot[3] = 0.5*(q0*w_z+q1*w_y-q2*w_x)+(1-quatnorm(quat))*q3

#integrate quat_dot
quat = quat + dt_cal*quat_dot
quat = quatnormalize(quat)

#calc euler angles
euler[0] = math.atan2((2*(quat[0]*quat[1]+quat[2]*quat[3])),
(1-2*(quat[1]**2+quat[2]**2)))
euler[1] = math.asin(2*(quat[0]*quat[2]-quat[3]*quat[1]))
euler[2] = math.atan2((2*(quat[0]*quat[3]+quat[1]*quat[2])),
(1-2*(quat[2]**2+quat[3]**2)))
ph = euler[0]
th = euler[1]
psi = euler[2]

#calc earth frame acceleration
EulM = array(((math.cos(th)*math.cos(psi),math.cos(th)*math.sin(psi),-math.sin(th)),\
(math.sin(ph)*math.sin(th)*math.cos(psi)-math.cos(ph)*math.sin(psi),
math.sin(ph)*math.sin(th)*math.sin(psi)+math.cos(ph)*math.cos(psi),
math.sin(ph)*math.cos(th)),\
(math.cos(ph)*math.sin(th)*math.cos(psi)+math.sin(ph)*math.sin(psi),
math.cos(ph)*math.sin(th)*math.sin(psi)-math.sin(ph)*math.cos(psi),
math.cos(ph)*math.cos(th))))

Acc_e = dot(Acc_b,EulM)
Acc_e_x = Acc_e[0] + Acc_e_x
Acc_e_y = Acc_e[1] + Acc_e_y
Acc_e_z = Acc_e[2] + Acc_e_z
AccEcalib = AccEcalib + 1
if AccEcalib == num_AccEcalib - 1:
    calib2_time = str(calib2_time - calib_time)
    print '***Acceleration Calibration Done***\n'
    print 'Acceleration Calibration Duration: ', calib2_time[0:5], 'seconds\n'
    print '***Recording Data***\n'
    Acc_e_x_off = Acc_e_x/(num_AccEcalib-1)
    Acc_e_y_off = Acc_e_y/(num_AccEcalib-1)
    Acc_e_z_off = Acc_e_z/(num_AccEcalib-1)
    Acc_e_x = 0
    Acc_e_y = 0
    Acc_e_z = 0
    Vx, Vy, Vz = 0, 0, 0
    Sx, Sy, Sz = 0, 0, 0
    euler = array((0,0,0))*1.0
    w_x, w_y, w_z = 0, 0, 0
    AngB_zSet = AngB_z

# if motors are running unmeant, stop them before launch
mot_stop = 0
while mot_stop < 100:
    srv3_in = 0 + center_y
    srv3_in = int(srv3_in)
    srv2_in = 0 + center_x
    srv2_in = int(srv2_in)
    mot1_in = 0
    mot1_in = int(mot1_in)
    mot0_in = 0
    mot0_in = int(mot0_in)

```



```

# send inputs to serial output
ser_out = motespeak(1, 0, mot1_in, mot0_in, srv1_in, srv2_in, srv3_in)
ser.write(ser_out)
mot_stop = mot_stop + 1

else: # if package size != 30, package is discarded
    pkg = ''
    break

# start calculation for recording data
#reset time
t = time.clock() - calib2_time

#time step dt
dt = t - t2
t2 = t

# temperatures
temp1 = dat[8]
temp2 = dat[9]

#count
count = dat[0]

# calculate angular rate w
GY_x, GY_y, GY_z = dat[5], dat[6], dat[7]
w_x = gx*GY_x - gyr_off_x/precalib
w_y = -(gy*GY_y - gyr_off_y/precalib)
w_z = gz*GY_z - gyr_off_z/precalib

# calculate body frame angle
AngB_x = AngB_x + w_x*dt
AngB_y = AngB_y + w_y*dt
AngB_z = AngB_z + w_z*dt

# calculate body frame acceleration
#(x and y interchanged due to coordinate system set.)
XL_x, XL_y, XL_z = dat[2], dat[3], dat[4]
Acc_b_x = cy*XL_y + dy
Acc_b_y = cx*XL_x + dx
Acc_b_z = cz*XL_z + dz
Acc_b[0] = Acc_b_x
Acc_b[1] = Acc_b_y
Acc_b[2] = Acc_b_z

#rotate quaternion
q0 = quat[0]
q1 = quat[1]
q2 = quat[2]
q3 = quat[3]
quat_dot[0] = -0.5*(q1*w_x+q2*w_y+q3*w_z)+(1-quatnorm(quat))*q0
quat_dot[1] = 0.5*(q0*w_x+q2*w_z-q3*w_y)+(1-quatnorm(quat))*q1
quat_dot[2] = 0.5*(q0*w_y+q3*w_x-q1*w_z)+(1-quatnorm(quat))*q2
quat_dot[3] = 0.5*(q0*w_z+q1*w_y-q2*w_x)+(1-quatnorm(quat))*q3

#integrate quat_dot
quat = quat + dt*quat_dot
quat = quatnormalize(quat)

#calc euler angles
euler[0] = math.atan2((2*(quat[0]*quat[1]+quat[2]*quat[3])), (1-2*(quat[1]**2+quat[2]**2)))
euler[1] = math.asin(2*(quat[0]*quat[2]-quat[3]*quat[1]))
euler[2] = math.atan2((2*(quat[0]*quat[3]+quat[1]*quat[2])), (1-2*(quat[2]**2+quat[3]**2)))
ph = euler[0]
th = euler[1]
psi = euler[2]

#calc earth frame acceleration
EulM = array(((math.cos(th)*math.cos(psi),math.cos(th)*math.sin(psi),-math.sin(th)),\
    math.sin(ph)*math.sin(th)*math.sin(psi)+math.cos(ph)*math.cos(psi),math.sin(ph)*math.cos(th)),\
    math.cos(ph)*math.sin(th)*math.sin(psi)-math.sin(ph)*math.cos(psi),math.cos(ph)*math.cos(th))))

Acc_e = dot(Acc_b,EulM)
Acc_e_x = Acc_e[0] - Acc_e_x_off
Acc_e_y = Acc_e[1] - Acc_e_y_off
Acc_e_z = Acc_e[2] - Acc_e_z_off

# velocity
Vx = Vx + dt*-Acc_e_x*9.81
Vy = Vy + dt*-Acc_e_y*9.81
Vz = Vz + dt*-Acc_e_z*9.81
Sx = Sx + dt*Vx

```

```

Sy = Sy + dt*Vy
Sz = Sz + dt*Vz

# PID Control
AngB_zErr = AngB_zSet - AngB_z
if AngB_zErr < 0:
    AngB_zErr = -AngB_zErr%6.283
    AngB_zErr = -AngB_zErr
elif AngB_zErr > 0:
    AngB_zErr = AngB_zErr%6.283

AngB_zCorrI = AngB_zCorrI + AngB_zErr*dt
AngB_zCorrD = (AngB_zErr - AngB_zPrevE)/dt
AngB_zOut = Kp_z*AngB_zErr + Ki_z*AngB_zCorrI + Kd_z*AngB_zCorrD

if AngB_zErr < 0:
    AngB_zPrevE = -AngB_zErr%6.283
    AngB_zPrevE = -AngB_zPrevE
elif AngB_zErr > 0:
    AngB_zPrevE = AngB_zErr%6.283

if AngB_zOut < 0:
    mot1_in = -AngB_zOut*500
    mot1_in = int(mot1_in)
    mot0_in = 0
elif AngB_zOut > 0:
    mot0_in = AngB_zOut*500
    mot0_in = int(mot0_in)
    mot1_in = 0

# Auto Gimbaled Nozzle

if Vz > 0.5:
    nz_enable = 1

if nz_enable == 1:
    nz_count = nz_count + 1

if nz_count > 10:
    srv2_in = AngB_y * 900 + center_x
    srv2_in = int(srv2_in)
    srv3_in = AngB_x * 900 + center_y
    srv3_in = int(srv3_in)
    '''if nz_count >= 666:
        srv2_in = (AngB_y + math.pi/4) * 900 + center_x
        srv2_in = int(srv2_in)
        srv3_in = (AngB_x + math.pi/4) * 900 + center_y
        srv3_in = int(srv3_in)'''

# Nozzle Maxima
if srv2_in < -800:
    srv2_in = -800
if srv2_in > -25:
    srv2_in = -25
if srv3_in < -450:
    srv3_in = -450
if srv3_in > 190:
    srv3_in = 190

'''srv2_in = AngB_y * 800
srv2_in = int(srv2_in)
srv3_in = AngB_x * 800
srv3_in = int(srv3_in)'''

#write into file
file.write('(t)10f \t %(Acc_e_x)10f \t %(Acc_e_y)10f \t %(Acc_e_z)10f\
\t %(Vx)10f \t %(Vy)10f \t %(Vz)10f \t %(Sx)10f \t %(Sy)10f \t %(Sz)10f\
\t %(ph)10f \t %(th)10f \t %(psi)10f \t %(Acc_b_x)10f \t %(Acc_b_y)10f \t %(Acc_b_z)10f \t\
\t %(w_x)10f \t %(w_y)10f \t %(w_z)10f \t %(AngB_x)10f \t %(AngB_y)10f \t %(AngB_z)10f \t\
\t %(XL_x)d \t %(XL_y)d \t %(XL_z)d \t %(GY_x)d \t %(GY_y)d\
\t %(GY_z)d \t %(temp1)d \t %(temp2)d \t %(count)d \t %(drop_pkg)d \t %(lqi)d \t %(rssi)d \t\
\t %(q0)10f \t %(q1)10f \t %(q2)10f \t %(q3)10f \t %(srv2_in)d \t %(srv3_in)d \t %(mot0_in)d\
\t %(mot1_in)d \n' % vars())

```

```

        # listen to joystick events
        for e in pygame.event.get(): # iterate over event stack
            if e.type == pygame.locals.JOYAXISMOTION:
                joyLx , joyLy = j.get_axis(0), j.get_axis(1)
            if e.type == pygame.locals.JOYAXISMOTION:
                joyRx , joyRy = j.get_axis(2), j.get_axis(3)
            elif e.type == pygame.locals.JOYBALLMOTION:
                break
            elif e.type == pygame.locals.JOYHATMOTION:
                break
            elif e.type == pygame.locals.JOYBUTTONDOWN:
                break
            elif e.type == pygame.locals.JOYBUTTONUP:
                joyLx, joyLy, joyRx, joyRy = 0, 0, 0, 0

        # scale joystick events
        srv3_in = joyLx*700
        srv3_in = int(srv3_in)
        srv2_in = joyLy*700
        srv2_in = int(srv2_in)
        mot1_in = joyRx*1000
        mot1_in = int(mot1_in)
        mot0_in = -joyRy*1000
        mot0_in = int(mot0_in)

        print srv2_in, '\t', srv3_in, '\t', mot0_in, '\t', mot1_in

        # send inputs to serial output
        #mot0_in = 0
        #mot1_in = 0
        #srv2_in = center_y
        ser_out = motespeak(1, 0, mot1_in, mot0_in, srv1_in, srv2_in, srv3_in)
        ser.write(ser_out)
        #print Sz, mot0_in, mot1_in, AngB_zErr, AngB_zOut

    runs = runs + 1
    pkg = ''
    dat = ''

    else: # if package size != 30, package is discarded
        pkg = ''
        num_error = num_error + 1 # and an Error is counted
        break

except: #stops while 1 loop
    print '***Transmission Done***\n\n'
    print runs, 'Packages Transmitted'
    print num_error, 'Errors occurred'
    print drop_pkg, 'Packages dropped\n'
    comments = raw_input('Comments: ')
    pkg_trnsm = runs - num_error

# if motors are running unmeant, stop them before launch
mot_stop = 0
while mot_stop < 1000:
    srv3_in = 0 + center_y
    srv3_in = int(srv3_in)
    srv2_in = 0 + center_x
    srv2_in = int(srv2_in)
    mot1_in = 0
    mot1_in = int(mot1_in)
    mot0_in = 0
    mot0_in = int(mot0_in)

    # send inputs to serial output
    ser_out = motespeak(1, 0, mot1_in, mot0_in, srv1_in, srv2_in, srv3_in)
    ser.write(ser_out)
    mot_stop = mot_stop + 1

#write footer
file.write('\nComments: ' + comments)
file.write('\nCalibration Cycles: ' + str(num_calib))
file.write('\nPackages transmitted: ' + str(pkg_trnsm))
file.write('\nErrors occurred: ' + str(num_error) + '\n\n')
file.write('Accelerometer Scaling Factors:\n' + str(cnx, cy, cz: '\n'
+ str(cx) + '\t' + str(cy) + '\t' + str(cz) + '\n\n')
file.write('Accelerometer Offset Factors:\n' + str(dx, dy, dz: '\n'
+ str(dx) + '\t' + str(dy) + '\t' + str(dz) + '\n\n')
file.write('Gyro Scaling Factors:\n' + str(gx, gy, gz: '\n'
+ str(gx) + '\t' + str(gy) + '\t' + str(gz) + '\n\n')
file.write('Calibration Cycles:\n' + str(num_gyr, num_acc: '\n'
+ str(num_calib) + '\t' + str(num_acc) + '\n\n')

```

```

        # closes serial port and file, ends program
        ser.close()
        file.close()
        end = raw_input('Press any key to end program')
        sys.exit()

quaternion.py
*****

import math
from math import sin, cos, tan, pi
from numpy import array, dot

def quat2dcm(q):
    dcm = array(((0,0,0),(0,0,0),(0,0,0))) * 1.0

    dcm[0,0] = q[0]**2 + q[1]**2 - q[2]**2 - q[3]**2;
    dcm[0,1] = 2.*(q[1]*q[2] + q[0]*q[3]);
    dcm[0,2] = 2.*(q[1]*q[3] - q[0]*q[2]);
    dcm[1,0] = 2.*(q[1]*q[2] - q[0]*q[3]);
    dcm[1,1] = q[0]**2 - q[1]**2 + q[2]**2 - q[3]**2;
    dcm[1,2] = 2.*(q[2]*q[3] + q[0]*q[1]);
    dcm[2,0] = 2.*(q[1]*q[3] + q[0]*q[2]);
    dcm[2,1] = 2.*(q[2]*q[3] - q[0]*q[1]);
    dcm[2,2] = q[0]**2 - q[1]**2 - q[2]**2 + q[3]**2;

    return dcm

def quatinv(q):
    q = -q
    q[0] = -q[0]
    return q

def eul2quat(ph, th, psi):
    cp = cos(ph/2.0); ct = cos(th/2.0); cs = cos(psi/2.0)
    sp = sin(ph/2.0); st = sin(th/2.0); ss = sin(psi/2.0)

    return array((
        cp * ct * cs + sp * st * ss,
        sp * ct * cs - cp * st * ss,
        cp * st * cs + sp * ct * ss,
        cp * ct * ss - sp * st * cs
    ))

def quatrotate(q, v):
    return(dot(quat2dcm(q), v))

def quatnorm(q):
    q = math.sqrt(q[0]**2+q[1]**2+q[2]**2+q[3]**2)
    return q

def quatnormalize(q):
    q = q/quaternion(q)
    return q

def quat_v_rotate_test(q,v):
    quat_v_rot = array(((0,0,0),(0,0,0),(0,0,0))) * 1.0

    quat_v_rot[0,0] = (1-2*q[2]**2-2*q[3]**2)
    quat_v_rot[0,1] = 2*(q[1]*q[2]+q[0]*q[3])
    quat_v_rot[0,2] = 2*(q[1]*q[3]-q[0]*q[2])
    quat_v_rot[1,0] = 2*(q[1]*q[2]-q[0]*q[3])
    quat_v_rot[1,1] = (1-2*q[1]**2-2*q[3]**2)
    quat_v_rot[1,2] = 2*(q[2]*q[3]+q[0]*q[1])
    quat_v_rot[2,0] = 2*(q[1]*q[3]+q[0]*q[2])
    quat_v_rot[2,1] = 2*(q[2]*q[3]-q[0]*q[1])
    quat_v_rot[2,2] = (1-2*q[1]**2-2*q[2]**2)

    v = dot(quat_v_rot,v)
    return v

```