DESIGN OF AN INERTIAL NAVIGATION UNIT USING MEMS SENSORS

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by
Maksim Eskin
Project Advisor: Bruce Land
Degree Date: January 2006

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title: Design of an Inertial Navigation Unit using MEMS sensors

Author: Maksim Eskin

Abstract: Inertial navigation systems are used in many situations where the use of an external reference to measure position is impractical or unreliable. Typical inertial navigation systems used in aeronautics and marine applications are highly advanced pieces of equipment costing thousands of dollars. However, inexpensive accelerometers and angular rate sensors (gyros) can be used to make a far less accurate inertial navigation unit for around $100. Such a design is implemented. The calibration of the sensors to minimize error is discussed. Finally, static tests are carried out to estimate the length of time over which the system can be considered reliable. The system is found to have an approximate position error of 0.17 m and velocity error of 0.3 m/s after one second, making it potentially useful for increasing accuracy of a GPS system, where typical accuracies are in the 1 to 3 meter range.

Report Approved by
Project Advisor:_____Date:_____

Executive Summary

Inertial navigation systems are used in many situations where the use of an external reference to measure position is impractical or unreliable. Typical inertial navigation systems used in aeronautics and marine applications are highly advanced pieces of equipment costing thousands of dollars. However, inexpensive accelerometers and angular rate sensors (gyros) can be used to make a far less accurate inertial navigation unit for around $100.

Applications of such systems include human motion tracking for capturing gestures, or enhancement of existing sensor systems like GPS or magnetic compasses. The design implemented in this report uses three Analog Devices MEMS rate gyros, a three-axis Kionix MEMS accelerometer, and a Microchip dsPIC 16-bit microcontroller.

Proper calibration is explored as a means of improving the system accuracy, as the parameters of the sensors used are not as stable or as closely specified as their more advanced counterparts. A least squares approach is used to estimate the accelerometer and the gyro bias parameters by holding the inertial navigation system in a set of different orientations.

A software design for the system is presented and the performance is evaluated using static tests. The system is found to have an approximate position error of 0.17 m and velocity error of 0.3 m/s after one second, making it potentially useful for increasing accuracy of a GPS system, where typical accuracies are in the 1 to 3 meter range.

Contents

I Introduction

1. Overview of Navigation

There are many instances where it is desirable to know the position and velocity of an object for purposes of navigation or guidance. These include household robots, hikers, land, sea, and air vehicles, missiles, and spacecraft. Position information is also used for surveying or mapping, as well as remote tracking of position. There are many other examples.

Methods of measuring position and velocity are just as numerous. They include fixed land references such as beacons, devices measuring motion relative to a fixed medium such as the ground, atmosphere, or earth's magnetic field, and satellite navigation systems like GPS. However varied all these methods may be, they all resemble each other in that they involve measurement with respect to a reference with known position and velocity. On one hand, this is part of the nature of the problem, since position is by definition relative. However, there are numerous cases where a moving object's initial position is known, but its subsequent motions cannot be conveniently tracked with respect to a reference. In these cases, an inertial navigation system is used.

Examples of such applications include submarines, which cannot use radio navigation due to water's opacity to radio waves, and are a prominent early example of inertial navigation [12]. Aircraft are perhaps the most important application today, because they often travel in conditions of low visibility and must be able to maintain level flight without ground references. Missiles, especially ballistic missiles, are designed to operate in an environment where any given reference might be jammed or destroyed by the enemy.

Even in cases where a reference is normally available, it might be necessary to use an inertial navigation system in case of a momentary outage.

This specifically applies to GPS and other radio navigation systems, which can become unavailable unexpectedly and for minutes at a time due to bad weather or other blockage of the signal. In cities and other obstacle- rich environments, a GPS signal can be quite difficult to obtain, which impedes GPS navigation by car.

Finally, even in cases where the external reference is currently available, it is often desirable to combine the outputs of multiple sensors, in order to increase precision. These latter two cases are where an inertial navigation system like the one described in this report can be helpful.

2. Inertial Navigation

An inertial navigation system (INS) uses two types of sensors called accelerometers and gyros to measure its motion parameters. A prototypical accelerometer contains a mass suspended on a spring, with some way of measuring the extent to which the spring is compressed. When the accelerometer's body is accelerated, a force is transmitted to the mass through the spring, causing the spring to stretch or contract. This can then be measured, and results in a value proportional to the accelerometer's acceleration.

A gyro is a device that measures the rate of rotation around the gyro axis. The earliest gyros were actual spinning gyroscopes which, when rotated perpendicularly to their spin axis, will produce a force which can be measured. Today, Ring Laser Gyros and Fiber Optic Gyros are the most common types of high-end gyro [8]. At the low end, numerous designs of MEMS gyros exist. The type used in this project measures the coriolis effect on a vibrating structure. This essentially the same principle as a spinning wheel gyro, but instead of a continuous rotation, an oscillation is exploited. Since the resonant frequency is higher for smaller structures, it is actually possible to achieve greater sensitivity for smaller gyros [7].

In the early days of inertial navigation, the gyros were used in a feedback loop with a motorized gimbal to keep the sensor in a desired orientation for the duration of the mission. This is termed a stable platform INS. The accelerometer readings then corresponded to the acceleration of the vehicle in some useful coordinate frame. These could then be integrated to produce velocity and again to produce position.

It is easy to see that over time, even a small inaccuracy in acceleration measurement will result in an enormous error in position. This is the essential limitation of an inertial navigation system. The AIRS (Advanced Inertial

Reference Sphere), which is used in the Minuteman III ICBM, has gyro drift rates of 1.5×10^{-5} degrees per hour [1] (i.e. the estimated orientation will be off by that much after an hour from the initial). However, it costs over a million dollars and takes a year to assemble. A typical INS used on a ship or in an airplane will have a drift rate of around 0.01 degrees per hour. Such a device will have a position error drift of around 0.6 miles per hour [8], its cost being around $20,000. The MEMS gyros used in this project might be expected to have a drift of 1 to 10 degrees per hour, and the position error drift over an hour is essentially infinite, but the cost might be around $100.

So-called strapdown systems, of which this project is an example, do not use a stabilized platform, and use the gyro outputs to calculate the orientation of the unit in a global coordinate frame, then perform a rotation on the local acceleration to find acceleration in the global coondinate frame. Although this eliminates the mechanical components of the gimbal, it is significantly more demanding computationally. It also requires the gyros to have a greater measurement range. However, in the context of a low-cost MEMS system, a gimbal would not be justified, and it would cause more power consumption than desirable in a small handheld or automotive device.

3. Potential of a MEMS INS

One of the inspirations for this project was an article called Navigating the City in GPS World [5]. This article describes the use of a strapdown MEMS INS in conjunction with GPS to reliably navigate a city in a car or by foot. The periodic stops that a car makes at intersections were used to reset the INS using the GPS estimate. This prevented the position estimate from diverging, and allowed a reasonable trajectory plot.

A very meticulous design that used a commercial MEMS inertial measurement system achieved around 1 km error in 5 minutes [11]. This is a typical result for such systems, and makes them all but useless over times much longer than a second. However, applying what are called non holonomic constraints, which take into account the fact that the INS is mounted on a vehicle and only certain types of motion are possible, reduced the error to tens of meters over 20 minutes (varying with vehicle velocity).

If we casually assume that the rate of error divergence is linear, then a 5-minute 1 km error would lead to a 1-second error of 3 meters. Averaging N measurements with equal variance reduces the variance by $1/\sqrt{N}$ (assuming Gaussian noise). So given a GPS unit that outputs 3-meter RMS error

Parameter	Value
Position error	3 m per second
Velocity error	1 m/s per second
Cost	$100

Table 1: Desired Performance for this INS design

positions once a second, the total error of the system will go down to around 2 meters. At the end of every measurement cycle, the new position is fed back to the INS as a new initial value.

Adding the aforementioned non-holonomic constraints, or other constraints that depend on the specific nature of the problem, can also improve things. Human motion tracking is a popular application where motion is obviously greatly constrained by the possible trajectories that a human appendage is capable of making. Small unmanned aerial vehicles are another. In short, there is a wide range of applications for a MEMS INS where one or both of the following conditions are met:

- A model exists to constrain what motion is possible
- The INS is augmented by another sensor or set of sensors

The desired goals for this INS design are therefore not very restrictive, but a demonstrated performance of a few kilometers error in a few minutes would place it in the range of existing designs. To be at all practical, the MEMS INS must be significantly cheaper to justify its lack of precision. Table 1 summarizes desired project performance.

II Hardware Design

1. Component Selection

The most important component of the design is, of course, the MEMS sensors themselves. These were not selected at all, because some were available on hand, and all MEMS sensors that are available today are fairly comparable to each other. The accelerometer used is a Kionix KXM52-1050 three-axis accelerometer with a dynamic range of $\pm$2g, costing around $20 per unit. The gyros are three Analog Devices ADXRS401s which have a range of $\pm 70^o/s$

Model	Range	RMS Noise	Sensitivity	Bandwidth	Cost
Gyros					
Analog ADXRS401	$\pm 70^o$/s	0.05 $^o/s/\sqrt{Hz}$	15 mV/^{o}/s	40 Hz	$22.00
Silicon Sensing CRS03	$\pm 100^o$/s	0.05 $^o/s/\sqrt{Hz}$	20 mV/^{o}/s	10 Hz	$304.00
Accelerometers					
Kionix KXM52	$\pm 2g$	50 $\mu g/\sqrt{Hz}$	1 V/g	3 kHz	$20.00
Analog ADXL322	$\pm 2g$	220 $\mu g/\sqrt{Hz}$	420 mV/g	2.5 kHz	$3.75
MEMSic MXR7202GL	$\pm 2g$	300 $\mu g/\sqrt{Hz}$	300 mV/g	20 Hz	$9.30
STMicro LIS3L02	$\pm 2g$	50 $\mu g/\sqrt{Hz}$	1 V/g	100 Hz	?

Table 2: A comparison of some available MEMS sensors (bold indicates what was used)

and likewise cost around $20 per unit. This leaves around $20 for the rest of the hardware. Table 2 summarizes the performance of the MEMS sensors, with some other models included for comparison. The sensitivity, RMS noise, and bandwidth all come into play in the final performance of the INS, but in ways that aren't very straightforward. According to a qualitative table in Bar-Shalom ([4], p.500), both of these sensors are in the medium to low category of inertial sensor performance based on their noise specifications. This means they would not even be considered for a true INS in an aircraft or missile, but we will use them.

The design of the hardware is primarily driven by cost. Given greater computing resources, the performance of the INS could be increased somewhat, but it seems that more than doubling the price for a 40 MHz 32-bit PowerPC type processor is not worth the gain in performance that is in the end still limited by the sensors. There are plenty of cheap 8-bit microcontrollers on the market. However, the Microchip dsPIC platform was selected for its 16-bit capability. The dsPIC 30F2012 was used for this design. It includes 10 12-bit A/D channels, and can run at up to 30 MHz. It also comes in a convenient DIP package, has free development tools, and is easy to program. The 16-bit processing, however, is the crucial bit, since it halves the processing time for 16-bit quantities, which all of our state parameters must be to have any acceptable precision. Hardware floating point would be even nicer, but no inexpensive microcontrollers have one.

The dsPIC 30F2012 cost $10, bringing the total cost of the design to $90

excluding discrete parts, connectors, and the printed circuit board. In large quantities the total will not exceed $100 in parts, or at least not by much.

2. Circuit Design

A printed circuit board was fabricated using the ExpressPCB service with a ground plane, separate power buses for analog and digital circuits, and plenty of decoupling capacitors. Appendix 1 includes the circuit diagram and the PCB layout. See Figure 1 for photographs of the board. Overall, the noise reduction techniques practiced seem to have helped, given that the RMS noise on the analog power line was measured at 8 mV at the A/D reference input, in comparison to the RMS noise on the digital line which was 50 mV. The gyro power lines had an RMS noise of 3 mV. Clearly, the less noise the better, because, as explained above, the smallest error accumulates over time.

A serial interface chip is included on the board for PC communication, and four pins are reserved for data transfer to another device (say, one that does INS/GPS integration). Three of the ADC channels are used for the accelerometer axes, three for the gyro axes, and one for a gyro temperature output which is used to compensate for bias drift due to temperature. A 20 MHz crystal using the 4/3 on-chip PLL achieves a clock rate of 26.6 MHz.

The total current draw at 9V for the circuit was found to be 0.1 A, which is considerable, but could be reduced by around a factor of three using switching and/or low-dropout regulators (linear voltage regulators are about 37% efficient).

3. Processor Time Budget

There are three essential things that the INS must do. First, it must read the A/D inputs at the required rate, then convert them into some useful form, then output them. It may also need to accept initial conditions or commands as inputs.

To start with, the sampling rate for the sensors must be more than twice the sensor bandwidth (the Nyquist Criterion). For the gyros this means more than 80 Hz. Note that in an INS on a military jet , the gyro would be updated at a much higher frequency, 2 kHz or so. For a gimballed system, 30 Hz would be sufficient [8]. For the accelerometers, it means more than 6 kHz. Since fundamentally, the position estimation can only be updated as

often as the gyro data, the accelerometer data can be sampled at the full rate and averaged, or low pass filtered and only sampled at the gyro rate. It seems that the first approach should be better for noise.

The time to sample and convert three channels from the A/D converter was determined experimentally to be 84μs. If sleep mode is entered before sampling (to reduce noise from the microcontroller operation), the time goes up to a minimum of 90μs but can go as high as 400μs. This depends on how long the oscillator takes to restart after being shut off. Getting all 7 inputs in sleep mode takes at least 350μs. Since the gyro inputs are sampled less often than the accelerometers, it makes sense to use sleep mode for them, and not use sleep mode for the accelerometers. 80 times 350 μs is 28 ms. 6000 times 84 μs is 0.5 s. So, using the full sampling rate, around half the processor time is consumed sampling the gyro and accelerometer data. This can be decreased should there be time constraints. This leaves 2166 instruction cycles per accelerometer sample. We will see in the software design section just what needs to be done in this time.

III INS Mechanization

1. Basic Concepts

Mechanization is a fancy term used to describe the mathematical analysis of INS data to generate velocity and position information in a useful coordinate frame. In this section the basic design for the INS mechanization will be described. The implementation will be discussed in more detail in the software design section.

The basic information found by the sensors is the acceleration $\hat{a}$ and angular velocity $\hat{\omega}$. The main complication is that $\hat{a}$ includes the actual acceleration, the force of gravity, and (to a small extent) the coriolis effect of earth's rotation on a moving object. If the direction of the gravity vector is known exactly, it can be subtracted from the $\hat{a}$ and all is well. However, the estimated attitude of the INS will gradually diverge from its true attitude because of errors in the gyros. As this happens, the gravity vector will no longer be correctly subtracted from $\hat{a}$ and then the acceleration will appear to be bigger than it actually is. This causes the velocity and position estimates to diverge. Hence, estimating attitude is just as important if not more so than estimating velocity itself.

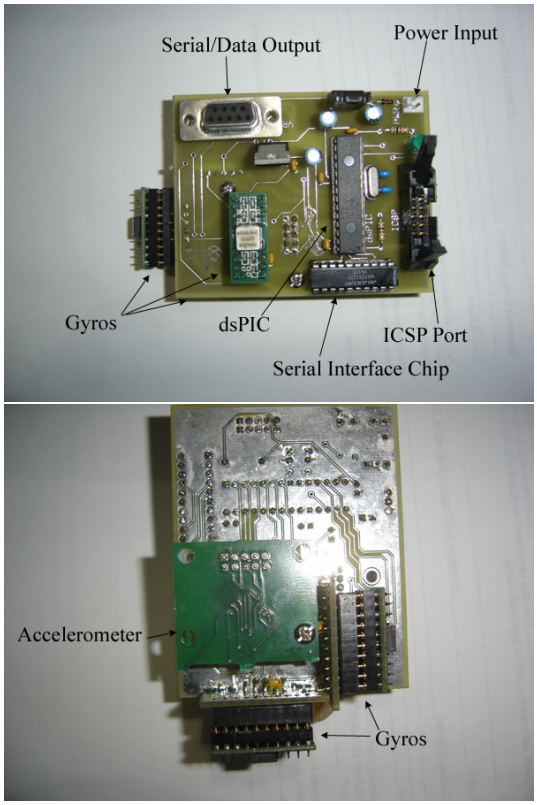


Figure 1: Top and bottom views of INS board

Various coordinate frames have been referred to casually in this report but it is now time to define them with slightly more rigor. The local coordinate frame of the INS, with respect to which it makes all of its measurements, will be called the body frame. There are numerous possible choices for a more global frame, but we will choose what are called navigation coordinates, which use the local vertical, north, and east directions centered on the INS. Many systems use earth centered coordinates that make it easier to take earth's curvature into account, but since our INS is useless over the time it would take to travel a significant distance on the earth, the local coordinates will suit us just fine. Furthemore, they are intuitive to people, which makes them easy to use. Further, instead of working in terms of latitude, longitude, and altitude for position, we will work with location relative to the initial position and convert as necessary.

At every measurement cycle, the INS must find the coordinate transformation matrix from body to navigation coordinates based on the integrated gyro measurements, then use that to transform the accelerometer readings into the navigation frame. At that point the gravity vector is subtracted, and the velocity, attitude, and position are updated.

2. Some Math

The best treatment of the INS measurement equations, online or in print, has been found in Shin [11]. It appears that the traditional method of passing along knowledge about INS design is through expensive seminars taught by INS engineers. Perhaps this is due to the fact that INS have historically been developed by defense companies that do not have the same level of openness as academia. In any case, it seems that few sources exist that offer a comprehensive discussion of INS design and are written after 1970 (all INS developed before about 1980 were gimballed). Shin himself got his information from one of the aforementioned seminars. In any case, the mechanization equations are mostly based on his presentation.

High-performance INS must take into account the earth's angular velocity about its axis, which is easily computed as $\Omega_e = 7.27 \times 10^{-5}$ rad/sec. Our gyros have a sensitivity of 15 mV/^{o}/s, so the earth's rotation would translate to 1 μV. It can therefore be safely ignored. The coriolis effect mentioned above is the angular velocity of the navigation frame with respect to an inertial frame (usually a nonrotating frame centered at the earth's center). The navigation frame rotates due to the earth's rotation, as well as the INS motion along the earth's surface. Thus,

$$\omega_{in} = \begin{bmatrix} \Omega_e \cos(\phi) + v_E/R_e \\ -v_N/R_e \\ -\Omega_e \sin(\phi) - v_E \tan(\phi)/R_e \end{bmatrix}$$

Where ϕ is the current latitude and v_E and v_N are east and north velocity. $R_e \sim 6 \times 10^6$ m is earth's radius. Given a latitude of 45^o and velocity of 1 m/s both east and north, the resulting acceleration is $a_c = \omega_{in} \times v$. $\|a_c\| = 1.26 \times 10^{-4}$g. This is not quite as insignificant as the gyro effect. However, considering that the accelerometer sensitivity is 1 V/g, this comes out to 0.1 mV, ten times smaller than the A/D converter can sense. The v/R_e terms turn out to be much smaller than the earth rotation terms for small velocities, so $\|a_c\|$ is roughly linear in $\|v\|$. It becomes 1 mg at $\|v\|$=14 m/s,

which is 32 miles per hour. So this term might be of marginal importance in highway driving, but certainly not in human motion. It will be included for completeness.

So the first two measurement equations for our INS are:

$$\begin{aligned} \dot{r}_n &= v_n & (1)\\ \dot{v}_n &= C_b^n \hat{a} - a_c - g & (2) \end{aligned}$$

where r is position and v is velocity (the n subscript indicates that it is in the navigation frame). C_b^n is the coordinate transformation matrix, which must be now found.

A mathematical entity called the quaternion comes into play. This is the generally used method of coordinate rotation for strapdown systems [7]. There is much to be said about them, but they will be treated here simply as four-vectors. Given three angles θ_x, θ_y, θ_z, representing roll, pitch, and yaw, the corresponding quaternion is:

$$q = \begin{bmatrix} \sin(\frac{1}{2}\theta_x)\cos(\frac{1}{2}\theta_y)\cos(\frac{1}{2}\theta_z) - \cos(\frac{1}{2}\theta_x)\sin(\frac{1}{2}\theta_y)\sin(\frac{1}{2}\theta_z) \\ \cos(\frac{1}{2}\theta_x)\sin(\frac{1}{2}\theta_y)\cos(\frac{1}{2}\theta_z) + \sin(\frac{1}{2}\theta_x)\cos(\frac{1}{2}\theta_y)\sin(\frac{1}{2}\theta_z) \\ \cos(\frac{1}{2}\theta_x)\cos(\frac{1}{2}\theta_y)\sin(\frac{1}{2}\theta_z) - \sin(\frac{1}{2}\theta_x)\sin(\frac{1}{2}\theta_y)\cos(\frac{1}{2}\theta_z) \\ \cos(\frac{1}{2}\theta_x)\cos(\frac{1}{2}\theta_y)\cos(\frac{1}{2}\theta_z) + \sin(\frac{1}{2}\theta_x)\sin(\frac{1}{2}\theta_y)\sin(\frac{1}{2}\theta_z) \end{bmatrix} \quad (3)$$

This is useful for initialization. Further, the derivative of a quaternion is simply $\dot{q} = \frac{1}{2} q_\omega q$, which in matrix form is:

$$\dot{q} = \frac{1}{2} \begin{bmatrix} 0 & \omega_z & \omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix} q \quad (4)$$

Shin presents the corresponding equation in discrete time ([11], p.23):

$$\begin{aligned} \Delta\theta &= \sqrt{\Delta\theta_x^2 + \Delta\theta_y^2 + \Delta\theta_z^2} & (5)\\ s &= \frac{2}{\Delta\theta}\sin(\frac{\Delta\theta}{2}) & (6)\\ c &= 2(\cos(\frac{\Delta\theta}{2} - 1) & (7)\\ q_{k+1} &= q_k + \frac{1}{2} \begin{bmatrix} c & s\Delta\theta_z & -s\Delta\theta_y & s\Delta\theta_x \\ -s\Delta\theta_z & c & s\Delta\theta_x & s\Delta\theta_y \\ s\Delta\theta_y & -s\Delta\theta_x & c & s\Delta\theta_z \\ -s\Delta\theta_x & -s\Delta\theta_y & -s\Delta\theta_z & c \end{bmatrix} q_k & (8) \end{aligned}$$

$$= \frac{1}{2}\Omega(k)q_k \tag{9}$$

$$\tag{10}$$

Where the elements of $\Delta\theta$ are just the elements of $\hat{\omega}$ multiplied by the gyro sampling period.

Finally, the matrix C_b^n is formed from q:

$$C_b^n = \begin{bmatrix} (q_1^2 - q_2^2 - q_3^2 + q_4^2) & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & (q_2^2 - q_1^2 - q_3^2 + q_4^2) & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & (q_3^2 - q_1^2 - q_2^2 + q_4^2) \end{bmatrix} \tag{11}$$

This is the last equation. A quaternion can be used to multiply a vector directly, but it amounts to the same number of operations, and coordinate transformation matrices are more intuitive (or at least more familiar).

We will leave these equations for now, and return to them in the software design section. In the meantime, we must consider how to get $\hat{a}$ and $\hat{\omega}$.

3. Gyro Measurement Model

The gyros output a voltage nominally proportional to the rate of rotation around their sense axis. Thus the rotation rate may be written as:

$$\hat{\omega} = A(V_s - V_b)$$

With V_s being the output voltage, V_b being the sensor bias voltage, A being the sensor voltage gain, and ω being the actual rotation rate. In practice, there are variations in V_b and A that might need to be accounted for in order to have a reasonably accurate measurement of ω. Table 3 shows the effect of various parameters on bias and gain, based on the Analog Devices ADXRS150 datasheet.

Note that a 12-bit ADC can sense one part in 4096 voltage variations, which corresponds to 0.02%. It seems from the table that the most significant variations are those in temperature. To estimate the effect of V_{cc} variations we must consider the potential variation in voltage. It turns out that this is around 8 mV RMS when the system is running. Then the voltage variations become around 0.0048% total, which is trivial, and because the voltage variations are approximately zero mean (the voltage does not drift over time), their effect on the bias will average out to be even smaller.

Although the three gyros used for angular velocity sensing are meant to be mounted at right angles to each other, in reality they are not exactly orthogonal. This situation can be fixed by a matrix $\mathbf{A}_R$. The diagonal terms of the matrix can also be used to incorporate the base gain of the gyros (uncompensated for temperature). The off-diagonals will be used to carry out an orthogonalization of the measurement axes. The total gain matrix thus looks like:

$$(\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T) = \begin{bmatrix} A_{11} + C_{T2}(1)\Delta T & 0 & 0 \\ (\hat{x}, \hat{y})A_{11} & A_{22} + C_{T2}(2)\Delta T & 0 \\ (\hat{x}, \hat{z})A_{11} & (\hat{y}, \hat{z})A_{22} & A_{33} + C_{T2}(3)\Delta T \end{bmatrix}$$

(The parentheses $(\hat{x}, \hat{y})$ represent an inner product between the axes).

A quadratic term will be included to model the nonlinearity. If we figure that the quadratic term is 0.1% of the full gain, and the temperature is 10 % of the full gain, then the temperature will have an effect of around 0.01% on the full measurement through the quadratic term. This effect can thus be safely neglected, so we do not need separate temperature coefficients for the quadratic gain.

The above analysis leads to the following definitions:

$$\begin{aligned} \Delta T &= T - 27^o C \\ V_b &= V_{b0} + C_{T1}\Delta T + \mathbf{C}_a a \\ \omega &= (\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)((V_s - V_b) + \text{diag}[A_2](V_s - V_b)^2) \end{aligned}$$

The bottom two equations are vector equations, with three V_b and three ω. There is only one ΔT because the temperature is only read from one sensor. diag of a vector means to form a matrix and place the vector's elements on the diagonal.

Parameter	Effect on bias	Effect on gain
Nominal	2.5V	12.5 mV/^{o}/s
Temperature	12%	8%
V_{cc}	0.6%/V	0.7 %/V
Acceleration	0.12%	–
Nonlinearity	–	0.1 %

Table 3: Sources of error in ADXRS150 gyro (Datasheet Rev.B)

4. Accelerometer Measurement Model

The basic accelerometer model is essentially the same as the gyro model. Table 4 lists the parameters that affect the accelerometer: Evidently, the

Parameter	Effect on bias	Effect on gain
Nominal	2.5V	1 V/g
Temperature	6%	2%
Nonlinearity	–	0.1%
Cross-Axis	–	2%

Table 4: Sources of error in KXM52-1050 accelerometer (Datasheet Rev.1.4)

accelerometer is much less sensitive to temperature. Furthermore, it does not have a built-in temperature sensor, so it is impossible to know the temperature of the actual die, which is what matters. As a result, the effect will be modeled as a linear function of the temperature of one of the gyros. This is based on the idea that the main source of heating is the operation of the accelorometer itself. Since the accelerometers and gyros are turned on at the same time, their temperature profiles should be approximately the same, except perhaps for a scale factor. This assumes that the ambient temperature is consistent around the whole unit, which can be assured by enclosing the unit in a container with plenty of room for air to circulate inside.

Because the accelerometer has sensors for all three axes on a single die, there is no way to geometrically measure the non-orthogonality of these sensors. The "Cross-axis" line in the table is meant to illustrate the extent to which the sensors may be non-orthogonal. Effectively this says that the cosine of the angles between the accelerometers may be as high as 0.02, which corresponds to the angles varying as much as 1.15^o from perfectly orthogonal.

The measurement model is then defined as follows:

$$\begin{aligned} \Delta T &= T - 27^oC \\ V_b &= V_{b0} + C_{T1}\Delta T \\ a &= (\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)((V_s - V_b) + \text{diag}[A_2](V_s - V_b)^2) \end{aligned}$$

IV INS Calibration

In order to actually use the accelerometer and gyro models to calculate values for a and ω, we need to find the values of the various parameters. The more precisely these parameters are known, the more accurate the measurements, and the slower the increase in position and velocity error. In some cases, these parameters may vary unpredictably in time. If so, that will also become evident in the calibration.

The process would be easy given access to devices for, say, rotating the INS at a consistent rate, which would allow the calibration of individual gyros. Unfortunately, no such devices were available for this project. A motor with a PWM control and position feedback could be used to produce controlled motion, but this still leaves the problem of precisely mounting the INS in a specified way relative to the direction of motion. Lacking access to precision machining capability, we must resort to other methods.

Fortunately, there is one source of constant acceleration, and that is gravity. By orienting the INS in a variety of different ways, it is possible to give it a set of different acceleration inputs, all with a magnitude of 1g. If there were a way to precisely orient the INS, this would be the end of the story. Sadly, there is no way to do that either (if we can't orient the INS with more than 1 degree precision, we have no hope of measuring the cross-axis error due to 1 degree accelerometer misalignment). Nonlinear least squares optimization will be used to find both the parameters and the orientations.

1. Linear and Nonlinear Least Squares

Least squares fitting is a method for using a large number of measurements to calculate the most likely set of values that produced those measurements. Suppose there is a vector of values x and the measurements z are these values transformed by a matrix $\mathbf{A}$, so $z = \mathbf{A}x$. If $\mathbf{A}$ is square, then x is simply $\mathbf{A}^{-1}z$. If $\mathbf{A}$ has more rows than columns, then we have a typical least squares problem. The solution is through what is called the normal equation, $x = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T z$.

The measurement model equations, however, are nonlinear. They are of the form $z = f(x)$, where f is some nonlinear function. A common method for finding x in this case is called the Gauss-Newton method. Like many other methods, it essentially follows the gradient of the function to a local minimum. The following basic algorithm is followed:

1. Set x_g to an initial guess which is close enough to the solution that the algorithm will converge to it rather than some other undesired solution.
2. Find $F = \nabla_x f$ evaluated at x_g.
3. Set the error $e = z - f(x_g)$.
4. Solve the linear least squares problem $e = F\Delta x$ using the equation $x = (F^T F)^{-1} F^T$. Δx will be our step – the guess for x will be changed by this value in the next iteration. However, in some cases it is possible that Δx is too high, and the algorithm will miss the solution entirely. For this reason we have the next few steps.
5. Find the cost $J = \frac{1}{2} e^T e$.
6. Our error at the next iteration will be $e_{new} = z - f(x_g + \alpha\Delta x)$, because we will add Δx to x_g. α is initially set to 1.
7. Find the cost $J_{new} = \frac{1}{2} e_{new}^T e_{new}$.
8. While $J_{new} > J$, set α to $\alpha/2$. Then re-evaluate J_{new}.
9. Once an alpha that makes $J_{new} \leq J$ has been found, set x_g to $x_g + \alpha\Delta x$. Go back to step to 2.

This algorithm is repeated until Δx becomes very small, which suggests that a local equilibrium has been reached.

A related algorithm, called the Levenberg-Marquardt method, adjusts the normal equation in step 4 of the above algorithm by replacing $F^T F$ with $F^T F + \lambda I$. λ is initialized with $10^{-3} \max(F^T F)_{ii}$. Then it is successively increased and Δx re-evaluated until the parameter ρ, defined as $(\|e\|^2 - \|e_{new}\|^2)/(\Delta x^T(\lambda\Delta x + F^T e))$, becomes greater than 0. α is not used.

In essence, this method propagates the α of the Gauss-Newton method through the the measurement function f, and thus takes the nonlinearities into account while selecting the step size. This is the method that actually succeeded in finding the accelerometer parameters. Here is a step-by-step listing of the method (Based on [9]):

1. Set x_g to an initial guess which is close enough to the solution that the algorithm will converge to it rather than some other undesired solution. Set ν to 2.

2. Find $F = \nabla_x f$ evaluated at x_g.

3. Set the error $e = z - f(x_g)$.

4. If this is our first iteration, set $\lambda = 10^{-3} \max(F^T F)_{ii}$.

5. Solve the linear least squares problem $e = F\Delta x$ using the equation $x = (F^T F + \lambda I)^{-1} F^T$.

6. Find the cost $J = \frac{1}{2} e^T e$.

7. Our error at the next iteration will be $e_{new} = z - f(x_g + \Delta x)$, because we will add Δx to x_g.

8. Find the value of $\rho = (\|e\|^2 - \|e_{new}\|^2)/(\Delta x^T(\lambda \Delta x + F^T e))$.

9. While $\rho \leq 0$, set λ to $\nu * \lambda$. Set ν to 2ν (effectively achieving an exponential increase in λ). Go back to step 5.

10. Once a λ satisfying $\rho > 0$ has been found, set x_g to $x_g + \Delta x$. Set λ to $\lambda \max(\frac{1}{3}, 1 - (2\rho - 1)^3)$ and reset ν back to 2.

2. Finding the Accelerometer Parameters

The experimental setup is as follows. The INS is placed in an adjustable clamp. Some readings of the accelerometer outputs and the temperature are recorded. Then the clamp is rotated and more readings are recorded in this different orientation. Each of these orientiations gives a new set of equations:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \mathbf{R}(\theta, \phi) M(p_a, V_s, \Delta T) = h(\theta, \phi, p_a, V_s, \Delta T)$$

$$\mathbf{R}(\theta, \phi) = \begin{bmatrix} \cos(\theta) & \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) \\ 0 & \cos(\phi) & \sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

$\mathbf{R}$ is a rotation matrix, with the two angles θ and ϕ (pitch and roll). Since the gravitational field is symmetric around the z axis, there is no need to include yaw. p_a is the set of parameters which we wish to find. V_s and ΔT are the voltage and temperature measurements that are taken at each sample. p_a has 4 parameters for each sensor, plus 6 for $\mathbf{A}_R$, yielding 18 total parameters.

Parameter	Derivative
θ	$\begin{bmatrix} -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \\ 0 & 0 & 0 \\ -\cos(\theta) & -\sin(\theta)\sin(\phi) & -\sin(\theta)\cos(\phi) \end{bmatrix} M(p_a, V_s, \Delta T)$
ϕ	$\begin{bmatrix} 0 & \sin(\theta)\cos(\phi) & -\sin(\theta)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \\ -\sin(\theta) & \cos(\theta)\cos(\phi) & -\cos(\theta)\sin(\phi) \end{bmatrix} M(p_a, V_s, \Delta T)$
V_{b0}	$-\mathbf{R}(\theta,\phi)(\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)(1 + 2\text{diag}[A_2](V_s - V_b))$
C_{T1}	$\Delta T \partial h / \partial V_{b0}$
$\mathbf{A}_R$	$\mathbf{R}(\theta,\phi)((V_s - V_b) + \text{diag}[A_2](V_s - V_b)^2)$
C_{T2}	$\Delta T \partial h / \partial \mathbf{A}_R$
A_2	$-\mathbf{R}(\theta,\phi)(\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)(V_s - V_b)^2$

Table 5: Derivatives of the accelerometer equations with respect to their parameters

There is a separate θ and ϕ for each set of equations, but since there are three equations in each set, we still only need a total of 18 equations.

The next step is to find the derivative of $h(\theta, \phi, p_a, V_s, \Delta T)$ with respect to each of the parameters. They are shown in Table 5.

Now the algorithm can be written. See appendix A-2 for code. The question moves to, what should be the orientations used for the measurements? In principle, any set of distinct orientiations would do, but it is best to have them separated as far from each other as possible, to maximize observability. Considering that there are 18 measurements, the first 18 positions will be 9 evenly spaced rotations around the x axis of the INS and 9 evenly spaced rotations around the y axis. These are shown in Figure 2.

Roughly knowing these orientations is important in order to provide an initial guess to the algorithm. The initial guess that was chosen uses the nominal accelerometer parameters of 2.5V bias and 1 V/g gain, and the nominal orientations, first rotating around the y axis by 40^o increments, then doing the same for the x axis. For a photo of the setup, see Figure 3.

3. Finding the Gyro parameters

The static gyro parameters (i.e. the bias parameters) can be found using the same data used for the accelerometers. The gyro outputs can be described

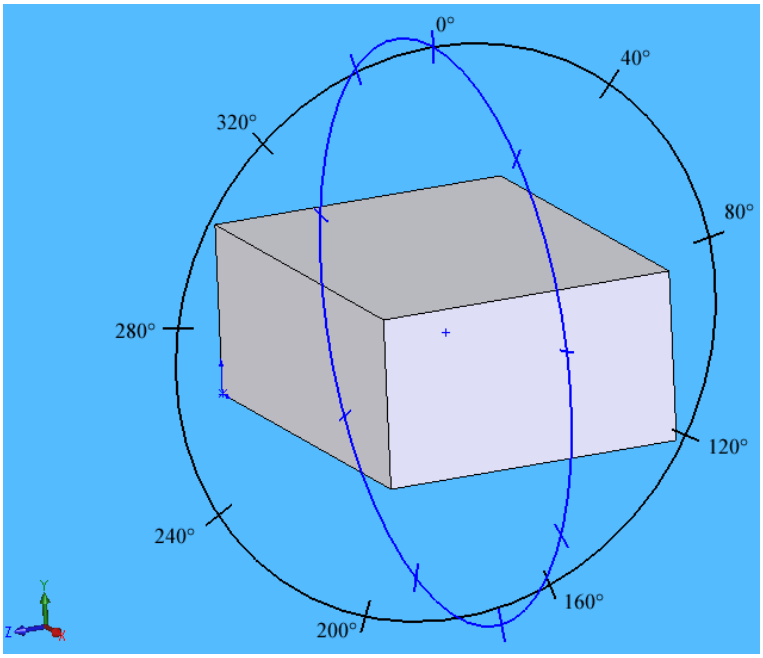


Figure 2: Angles at which to measure gravity

by the equation $V_s = V_{b0} + C_{T1}\Delta T + \mathbf{C}_a a$. Because this equation is linear, it can be solved directly via the normal equation.

Finding the dynamic gyro parameters is more challenging because while gravity provides a reference for acceleration, there is no reference for rotational velocity, absent a special device to do the job. However, because the orientations in the accelerometer calibration process above were optimized, they are known fairly precisely (about as precisely as the accelerometer parameters). Thus the change in orientation from position to position is well known. The mere integral of the gyro angular velocities, however, will not suffice for comparison. This is because the local coordinate system changes every time the INS rotates, so the integral is no longer the same as in the global system.

What will be compared instead are the quaternions of the rotations. The quaternion for the total change in orientation can be formed using equation 3. It should match the quaternion formed from the repeated application of equation 8 to the gyro data. The derivative of the quaternion is shown in equation 4 and will be used in forming the gradient H.

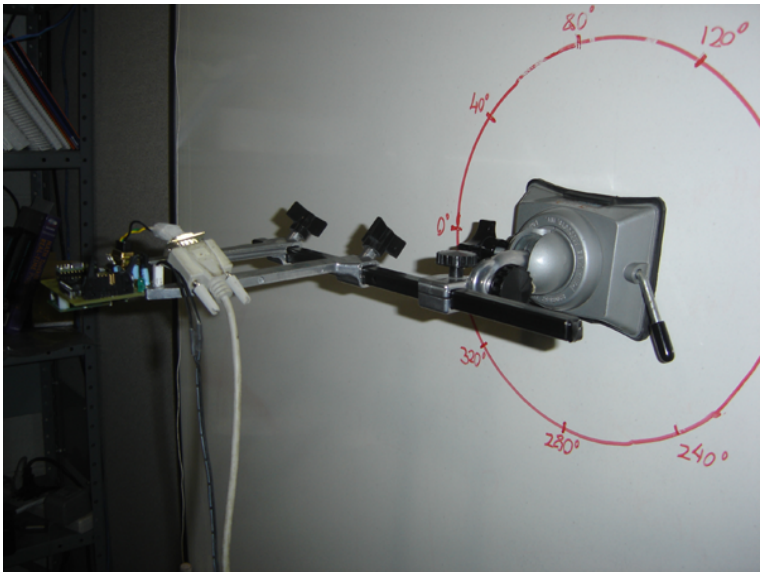


Figure 3: Accelerometer Calibration Setup

So, given the set of $\hat{\omega}(k)$ from k_0 to k_f,

$$\begin{aligned} q_{kf} &= (I + \frac{1}{2}\Omega(k_{f-1}))q_{kf-1} \\ &= (I + \frac{1}{2}\Omega(k_{f-1}))(I + \frac{1}{2}\Omega(k_f - 1))q_{kf-2} \\ &= \prod_{i=1}^{n}(I + \frac{1}{2}\Omega(k_{i-1}))q_{k0} \end{aligned}$$

where n is the number of steps from k_0 to k_f, and

$$\frac{\partial q_{kf}}{\partial p} = \frac{1}{2}\frac{\partial \Omega_c}{\partial p}q_{kf}$$

Where Ω_c is the continuous time version of $\Omega(k)$, as shown in eqn. 4. Table 6 shows the derivatives of the components of $\hat{\omega}$ with respect to the various elements of the parameters p. q_{k0} in this case is $[0\ 0\ 0\ 1]^T$.

4. Accelerometer Calibration Results

The Gauss-Newton method failed to produce reasonable values, but the Levenberg-Marquard method succeded. To verify the stability of the pa-

rameters, eight separate trials were conducted. The residual errors for the trials were all well below 1% indicating a close fit of the model to the parameters. The mean and standard deviation of the parameters are shown in Table 7. Clearly, some of the parameters are more statistically significant than others. The bias and the gain, the two most important parameters, are seen to have very low standard deviation. The other parameters show quite a bit more variation, but this is not unexpected since their magnitudes are at the very limits of the measuring ability of the INS.

Bar-Shalom ([4], p.156) provides a criterion for determining whether a parameter is statistically significant. To make a long story short, if for parameter x, $c = |\hat{x}|/\sigma_x$, then $1 - 2\mathcal{G}(c)$ is the probability that the parameter is significant, with $\mathcal{G}$ being the normal cdf with mean 0 and variance 1. For the accelerometer parameters, the gain and bias have very high significance (¿ 95%), while the other parameters have very low significance. This suggests that the basic linear model is adequate for the accelerometer, which is in one sense a good thing. On the other hand, it shows that the variation in parameters can't be modeled effectively beyond the two basic parameters, in other words that any parameter variation that is encountered is essentially random.

Since most of the parameters proved to be insignificant, we have redo the parameter fit to the reduced order model. Table 8 shows the final values.

5. Gyro Calibration Results

The bias calibration was performed for all eight of the datasets used in the accelerometer calibration. The residuals were around 0.04% for each dataset. Table 9 shows the values and standard deviations of the parameters. A sig-

Parameter	Derivative
V_{b0}	$-(\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)(1 + 2\text{diag}[A_2](V_s - V_b))$
C_{T1}	$\Delta T \partial h / \partial V_{b0}$
$\mathbf{A}_R$	$((V_s - V_b) + \text{diag}[A_2](V_s - V_b)^2)$
C_{T2}	$\Delta T \partial h / \partial \mathbf{A}_R$
A_2	$-(\mathbf{A}_R + \text{diag}[C_{T2}]\Delta T)(V_s - V_b)^2$
$\mathbf{C}_a(i,j)$	$a_j \partial h_i / \partial V_{b0}$

Table 6: Derivatives of the gyro equations with respect to their parameters

nificance analysis was performed just as for the accelerometer calibration. The biases themselves and most elements of the $\mathbf{C}_a$ matrix (except elements (1,2) and (2,1)) had greater than 95% significance. The only temperature parameter to have significance was $C_{T1}(3)$, which is the one whose temperature is being measured. This suggests that the gyro temperatures must vary a great deal from gyro to gyro, and each temperature must be measured separately in order to do temperature compensation properly. Clearly, the gyros are significantly less ideal devices than the accelerometers. This is not surprising given that the physical phenomenon that they measure is much more complex, but it does not bode well for our ability to accurately measure angular velocity.

The bias parameters, and the nominal values of gain (1.39 rad/s/V) are used to initialize p for the fitting to dynamic data. This turns out to be a rather time consuming procedure in MATLAB, with a single iteration of the LM algorithm taking around 40 seconds. Unfortunately, the algorithm was unsuccessful, with initial errors around 70% and coming down to around 30% when the algorithm converged.

One possible explanation for this lies in the fact that while we ignore

Parameter	x$\pm\sigma$	y$\pm\sigma$	z$\pm\sigma$	Unit
V_{b0}	2.4446±0.12663	2.5263±0.068762	2.6491±0.020407	Volts
C_{T1}	-45.514±85.582	-20.426±50.528	-30.180±13.532	mV/^{o}C
$\mathbf{A}_R(i,i)$	0.93686±0.014344	-0.91193±0.080782	1.0127±0.017384	g/Volt
$\mathbf{A}_R(2,1)$	–	10.609±20.867	–	mg/Volt
$\mathbf{A}_R(3,1)$	–	–	-3.4143±16.583	mg/Volt
$\mathbf{A}_R(3,2)$	–	–	37.849±0.11101	mg/Volt
C_{T2}	-9.5187±28.005	-18.822±64.064	7.0169±11.530	mg/(Volt ^{o}C)
A_2	6.004±8.0378	-1.1454±10.981	-2.5614±7.5659	1×10^{-3}/Volts

Table 7: Accelerometer parameters

Parameter	x$\pm\sigma$	y$\pm\sigma$	z$\pm\sigma$	Unit
V_{b0}	2.5073±0.0005	2.5171±0.0006	2.6509±0.0014	Volts
$\mathbf{A}_R(i,i)$	0.9289±0.0001	-0.9271±0.0001	1.0257±0.0001	g/Volt

Table 8: Accelerometer parameters, Reduced order model

the change in z-axis orientation (yaw) for the accelerometer calibration, because gravity is symmetric about the z axis, the total change in orientation as recorded by the gyros will nevertheless have some (possibly small) value for that axis. However, there is no way to infer this change based on the accelerometer readings, so it is assumed to be zero. It is doubtful that this explains the entire error, however. Perhaps there is a mistake in the algorithm, but it could not be located as of this report's writing.

An alternative method for gyro calibration is to attach the INS to a swinging pendulum. The maximum swing of the pendulum can be measured through the accelerometers, and then the theoretical angular velocity can straightforwardly be calculated. However, there was not enough time to implement this method. For the remainder of this project, the gyro gains will be assumed to equal 12.5 V/^{o}/s, the nominal value in the datasheet, which is 1.3963 rad/s/V.

V Software Design

1. Raw Output

We now move into the matter of how to program the INS with the mechanization equations to enable it to calculate position and velocity. First, the code that was used to produce raw output for the calibration setup will be presented, both for its own sake and as a stepping stone to the full INS code.

A pseudocode version of the raw output code runs as follows (for the full code, see Appendix A-3):

```
main() {
    Set up A/D converter and UART
    Program timer to interrupt after n cycles
```

Parameter	x$\pm\sigma$	y$\pm\sigma$	z$\pm\sigma$	Unit
V_{b0}	2.5453±0.001	2.4231±0.004	2.3841±0.004	Volts
C_{T1}	-1.1533±1.4341	-1.9233±2.3982	-3.2033±0.64503	mV/^{o}C
$\mathbf{C}_a(1,:)$	2.9885±0.38354	-1.4008±0.76318	-1.0364±0.30870	mV/g
$\mathbf{C}_a(2,:)$	0.39261±0.38013	-2.4646±0.73646	1.7959±0.58858	mV/g
$\mathbf{C}_a(3,:)$	-1.7667±0.48504	-1.2477±0.47567	1.4351±0.07497	mV/g

Table 9: Static gyro parameters

```
    Set sample mode to read all seven inputs in sleep mode
    while(1) {
        if(Temperature updated) {
            Send temperature over serial
            Clear temperature update flag
        }

        if(Gyro data updated) {
            Send gyro data over serial
            Clear gyro update flag
        }

        if(Accelerometer data updated) {
            Send accelerometer data over serial
            Clear accelerometer update flag
        }
    }
}

Timer interrupt {
    Reset timer
    Configure A/D converter based on desired sample mode
    Start conversion
    Go to sleep, if desired
}

Conversion complete interrupt {
    Read in data to appropriate variables
    Set update flags for variables that were updated
    Set sample mode for next conversion based on desired sampling frequencies
}
```

The time from entering the timer interrupt to exiting the conversion complete interrupt was determined to be 60 μs for the no-sleep-mode accelerometer only readings, and 1.16 ms for readings of all channels with sleep mode (all timing is determined by toggling a port pin in the code and monitoring it with an oscilloscope). If we sample at 6 kHz, which is the approximate Nyquist frequency of the accelerometers, we will use 36% of the processor time on that every second. Sampling the gyros at 80 Hz in sleep mode will take roughly 10% of processor time.

Sampling modes are defined as follows:

- **sampmode=0** Sample accelerometers only
- **sampmode=1** Sample accelerometers and gyros
- **sampmode=2** Sample accelerometers, gyros, and temperature
- **sampmode=4-6** Same as above, but with sleep mode enabled

T			G1			G2			G3		
Lo	Hi	0x01	Lo	Hi	0x02	Lo	Hi	0x03	Lo	Hi	0x04
A1			A2			A3			EOL		
Lo	Hi	0x05	Lo	Hi	0x06	Lo	Hi	0x07	\r	\n	

Table 10: Raw output format (T=Temperature, Gx=Gyro x, Ax=Accelerometer x)

The conversion complete interrupt routine selects the appropriate sample mode based on a counter that resets every 6000 cycles. Every 75 cycles (80 Hz), sample mode 5 (acc. and gyros in sleep mode) is used, and every 6000 cycles, sample mode 6 (everything in sleep mode) is used. The rest of the time, sample mode 0 is used. Sampling in sleep mode is time consuming, however it has been seen to decrease the noise by a factor of 4 or more. This is especially true for the gyro inputs, which are right next to the clock pins of the dsPIC. When the clock is shut down, the noise on these pins is decreased dramatically.

The timer interrupt frequency had to be determined experimentally, because in sleep mode, the main clock is shut down, and the timer stops incrementing. Interrupting every 4205 cycles proved to be the right rate for a 6 kHz sample frequency. This also tells us that we have 315,375 cycles per gyro data update to do all of our processing.

At a serial output rate of 57.6 kbps, there is not nearly enough time to output all of the accelerometer readings to the computer. Therefore, the readings are averaged together and output at the same rate as the gyro data. The serial data format is shown in Table 10.

If a field does not need to be sent because it has not been updated, it is simply omitted. This allows the data to be transmitted more efficiently and flexibly. If the gyro data and accelerometer data are sent 80 times a second, and temperature once a second, that translates to around 15kbps, which easily fits in the available bandwidth.

On the PC side, capturing the data is accomplished by a small program written in C#, which reads the data, translates it into comma separated text format, and saves it to a file. This allows for easy processing in MATLAB. See Appendix A-5 for the code.

2. The INS update procedure

Now it is time to implement the mechanization equations. They will be placed in the same main loop where the serial output code was in the pseudocode shown above. Here is the pseudocode for the mechanization equations:

```
// Accelerometer bias and gain
Vb_a={Vb_a_1,Vb_a_2,Vb_a_3};
A_a={A_a_1,A_a_2,A_a_3};

// Gyro bias parameters
Vb0_g={Vb0_1,Vb0_2,Vb0_3};
Ct1_g={Ct1_1,Ct1_2,Ct1_3};
Aa_g={Aa_11,Aa_12,Aa_13,Aa_21,Aa_22,Aa_23,Aa_31,Aa_32,Aa_33};
A_g; // Gyro scale factor
At; // Temperature scale factor
fs; // Sampling frequency (80 Hz)

if(Temp data updated) {
    DeltaT=(raw_temp - 2048)*At;
}

if(Gyro data updated and Acc data updated) {
    // Find ahat
    ahat[0]=(raw_acc[0] - Vb_a[0])*A_a[0];
    (repeat for ahat 1 and 2)

    // Find omegahat
    Vb_g[0]=Vb0_g[0]+Ct1_g[0]*DeltaT+Aa_g[0] . . .
    (repeat for Vb 1 and 2)
    omegahat[0]=(raw_gyro[0] - Vb_g[0])*A_g/fs;
    (repeat for omegahat 1 and 2)

    // Update q
    dtheta=sqrt(omegahat[0]^2 + omegahat[1]^2 + omegahat[2]^2);
    s=(2/dtheta)*sin(dtheta/2);
    c=2*(cos(dtheta/2)-1);
    q[0]=q[0]+0.5*(c*q[0] + s*omegahat[3]*q[1] + . . .
    (continues like Equation 8)

    // Find rotation matrix
    q2={q[0]^2,q[1]^2,q[2]^2,q[3]^2};
    C[0,0]=q2[0]-q2[1]-q2[3]+q2[4];
    (continues like Equation 11)

    // Find true acceleration
    a[0]=C[0,0]*ahat[0]+C[0,1]*ahat[1]+C[0,2]*ahat[2];
    a[1]=C[1,0]*ahat[0]+C[1,1]*ahat[1]+C[1,2]*ahat[2];
    a[2]=C[2,0]*ahat[0]+C[2,1]*ahat[1]+C[2,2]*ahat[2] - 1;

    vn[0]=vn[0]+a[0]/fs;
    (repeat for vn 1 and 2)

    rn[0]=rn[0]+vn[0]/fs;
    (repeat for rn 1 and 2)
}
```

The full code is included in Appendix A-4. A first pass at this code in full (software) floating point yields 12,579 instruction cycles in the simulator. When run on the actual dsPIC, it completed in 1.6ms, around a tenth of the time between gyro updates. Note that in an INS on a military jet, the gyro would be updated at a much higher frequency, 2 kHz or so. For a gimballed system, 30 Hz would be sufficient [8]. So the amount of processing power necessary is very problem-specific, but for these particular gyros, a dsPIC running at 26 MHz is more than enough.

One detail that complicates matters is figuring out the initial orientation. Aviation grade INS generally accomplish this through gyrocompassing, where the INS is held stationary for a period of time and the gyros are used to sense which way the earth is rotating. This allows accurate computation of the orientation of the INS in all axes. It is also quite practical since aircraft typically spend some time parked on the tarmac before taking off, so there is time to perform the alignment.

In the case of this INS, the earth's rotation can't be sensed. Initial roll and pitch alignment must therefore be performed using the accelerometers as tilt sensors. The heading can't be determined without an additional sensor such as a magnetometer or a GPS unit. However, the heading is not important for subtracting the gravity vector anyway, so it can be initialized at zero for now.

If the INS is stationary and it is only sensing gravity, $\|\hat{a}\|$ will be 1. The roll can be calculated as $\theta_x = \sin^{-1}(\hat{a}_y)$ and the pitch as $\theta_y = \sin^{-1}(\hat{a}_x)$. A small complication arises from the fact that if the INS is upside down, the direction of gravity is reversed. The simplest way of dealing with this problem is to require that the INS be right side up when turned on. Alternatively, we can take the direction of the largest magnitude accelerometer signal (if the INS is close to level, it will be $\hat{a}_z$), and if that is negative, negate θ_x and θ_y and add 180^o to them. Here it is in pseduocode:

```
int maxa;
if(abs(ahat[0])>abs(ahat[1]))
    maxa=ahat[0];
else
    maxa=ahat[1];
if(abs(ahat[2])>abs(maxa))
    maxa=ahat[2];
if(maxa > 0) {
    thetax = asin(ahat[1]);
    thetay = asin(ahat[0]);
} else {
    thetax = 180-asin(ahat[1]);
    thetay = -asin(ahat[0]);
}
thetaz=0;
```

Then equation 3 is then applied to generate the initial value for the quaternion vector. It also helps to average over multiple accelerometer samples in order to obtain a better estimate of the attitude.

3. INS Code results

As a start, the INS was programmed to only output $\hat{a}$ and $\hat{\omega}$. The serial output was recorded over a period of twenty minutes while the INS was held stationary. Table 11 shows some representative statistics for this dataset. The first thing to note is that the accelerometers overall show an output very close to 1 g over the 20 minute period, and that their null (i.e. the zero offset) drifts very little over time. The gyros, on the other hand, show an average angular velocity of almost 0.5 ^{o}/s, and drift almost 2 degrees per second in an hour. This is what was expected. Figure 4 shows a plot of the three gyro signals averaged together and smoothed with an 80-step sliding window average.

The noise level closely matches the manufacturer specifications. The accelerometer noise was specified at 50 $\mu g/\sqrt{\text{Hz}}$, and the gyro noise was specified at 0.05 $^o/\text{s}/\sqrt{\text{Hz}}$. This shows that the sensors are being used correctly and that the analog to digital conversion process does not adversely affect the signals.

	$\hat{a}$	$\hat{\omega}$
Norm of the mean	0.9947 g	0.4662 ^{o}/s
Norm of the std. dev.	2.3959 mg	0.4475 ^{o}/s
std.dev. / $\sqrt{\text{Hz}}$	43.74 $\mu g/\sqrt{\text{Hz}}$	0.07075 $^o/\text{s}/\sqrt{\text{Hz}}$
Null drift	-2.8607 mg/hr	1.6347 ^{o}/s/hr

Table 11: Statistics from a 20-minute run

Next, the measured acceleration (the right hand side of equation 2) was recorded over about a minute while the INS was stationary. Figure 5 shows the norm of the acceleration and each acceleration component over time. What these plots really tell us is how fast the attitude estimates diverge, causing the gravity vector to be subtracted in the wrong direction. We see that, at least in the stationary case, we diverge almost linearly at around 10 mg/sec. The linearity is due to the fact that the divergence is small, so the sine of the error is approximately linear.

Since velocity is the integral of the acceleration, it can be expected to diverge quadratically at a rate of $0.5 \cdot 0.01 \cdot 9.8 = 0.049 m/s^2$. Indeed, as

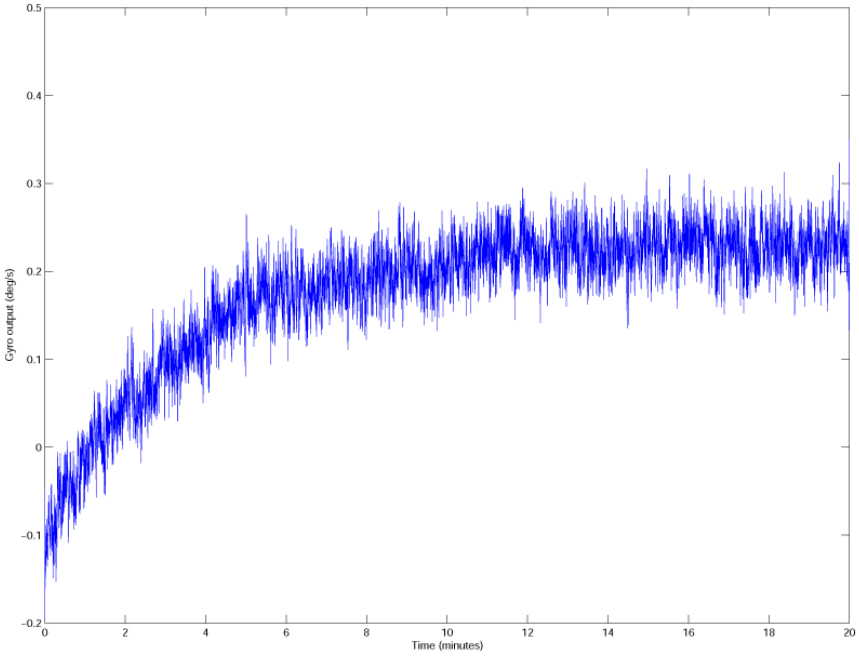


Figure 4: A plot of the gyro drift over 20 minutes

Figure 6 shows, this is approximately what happens. Lastly, Figure 7 shows the divergence of the position, which should diverge with the cube of time at the rate of $0.0163 m/s^3$ (in practice it's not quite a cubic function). Therefore, the divergence is quite a bit faster than the INS that experienced 5km in 5 minutes that was mentioned in the introduction. On the other hand, that setup used a commercial integrated sensor unit which may have had marginally better performance. However, divergence after one second is only around 0.17 m, and velocity error is around 0.3 m/s, which is much better than the typical error of a GPS unit, for instance. So we see that, if the INS were integrated with another sensor, it could potentially be useful.

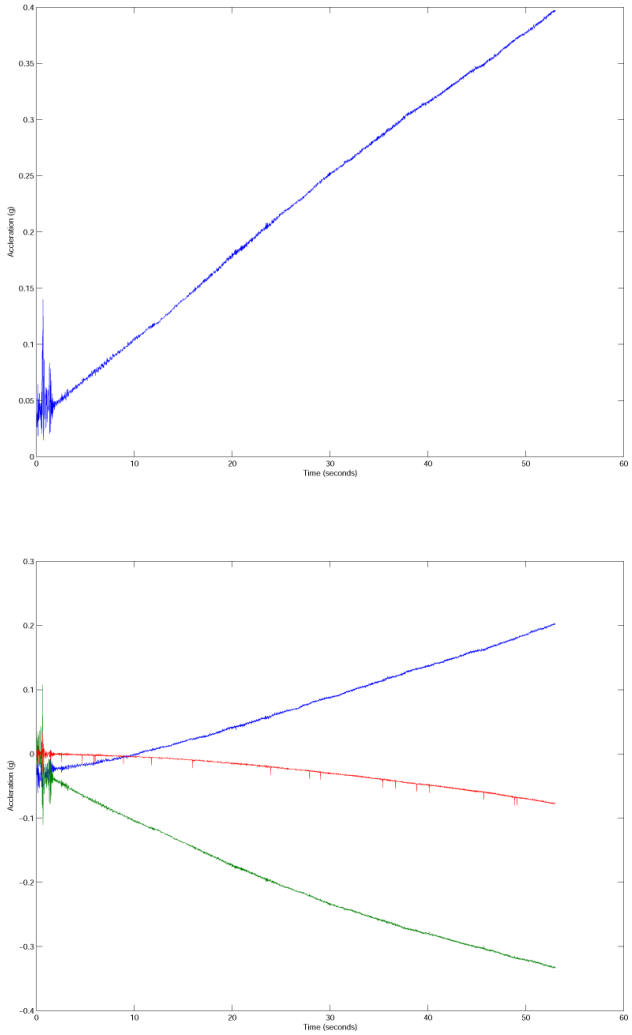


Figure 5: Plots of the acceleration (top) and each component (bottom) for a stationary INS

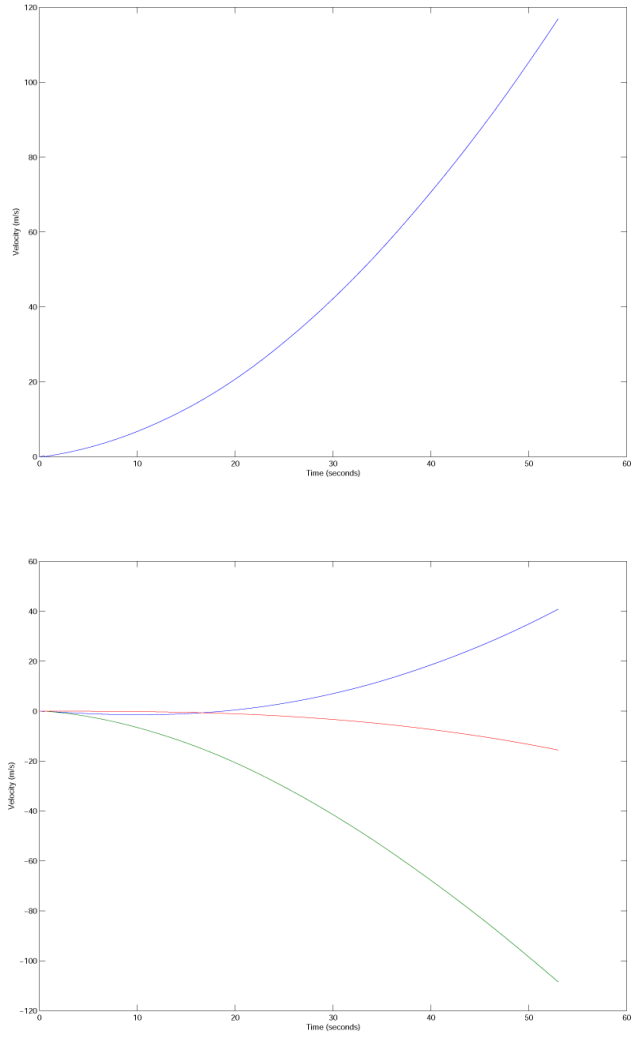


Figure 6: Plots of the velocity (top) and each component (bottom) for a stationary INS

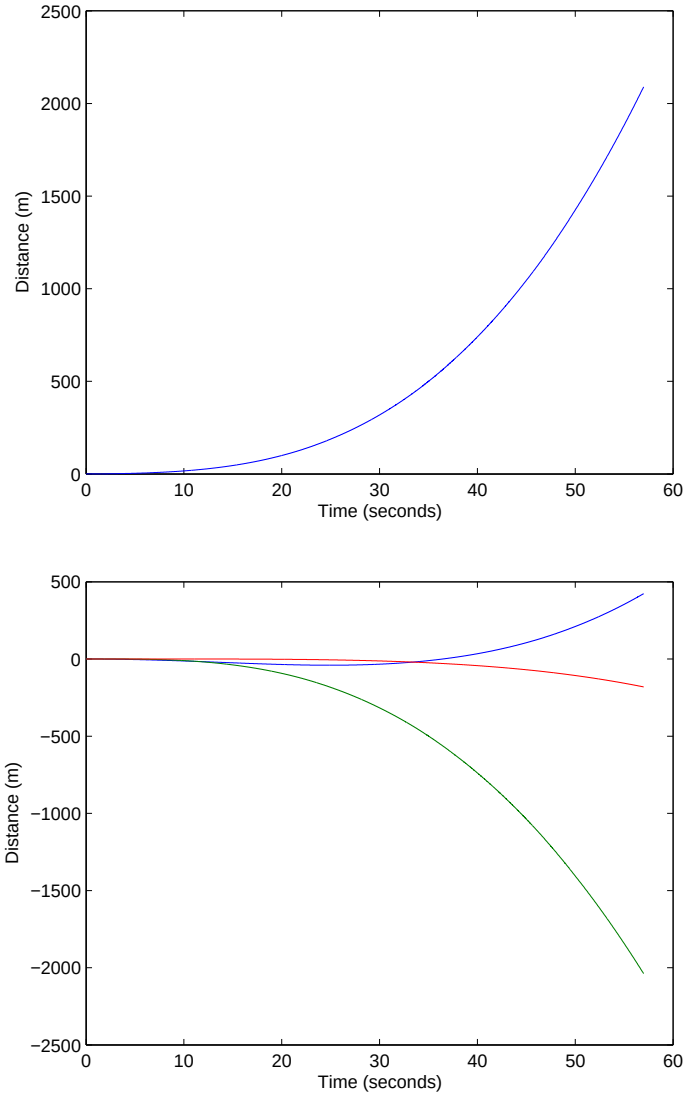


Figure 7: Plots of the distance (top) and each component (bottom) for a stationary INS

VI Conclusion

A design has been presented which can be used to measure acceleration, velocity, and position over time scales of several seconds. An overall position error of 0.17 meters and velocity error of 0.3 m/s can be expected after one second of operation.

Some aspects of the calibration process to identify parameters for the sensors have been described as an exploration of ways to improve the system's accuracy. The calibration showed that neither temperature nor higher order terms nor cross axis terms are useful in modeling the behavior of this particular accelerometer model, given the level of accuracy obtained from the digital to analog converter and the rest of the system. There have been claims [3] that a quadratic model can improve readings, but for this sensor the quadratic term was found to be insignificant.

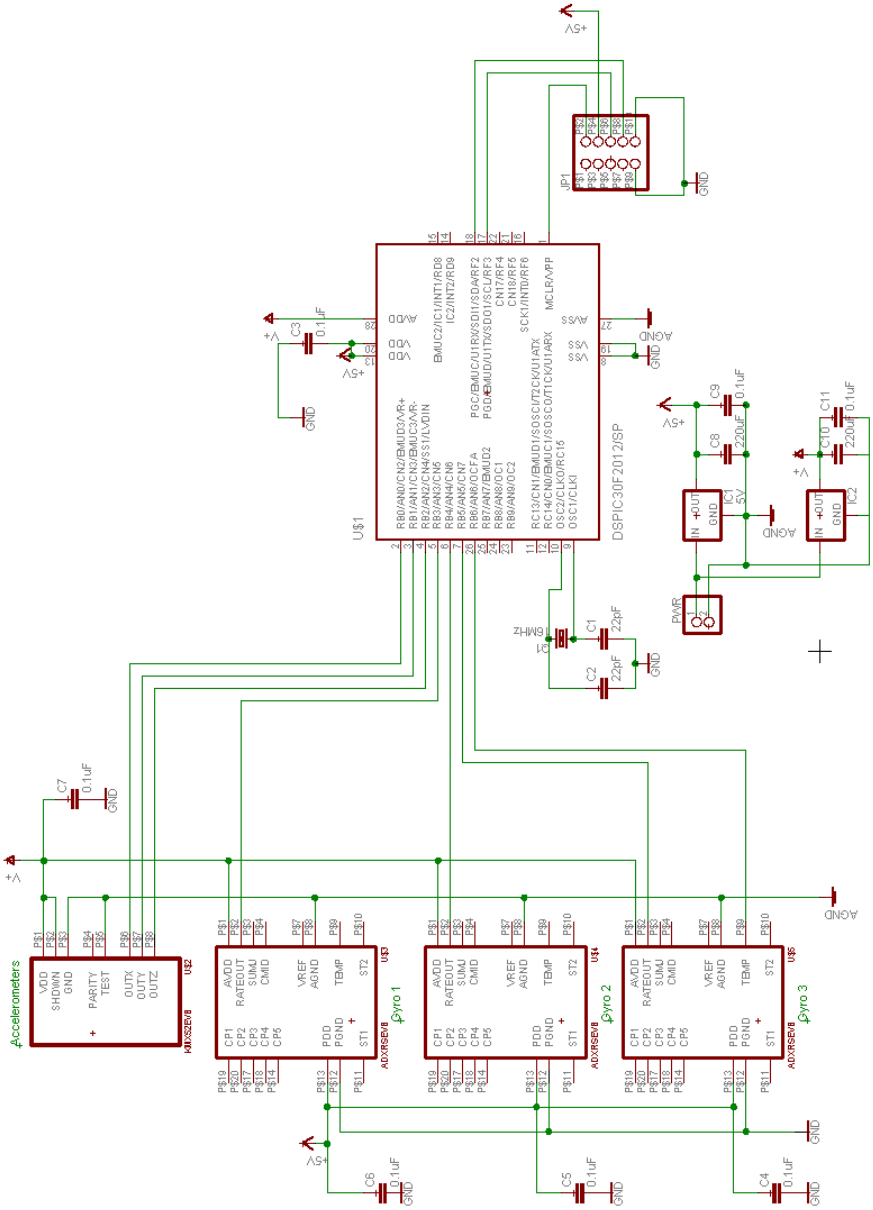
The gyro bias was found to depend substantially on temperature and acceleration, with both effects incorporated into the software. The gyro gain could not be accurately estimated using the method presented, although there may be other methods, such as using a pendulum to produce a controllerd angular velocity, that could be used to calibrate the gyro gain in the absence of precision equipment.

One of the biggest limitations of the inertial navigation system is that it is unable to distinguish between the force used to accelerate it and the force of gravity. Therefore, the force of gravity must be continually subtracted from the accelerometer readings. In order to know the direction of the gravitational force, rotations of the inertial navigation system must be sensed with the highest accuracy possible.

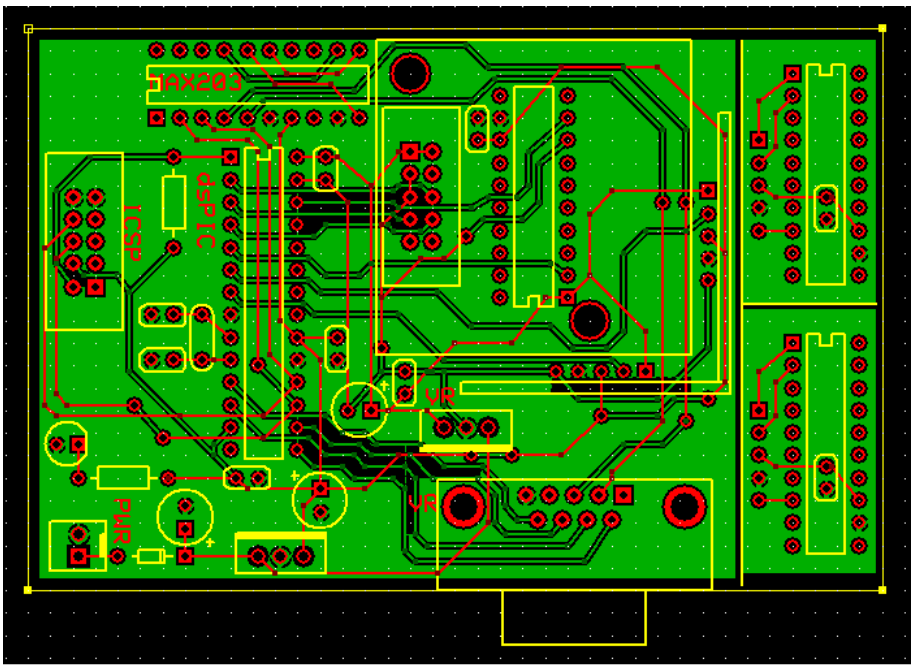
Inevitably, all inertial navigation systems experience divergence of their position and velocity measurements as the errors from their sensors accumulate. Thus, an inertial navigation system for a particular application must be designed on the basis of expected time ranges over which the system is expected to give accurate readings.

It might seem that a system that can only be used for a few seconds can't possibly be useful. However this is not so, as long as it can be reset at regular intervals by another, slower sensor such as a GPS system. Integrated GPS/INS systems are a popular way of enhancing accuracy. Inertial navigation systems such as the one presented have the potential to bring improved navigation to applications where it has formerly been too expensive to implement.

A-1 Circuit Board Schematic and Layout



INS Board schematic



INS Board PCB layout

A-2 MATLAB Code for INS calibration

1. ins_calib routine

This routine is the entry point to the calibration procedure. The first two lines specify the location of the data files used by the routine.

```
function [pa,pg,aerr,gberr]=ins_calib(do_acc,ind)
    disp('Loading data');
    data1=load(['C:\Documents and Settings\Administrator.CUDGCKAYAK\Desktop\data\y9turn' num2str(ind) '.txt']);
    data2=load(['C:\Documents and Settings\Administrator.CUDGCKAYAK\Desktop\data\x9turn' num2str(ind) '.txt']);

    % Fill in blank areas in data with surrounding values
    disp('Splitting into stationary regions')
    gbreak=ones(100,1)*[0 0 0 0 1700 1700 1700 0];
    odata=[gbreak;data1;gbreak;data2];
    data=odata;
    n=length(data);
    clear data1 data2;
    for i=1:n
        for j=1:8
            if(data(i,j)==-1)
                if(i<2)
                    data(i,j)=0;
                else
                    data(i,j)=data(i-1,j);
                end
            end
        end
    end

    % Break data up into regions where INS was held still
```

```
    [regions, ranges]=findregiondata(data,odata,1);

    n=size(regions,1);

    acc_Vs=regions(:,2:4)';
    gyro_b_Vs=fliplr(regions(:,5:7)');
    acc_temp=regions(:,8);

    bits2volts=5/4096;
    acc_Vs=acc_Vs*bits2volts;
    gyro_b_Vs=gyro_b_Vs*bits2volts;

    % Run accelerometer calibration routine
    if(do_acc==1)
        disp('Finding accelerometer parameters');
        [pa,aerr]=acc_short_calib(acc_Vs,findT(acc_temp));
    else
        load pa;
    end
    pa=[pa(1:3);zeros(3,1);pa(4:6);zeros(9,1);pa(7:end)];
    disp(['Final acc. err:' num2str(100*aerr(end)) '%']);

    % Run gyro calibration routine
    disp('Performing gyro bias calibration');
    [pg,gberr]=gyro_bias_calib(acc_Vs,pa,gyro_b_Vs,findT(acc_temp));
    disp(['Final acc. err:' num2str(100*gberr(end)) '%']);
    return;
    pg(7:9)=[1.39; -1.39; 1.39];

    lmax=max([0; ranges(1:end-1,1)]-ranges(:,2));

    % Prepare data for gyro dynamic calibration
    disp('Performing gyro dynamic parameter calibration');
    O=fliplr(reshape(pa(19:end),2,18)');
    O=[O zeros(length(O),1)];
    DeltaO=[];
    gyro_Vs=zeros(3,lmax,16);
    gyro_temp=zeros(lmax,16);
    gyro_a=zeros(3,lmax,16);
    gyro_lengths=zeros(lmax,1);
    samps=1;
    for k=[1 10;
           8 17];
        for i=k(1):k(2)
            start=ranges(i,2);
            stop=ranges(i+1,1);

            r1=odata(start:stop,5:7);
            nn_ind=find(r1(:,1)~=-1);
            r1=r1(nn_ind,:);
            r1=fliplr(r1)'*bits2volts;
            gyro_Vs(:,1:size(r1,2),samps)=r1;

            r1=findT(data(start+nn_ind-1,8));
            gyro_temp(1:length(r1),samps)=r1;
            for j=1:length(nn_ind)
               Vs=data(start+nn_ind(j)-1,1:3)'*bits2volts;
               gyro_a(:,j,samps)=findM(pa,Vs,gyro_temp(j,samps),0);
            end
            gyro_lengths(samps)=length(nn_ind);
            DeltaO=[DeltaO; (O(i+1,:)-O(i,:))'];
            samps=samps+1;
        end
    end

    % Run dynamic calibration routine
    [pg, err]=gyro_calib(pg,gyro_Vs,gyro_temp,gyro_a,DeltaO,gyro_lengths);


%This function splits the data up into regions where the
%INS is not moving, based on the gyro data
function [aregions,ranges]=findregiondata(data,odata,nsplit)
    n=length(data);
    raw_a=data(:,2:4);
    raw_r=fliplr(data(:,5:7));
    raw_temp=data(:,8);
```

```
    nr=sum(raw_r.^2,2);
    nr=abs(nr-mean(nr));
    cutoff=1.885e5;
    ranges=[];
    while(length(ranges) ~= 18)
        ischng=smooth(nr,301)>cutoff;
        ranges=[];
        on=0;
        for i=2:n
            if(ischng(i) < ischng(i-1) & on==0 & n-i > 200)
                on=1;
                ranges=[ranges; [i 0]];
            end
            if(ischng(i) > ischng(i-1) & on==1)
                on=0;
                ranges(end,2)=i;
            end
        end
        if(on==1)
            ranges(end,2)=n;
        end
        cutoff=cutoff+100;
     end
%     cutoff
%      plot(1:n,smooth(nr,301),1:n,ischng*3e6)
    aregions=[];
    for i=1:size(ranges,1)
        ln1=floor((ranges(i,2)-ranges(i,1))/nsplit);
        for j=1:nsplit
            start=ranges(i,1)+ln1*(j-1);
            stop=ranges(i,1)+ln1*j;
            nn_ind=find(odata(start:stop,2)~=-1);
            aregions=[aregions; mean(data(start+nn_ind-1,:),1)];
        end
    end
```

2. acc_calib

This is where the parameter fitting for the accelerometers takes place. A second script, acc_short_calib, does the same for the reduce order model (only the biases and the gains).

```
function [p,err]=acc_calib(Vs,DeltaT)
    n=length(DeltaT);
    p=[2.5;
       2.5;
       2.5;
       0;
       0;
       0;
       1;
       -1;
       1;
       0;
       0;
       0;
       0;
       0;
       0;
       0;
       0;
       0;
       reshape(([40; 0]*(0:8)*pi/180),n,1);
       reshape(([0;-40]*(0:8)*pi/180),n,1)];


    err=[];

    iters=0;
    finished=0;
```

```
    % z is just the gravity vector
    % pointing down
    z=[0;0;1]*ones(1,n);
    z=reshape(z,n*3,1);

    h=waitbar(0,'Calibrating Accelerometers');
    maxiters=200;
    L=-1;
    while(~finished)
        waitbar(iters/maxiters,h);
        iters=iters+1;
        % This is where all the action happens
        [deltap, L]=findlmJ(p,Vs,DeltaT,L);
        p=p+deltap;
        err=[err norm(z-findh(p,Vs,DeltaT))];
        if(norm(deltap) < 1e-15 | iters==maxiters)
            finished=1;
        end
    end
    close(h);

% Gauss-Newton least squares estimation
% (Does not converge)
function [deltap,alpha]=findJ(p,Vs,DeltaT)
    H=findgradh(p,Vs,DeltaT);
    n=length(DeltaT);
    z=[0;0;1]*ones(1,n);
    z=reshape(z,n*3,1);

    [Qv,Rv]=qr(H'*H);
    err=(z-findh(p,Vs,DeltaT));
    mnew=Qv'*H'*err;
    deltap=Rv\mnew;

    alpha=1;
    J=err'*err;
    errnew=(z-findh(p+alpha*deltap,Vs,DeltaT));
    Jnew=errnew'*errnew;

    while(Jnew >= J)
        alpha=alpha/2;
        errnew=(z-findh(p+alpha*deltap,Vs,DeltaT));
        Jnew=errnew'*errnew;
        if(alpha < 1e-15)
            break;
        end
    end

% Levenberg-Marquard estimation - use this one
function [deltap,Lnext]=findlmJ(p,Vs,DeltaT,L)
    tau=1e-3;
    epserr=1e-15;
    k=100;
    nu=2;
    alpha=1;

    rho=0;
    iter=0;
    while(rho <= 0)
        if(iter > 0)
            L=L*nu;
            nu=2*nu;
        end

        H=findgradh(p,Vs,DeltaT);
        n=length(DeltaT);
        z=[0;0;1]*ones(1,n);
        z=reshape(z,n*3,1);
        A=H'*H;

        if(L==-1)
            L=tau*max(diag(A));
        end

        err=(z-findh(p,Vs,DeltaT));
```

```
        [Qb,Rb]=qr(A+L*eye(size(A)));
        deltap=Rb\Qb'*H'*err;
        errnew=(z-findh(p+deltap,Vs,DeltaT));
        rho=(norm(err)^2-norm(errnew)^2)/(deltap'*(L*deltap + H'*err));
        iter=iter+1;
    end
    Lnext=L*max(1/3,1-(2*rho-1)^3);

% Finds the gradient of h by numerical differentiation
% useful for checking accuracy of analytical function
function H=findnumgradh(p,Vs,DeltaT,delta)
    np=length(p);
    H=[];
    for i=1:np
       pplus=p;
       pplus(i)=pplus(i)+delta;
       pminus=p;
       pminus(i)=pminus(i)-delta;
       hplus=findh(pplus,Vs,DeltaT);
       hminus=findh(pminus,Vs,DeltaT);
       Hcol=(hplus-hminus)/(2*delta);
       H=[H Hcol];
    end

% Finds the gradient of h analytically
function H=findgradh(p,Vs,DeltaT)
    H=[];
    n=length(DeltaT);
    for i=1:n
        [R,DRmat_th,DRmat_ph]=findR(p(17+i*2),p(18+i*2),1);
        [M,gradM]=findM(p,Vs(:,i),DeltaT(i),1);

        newRow=[R*gradM zeros(3,2*(i-1))];
        newRow=[newRow DRmat_th*M DRmat_ph*M];
        newRow=[newRow zeros(3,2*(n-i))];
        H=[H; newRow];
    end

% Finds h (the measurement function)
function h=findh(p,Vs,DeltaT)
    h=[];
    for i=1:length(DeltaT)
        h=[h; findR(p(17+i*2),p(18+i*2),0)*findM(p,Vs(:,i),DeltaT(i),0)];
    end
```

3. acc_short_calib

```
function [p,err]=acc_short_calib(Vs,DeltaT)
    n=length(DeltaT);
    p=[2.5;
       2.5;
       2.5;
       1;
       -1;
       1;
       reshape(([40; 0]*(0:8)*pi/180),n,1);
       reshape(([0;-40]*(0:8)*pi/180),n,1)];


    err=[];

    iters=0;
    finished=0;

    z=[0;0;1]*ones(1,n);
    z=reshape(z,n*3,1);

    h=waitbar(0,'Calibrating Accelerometers');
    maxiters=200;
    L=-1;
    while(~finished)
        waitbar(iters/maxiters,h);
        iters=iters+1;
        [deltap, L]=findlmJ(p,Vs,DeltaT,L);
```

```
        p=p+deltap;
        err=[err norm(z-findh(p,Vs,DeltaT))];

        if(norm(deltap) < 1.5e-9 | iters==maxiters)
            finished=1;
        end
    end
    close(h);


function [deltap,alpha]=findJ(p,Vs,DeltaT)
    H=findgradh(p,Vs,DeltaT);
    n=length(DeltaT);
    z=[0;0;1]*ones(1,n);
    z=reshape(z,n*3,1);

    [Qv,Rv]=qr(H'*H);
    err=(z-findh(p,Vs,DeltaT));
    mnew=Qv'*H'*err;
    deltap=Rv\mnew;

    alpha=1;
    J=err'*err;
    errnew=(z-findh(p+alpha*deltap,Vs,DeltaT));
    Jnew=errnew'*errnew;

    while(Jnew >= J)
        alpha=alpha/2;
        errnew=(z-findh(p+alpha*deltap,Vs,DeltaT));
        Jnew=errnew'*errnew;
        if(alpha < 1e-15)
            break;
        end
    end

function [deltap,Lnext]=findlmJ(p,Vs,DeltaT,L)
    tau=1e-3;
    epserr=1e-15;
    k=100;
    nu=2;
    alpha=1;

    rho=0;
    iter=0;
    while(rho <= 0)
        if(iter > 0)
            L=L*nu;
            nu=2*nu;
        end

        H=findgradh(p,Vs,DeltaT);
        n=length(DeltaT);
        z=[0;0;1]*ones(1,n);
        z=reshape(z,n*3,1);
        A=H'*H;

        if(L==-1)
            L=tau*max(diag(A));
        end

        err=(z-findh(p,Vs,DeltaT));

        [Qb,Rb]=qr(A+L*eye(size(A)));
        deltap=Rb\Qb'*H'*err;
        errnew=(z-findh(p+deltap,Vs,DeltaT));
        rho=(norm(err)^2-norm(errnew)^2)/(deltap'*(L*deltap + H'*err));
        iter=iter+1;
    end
    Lnext=L*max(1/3,1-(2*rho-1)^3);

function H=findnumgradh(p,Vs,DeltaT,delta)
    np=length(p);
    H=[];
    for i=1:np
       pplus=p;
       pplus(i)=pplus(i)+delta;
       pminus=p;
```

```
        pminus(i)=pminus(i)-delta;
        hplus=findh(pplus,Vs,DeltaT);
        hminus=findh(pminus,Vs,DeltaT);
        Hcol=(hplus-hminus)/(2*delta);
        H=[H Hcol];
    end

function H=findgradh(p,Vs,DeltaT)
    H=[];
    n=length(DeltaT);
    for i=1:n
        [R,DRmat_th,DRmat_ph]=findR(p(5+i*2),p(6+i*2),1);
        [M,gradM]=findM([p(1:3);zeros(3,1);p(4:6);zeros(9,1)],Vs(:,i),DeltaT(i),1);

        newRow=[R*gradM(:,[1 2 3 7 8 9]) zeros(3,2*(i-1))];
        newRow=[newRow DRmat_th*M DRmat_ph*M];
        newRow=[newRow zeros(3,2*(n-i))];
        H=[H; newRow];
    end


function h=findh(p,Vs,DeltaT)
    h=[];
    for i=1:length(DeltaT)
        h=[h; findR(p(5+i*2),p(6+i*2),0)*findM([p(1:3);zeros(3,1);p(4:6);zeros(9,1)],Vs(:,i),DeltaT(i),0)];
    end
```

4. gyro_bias_calib.m

This is a simple linear least squares fit for the gyro bias.

```
function [pg,err]=gyro_bias_calib(acc_Vs,pa,gyro_b_Vs,DeltaT);
    n=length(DeltaT);
    a=[];
    z=[];
    for i=1:n
        a=[a findM(pa,acc_Vs(:,i),DeltaT(i),0)];
        z=[z; gyro_b_Vs(:,i)];
    end

    H=[];
    for i=1:n
        newRow=[eye(3)...
                DeltaT(i)*eye(3)...
                [a(:,i)'; zeros(2,3)]...
                [zeros(1,3); a(:,i)'; zeros(1,3)]...
                [zeros(2,3); a(:,i)']];
        H=[H; newRow];
    end

    [Q,R]=qr(H'*H);
    pg_=R\Q'*H'*z;
    pg=zeros(27,1);
    pg(1:6)=pg_(1:6);
    pg(19:end)=pg_(7:end);
    err=norm(z-H*pg_)/norm(z);
```

5. gyro_calib

This is where the fitting is done for the gyro gain. This routine does not provide acceptable results as is, and is very slow.

```
function [p,err]=gyro_calib(p,Vs,DeltaT,a,DeltaO,lengths)
    n=length(lengths);

    err=[];

    iters=0;
```

```
    finished=0;

    z=[];
    for i=1:n
        k=eul2q(DeltaO(3*i-2:3*i));
        z=[z; k];
    end

    h=waitbar(0,'Calibrating Gyros');
    maxiters=200;
    L=-1;
    while(~finished)
        waitbar(iters/maxiters,h);
        iters=iters+1;
        [deltap, L,nerr]=findlmJ(p,Vs,DeltaT,a,z,lengths,L);
        p=p+deltap;
        err=[err nerr];
        disp(['Error:' num2str(100*norm(nerr)/norm(DeltaO)) '%']);
        plot(reshape(nerr*180/pi,3,16)')
        if(norm(deltap) < 1e-15 | iters==maxiters)
            finished=1;
        end
    end
    close(h);

function [deltap,Lnext,errout]=findlmJ(p,Vs,DeltaT,a,z,lengths,L)
    tau=1e-3;
    epserr=1e-15;
    k=100;
    nu=2;
    alpha=1;

    rho=0;
    iter=0;
    while(rho <= 0 && L < 1e18)
        if(iter > 0)
            L=L*nu;
            nu=2*nu;
        end
        [h,H]=findh(p,Vs,DeltaT,a,lengths,0);
         H=numgradh(p,Vs,DeltaT,a,lengths);
        A=H'*H;

        if(L==-1)
            L=tau*max(diag(A));
        end

        err=(z-h);

        [Qb,Rb]=qr(A+L*eye(size(A)));
        deltapshort=Rb\Qb'*H'*err;
        deltap=[zeros(6,1); deltapshort; zeros(9,1)];
        errnew=(z-findh(p+deltap,Vs,DeltaT,a,lengths,0));
        rho=(norm(err)^2-norm(errnew)^2)/(deltapshort'*(L*deltapshort + H'*err));
        iter=iter+1;
    end
    Lnext=L*max(1/3,1-(2*rho-1)^3);
    errout=[];
    for i=1:16
        errout=[errout; (q2eul(z(4*i-3:4*i))-q2eul(h(4*i-3:4*i)))];
    end

function H=numgradh(p,Vs,DeltaT,a,lengths)
   H=[];
   for j=7:18
       j
      delta=1e-6;
      pplus=p;
      pminus=p;
      pplus(j)=p(j)+delta;
      pminus(j)=p(j)-delta;
      hplus=findh(pplus,Vs,DeltaT,a,lengths,0);
      hminus=findh(pminus,Vs,DeltaT,a,lengths,0);
      H=[H (hplus-hminus)/(2*delta)];
   end
   %[h,H2]=findh(p,Vs,DeltaT,a,lengths,1);
   %mesh(1:64,1:12,H2-H);
```

```
function [h,H]=findh(p,Vs,DeltaT,a,lengths,dervflag)
    h=[];
    H=[];
    for i=1:length(lengths)
        q=[0;0;0;1];
        for j=1:lengths(i)
            if(j==lengths(i))
                [N,gradN]=findN(p,Vs(:,j,i),DeltaT(j,i),a(:,j,i),dervflag);
            else
                N=findN(p,Vs(:,j,i),DeltaT(j,i),a(:,j,i),0);
            end
            bigomega_d=s2bomega_d(N);
            q=(eye(4)+0.5*bigomega_d)*q;
        end
        h=[h; q];
        if(dervflag==1)
            newRow=[];
            for j=7:18
                newRow=[newRow s2bomega_c(gradN(:,j))*q];
            end
            H=[H; 0.5*newRow];
        end
    end

function thetas=q2eul(q)
    dcm=q2dcm(q);
    thetas=zeros(3,1);
    thetas(1)=atan2(dcm(2,3),dcm(3,3));
    thetas(2)=asin(-dcm(1,3));
    thetas(3)=atan2(dcm(1,2),dcm(1,1));

function dcm=q2dcm(q)
    q2=q.^2;
    dcm=[(q2(1)-q2(2)-q2(3)+q2(4)) 2*(q(1)*q(2)+q(3)*q(4)) 2*(q(1)*q(3)-q(2)*q(4));...
         2*(q(1)*q(2)-q(3)*q(4)) (q2(2)-q2(1)-q2(3)+q2(4)) 2*(q(2)*q(3)+q(1)*q(4));...
         2*(q(1)*q(3)+q(2)*q(4)) 2*(q(2)*q(3)-q(1)*q(4)) (q2(3)-q2(1)-q2(2)+q2(4))];

function q=eul2q(thetas)
    c=cos(thetas/2);
    s=sin(thetas/2);
    q=[s(1)*c(2)*c(3)-c(1)*s(2)*s(3);...
      c(1)*s(2)*c(3)+s(1)*c(2)*s(3);...
      c(1)*c(2)*s(3)-s(1)*s(2)*c(3);...
      prod(c)+prod(s)];

function bigomega=s2bomega_c(omega)
    bigomega=[0 omega(3) -omega(2) omega(1);...
              -omega(3) 0 omega(1) omega(2);...
              omega(2) -omega(1) 0 omega(3);...
              -omega(1) -omega(2) -omega(3) 0];

function bigomega=s2bomega_d(omega)
    dtheta=omega/80;
    ndtheta=norm(dtheta);
    dtheta=(2/ndtheta)*sin(ndtheta/2)*dtheta;
    c=2*(cos(ndtheta/2) -1);
    bigomega=[c dtheta(3) -dtheta(2) dtheta(1);...
              -dtheta(3) c dtheta(1) dtheta(2);...
              dtheta(2) -dtheta(1) c dtheta(3);...
              -dtheta(1) -dtheta(2) -dtheta(3) c];
```

6. findM, findN, findT, and findR

These are used to evaluate the accelerometer measurement function, the gyro measurement function, the temperature conversion from volts to degrees, and the rotation matrix, respectively.

```
function [Mvec,gradM]=findM(p,Vs,DeltaT,dervflag)
    Vb0=p(1:3);
```

```
    Ct1=p(4:6);
    Ar=diag(p(7:9));
    Ar(2,1)=p(10);
    Ar(3,1)=p(11);
    Ar(2,3)=p(12);
    Ar(3,2)=Ar(2,3);
    Ct2=p(13:15);
    A2=p(16:18);
    Vb=Vb0+Ct1*DeltaT;
    Mvec=(Ar + diag(Ct2)*DeltaT)*((Vs-Vb) + A2.*(Vs-Vb).^2);

    gradM=zeros(3,18);
    if(dervflag==1)
        S=((Vs-Vb) + A2.*(Vs-Vb).^2);
        gradM(:,1:3)=-diag((Ar + diag(Ct2)*DeltaT)*(1+2*A2.*(Vs-Vb)));
        gradM(:,4:6)=gradM(:,1:3)*DeltaT;
        gradM(:,7:9)=diag(S);
        gradM(:,10)=[0; S(1); S(1)];
        gradM(:,11)=[0; 0; S(1)];
        gradM(:,12)=[0; 0; S(2)];
        gradM(:,13:15)=DeltaT*gradM(:,7:9);
        gradM(:,16:18)=diag((Ar  + diag(Ct2)*DeltaT)*(Vs-Vb).^2);
    end

function [Nvec, gradN]=findN(p,Vs,DeltaT,a,dervflag)
    Vb0=p(1:3);
    Ct1=p(4:6);
    Ar=diag(p(7:9));
    Ar(2,1)=p(10);
    Ar(3,1)=p(11);
    Ar(3,2)=Ar(2,3);
    Ct2=p(13:15);
    A2=p(16:18);
    Ca=reshape(p(19:27),3,3);
    Vb=Vb0+Ct1*DeltaT+Ca*a;
    Nvec=(Ar + diag(Ct2)*DeltaT)*((Vs-Vb) + A2.*(Vs-Vb).^2);

    gradN=zeros(3,27);
    if(dervflag==1)
        S=((Vs-Vb) + A2.*(Vs-Vb).^2);
        gradN(:,1:3)=-diag((Ar + diag(Ct2)*DeltaT)*(1+2*A2.*(Vs-Vb)));
        gradN(:,4:6)=gradN(:,1:3)*DeltaT;
        gradN(:,7:9)=diag(S);
        gradN(:,10)=[0; S(1); S(1)];
        gradN(:,11)=[0; 0; S(1)];
        gradN(:,12)=[0; 0; S(2)];
        gradN(:,13:15)=DeltaT*gradN(:,7:9);
        gradN(:,16:18)=diag((Ar  + diag(Ct2)*DeltaT)*(Vs-Vb).^2);
        gradN(:,19:21)=gradN(:,1:3)*a(1);
        gradN(:,22:24)=gradN(:,1:3)*a(2);
        gradN(:,25:27)=gradN(:,1:3)*a(3);
    end

function DeltaT=findT(temp)
    bits2volts=5/4096;
    tempbias=2048;
    tempoffset=0;%=27;
    temp_Vperdeg=0.0084;
    DeltaT=(temp-tempbias)*bits2volts/temp_Vperdeg + tempoffset;

function [Rmat,DRmat_th,DRmat_ph]=findR(theta,phi,dervflag)
    Rmat=[-cos(theta) -sin(theta)*sin(phi)  -sin(theta)*cos(phi);
          0                      -cos(phi)  sin(phi);
          -sin(theta)  cos(theta)*sin(phi)  cos(theta)*cos(phi)];

    if(dervflag==1)
        DRmat_th=[sin(theta)  -cos(theta)*sin(phi) -cos(theta)*cos(phi);
                  0           0                    0;
                  -cos(theta) -sin(theta)*sin(phi) -sin(theta)*cos(phi)];

        DRmat_ph=[0  -sin(theta)*cos(phi)  sin(theta)*sin(phi);
                  0              sin(phi)  cos(phi);
                  0   cos(theta)*cos(phi)  -cos(theta)*sin(phi)];
    else
        DRmat_th=zeros(3);
        DRmat_ph=zeros(3);
    end
```

A-3 INS Raw output code

This code dumps the gyro and accelerometer data directly to the serial port (used MPLAB and C30 compiler v.2.20).

```
#include "p30f2012.h"
#include "stdio.h"

void printSerial_int(unsigned int val,char sep) {
PORTDbits.RD9=1;
while(U1STAbits.UTXBF);
U1TXREG=val & 0xFF;
while(U1STAbits.UTXBF);
U1TXREG=val >> 8;
while(U1STAbits.UTXBF);
U1TXREG=sep;
PORTDbits.RD9=0;
}

void printSerial_str(char * str) {
char i = 0;
while(str[i]!=0) {
while(U1STAbits.UTXBF);
U1TXREG=str[i];
i++;
}
}


char * hex_convert(char *buf, unsigned long value, char lastCar) {
char num[32];
int pos;

*buf++='0';
*buf++='x';

pos=0;
while(value!=0) {
char c = value & 0x0F;
num[pos++]="0123456789ABCDEF"[(unsigned) c];
value=(value >> 4) & (0x0fffffffL);
}
if(pos==0) num[pos++]='0';
while(--pos >= 0) *buf++=num[pos];

*buf++ = lastCar;
*buf=0;
return buf;
}

void printSerial_nbr(unsigned long nbr, char lastCar) {
char strNbr[12];
hex_convert(strNbr,nbr,lastCar);
printSerial_str(strNbr);
}

void timersetup(void) {
// Set up timer to trigger every 4410 cycles
T1CONbits.TCS=0;
T1CONbits.TCKPS=0;
T1CONbits.TGATE=0;
T1CONbits.TSIDL=1;
TMR1=0;
PR1=0x106D;    /* Interrupt after this many cycles*/
IPC0bits.T1IP = 4; /* set priority level */
IFS0bits.T1IF = 0; /* clear interrupt flag */
SRbits.IPL = 3; /* enable CPU priority levels 4-7 */
T1CONbits.TON=1;
IEC0bits.T1IE = 1; /* enable interrupts */
}

void adcsetup(void) {
ADPCFG=~127;  // AN0-AN6 A/D mode, rest are just ports
ADCSSL=127;  // Scan AN0-AN6 pins for data
```

```
ADCHS=0;

ADCON3bits.ADRC=1; // Use ADC internal clock

ADCON2bits.ALTS=0;  // Always use MUXA for inputs
ADCON2bits.BUFM=0; // Split buffers into 8 and 8 words
ADCON2bits.SMPI=13; // Sample/Convert sequences per interrupt
ADCON2bits.CSCNA=1; // Scan through multiple inputs
ADCON2bits.VCFG=0; // Select AVDD, AVSS for VREF

ADCON1bits.ASAM=0; // Auto start sampling
ADCON1bits.SSRC=7;  // Automatically convert after sampling
ADCON1bits.FORM=0;  // Results stored in integer form
ADCON1bits.ADSIDL=0;// Don't stop in idle mode
IEC0bits.ADIE=1;
ADCON1bits.ASAM=0;
ADCON1bits.ADON=1;
}

void uartsetup(void) {
U1MODEbits.STSEL=0; // 1 Stop bit
U1MODEbits.PDSEL=0; // 8 bit no parity
U1MODEbits.ALTIO=0; // Use regular tx/rx pins
U1MODEbits.UARTEN=1; // enable UART
U1BRG=28; // At Fcy=26MHZ, 57600 baud
U1STAbits.UTXEN=1; // enable TX
}

/*
 * Sampmode determines things about the next A/D conversion:
 * 0 - Accelerometers only
 * 1 - Accelerometers and gyros
 * 2 - Accelerometers, gyros, and temperature
 * Bit 3 set to 1 enables sleep mode during the conversion
 */
unsigned char sampmode=0;

// Calib mode on means we always stay in sampmode=6
unsigned char calibmode=0;

// The number of conversions so far this second
unsigned int convcnt=0;

typedef struct {
unsigned int temperature;
unsigned long mu1;
unsigned long mu2;
unsigned long mu3;
unsigned int a1;
unsigned int a2;
unsigned int a3;
unsigned int r1;
unsigned int r2;
unsigned int r3;

unsigned char tempupd; // Whether or not we have sent the current value of temperature
unsigned char r_upd;   // Whether or not we have sent the current values or r1-3
unsigned int  a_upd;   // The index up to which we have sent a1-3
} ConvData;

ConvData convData;

int main(void)
{
unsigned int data[8];
char sent;


TRISD=0;
PORTD=0x0000;

timersetup();
adcsetup();
uartsetup();

while(1) {
sent=0;
```

```
if(convData.tempupd) {
printSerial_int(convData.temperature,1);
convData.tempupd=0;
sent++;
}
if(convData.r_upd) {
printSerial_int(convData.r1,2);
printSerial_int(convData.r2,3);
printSerial_int(convData.r3,4);
convData.r_upd=0;
sent++;
}
if(convData.a_upd) {
printSerial_int(convData.a1,5);
printSerial_int(convData.a2,6);
printSerial_int(convData.a3,7);
convData.a_upd=0;
sent++;
}
if(sent) printSerial_int('\r','\n');
}

return 0;
}

void __attribute__((__interrupt__)) _ADCInterrupt(void)
{
IFS0bits.ADIF=0;
ADCON1bits.ASAM=0;

switch(sampmode) {
case 2:
// Temp, gyro, acc (no sleep)
convData.temperature=ADCBUF6;
convData.tempupd=1;
case 1:
// gyro, acc (no sleep)
convData.r1 = ADCBUF3;
convData.r2 = ADCBUF4;
convData.r3 = ADCBUF5;
convData.r_upd=1;
case 0:
// acc (no sleep)
convData.mu1+=ADCBUF0;
convData.mu2+=ADCBUF1;
convData.mu3+=ADCBUF2;
break;

case 4:
// acc (sleep)
convData.mu1+=ADCBUF3;
convData.mu2+=ADCBUF4;
convData.mu3+=ADCBUF5;
break;
case 5:
// gyro, acc (sleep)
convData.mu1+=ADCBUF6;
convData.mu2+=ADCBUF7;
convData.mu3+=ADCBUF8;
convData.r1 = ADCBUF9;
convData.r2 = ADCBUFA;
convData.r3 = ADCBUFB;
convData.r_upd=1;
break;
case 6:
// Temp, gyro, acc (sleep)
convData.mu1+=ADCBUF7;
convData.mu2+=ADCBUF8;
convData.mu3+=ADCBUF9;
convData.r1 = ADCBUFA;
convData.r2 = ADCBUFB;
convData.r3 = ADCBUFC;
convData.temperature=ADCBUFD;
convData.r_upd=1;
convData.tempupd=1;
break;
}
```

```
/*
 *  Figure out what the next conversion should measure
 */
convcnt++;
sampmode=0;
if((convcnt % 75)==0) {
IEC0bits.T1IE = 0;
convData.a1=convData.mu1/75;
convData.a2=convData.mu2/75;
convData.a3=convData.mu3/75;
convData.a_upd=1;
convData.mu1=0;
convData.mu2=0;
convData.mu3=0;
IEC0bits.T1IE = 1;
}

  if(calibmode) {
sampmode=6;
} else if(convcnt>=6000) {
// 6020 conversion cycles per second; once a second, sample gyro
// and temp gauge in sleep mode
convcnt=0;
sampmode=6;
} else if((convcnt % 75)==0) {
// At 140 Hz, sample the gyro in sleep mode
sampmode=5;
}

PORTDbits.RD9=0;
return;
}

void __attribute__((__interrupt__)) _T1Interrupt(void)
{
// Reset the timer
IEC0bits.T1IE = 0;
timersetup();
IEC0bits.T1IE = 1;

PORTDbits.RD9=1;

switch(sampmode & 0x03) {
case 0:
ADCSSL=7;  // Scan AN0-AN2 pins for data
ADCON2bits.SMPI=2; // Sample/Convert sequences per interrupt
break;
case 1:
ADCSSL=63;  // Scan AN0-AN5 pins
ADCON2bits.SMPI=5; // Sample/Convert sequences per interrupt
break;
case 2:
ADCSSL=127;  // Scan AN0-AN6 pins
ADCON2bits.SMPI=6; // Sample/Convert sequences per interrupt
break;
}

// Add extra conversions to make sure we have entered sleep mode before
// doing the main ones
if(sampmode & 0x04)
ADCON2bits.SMPI=2*ADCON2bits.SMPI+1;

ADCON1bits.ASAM=1;

if(sampmode & 0x04) Sleep();

return;
}
```

A-4 INS main code

This code performs the actual INS processing. The main.c file contains the bulk of the code. main.h contains parameter values and definitions. serial_stuff.c contains functions for serial output.

main.c

```
#include "p30f2012.h"
#include "stdio.h"
#include "math.h"
#include "serial_stuff.h"
#include "main.h"


void timersetup(void) {
// Set up timer to trigger every 4410 cycles
T1CONbits.TCS=0;
T1CONbits.TCKPS=0;
T1CONbits.TGATE=0;
T1CONbits.TSIDL=1;
TMR1=0;
PR1=0x106D;    /* Interrupt after this many cycles*/
IPC0bits.T1IP = 4; /* set priority level */
IFS0bits.T1IF = 0; /* clear interrupt flag */
SRbits.IPL = 3; /* enable CPU priority levels 4-7 */
T1CONbits.TON=1;
IEC0bits.T1IE = 1; /* enable interrupts */
}

void adcsetup(void) {
ADPCFG=~127;  // AN0-AN6 A/D mode, rest are just ports
ADCSSL=127;  // Scan AN0-AN6 pins for data

ADCHS=0;

ADCON3bits.ADRC=1; // Use ADC internal clock

ADCON2bits.ALTS=0;  // Always use MUXA for inputs
ADCON2bits.BUFM=0; // Split buffers into 8 and 8 words
ADCON2bits.SMPI=13; // Sample/Convert sequences per interrupt
ADCON2bits.CSCNA=1; // Scan through multiple inputs
ADCON2bits.VCFG=0; // Select AVDD, AVSS for VREF

ADCON1bits.ASAM=0; // Auto start sampling
ADCON1bits.SSRC=7;  // Automatically convert after sampling
ADCON1bits.FORM=0;  // Results stored in integer form
ADCON1bits.ADSIDL=0;// Don't stop in idle mode
IEC0bits.ADIE=1;
ADCON1bits.ASAM=0;
ADCON1bits.ADON=1;
}

void uartsetup(void) {
U1MODEbits.STSEL=0; // 1 Stop bit
U1MODEbits.PDSEL=0; // 8 bit no parity
U1MODEbits.ALTIO=0; // Use regular tx/rx pins
U1MODEbits.UARTEN=1; // enable UART
U1BRG=28; // At Fcy=26MHZ, 57600 baud
U1STAbits.UTXEN=1; // enable TX
}

int main(void)
{
unsigned char *aff;
unsigned int *dw;
unsigned int da;
double deltat, dtheta, s_, c_;
double ahat[3]={0,0,0};
double Vbg[3]={0,0,0};
double omegahat[3]={0,0,0};
double q[4]={0,0,0,0};
double q2[4];
double C[3][3];
```

```
double a[3]={0,0,0};
double vn[3]={0,0,0};
double rn[3]={0,0,0};

double tx2, ty2, ctx2, cty2, stx2, sty2, maxa;

// This determines how long we average accelerometers to measure initial tilt
int startup=STARTUPTIME;

// This tell us if we need to send a newline at the end of some sent data
char sent;

tx2=0;
ty2=0;

TRISD=0;
PORTD=0x0000;

timersetup();
adcsetup();
uartsetup();

while(1) {
sent=0;

if(startup > 0) {
if(convData.a_upd) {
convData.a_upd=0;

// Add up the accelerometer data (will divide later)
ahat[0]+=(((double)convData.a1)*bits2volts-Vb_a0)*A_a0;
ahat[1]+=(((double)convData.a2)*bits2volts-Vb_a1)*A_a1;
ahat[2]+=(((double)convData.a3)*bits2volts-Vb_a2)*A_a2;

// This means we have all our samples
if(startup==1) {
// Average accelerometer data
ahat[0]=ahat[0]/STARTUPTIME;
ahat[1]=ahat[1]/STARTUPTIME;
ahat[2]=ahat[2]/STARTUPTIME;

maxa=ahat[0];
// Find maximum accelerometer vector
if(ahat[1] > ahat[0] || -ahat[1] > -ahat[0])
maxa=ahat[1];
if(ahat[2] > maxa || -ahat[2] > -maxa)
maxa=ahat[2];

// Find thetax and thetay
// ahat should not exceed 1, but it might
// by a little bit.
if(maxa > 0) {
if(ahat[1] >1 || ahat[1] < -1)
tx2=ahat[1] > 0 ? -PI/4:PI/4;
else
tx2=-asin(ahat[1])/2;

if(ahat[0] >1 || ahat[0] < -1)
ty2=ahat[0] > 0 ? PI/4:-PI/4;
else
ty2=asin(ahat[0])/2;
} else {
if(ahat[1] >1 || ahat[1] < -1)
tx2=ahat[1] > 0 ? 3*PI/4:-3*PI/4;
else
tx2=(PI+asin(ahat[1]))/2;

if(ahat[0] >1 || ahat[0] < -1)
ty2=ahat[0] > 0 ? -PI/4:PI/4;
else
ty2=-asin(ahat[0])/2;
}

// Initialize quaternion
ctx2=cos(tx2);
cty2=cos(ty2);
stx2=sin(tx2);
```

```
sty2=sin(ty2);
q[0]=stx2*cty2;
q[1]=ctx2*sty2;
q[2]=-stx2*sty2;
q[3]=ctx2*cty2;

/* while(1) {
printSerial_float(tx2,1);
printSerial_float(ty2,3);
printSerial_float(q[0],5);
printSerial_float(q[1],7);
printSerial_float(q[2],9);
printSerial_float(q[3],11);
printSerial_int('\r','\n');

printSerial_float(ahat[0],1);
printSerial_float(ahat[1],3);
printSerial_float(ahat[2],5);
printSerial_int('\r','\n');
}*/
}
startup--;
}
} else {
// Update temperature
if(convData.tempupd) {
deltat=(convData.temperature-2048)*bits2volts*GainTemp;
convData.tempupd=0;
}

// This is the main INS update step
if(convData.r_upd && convData.a_upd) {
convData.r_upd=0;
convData.a_upd=0;
PORTDbits.RD8=1;

// Calculate acceleration
ahat[0]=(((double)convData.a1)*bits2volts-Vb_a0)*A_a0;
ahat[1]=(((double)convData.a2)*bits2volts-Vb_a1)*A_a1;
ahat[2]=(((double)convData.a3)*bits2volts-Vb_a2)*A_a2;

// Calculate gyro bias
Vbg[0]=Vb0_g0+Ct1_g0*deltat+Aa_g0*ahat[0]+Aa_g1*ahat[1]+Aa_g2*ahat[2];
Vbg[1]=Vb0_g1+Ct1_g1*deltat+Aa_g3*ahat[0]+Aa_g4*ahat[1]+Aa_g5*ahat[2];
Vbg[2]=Vb0_g2+Ct1_g2*deltat+Aa_g6*ahat[0]+Aa_g7*ahat[1]+Aa_g8*ahat[2];

// Calculate angular velocity * time step = change in angle
omegahat[0]=(((double)convData.r1)*bits2volts-Vbg[0])*GainGyro/FSAMP;
omegahat[1]=-(((double)convData.r2)*bits2volts-Vbg[1])*GainGyro/FSAMP;
omegahat[2]=(((double)convData.r3)*bits2volts-Vbg[2])*GainGyro/FSAMP;

// Update quaternion
dtheta=sqrt((omegahat[0]*omegahat[0] + omegahat[1]*omegahat[1] + omegahat[2]*omegahat[2]));
s_=sin(dtheta/2)/dtheta;
c_=cos(dtheta/2);
q[0]=c_*q[0]+s_*(omegahat[2]*q[1]-omegahat[1]*q[2]+omegahat[0]*q[3]);
q[1]=c_*q[1]+s_*(-omegahat[2]*q[0]+omegahat[0]*q[2]+omegahat[1]*q[3]);
q[2]=c_*q[2]+s_*(omegahat[1]*q[0]-omegahat[0]*q[1]+omegahat[2]*q[3]);
q[3]=c_*q[3]+s_*(-omegahat[0]*q[0]-omegahat[1]*q[1]-omegahat[2]*q[2]);

// Update rotation matrix
q2[0]=q[0]*q[0];
q2[1]=q[1]*q[1];
q2[2]=q[2]*q[2];
q2[3]=q[3]*q[3];
C[0][0]=q2[0]-q2[1]-q2[2]+q2[3];
C[0][1]=2*(q[0]*q[1]+q[2]*q[3]);
C[0][2]=2*(q[0]*q[2]-q[1]*q[3]);
C[1][0]=2*(q[0]*q[1]-q[2]*q[3]);
C[1][1]=q2[1]-q2[0]-q2[2]+q2[3];
C[1][2]=2*(q[1]*q[2]+q[0]*q[3]);
C[2][0]=2*(q[0]*q[2]+q[1]*q[3]);
C[2][1]=2*(q[1]*q[2]-q[0]*q[3]);
C[2][2]=q2[2]-q2[0]-q2[1]+q2[3];

// This is our actual acceleration (minus gravity)
a[0]=C[0][0]*ahat[0]+C[0][1]*ahat[1]+C[0][2]*ahat[2];
```

```
a[1]=C[1][0]*ahat[0]+C[1][1]*ahat[1]+C[1][2]*ahat[2];
a[2]=C[2][0]*ahat[0]+C[2][1]*ahat[1]+C[2][2]*ahat[2]-1;

// Update velocity
vn[0]+=a[0]/FSAMP;
vn[1]+=a[1]/FSAMP;
vn[2]+=a[2]/FSAMP;

// Update position
rn[0]+=vn[0]/FSAMP;
rn[1]+=vn[1]/FSAMP;
rn[2]+=vn[2]/FSAMP;

// Output it over serial port
printSerial_float(vn[0],1);
printSerial_float(vn[1],3);
printSerial_float(vn[2],5);
printSerial_float(rn[0],7);
printSerial_float(rn[1],9);
printSerial_float(rn[2],11);

sent=1;
PORTDbits.RD8=0;
}
}
if(sent) printSerial_int('\r','\n');
}

return 0;
}

void __attribute__((__interrupt__)) _ADCInterrupt(void)
{
IFS0bits.ADIF=0;
ADCON1bits.ASAM=0;

switch(sampmode) {
case 2:
// Temp, gyro, acc (no sleep)
convData.temperature=ADCBUF6;
convData.tempupd=1;
case 1:
// gyro, acc (no sleep)
convData.r1 = ADCBUF3;
convData.r2 = ADCBUF4;
convData.r3 = ADCBUF5;
convData.r_upd=1;
case 0:
// acc (no sleep)
convData.mu1+=ADCBUF0;
convData.mu2+=ADCBUF1;
convData.mu3+=ADCBUF2;
break;

case 4:
// acc (sleep)
convData.mu1+=ADCBUF3;
convData.mu2+=ADCBUF4;
convData.mu3+=ADCBUF5;
break;
case 5:
// gyro, acc (sleep)
convData.mu1+=ADCBUF6;
convData.mu2+=ADCBUF7;
convData.mu3+=ADCBUF8;
convData.r1 = ADCBUF9;
convData.r2 = ADCBUFA;
convData.r3 = ADCBUFB;
convData.r_upd=1;
break;
case 6:
// Temp, gyro, acc (sleep)
convData.mu1+=ADCBUF7;
convData.mu2+=ADCBUF8;
convData.mu3+=ADCBUF9;
convData.r1 = ADCBUFA;
convData.r2 = ADCBUFB;
```

```
convData.r3 = ADCBUFC;
convData.temperature=ADCBUFD;
convData.r_upd=1;
convData.tempupd=1;
break;
}

/*
 *  Figure out what the next conversion should measure
 */
convcnt++;
sampmode=0;
if((convcnt % gyrofrac)==0) {
IEC0bits.T1IE = 0;
convData.a1=convData.mu1/gyrofrac;
convData.a2=convData.mu2/gyrofrac;
convData.a3=convData.mu3/gyrofrac;
convData.a_upd=1;
convData.mu1=0;
convData.mu2=0;
convData.mu3=0;
IEC0bits.T1IE = 1;
}

if(convcnt>=numcyc) {
// 6000 conversion cycles per second; once a second, sample gyro
// and temp gauge in sleep mode
convcnt=0;
sampmode=6;
} else if((convcnt % gyrofrac)==0) {
// At 80 Hz, sample the gyro in sleep mode
sampmode=5;
}

PORTDbits.RD9=0;
return;
}

void __attribute__((__interrupt__)) _T1Interrupt(void)
{
// Reset the timer
IEC0bits.T1IE = 0;
timersetup();
IEC0bits.T1IE = 1;

PORTDbits.RD9=1;

switch(sampmode & 0x03) {
case 0:
ADCSSL=7;  // Scan AN0-AN2 pins for data
ADCON2bits.SMPI=2; // Sample/Convert sequences per interrupt
break;
case 1:
ADCSSL=63;  // Scan AN0-AN5 pins
ADCON2bits.SMPI=5; // Sample/Convert sequences per interrupt
break;
case 2:
ADCSSL=127;  // Scan AN0-AN6 pins
ADCON2bits.SMPI=6; // Sample/Convert sequences per interrupt
break;
}

// Add extra conversions to make sure we have entered sleep mode before
// doing the main ones
if(sampmode & 0x04)
ADCON2bits.SMPI=2*ADCON2bits.SMPI+1;

ADCON1bits.ASAM=1;

if(sampmode & 0x04) Sleep();

return;
}
```

main.h

```
/*
```

```
 * Sampmode determines things about the next A/D conversion:
 * 0 - Accelerometers only
 * 1 - Accelerometers and gyros
 * 2 - Accelerometers, gyros, and temperature
 * Bit 3 set to 1 enables sleep mode during the conversion
 */
unsigned char sampmode=0;

// The number of conversions so far this second
unsigned int convcnt=0;

typedef struct {
unsigned int temperature;
unsigned long mu1;
unsigned long mu2;
unsigned long mu3;
unsigned int a1;
unsigned int a2;
unsigned int a3;
unsigned int r1;
unsigned int r2;
unsigned int r3;

unsigned char tempupd; // Whether or not we have sent the current value of temperature
unsigned char r_upd;   // Whether or not we have sent the current values or r1-3
unsigned char  a_upd;   // The index up to which we have sent a1-3
} ConvData;

ConvData convData;

#define printSerial_float(x,y) aff=(char *) &(x); \
       dw=(unsigned int *) aff; \
     printSerial_int(dw[0],y); \
 printSerial_int(dw[1],y+1); \


// Sampling parameters
#define bits2volts 5/4096
#define FSAMP 80
#define numcyc 6000
#define gyrofrac 75

#define GainTemp 0.0084 // Temperature degrees per volt
#define Vb_a0 2.50731184854958 // x accelerometer bias
#define Vb_a1 2.51708344822763 // y accelerometer bias
#define Vb_a2 2.65085336425370 // z accelerometer bias
#define A_a0 0.92889698838059 // x accelerometer gain
#define A_a1 -0.92712035053800 // y accelerometer gain
#define A_a2 1.02570005604879 // z accelerometer gain

#define Vb0_g0 2.54550112933003 // x gyro bias
#define Vb0_g1 2.42309372110485 // y gyro bias
#define Vb0_g2 2.38393062104104 // z gyro bias
#define Ct1_g0 -0.00127332045947   // x gyro bias temp. gain
#define Ct1_g1 -0.00193392098413   // y gyro bias temp. gain
#define Ct1_g2 -0.00309753818907   // z gyro bias temp. gain
#define Aa_g0 0.00301424106840 // Aa(1,1)
#define Aa_g1 -0.00008541220703 // Aa(1,2)
#define Aa_g2 -0.00103822790568 // Aa(1,3)
#define Aa_g3 0.00038442336069 // Aa(2,1)
#define Aa_g4 -0.00253183153360 // Aa(2,2)
#define Aa_g5 0.00172178312521 // Aa(2,3)
#define Aa_g6 -0.00179240853058 // Aa(3,1)
#define Aa_g7 -0.00129305222382    // Aa(3,2)
#define Aa_g8 0.00142030748636 // Aa(3,3)
#define GainGyro 1.3963 // gyro gain

#define STARTUPTIME 20

#define PI 3.14159
```

serial_stuff.c

```
#include "p30f2012.h"

void printSerial_int(unsigned int val,char sep) {
```

```
PORTDbits.RD9=1;
while(U1STAbits.UTXBF);
U1TXREG=val & 0xFF;
while(U1STAbits.UTXBF);
U1TXREG=val >> 8;
while(U1STAbits.UTXBF);
U1TXREG=sep;
PORTDbits.RD9=0;
}

void printSerial_str(char * str) {
char i = 0;
while(str[i]!=0) {
while(U1STAbits.UTXBF);
U1TXREG=str[i];
i++;
}
}

char * hex_convert(char *buf, unsigned long value, char lastCar) {
char num[32];
int pos;

*buf++='0';
*buf++='x';

pos=0;
while(value!=0) {
char c = value & 0x0F;
num[pos++]="0123456789ABCDEF"[(unsigned) c];
value=(value >> 4) & (0x0fffffffL);
}
if(pos==0) num[pos++]='0';
while(--pos >= 0) *buf++=num[pos];

*buf++ = lastCar;
*buf=0;
return buf;
}

void printSerial_nbr(unsigned long nbr, char lastCar) {
char strNbr[12];
hex_convert(strNbr,nbr,lastCar);
printSerial_str(strNbr);
}
```

serial_stuff.h

```
void printSerial_int(unsigned int val,char sep);
void printSerial_str(char * str);
char * hex_convert(char *buf, unsigned long value, char lastCar);
void printSerial_nbr(unsigned long nbr, char lastCar);
```

A-5 PC serial capture code

This C# code was used to do all of the serial data capture. The radio buttons allow the selection of capturing integer values from the raw output code or floating point data from the full INS code.

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO.Ports;
```

```
using System.IO;
using System.Threading;


namespace RawIMU
{
    public partial class Form1 : Form
    {
        private SerialPort com;
        private List<dataline> data;
        private List<INSdataline> INSdata;
        private Thread rxThread;
        private int logtime;

        public Form1()
        {
            InitializeComponent();
            radioButton_raw.Checked = true;
            com = new SerialPort("COM1", 57600);

            data = new List<dataline>();
            INSdata = new List<INSdataline>();
            timer1.Enabled = true;

        }

        ~Form1() {
            com.Close();
            com.Dispose();
        }

        private void OpenButton_Click(object sender, EventArgs e)
        {
            if (!this.com.IsOpen) {
                data.Clear();
                INSdata.Clear();
                com = new SerialPort("COM1", 57600,Parity.None,8,StopBits.One);

                rxThread = new Thread(new ThreadStart(Receive));
                rxThread.Start();
                try {
                    com.Open();
                }
                catch {
                    MessageBox.Show("Could not open serial port");
                    rxThread.Abort();
                    return;
                }
                logtime = 0;
                ((Button)sender).Text = "Close Port";
            }
            else {
                com.Close();
                rxThread.Abort();
                ((Button)sender).Text = "Open Port";
            }
        }

        void Receive() {
            try {
                byte[] buf=new byte[1024];
                byte[] buf2 = new byte[1024];
                int bufptr = 0;

                while (true) {
                    if (!com.IsOpen)
                        continue;
                    try {
                        while (true) {
                            buf[bufptr++] =(byte) com.ReadByte();
                            if (bufptr == 1024) {
                                bufptr = 0;
                                break;
                            }
                        }
                    }
                    catch (IOException) {
```

```
                    continue;
                }

                int mark=-1;
                for (int i = 3; i < 1024; i++) {
                    if ((buf[i - 1] == 10) && (buf[i - 2] == 0) && (buf[i - 3] == 13)) {
                        if (mark != -1) {
                            if (radioButton_raw.Checked)
                                data.Add(new dataline(buf2, i - mark - 3));
                            else if (radioButton_ins.Checked)
                                INSdata.Add(new INSdataline(buf2, i - mark - 3));
                        }
                        mark = i;
                    }
                    if (mark != -1)
                        buf2[i - mark] = buf[i];
                }
                buf = buf2;
            }
        }
        catch (ObjectDisposedException) {
            if (rxThread != null)
                rxThread = null;
        }
    }

    SaveFileDialog dlg;

    private void savebutton_Click(object sender, EventArgs e) {
        dlg = new SaveFileDialog();
        dlg.FileOk += new CancelEventHandler(dlg_FileOk);
        dlg.ShowDialog();
    }

    void dlg_FileOk(object sender, CancelEventArgs e) {
        string name = dlg.FileName;
        using (StreamWriter sw = new StreamWriter(name)) {
            for (int i = 0; i < data.Count; i++) {
                sw.WriteLine(data[i].ToString());
            }
            for (int i = 0; i < INSdata.Count; i++) {
                sw.WriteLine(INSdata[i].ToString());
            }
        }
    }

    private void timer1_Tick(object sender, EventArgs e) {
        if (com.IsOpen) {
            logtime++;
            logtimelabel.Text = logtime.ToString();
            if(radioButton_raw.Checked)
                loglengthlabel.Text = data.Count.ToString();
            else if(radioButton_ins.Checked)
                loglengthlabel.Text = INSdata.Count.ToString();
        }
    }

}

public struct dataline {
    public int ind;
    public int a1;
    public int a2;
    public int a3;
    public int r1;
    public int r2;
    public int r3;
    public int temp;

    public dataline(byte[] data,int length) {
        ind = (int) System.DateTime.Now.Second;
        a1 = -1;
        a2 = -1;
        a3 = -1;
        r1 = -1;
        r2 = -1;
        r3 = -1;
```

```
        temp = -1;

        int i = 0;
        if(length < 3) return;
        while (i <= (length-3)) {
            int val = data[i] + (data[i + 1] << 8);
            switch (data[i + 2]) {
                case 1:
                    temp = val;
                    break;
                case 2:
                    r1 = val;
                    break;
                case 3:
                    r2 = val;
                    break;
                case 4:
                    r3 = val;
                    break;
                case 5:
                    a1 = val;
                    break;
                case 6:
                    a2 = val;
                    break;
                case 7:
                    a3 = val;
                    break;
            }
            i += 3;
        }
    }

    public override string ToString() {
        string str = "";
        str += ind.ToString();
        str += ",";
        str += a1.ToString();
        str += ",";
        str += a2.ToString();
        str += ",";
        str += a3.ToString();
        str += ",";
        str += r1.ToString();
        str += ",";
        str += r2.ToString();
        str += ",";
        str += r3.ToString();
        str += ",";
        str += temp.ToString();
        return str;
    }
}

public struct INSdataline {
    public int ind;
    public double v1;
    public double v2;
    public double v3;
    public double r1;
    public double r2;
    public double r3;

    long v1_;
    long v2_;
    long v3_;
    long r1_;
    long r2_;
    long r3_;

   double bin2float(long bla) {
        int offset = 8388608;
        double mant = ((double)(bla & 0x7FFFFF)) / offset;
        long exp = ((bla & 0x7F800000) >> 23) - 127;
        int sgn = (bla & 0x80000000) == 0 ? 1 : -1;
        return sgn * (1 + mant) * Math.Pow(2, exp);
    }
```

```
public INSdataline(byte[] data,int length) {
    v1_ = 0;
    v2_ = 0;
    v3_ = 0;
    r1_ = 0;
    r2_ = 0;
    r3_ = 0;
    ind = (int) System.DateTime.Now.Second;
    v1 = double.NaN;
    v2 = double.NaN;
    v3 = double.NaN;
    r1 = double.NaN;
    r2 = double.NaN;
    r3 = double.NaN;

    int i = 0;
    if(length < 3) return;
    while (i <= (length-3)) {
        int val = data[i] + (data[i + 1] << 8);
        switch (data[i + 2]) {
            case 1:
                v1_=val;
                break;
            case 2:
                v1_+=val << 16;
                v1=bin2float(v1_);
                break;
            case 3:
                v2_=val;
                break;
            case 4:
                v2_ += val<<16;
                v2=bin2float(v2_);
                break;
            case 5:
                v3_=val;
                break;
            case 6:
                v3_+=val<<16;
                v3=bin2float(v3_);
                break;
            case 7:
                r1_=val;
                break;
            case 8:
                r1_+=val<<16;
                r1=bin2float(r1_);
                break;
            case 9:
                r2_=val;
                break;
            case 10:
                r2_+=val<<16;
                r2=bin2float(r2_);
                break;
            case 11:
                r3_=val;
                break;
            case 12:
                r3_+=val<<16;
                r3=bin2float(r3_);
                break;
        }
        i += 3;
    }
}

public override string ToString() {
    string str = "";
    str += ind.ToString();
    str += ",";
    str += v1.ToString();
    str += ",";
    str += v2.ToString();
    str += ",";
```

```
            str += v3.ToString();
            str += ",";
            str += r1.ToString();
            str += ",";
            str += r2.ToString();
            str += ",";
            str += r3.ToString();
            return str;
        }
    }
}
```

References

[1] Advanced Inertial Reference Sphere. Web page `http://nuclearweaponarchive.org/Usa/Weapons/Airs.html`. Accessed on December 17, 2005.

[2] Andreyev, V. D. *Teoriia Inerstialnoi Navigatsii*. Moscow, 1966.

[3] Ang, Wei Tech. Physical Model of a MEMS Accelerometer for Low-g Motion Tracking Applications. Proceedings of the 2004 IEEE Intl. Conference on Robotics & Automation, New Orleans, LA, April 2004, pp. 1345-1351.

[4] Bar-Shalom, Yakov. *Estimation with Applications to Tracking and Navigation*. Wiley-Interscience, New York, 2001.

[5] Collin, Jussi. Navigating the City. GPS World, June 1, 2002. Available at `http://www.gpsworld.com/gpsworld/article/articleDetail.jsp?id=118816&&pageID=4`.

[6] Crenshaw, Jack W. *Math Toolkit for Real-Time Programming*. CMP Books, Lawrence, Kansas, 2000.

[7] Grewal, Mohinder S. *Global Positioning Systems, Inertial Navigation, and Integration*. John Wiley, New York, 2001.

[8] King, A. D. Inertial Navigation - Forty Years of Evolution. GEC Review, v.13, no. 3, 1998, pp. 140-149. Available at `http://www.imar-navigation.de/download/inertial_navigation_introduction.pdf`.

[9] Lourakis, Manolis I. A. A Brief Description of the Levenberg-Marquardt Algorithm Implemented by `levmar`. Institute of Computer Science, Heraklion, Crete, Greece, 2005. Available at `http://www.ics.forth.gr/ lourakis/levmar/levmar.pdf`.

[10] Marins, João. An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors. Proceedings of the 2001 IEEE Intl. Conference on Intelligent Robotics and Systems, Oct. 29, 2001, pp 2003-2011.

[11] Shin, Eun-Hwan. Accuracy Improvement of Low Cost INS/GPS for Land Applications. Masters thesis, University of Calgary, Dec. 2001. Available at `http://www.geomatics.ucalgary.ca/Papers/Thesis/NES/01.20156.EHShin.pdf`.

[12] O'Donnell, C.F. *Inertial Navigation Analysis and Design*. McGraw-Hill, New York, 1964.