Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

An Evaluation of Low Cost Inertial Sensors for Hierarchical Sensing

Semester thesis

Aron Hjartarson & Kári Hreinsson

November 2011 to March 2012

Supervisor: Bernhard Buchli
Professor: Dr. Lothar Thiele

Abstract

In this project a sensor circuit was designed to reduce power consumption of GPS nodes used in the PermaSense project and increase the relevance of the GPS data measured. The sensor circuit serves as an inertial measurement unit which can activate the GPS chip upon movement detection instead of relying on predefined duty cycles. A communication protocol between the sensor circuit and a host was designed. An algorithm to extract additional spatial information from the sensor data was implemented and methods for movement detection were considered. The spatial information was extracted with mixed results and could be done more reliably in further work by implementing sensor fusion algorithms and advanced filtering. On–site evaluation using real data is also necessary to assess the parameters of the detection mechanisms considered.

Contents

Tables

Figures

1

Introduction

1.1 Motivation

1.1.1 PermaSense / X–Sense Project

The focus of the PermaSense / X–Sense project [23] is to establish wireless sensor networks in the alpine landscape of Switzerland that can sustain long term and high quality sensing of geological data, such as movement of rock formations and the effects of climate change. The project is conducted by researchers from ETH Zurich, University of Zurich and University of Basel. The data obtained can be used in decision making for harsh terrains as well as predicting natural hazards.

1.1.2 GPS Nodes

GPS nodes [2] are used to monitor the movement of rock formations. The GPS chips in those nodes consume a lot of power and having them turned on for multiple consecutive hours or days for measurements is not viable. Currently the GPS chips are turned on based on predefined time cycles with long measurement blocks. Therefore, during the off–time of the GPS chips, important activity might be missed. Equipping each node with an excessively large battery or developing more power efficient GPS chips are currently not interesting solutions. Instead of using these predefined time cycles, the on–time of the GPS chips could be optimized based on additional data, possibly increasing their power efficiency and more importantly increasing the relevance of the data.

1.2 Aim

The aim of this thesis is to create a standalone sensor circuit equipped with low–power sensors that can be implemented into the GPS nodes, equipped with movement detection mechanisms and the capability to provide additional spatial information about the GPS node to support the GPS readings. Based on that information, we could limit the on time of the GPS circuitry to periods of actual activity, possibly reducing the energy consumption of the nodes and eliminating gaps in data due to periodic off times. The sensors considered are an accelerometer, a gyroscope and a compass, all in the form of integrated circuits. We aim to develop a low overhead algorithm to extract spatial information from the data and for movement detection. In a future version, the sensor circuit could be connected to a TinyNode module [24] that would turn on the GPS circuitry upon receiving an interrupt signal from the microprocessor on which the algorithm would be written.

1.3 Problem Statement

A blueprint for how to implement the overall hardware design has to be established, i.e. whether to base it around the CoreStation [3], or to use a standalone microprocessor. Appropriate hardware, such as sensors and microprocessors has to be found and purchased for the project. The circuit sensors would have to be chosen with respect to low power, high precision and good availability. Once the circuit is assembled an efficient and preferably simple method has to be developed to ”multiplex” the data from the different sensors to an external device, so a solution has to be implemented on the processing unit chosen for the project. A somewhat realistic test setup has to be designed to test the sensor circuit and log some trial data for analysis.

An algorithm involving relevant filtering and processing methods has to be developed to extract additional information from the data and for movement detection. The methods of extracting additional information from the sensor data have to be realized and evaluated. Different movement detection strategies should be studied, based on either raw sensor data or the spatial data extracted.

1.4 Thesis Contributions

The contributions from this semester thesis are the following:

- A standalone sensor circuit that can support a GPS node using movement detection mechanisms and providing additional spatial information.
- A communication protocol between the circuit and an external host.
- Methods and code to process and extract data from the sensors for the purposes of movement detection.

1.5 Related Work

The implementation of environmental sensor networks has recently been studied [19, 17] and online GPS nodes for use in the alpine landscape have been developed and evaluated as a part of the PermaSense project [15, 2]. Estimating the spatial orientation of an aerial or space vehicle is more commonly referred to as attitude estimation and the design of such attitude determination systems has been well studied [13]. A miniature attitude and heading reference system using an accelerometer, gyroscope and a magnetic field sensor has been developed for human motion analysis, using complementary filters and sensor fusion algorithms with promising results

[16]. Nonlinear complementary filters have been developed for attitude estimation using an inertial measurement unit, which normally consists of an accelerometer, gyroscope and a magnetic sensor [18]. A similar study was done using a combination of an inertial measurement unit and a small camera [1].

2

Hardware and Software Implementation

2.1 Choosing the Hardware

After exploring different sensors and microprocessors we settled on the following integrated circuits as they were a good compromise of functionality, power usage, price and availability.

2.1.1 Accelerometer (LIS302DL)

The LIS302DL [8] is a low power three axes linear accelerometer with an I^{2}C/SPI serial interface. It has two user selectable measurement ranges, $\pm$ 2 g and $\pm$ 8 g, draws a supply current of 0.3 mA and has a sensitivity of 72 mg/digit at full measurement range. It can measure acceleration at an output data rate of 100 Hz or 400 Hz and is frequently used in applications such as free fall detection, motion activated functions and gaming devices to name a few.

Default setting as shown in the data sheet were used with the following exceptions:

Data rate (DR) was set to 1 (400 Hz).

Power down control (PR) was set to 1 (active mode).

Full scale (FS) was set to either 0 or 1 in different test runs.

Filtered data selection (FDS) was set to either 0 or 1 in different test runs.

High-pass filter cutoff frequency (HP coeff1/HP coeff2) was set to different values during test runs.

2.1.2 Gyroscope (L3G4200D)

The L3G4200D [7] is a low power three axis angular rate sensor with an I^{2}C/SPI serial interface. It has three user selectable measurement ranges, $\pm$ 250, $\pm$ 500 and $\pm$ 2000 dps, draws a supply current of 6.1 mA and has a sensitivity of 70 mdps/digit at full measurement range. It can measure angular rate at an output data rate of 100, 200, 400 or 800 Hz and is frequently used in applications such as GPS navigation systems, robotics and gaming devices.

Default setting as shown in the data sheet were used with the following exceptions:

Output data rate (DR1–DR0) was set to 10 (400 Hz).

Bandwidth selection (BW1–BW0) was set to 10 (50 Hz).

Power down control (PD) was set to 1 (active mode).

2.1.3 Compass (HMR3400)

The HMR3400 [6] a tilt compensated precision compass with a USART serial data interface. The tilt compensation is achieved using a built in three axes accelerometer. It draws a supply current of 15 mA and has heading accuracy of 3.0° RMS at tilt levels ranging from 0 to $\pm$ 30° and resolution of 0.1°. It can measure heading at an output data rate of 8 Hz and is typically used in navigation and precision pointing applications. The supply voltage for the compass is 5 V unlike the other components which use 3.3 V.

Default settings were used with the compass except that we switched to magnetic instead of heading data using the `*M<cr><lf>` command.

2.1.4 Microcontroller (ARM)

The STM32F100RBT6B [9] is an ARM based 32-bit, 64 pin MCU with 8 communication interfaces, including I^2C, SPI and USART. It is suitable for a wide range of applications including GPS platforms, industrial applications, PC peripherals and more. The ARM was mounted on a STM32VLDISCOVERY development board [10] containing an in-circuit ST–Link debugger/programmer.

2.1.5 Accelerometer Development Kit

An evaluation kit [12] was used to get a feel for the data acquisition and format of the accelerometer, and to possibly get an early start on determining the decision policy which would later be applied. The evaluation kit was based around an accelerometer very similar to the one used in this project. Communication with the evaluation kit was experimented with using serial commands processed using HyperTerminal (Windows) as well as with the pySerial library [11], visualizing the data using Python. The evaluation kit also came with devoted software which was used initially to some extent.

2.1.6 Development Software

After trying a few different development tools we settled on Atollic TrueSTUDIO [22] to program the ARM. The option `-std=gnu99` was added to the gcc compiler options for more familiar C syntax.

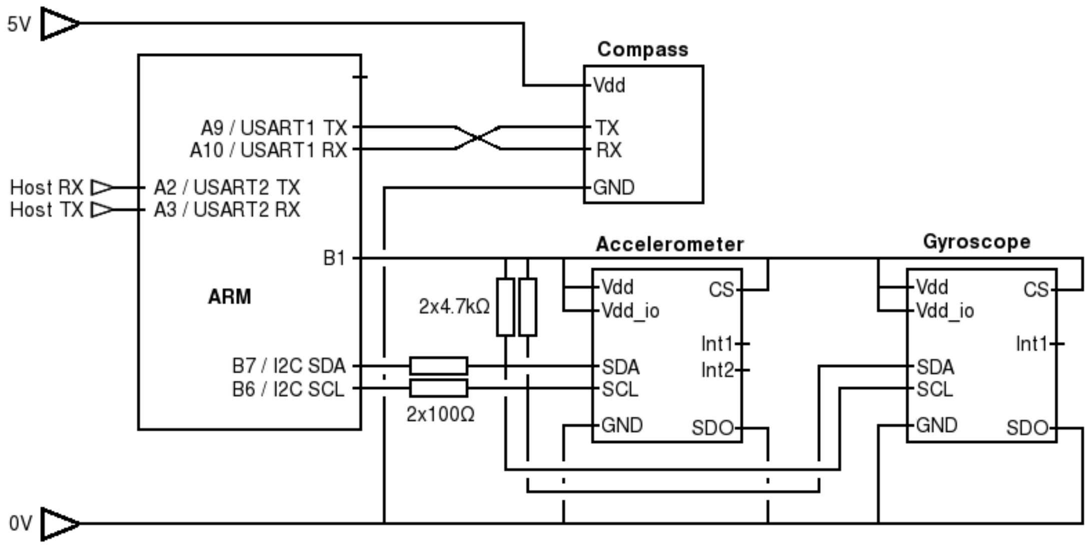

Figure 2-1
A high level circuit diagram. Only relevant pins on the ARM development board are shown and the USB connection is omitted.

2.2 Assembling Hardware & Communications

2.2.1 Preliminary Decisions

Before assembling the hardware, a decision had to be made about whether to do development on the CoreStation or create our own circuit to gather the data and forward it to an external host, as briefly mentioned in section 1.3. The CoreStation offered high level of abstraction, theoretically allowing us to communicate with the sensors in a regular operating system with various tools and programming languages at hand. The microprocessor based implementation on the other hand is closer to the desired end result of a standalone sensor circuit, but having to deal with low level hardware is often time consuming.

After experimenting with both methods the CoreStation was abandoned and focus was shifted towards the standalone microprocessor based solution. This was primarily due to software trouble on the CoreStation which lacked the necessary drivers to communicate with the sensors, and foreseeable time consuming work to fix that. Also, eventually the system had to be developed as a standalone unit and the software trouble on the CoreStation effectively eliminated its advantage.

I^2C support is required to communicate with the accelerometer and gyroscope while USART serial communications are required for the compass, and to send the combined data to an external device a separate USART channel is required. The microcontroller chosen for the project is described in section 2.1.4. The details of the I^2C protocol used for the devices are shown in their respective datasheets.

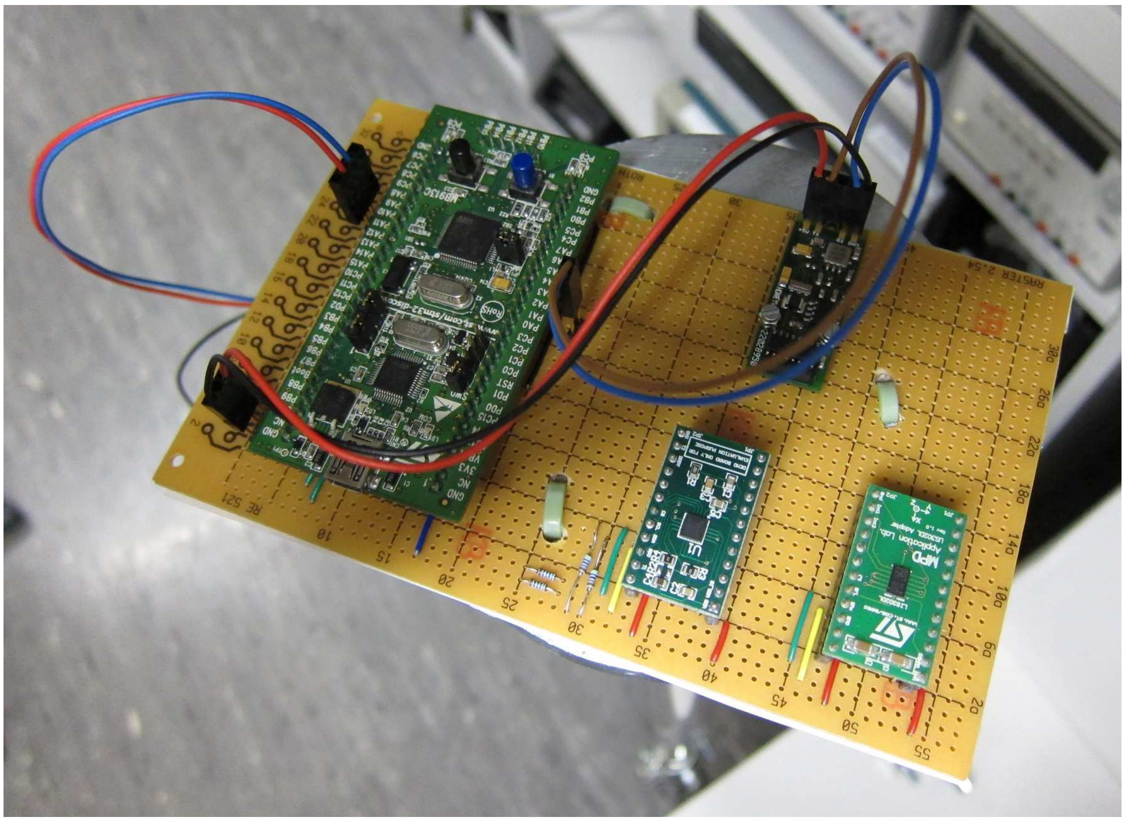

Figure 2-2
Circuit and sensors mounted on a prototyping board. From left to right we have the ARM mounted on its development board, the gyroscope (bottom middle), compass (top middle) and accelerometer (bottom right).

2.2.2 The Circuit

The circuit was assembled on a prototyping board. A high level diagram can be found in figure 2-1, excluding irrelevant outputs and components. As the accelerometer and gyroscope came on expansion boards, appropriate sockets were soldered onto the circuit board, whereas the compass was attached to the prototyping board and connected to the ARM through a small cable. Pull–up and series resistors were used for the I^2C channels following instructions in the ARM datasheet, keeping the lines high when idle. A photo of the assembled circuit can be seen in figure 2-2.

2.2.3 Reading from the Accelerometer and Gyroscope

Considerable time was spent on the I^2C communications due to scarce documentation and code examples. The biggest realization was that in order to get consistent behavior during development, the accelerometer had to be power cycled between test runs. This was due to the accelerometer often being left in a state which didn't allow establishing new communications because a previous test run had failed and improperly closed active communications. The power cycling is still done when the ARM program starts. It was left in the program code as a precaution in the (unlikely) case that the ARM restarts in the middle of I^2C communications.

After getting the accelerometer working it was straightforward to establish communications with the gyroscope, although it needed a slightly different approach due to each axis of the gyroscope having a resolution of two bytes as opposed to a single byte for the accelerometer. Since the devices were power cycled during each run of the microcontroller program, it proved most convenient to supply power to the devices directly from the ARM, as it outputs a 'high' signal of 3.3 V, which corresponds to the accelerometer and gyroscope supply voltages.

Serial communications were chosen to forward the data from the ARM to a computer for analysis. Since data was being read from several sensors, a simple protocol had to be designed to 'multiplex' the different data coming from the sensors into one stream. The programming was complicated by the way the accelerometer and gyro return their data. The sensors were set to a nominal sampling frequency of 400Hz, however outputs for the three axes are not written synchronously, often requiring more than one transmission to read a single three axes measurement.

2.2.4 Reading from the Compass

The compass can only be interfaced via USART serial communications and connecting it to the ARM was relatively straightforward, although it required a supply voltage of 5 V, thus not compatible with the outputs of the ARM. Luckily the development board, being powered via USB has 5 V outputs readily available. However, this makes controlling whether the compass is turned on or off impossible without external circuitry, which is irrelevant in the case of a prototype, but could be implemented in the final design.

The compass delivers straight ASCII data, so in order to get any meaningful numbers into the ARM program, they have to be converted from their text value into their numerical equivalents. As we are only interested in forwarding the data on to the serial line, we simply put a header byte in front of all compass bytes and forward them to the external host, being perhaps not particularly efficient but simple. The ASCII values can then be mapped to their numerical equivalents on the host side.

2.3 Sensor Data Protocol

When an axis value has been read by the ARM it is forwarded straight on to the serial line towards the destination host, even if the measurement is incomplete, containing only one or two axes. Each 'package' of data from the ARM contains a header byte, indicating to which sensor the following bytes originate from, what they represent, and the length of the sequence.

2.3.1 Implementation Details

Need for Synchronization

The data bits following a header byte can have any possible value. In order to realize what bytes are header bytes and what is data, a receiving program needs to be 'synchronized' before any useful meaning can be extracted from the the byte stream. To do this, a synchronization sequence is used that is not otherwise found in the raw data stream.

Dealing with Overflows

The time it takes to forward sensor data onto the serial line is quite long, initially causing overflows on the accelerometer and gyroscope before the main program loop started communicating with the sensors again. Interrupts and buffers are therefore utilized to avoid blocking the programs main loop during serial transmission. The interrupts came with a price, since they started disrupting the I^2C communications, presumably by the jump out of the main loop during some vital I^2C sequences. This is avoided by disabling the serial transmissions during I^2C communications.

Reducing Traffic

In order to reduce traffic on the serial line, a buffer is implemented that contains a single measurement from either the accelerometer or gyroscope that can combine up to three transmissions of different axes into one. This reduces a common case of transmitting one or two of the available data axes followed shortly by the missing axes, effectively compressing the data sent, avoiding unnecessary header bytes.

2.3.2 Header Byte

The purpose of the protocol is to identify what data comes from what sensor and what each bytes represents. Each sequence of data consists of a header byte followed by variable length content. The possible sequences with their corresponding header byte are described in table 2-1.

Header byte	Sequence type
`0b0000XXXX`	Compass sequence
`0b0100XXXX`	Accelerometer sequence
`0b1000XXXX`	Gyro sequence
`0b1100XXXX`	Synchronizing sequence

Table 2-1: Possible header bytes. `X` *corresponds to an undefined bit.*

As shown in the table, only the first four bits of a header sequence are used for identification while the last four bits are used to store additional data, depending on the type of sequence. The different sequence types will be described in detail in the following sections. In table A-1 we can see an example of how the data sensor protocol is interpreted.

2.3.3 Synchronization Sequence

The length of this sequence is equal to the longest possible sequence for compass, accelerometer or gyroscope data and consists of seven `0b11111111` bytes in a row. Since the other header bytes have a fixed value containing one or more zero bits, the synchronization sequence can be uniquely identified in the byte stream and we can be assured that the first byte containing a zero bit following seven or more 'full' bytes is a header byte and the following stream can be interpreted from there. If synchronization is lost for some reason, one simply has to wait for the next synchronization sequence and start again. The synchronization sequence is sent once every two hundred runs of the ARM program main loop, but this 'rate' is configurable in the ARM code.

2.3.4 Gyroscope Sequence

The last four bits of the gyro header byte contain information about the data that follows, as shown in figure 2-3. The fifth bit (from the left) indicates if any of the gyroscope axis had data overflow (a new value was measured and written before the existing one was read) while the last three indicate that following the header byte is data for the X, Y or Z axis. Each axis is represented by two bytes of data where the first byte is the more significant part of the value while the second is the less significant part. When joined (e.g. by left shifting the first byte by 8 bits and then "or"–ing them together) the outcome is a 16 bit two's–complement value for the corresponding axis.

1	0	0	0	F	X	Y	Z

F	Value of 1 indicates overflow.
X	Value of 1 indicates that two bytes of X data follow.
Y	Value of 1 indicates that two bytes of Y data follow.
Z	Value of 1 indicates that two bytes of Z data follow.

Figure 2-3
Header byte for the gyroscope.

As an example, if a header byte of `0b10000101` is read, we know that four bytes follow, of which the first two constitute the value of the X axis while the second two are the Z axis. Furthermore we know that no data was missed because of overflow in the gyro registers. The fifth byte following the header byte would be the first (header) byte of the next sequence.

2.3.5 Accelerometer Sequence

The accelerometer sequence, as shown in figure 2-4, is similar to the one for the gyroscope, except each axis of accelerometer is represented by one byte instead of two. The header byte is constructed in the exact same way with the last three bits indicating if X, Y or Z data follows while a `1` in the fifth bit position indicates an overflow.

0	1	0	0	F	X	Y	Z

F	Value of 1 indicates overflow.
X	Value of 1 indicates that one byte of X data follows.
Y	Value of 1 indicates that one byte of Y data follows.
Z	Value of 1 indicates that one byte of Z data follows.

Figure 2-4
Header byte for accelerometer.

2.3.6 Compass Sequence

The compass sequence is simply a header byte followed by a single byte of raw compass ASCII data. The last four bits of the compass header have no significance.

2.4 Assembling a Test Setup

To get data for different input signals for analysis and processing, the circuit was mounted on an aluminum pole. The entire circuit was powered through a USB cable from the computer. The USB cable supplies 5V and a regulator on the development board was used to get 3.3V which were needed for the accelerometer and the gyroscope. The serial data from the ARM was supplied through a serial-to-USB bridge to the computer. Software was developed to read the output from the circuit and log the values of the different sensors. The setup was then subjected to various stimulations while exploring different accelerometer settings. All in all, over one hundred signals were recorded, each of length either 20 or 30 seconds with accelerometer and gyroscope set to a sampling frequency of 400 Hz and the compass to the maximum of 8 Hz. A picture of the test setup used can be seen in figure 2-5.

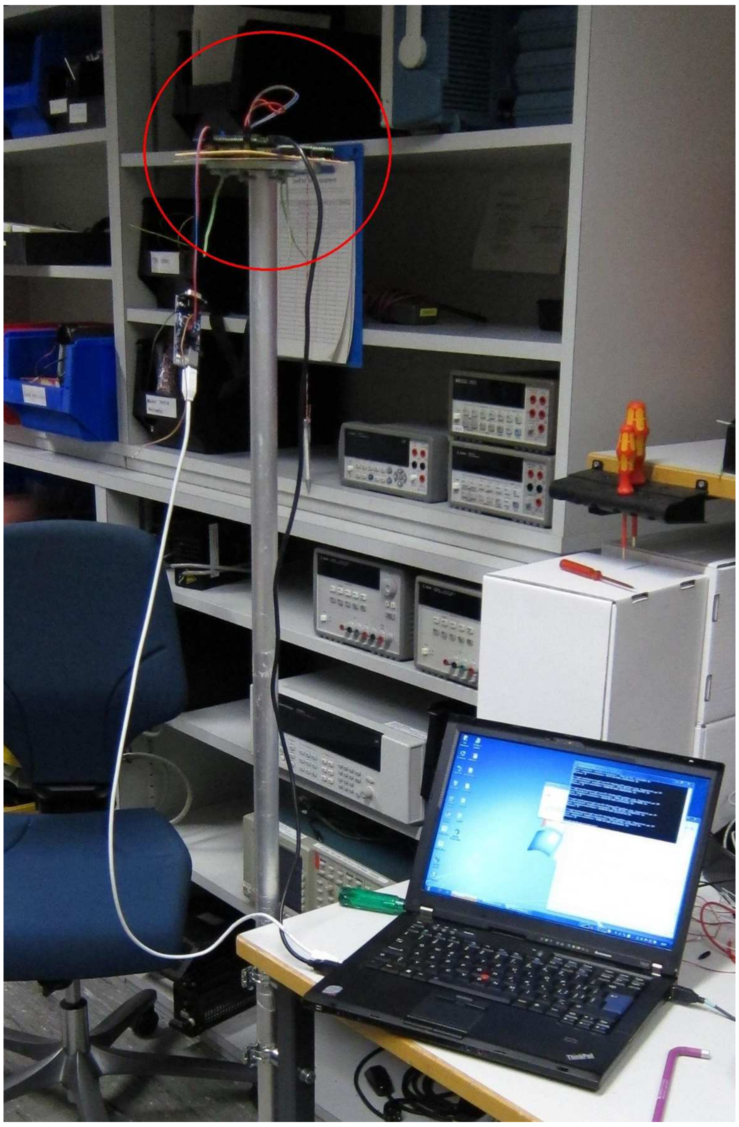

Figure 2-5
Test setup: Circuit is securely mounted to the top of the pole, it is then connected directly to a computer recording the output data from all the sensors.

3

Data Analysis and Processing

3.1 Extracting Information from the Sensor Data

3.1.1 The Sensor Data

The data received from the sensors consists of the following:

- 3-axes acceleration in digits [1]
- 3-axes angular rate in digits [1]
- 3-axes magnetic data in gauss

The accelerometer has two ranges of output, ± 2 g and ± 8 g. Depending on which range is used, different conversion parameters have to be used in order to get the correct units for the data. The accelerometer outputs one byte per axis or 8 bits, expressed in digits in the range 0 to $2^8 - 1$. According to the accelerometer datasheet [8], a sensitivity of 18 mg/digit should be expected for the lower range and 72 mg/digit for the upper. Thus for example, to convert ± 2 g data from digits to units of acceleration, the output is multiplied by 0.018 g. The accelerometer also has a high-pass filtering option implemented if needed, but eventually we disabled it, experiencing no advantage in using it for our purposes. It should be mentioned that by using the high-pass filter, the steady state value of Earth's gravitation is filtered out.

The gyroscope has three ranges of output ± 250, ± 500 and ± 1000 dps. Analogously to the accelerometer case, the output values were converted into degrees per second through sensitivity values from the datasheet, 8.75, 17.5 and 70 mdps/digit, respectively. Thus for example, for ± 250 dps, the output is multiplied by 0.00875.

No specific ranges were of concern in the compass output. The magnetic values were read from the compass, instead of the converted heading itself. Of course having a compass should directly give us the orientation, however we had some problems with the compass, where it seemed to occasionally give false readings and those

[1]We refer to digits as unitless numbers that need to be multiplied by a sensitivity parameter to gain a physical unit.

readings would change dramatically with the slightest change in position of the compass itself. This was discovered in the lab and it might have been caused by strong electromagnetic fields nearby, or possibly interference or length issues of the I^2C cables used in the circuit. Instead of using the formatted output from the compass, we used the raw magnetic data knowing that the heading can be post calculated from these values using inclination data from the accelerometer. In the datasheet, the sensitivity of the magnetic readings was expressed in gauss, expressing units of the magnetic field B, whereas the tesla unit (T) is normally used to express units of magnetic field B or the flux magnetic density, where B and H are closely related. It should be noted that regardless of whether G or T are used as units, the heading calculation should not be affected, since it only focuses on proportional relations between the 3-axes values. Figure 3-1 shows an example signal reading from the sensors outputs. The signal was generated by tapping the test pole once from the side.

3.1.2 Calibration & Rotation Adjustment

Calibrating the Gyroscope

Due to temperature dependence and variance in chip design, the output of the gyroscope is assumed to be shifted by an additive offset. An average of a predefined number of samples at the beginning of each sample vector is used to calculate the sensor offset, regardless of its inclination. The gyroscope is expected to be in a no movement state for the predefined number of samples used to calibrate the data.

Calibrating the Accelerometer

During no movement, the accelerometer constantly reads the gravitational vector, whereas the gyroscope shows zero readings on all axes, given that the offset has been accounted for. In the case of the accelerometer, the sensor would have to be carefully mounted in a horizontal plane in order to achieve such a factory offset calibration. Instead of perfectly aligning the accelerometer for calibration, a reference offset is calculated analogous to the gyroscope case, in order to show a zero reading on the x and y-axes and g on the z-axis. In that case, acceleration measured in opposite directions on the x and y-axes will have different signs, which is important when integrating the acceleration to estimate velocity and position.

Instead of only calibrating based on the first few samples of data, a dynamic calibration was also implemented as a second option, which recalibrates the accelerometer when no external forces have been detected over a given amount of time. Thus, if a rotational matrix would be applied to the data, this dynamic calibration would reduce the effects due to noise in the rotational matrix.

Rotational Matrices

A rotational matrix can be used to relate the orientation of two coordinate systems and we could use such a matrix to map our sensor data to Earth's coordinates, where true north could for example serve as the x-axis with the gravitational vector parallel to the z-axis. We tried implementing a rotational matrix for the accelerometer

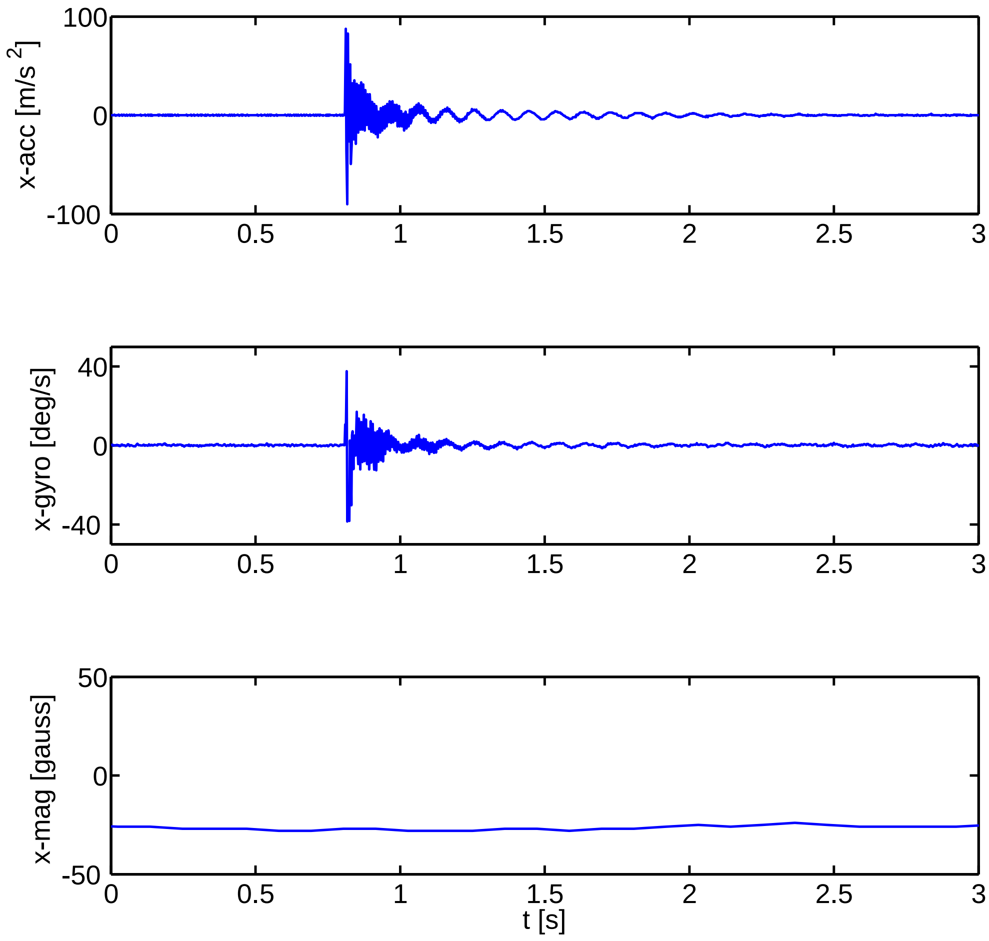

Figure 3-1
Example of a sample signal recorded using the test setup. The readings shown in the figure (only showing x-axis) were generated by tapping the test pole once.

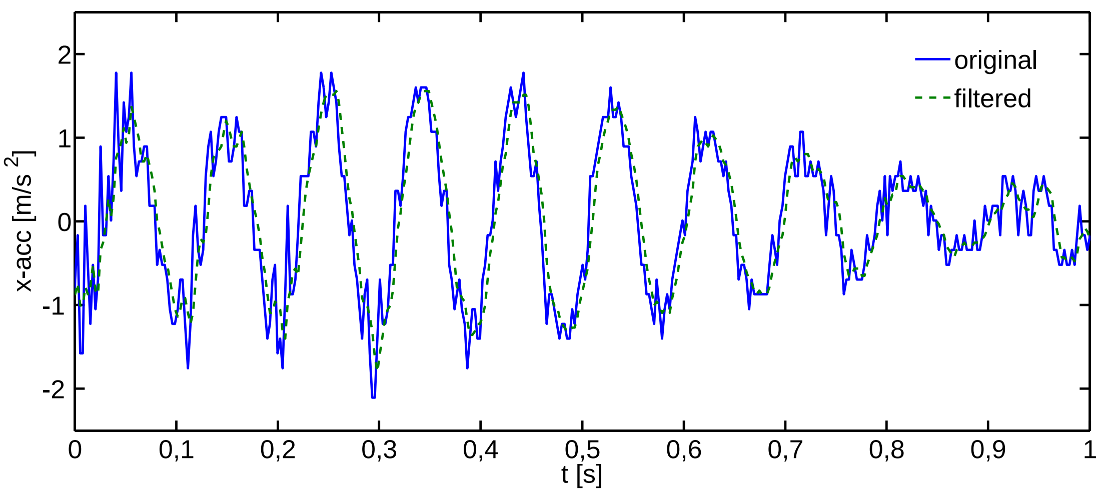

Figure 3-2
An example showing the results of filtering data through a moving average filter ($a_0 = 1, b_0 = 1, b_1 = 1, b_2 = 1, b_3 = 1$).

data, but since our compass data was not trustworthy, the matrix couldn't be effectively used. Since a rotational matrix is not applied, our current processing of the data only applies to low angles. Given more reliable compass data, a rotational matrix should be straightforward to implement [14]. A different approach is used to account for rotations in gyroscope data which could be implemented alongside the aforementioned rotational matrix [20]. The discussion in the following chapters applies equally to data that is corrected using rotational matrices and data that uses no rotational matrices, but is already aligned orthogonal to the gravitational vector and subjected to little or no rotations during measurements.

3.1.3 Filtering

A moving average and a second order Butterworth filter were implemented for data filtering[2]. The moving average is a simple low pass filter, in which the current output of the filter is a weighted average of a predefined number of recent inputs and outputs, as shown in equation 3.1. Thus the filter can be customized by changing the value of the weights $\{a_0, a_1, ..., a_m\}$ and $\{b_0, b_1, ..., b_m\}$. An example of application of a moving average filter is shown in figure 3-2.

$$a_0 y[n] + \ldots + a_m y[n-m] = b_0 x[n] + \ldots + b_m x[n-m] \tag{3.1}$$

The Butterworth low pass filter has an infinite impulse response and can be implemented in MATLAB where a specific cut-off frequency is defined, giving a slightly different approach to the filtering customization that in the case of a moving average. However, the Butterworth filter might be hard to implement on a host such as TinyNode etc., thus it might just serve as a tool for data analysis where the cut-off

[2]A complementary filter was also implemented, using the gyroscope measurements to give a better estimate of the accelerometer reading based on weights. That filter will not be discussed further, since it proved to have little or no advantage over the actual accelerometer reading.

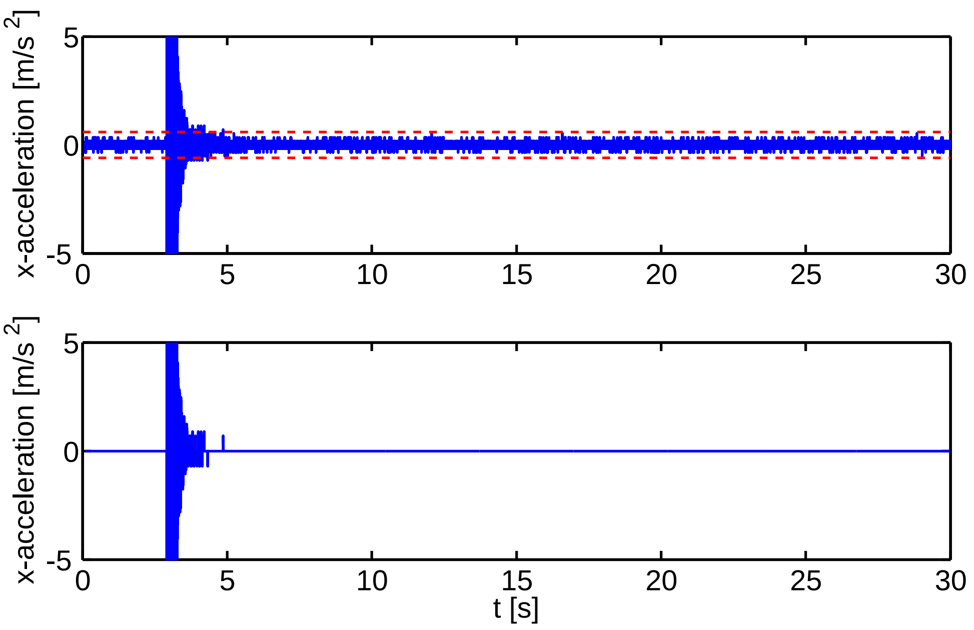

Figure 3-3
Example showing a discrimination window being applied to data. The discrimination window and the raw data are shown in the upper plot and the result is shown in the lower.

frequency can be easily controlled. It should be noted that low pass filtering can lead to poor transient response and lag.

3.1.4 Position

Calculating position from accelerometer readings can be done using double integration over time, going from acceleration to position. However, noise can be very difficult to deal with in that context, since any error in the acceleration reading grows exponentially through integration, resulting in shaky position estimates. Measures can be taken to deal with this issue, such as using filters, discrimination windows and velocity calibrations, as discussed below. It should be noted that for movement detection the actual position of the node is not of much significance, whereas the relative change in position over time is what we are interested in.

Calculating position from accelerometer data:

- Read and calibrate accelerometer data
- Apply filter (MA, BW etc.)
- Apply discrimination window
- Integrate to velocity
- Correct velocity
- Integrate to position

To account for mechanical noise in the accelerometer readings, a discrimination window was implemented. The idea of a discrimination window is to nullify all values

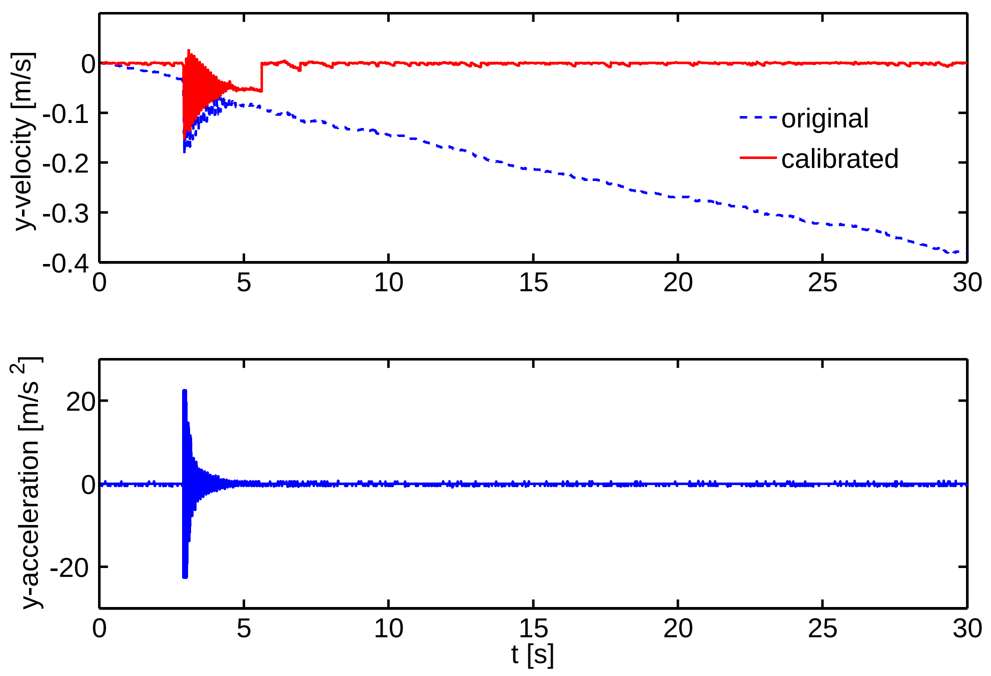

Figure 3-4
Example showing velocity correction after the first integration, forcing the velocity to zero after 25 consecutive samples of zero acceleration.

that are within a predefined range of the zero-offset as shown in figure 3-3, to prevent false velocity being summed up due to noise at steady state.

After the first integration, the velocity values are ”corrected” using a heuristic approach such that they fit the actual behavior expected of the GPS node [21]. Since it will be sitting on the top end of a pole that will be fixed to the ground, steady velocity over more than one or two handfuls of samples is not expected to be seen. Non-zero velocity values should only be expected when non-zero acceleration is observed. Thus, whenever a predefined number of consecutive zero acceleration samples have been observed, the velocity is forced to zero. This prevents a significant amount of errors from building up through the double integration. An example of velocity correction is shown in figure 3-4.

When integrating over time using a discrete set of values sampled from a continuous event, the estimate will have some error. The integration can be broken down to a simple sum with as many terms as the amount of samples used. In our case the sampling frequency determines the time interval between samples. A number of well known methods exist for calculating such integrals, such as using left sum, right sum, the midpoint rule etc. Which method to use for optimal results depends on the type of samples or function being integrated. We used trapezoidal integration for all integrations in this project. Trapezoidal integration approximates the area defined over a time interval as a trapezoid, with the two top corners at the function values at the two end points of the time interval.

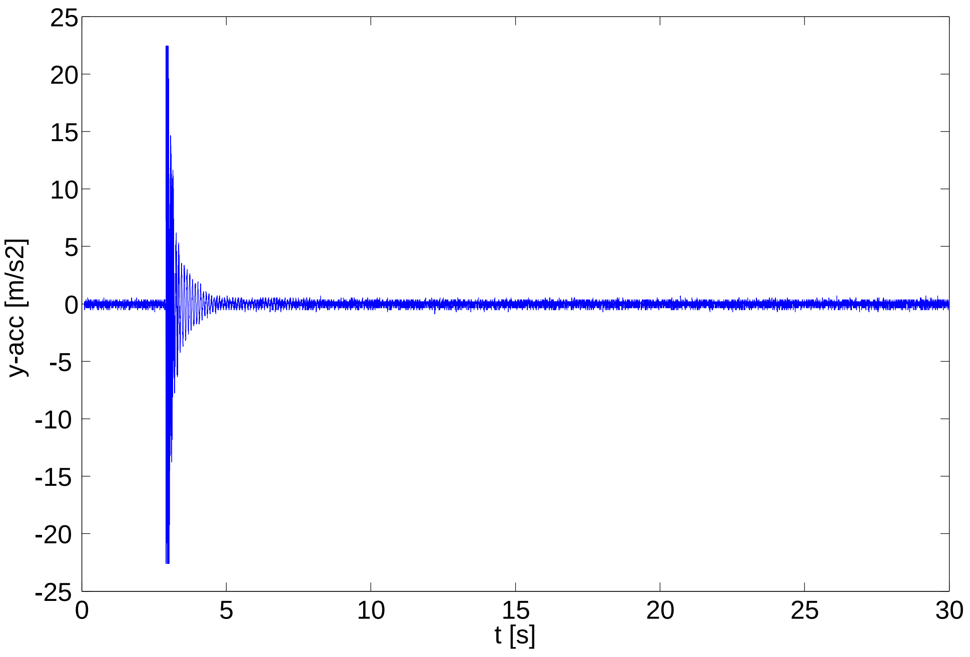

Figure 3-5
The acceleration signal used to generate the position estimate in figure 3-6.

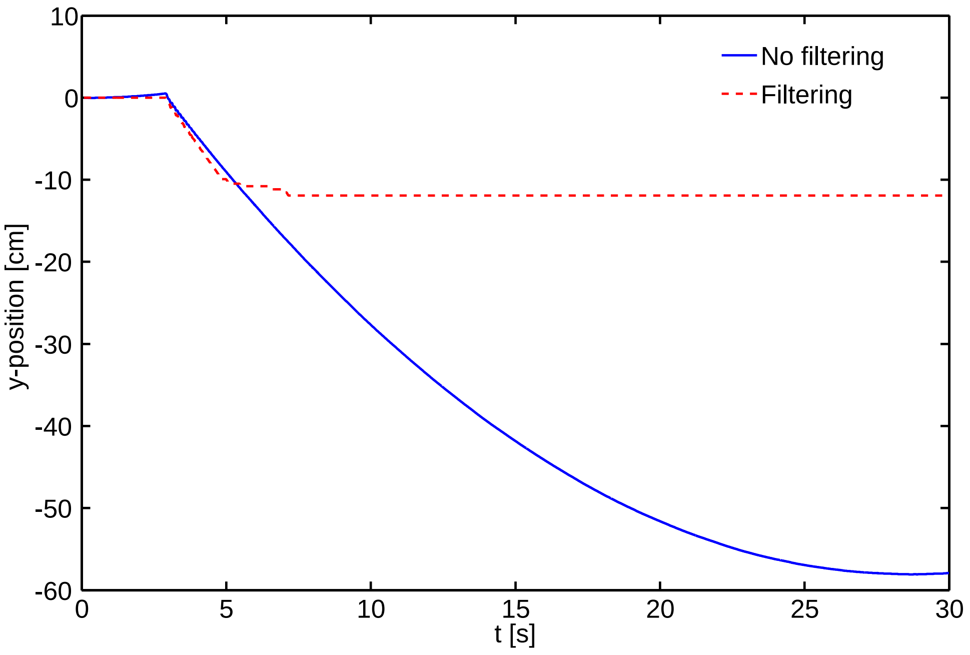

Figure 3-6
Example showing the effects of using a moving average filter, discrimination window and velocity calibration to reduce errors in position. The error is reduced by roughly 80% for this particular signal after estimating position for 30 seconds.

Despite the filtering, discrimination windows and velocity correction, the noise and errors introduced through the double integration lead to too much inaccuracy for the estimate to be trustworthy for the small scale movement considered in this project (mm to cm). Figures 3-5 and 3-6 show the effects of processing the data using filtering, a discrimination window and velocity correction. The acceleration signal was created by tapping the pole from our test setup once with a screwdriver. The pole was fixed and after 30 seconds it had returned to its starting position, as expected. In this specific case, the error was reduced by roughly 80% after 30 seconds of estimation using the filtering. However, at this time the position estimate deviates roughly 11 cm from the actual value, which should be zero. In case of further work, more accuracy could be achieved by optimizing the parameters involved in the processing and by implementing some new filters, such as a Kalman filter and complementary filters. It should be noted that significant relative changes in position could still be detected, given that they reach a given threshold of significance, reducing the likelihood of noise or errors to have induced the change.

3.1.5 Inclination

Inclination from a horizontal position can be calculated from both accelerometer and gyroscope data.

Calculating inclination from accelerometer data

- Read acceleration data
- Calculate pitch and roll angles using trigonometry

Calculation of inclination from the accelerometer data is very sensitive to noise and external forces. When calculating the inclination, the acceleration due to gravity (g) is assumed to be the only force acting on the node. In that situation, acceleration measured on the x and y-axes can be seen as the x and y components of the gravitational vector g [5]. Thus the inclination angles, also known as pitch (θ) and roll (ϕ), can be calculated using elementary trigonometry, as shown in equations 3.2 and 3.3. In our case, wind and other noise contributing factors can make such estimations hard. If the node senses acceleration on the x or y-axis, which would simply shift the node in the respective directions, false inclination would be returned. Some measures can be taken, such as only calculating inclination from the accelerometer when the 2-norm of the three acceleration values is close to 1g, e.g. 1g ± 0.07g.

$$\theta = \arcsin\left(\frac{A_x}{\mathrm{g}}\right) \tag{3.2}$$

$$\phi = \arcsin\left(\frac{A_y}{\mathrm{g}}\right) \tag{3.3}$$

The co-domain of the $\arcsin$ function is $[-90, 90]^\circ$ and whenever the absolute values of $\frac{A_x}{\mathrm{g}}$ or $\frac{A_y}{\mathrm{g}}$ become larger than 1, the angle is forced to a value of $\pm 90^\circ$ depending on the case, which should be fine, given the assumption that the node will never be inclined to angles of such extent, unless something has gone terribly wrong. Note that the yaw angle (heading) cannot be calculated from the accelerometer since its rotational axis is parallel to the gravitational vector g. However, it can be calculated using a gyroscope as shown in the next section.

Calculating inclination from gyroscope data

- Read and calibrate gyroscope data
- Integrate x- and y-axis angular rates over time

Calculating inclination from gyroscope readings is done by integrating the x and y-axis values over time to achieve the pitch and roll angles, respectively. It is assumed that the gyroscope data has been corrected using a rotational matrix, as briefly mentioned in section 3.1.2. The gyroscope is less prone to mechanical noise than the accelerometer, due to the nature of the quantity being measured and thus no noise filtering is applied to the data. The inclination estimates proved to be quite accurate for an integration period of 30 seconds, but drift is expected for longer time periods. Due to the no external forces criterion of the acceleration inclination estimate, a policy could be implemented such as only using the accelerometer values within that criterion and otherwise use the gyroscope estimate. The accelerometer estimate could also be used to regulate the drift in the gyroscope estimate periodically through comparison of estimates. Also, the integration imposes time dependency for the gyroscope estimate and should the gyroscope for some reason not be able to read continuously over time, an estimate from the accelerometer can be used for an initial value for the integration, given that the accelerometer has had successful inclination calculations recently.

3.1.6 Heading

The heading can be calculated from both the gyroscope and the magnetic data from the compass.

Calculating heading from compass data

- Read magnetic data
- Calculate inclination for tilt compensation
- Apply rotational equations to get horizontal heading

The rotational equations [4] used are shown in equations 3.4 to 3.6. These equations take in the pitch and roll angles θ and ϕ, which we can calculate from the accelerometer data, and the compass magnetic readings b_x, b_y and b_z. The output H_{comp} is the corresponding heading based on the magnetic data.

$$X_H = b_x \cos(\phi) + b_y \sin(\theta)\sin(\phi) - b_z cos(\theta) sin(\phi) \tag{3.4}$$

$$Y_H = b_y \cos(\theta) + b_z \sin(\theta) \tag{3.5}$$

$$\mathrm{H}_{comp} = \arctan\left(\frac{Y_H}{X_H}\right) \tag{3.6}$$

Calculating heading from gyroscope data

The heading can also be calculated from gyroscope data by integrating the angular rate around the z-axis over time. The angular rate obeys the right hand rule.

- Read and calibrate gyroscope data
- Integrate z-axis angular rate over time

3.2 What to Detect and How

As mentioned in section 3.1, we can extract spatial information such as velocity, position, inclination and heading from our sensor data. That information could serve as auxiliary information used to support the GPS node and provide a basis for movement detection. In the above sections we focused on how to extract this spatial information successfully, but in this section we will estimate the applicability of some estimation strategies and the viability of the raw and extracted data for such detection.

As will be pointed out in this section, detection thresholds have to be evaluated for mechanical and environmental noise such as wind and other natural phenomena using real data, preferably on-site. These thresholds could vary between nodes, depending on location and weather.

3.2.1 Raw Acceleration Data

We can use thresholds to detect changes in our raw acceleration data by comparing values from two separated moving average windows. Due to the abrupt nature of the acceleration data, we want to keep the windows short. The window length and the threshold values need to be carefully determined based on real on-site data and considering noise levels. The following is an example code for such detection.

```
# values contains last X acceleration samples
# oldmean contains old average
# newmean contains new average
# newvalue is the latest acceleration triplet (x,y,z)
# LENGTH is the window length (at most X / 2)

oldmean -= values[last] / LENGTH
pop(values) # remove the oldest sample
oldmean += values[last - LENGTH] / LENGTH

newmean -= values[first - LENGTH] / LENGTH
shift(values, newvalue) # add the new sample
newmean += values[first] / LENGTH

if abs(newmean - oldmean) > threshold
  trigger interrupt
else
  do nothing
```

Standard deviation can also be used to get an idea of what's happening, that is, telling us whether the system is in a relaxed state giving the current mean or whether some actual external dynamics are affecting the system, producing shaking or oscillations. The following is an example code for such detection.

```
# values contains last X acceleration samples

if standard_deviation(values) > threshold
  trigger interrupt
else
  do nothing
```

Another very simple implementation of a detection mechanism would be to focus on the euclidean norm of the axes values and see if it varies considerably from earth's gravitational vector g. The following is an example code for such detection.

```
# value contains last acceleration sample triplet (x,y,z)
# (or an average over recent samples)
# g is the value of earth's gravitational vector

if abs(norm(value) - g) > threshold
  trigger interrupt
else
  do nothing
```

3.2.2 Raw Gyroscope Data

The gyroscope has a zero 'relaxed state reading' unlike the accelerometer which always has g as a reference. Raw data from the gyroscope could be used to detect strong shaking or twisting, that is, fluctuations of angular rate that pass a certain threshold. *Note that the gyroscope is blind towards shifts in position without changes in inclination or heading.* The following is an example code for such detection.

```
# values contains last LENGTH gyroscope samples
# mean contains old average
# newvalue contains the latest gyroscope sample triplet (x,y,z)

mean -= values[last] / LENGTH
pop(values) # remove the oldest sample

shift(values, newvalue) # add the latest sample
mean += newvalue / LENGTH

if abs(mean) > threshold
  trigger interrupt
else
  do nothing
```

In this example, `newvalue` can contain an axis reading or a norm over the three axes.

3.2.3 Raw Magnetic Data

Changes in magnetic values can be used for movement detection through the use of thresholds. If the node tilts, changes in magnetic data will occur and if they exceed a certain threshold, movement can be signaled. Because the magnetic data is non–linear with respect to inclination, the thresholds may vary for different values of inclination. Because of the different strength in the x,y and z magnetic fields at a given point on Earth, different thresholds have to be implemented for different axes of the compass. In order to remedy this, the respective thresholds could be scaled by the magnitudes of its respective axis reading. A foreseeable problem with this approach is in the case of zero magnitude, for which the threshold will become zero. The following is an example code for such detection.

```
# values contains last X magnetic data samples
# oldmean contains old average
# newmean contains new average
# newvalue contains the latest gyroscope sample triplet (x,y,z)
# LENGTH is the window length (at most X / 2)

oldmean −= values[last] / LENGTH
pop(values) # remove the oldest sample
oldmean += values[last − LENGTH] / LENGTH

newmean −= values[first − LENGTH] / LENGTH
shift(values, newvalue) # add the latest sample
newmean += values[first] / LENGTH

if abs(newmean − oldmean) > threshold ∗ abs(oldmean)
  trigger interrupt
else
  do nothing
```

In this example, `values` contains readings from one of the three axes. Here the threshold is multiplied by the magnitude of the old mean to account for different scales of readings for the individual axes.

3.2.4 Velocity from the Accelerometer

The velocity data is the result of running the accelerometer data through an integrator, which serves as a very simple low pass filter. As mentioned above, discrete integration leads to accumulation of error and some heuristics and filtering are needed to get a somewhat realistic estimate. Even though acceleration has not been reaching thresholds over the given time intervals, it might have been non-zero for some time, thus accumulating velocity. If the velocity goes above a certain threshold, we should be able to detect actual movement. The following is an example code for such detection.

```
# velocity contains the integrated acceleration over last X samples
# values contains the sample data vector (length X)
# newvalue contains latest velocity triplet (x,y,z)

# subtract the old trapezoidal value
velocity −= (values[last] + values[last−1]) / 2
pop(values) # remove the oldest value

# add the latest value
shift(values, newvalue) # add the latest value
velocity += (values[first] + values[first−1]) / 2

if abs(velocity) > threshold:
  trigger interrupt
else
  do nothing
```

In this method, `velocity` should be pre-processed using filters and heuristics as described in section 3.1.

3.2.5 Position from the Accelerometer

Similar to the case of velocity, the position is the result of running velocity data through an integrator. Position would be our ultimate tool for movement detection, however due to the inaccuracy introduced through the double integration we cannot trust our position estimate. However, significant or 'obvious' changes could be detected using the position estimate through thresholds. The thresholds would be very conservative, preventing false positives as much as possible, and as mentioned above, the value of the threshold would have to be optimized based on real data and evaluations. We would have to evaluate the extent of the error and make sure that the threshold exceeds any such erroneous values however it may ultimately be implemented. The following is an example code for such detection.

```
# position contains the integrated velocity over last X samples
# values contains the sample data vector (length X)
# newvalue contains latest position triplet (x,y,z)

# subtract the old trapezoidal value
position -= (values[last] + values[last-1]) / 2
pop(values) # remove the oldest value

# add the latest value
shift(values, newvalue) # add the latest value
position += (values[first] + values[first-1]) / 2

if abs(position) > threshold:
  trigger interrupt
else
  do nothing
```

3.2.6 Inclination from the Accelerometer

We can detect changes in inclination by comparing values at two different points in time and see if they exceed a certain threshold. Due to the noise introduced through the accelerometer data, the estimate is not very accurate. To reduce error we can take average values over time periods and estimate from those. As mentioned in section 3.1.5, the inclination values become less viable as more external force is applied to the node.

Without calculating inclination, the raw accelerometer output with some filtering can be used to indirectly estimate changes in inclination. The difference between using the raw data and the inclination values is simply the extra mapping into degrees through the arcsin function, used in the inclination calculation. The following is an example code for such detection.

```
# values contains last X inclination samples
# oldmean contains old average
# newmean contains new average
# newvalue contains latest inclination value
# LENGTH is the window length
```

```
oldmean -= values[last] / LENGTH
pop(values) # remove the oldest value
oldmean += values[last - LENGTH] / LENGTH

newmean -= values[first - LENGTH] / LENGTH
shift(values, newvalue) # add the latest value
newmean += values[first] / LENGTH

if abs(newmean - oldmean) > threshold
  trigger interrupt
else
  do nothing
```

3.2.7 Inclination / Heading from the Gyroscope

The inclination value from the gyro is not affected by external forces and thus more robust than the accelerometer value, since it is derived by integrating the respective measured angular rate which is quite accurately measured. The inclination values from the gyro are relative, that is, if the node starts in an inclined position, the angular reading will give the angle from that specific position as time goes on. Conversely, the accelerometer can pinpoint its absolute inclination knowing earth's gravitational vector g. We can detect movement by comparing inclination values between two time points separated by a predefined interval. The following is an example code for such detection.

```
# movement contains the integrated rate over last X samples
# values contains the sample data vector (length X)
# newvalue contains latest inclination from integration

# subtract the old trapezoidal value
movement -= (values[last] + values[last-1]) / 2
pop(values) # remove the oldest value

# add the latest value
shift(values, newvalue) # add the latest value
movement += (values[first] + values[first-1]) / 2

if abs(movement) > threshold:
  trigger interrupt
else
  do nothing
```

3.2.8 Heading from Compass (and Accelerometer)

The heading values are extracted from the magnetic data, as mentioned in section 3.1.6. They represent the heading with reference to the magnetic north pole (as opposed to true north), whereas the gyroscope values are not absolute. The accelerometer values are used to calculate inclination which is needed for the rotational equations used to calculate the heading from the magnetic data. By comparing two heading values separated by a predefined time interval, we can detect whether the node has shifted its direction or not. As before, we base the detection mechanism on thresholds. An example of such detection is by using two running average windows separated by a predefined time interval, as shown below:

```
# values contains last X heading samples
# oldmean contains old average
# newmean contains new average
# newvalue contains latest heading sample
# LENGTH is the window length

oldmean -= values[last] / LENGTH
pop(values) # remove the oldest value
oldmean += values[last - LENGTH] / LENGTH

newmean -= values[first - LENGTH] / LENGTH
shift(values, newvalue) # add the latest value
newmean += values[first] / LENGTH

if abs(newmean - oldmean) > threshold
  trigger interrupt
else
  do nothing
```

3.3 Further Comments

The methods in section 3.2 could be combined using weights depending on the situation, e.g. giving more weight to the inclination value from the gyroscope than the one from the accelerometer when significant external forces are affecting the node. Also, a master detection signal could be implemented, such that its activation is based on a weighted average of detection signals from the individual movement detection mechanisms involved. The methods described above are only chosen examples of what can be done to detect movement of the device. Countless hours can be spent on fine tuning and combining these types of algorithms to achieve optimal movement detection for such a setup. A summary of the data types and the relevant detection mechanism is shown in table 3-1.

Data	**Detection examples**	**Pros**	**Cons**
Raw acceleration data	Thresholds / two moving averages, standard deviation, using norm	Time independent, accurate data	Needs on-site evaluation
Raw gyroscope data	Thresholds / integration window, (using norm)	Time independent, accurate data	Needs on-site evaluation, blind towards non-angular movement
Raw magnetic data	Thresholds / two moving averages	Time independent, accurate data	Needs on-site evaluation
Velocity from accelerometer	Thresholds / integration window, (using norm)	Detects accumulated acceleration, used for position	Integration, noisy & inaccurate, error accumulates
Position from accelerometer	Thresholds / integration window, (using norm)	Only method that gives position, could help GPS	Integration, noisy & inaccurate, error accumulates
Inclination from accelerometer	Thresholds / two moving averages	Time independent, used for tilt compensation	Sensitive to external forces, noisy
Inclination / heading from gyroscope	Thresholds / integration window	Accurate, used for tilt compensation	Integration, drift, short term
Heading from compass (and accelerometer)	Thresholds / two moving averages	Accurate	Blind to many common types of movement

Table 3-1
A summary of the data types and suggested detection mechanism.

4 Summary & Outlook

A sensor circuit was created that consists of a microcontroller, an accelerometer, a gyroscope and a compass. Communications were established between the sensors and the microcontroller and a communication protocol was designed for communication between the microcontroller and an external host. A test setup was built in the lab to test the circuit and process some basic signals. Additional spatial information was extracted from the sensor data such as velocity, position, inclination and heading using filters and other processing tools. Several movement detection mechanisms were studied and their applicability considered. The position estimate proved to be too inaccurate for the small scale movement we are interested in, but it might still be of some use in detecting significant events. The movement detection mechanisms considered were all based on a certain data element reaching a given threshold or standard deviation. Parameters such as these thresholds have to be evaluated using real data, preferably on-site, for further optimization of these detection mechanisms.

Overall, the concept of supporting the GPS nodes using an additional sensor circuit seems plausible to an extent, although on-site optimization is needed. In case of future work, the spatial properties could be extracted more efficiently from the sensor data by using sensor fusion algorithms and advanced filtering.

A

Sensor Data Protocol Example

Table A-1 shows an example of how the data sensor protocol output is interpreted.

Table A-1: Example of a byte sequence reading

Byte	Action
`10111111`	Synchronization sequence not found, ignoring.
`10000111`	Synchronization sequence not found, ignoring.
`11111111`	Part of synchronization sequence?
`11111111`	Part of synchronization sequence?
`11111111`	Part of synchronization sequence?
`11111111`	Part of synchronization sequence?
`11111111`	Part of synchronization sequence?
`11111111`	Part of synchronization sequence?
`11111111`	Seven ”full” bytes in a row, we are now synchronized.
`00000000`	Header byte for compass.
`00100000`	Compass data byte.
`01001010`	Header byte for accelerometer indicating that an overflow occurred before this reading and that a value for Y axis follows.
`01110010`	Accelerometer Y axis value.
`10000011`	Header byte for gyroscope indicating no overflow and that bytes for Y and Z axis follow.
`00101100`	More significant byte of Y axis.
`00010000`	Less significant byte of Y axis.
`01110100`	More significant byte of Z axis.
`00010011`	Less significant byte of Z axis.
`00000000`	Compass header byte.
`00111100`	Compass data byte.
`01000101`	Header byte for accelerometer indicating no overflow and that a value for X and Z axis follows.
`00010010`	Accelerometer X axis value.
`00110000`	Accelerometer Z axis value.

B

Example of an External Client Code

```
import time
import serial

s = serial.Serial('/dev/ttyUSB0', baudrate=115200, timeout=1)
s.flushInput()

join_bytes = lambda x: (x[0] << 8) | x[1]
to_bin_str = lambda n: n > 0 and to_bin_str(n >> 1) + str(n & 1) or ''
byte_str = lambda n: ('%8s' % to_bin_str(n)).replace(' ', '0')

AXIS = (('x', 1), ('y', 2), ('z', 4))
acc_overrun = 0
gyro_overrun = 0
acc_data = 0
gyro_data = 0
resets = 0
acc_res = dict([(x, []) for x, y in AXIS])
gyro_res = dict([(x, []) for x, y in AXIS])
start_found = False
compass = []

def from_twos_compliment(bytes, value):
    positive = value >> (bytes*8-1) == 0
    return value if positive else value - 2**(8*bytes)

class DataException(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return self.value

    def __repr__(self):
        return "<DataException: value=%s>" % self.value


class StartSignal(object):
    max_bytes = 7

    @classmethod
    def process(cls, data):
        global start_found
        if len(data) < cls.max_bytes:
            raise IndexError("Data array not long enough")
        elif all(map(lambda x: x == 0xff, data)):
            start_found = True
            return cls.max_bytes
        else:
            raise DataException("Not start signal")

class CompassSignal(object):
    max_bytes = 2

    @classmethod
    def process(cls, data):
        global compass

        if not (data[0] & 0b11000000) == 0b00000000:
            raise DataException("Not Compass signal")

        print "Compass: %s" % data[1]
        compass.append(data[1])

        return 2

class AccelerometerSignal(object):
    max_bytes = 4

    @classmethod
    def process(cls, data):
        global acc_overrun, acc_data, AXIS, acc_res

        if not (data[0] & 0b11000000) == 0b01000000:
            raise DataException("Not Accelerometer signal")

        out = dict([(x, None) for x, y in AXIS])
        pos = 1
        for i, j in AXIS:
            if (data[0] & j) == j:
                out[i] = data[pos]
                pos += 1

        acc_overrun += 1 if ((data[0] & 0b00001000) > 0) else 0
        acc_data += 1 if ((data[0] & 0b00000111) > 0) else 0

        for key, value in out.items():
            if value is not None:
                acc_res[key].append(value)

        print "Accelerometer: %8s %5s %5s %5s" %
            (byte_str(data[0]), out['x'], out['y'], out['z'])

        return pos

class GyroSignal(object):
    max_bytes = 7

    @classmethod
    def process(cls, data):
        global gyro_overrun, gyro_data, gyro_res, AXIS

        if not (data[0] & 0b11000000) == 0b10000000:
            raise DataException("Not Gyroscope signal")

        out = dict([(x, None) for x, y in AXIS])
        pos = 1
        for i, j in AXIS:
                if (data[0] & j) == j:
                    out[i] = from_twos_compliment(
                        2,
                        join_bytes(data[pos:pos+2])
                    )

                    pos += 2

        gyro_overrun += 1 if ((data[0] & 0b00001000) > 0) else 0
        gyro_data += 1 if ((data[0] & 0b00000111) > 0) else 0


        for key, value in out.items():
            if value is not None:
                gyro_res[key].append(value)

        print "Gyroscope: %8s %6s %6s %6s" % (
            byte_str(data[0]), out['x'], out['y'], out['z']
        )

        return pos

bytes = list()

options = [StartSignal, ]
while 1:
    if time.clock() >= 1:
        break

    if s.inWaiting() < 1:
        continue

    bytes.append(ord(s.read(1)))
    option = None
    missing_bytes = False
    for i in options[:]:
        try:
            used_bytes = i.process(bytes[0:i.max_bytes])
            bytes = bytes[used_bytes:]
            if i is StartSignal:
                options = [
                    StartSignal,
                    AccelerometerSignal,
                    GyroSignal,
                    CompassSignal
                ]
            option = i
            break
        except IndexError, e:
            missing_bytes = True
        except DataException, e:
            pass

    if option == None and not missing_bytes:
        if not start_found:
            print "Waiting for start signal.."
        else:
            print "Out of sync, resetting options to " \
                "start signal alone and discarding one byte."
            print bytes,
            print map(byte_str, bytes)
            options = [StartSignal, ]
            resets += 1
        bytes.pop(0)

print "Unprocessed bytes: %s" % bytes
print "Accelerometer samples/overruns: %s/%s" % (acc_data, acc_overrun)
print "Gyro samples/overruns: %s/%s" % (gyro_data, gyro_overrun)
print "Resets: %s" % resets
print "Compass data:"
print "".join(map(chr, compass)).strip()
```

Bibliography

[1] Grant Baldwin, Robert Mahony, Jochen Trumpf, Tarek Hamel, and Thibault Cheviron. Complementary Filter Design on the Special Euclidean Group SE(3). In *European Control Conference*, 2007.

[2] Bernhard Buchli, Felix Sutton, and Jan Beutel. GPS-equipped Wireless Sensor Network Node for High-Accuracy Positioning Applications. In *Proc. 9th European Conference on Wireless Sensor Networks (EWSN 2012)*, 2012. Trento, Italy.

[3] Bernhard Buchli, Mustafa Yuecel, Roman Lim, Tonio Gsell, and Jan Beutel. "Demo Abstract: Feature-Rich Platform for WSN Design Space Exploration". `http://www.tik.ee.ethz.ch/ bbuchli/pubs/BYLGB2011.pdf`.

[4] Michael J Caruso. Applications of Magnetoresistive Sensors in Navigation Systems. Technical report, Honeywell Inc., 1997.

[5] Michelle Clifford and Leticia Gomez. Measuring Tilt with Low-g Accelerometers. Technical report, Freescale Semiconductor Inc., 2005.

[6] HMR3400 (Compass) Datasheet. `http://www51.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Space-documents/Digital_Compass_Solution_HMR3400.pdf`.

[7] L3G4200D (Gyroscope) Datasheet. `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00265057.pdf`.

[8] LIS302DL (Accelerometer) Datasheet. `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00135460.pdf`.

[9] STM32F100RBT6B (ARM) Datasheet. `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00251732.pdf`.

[10] STM32VLDISCOVERY ARM development board user manual. `http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/ USER_MANUAL/CD00267113.pdf`.

[11] PySerial documentation. `http://pyserial.sf.net`.

[12] STEVAL-MKI005V1 (Accelerometer evaluation kit). `http://www.st.com/internet/com/TECHNICAL_RESOURCES/ TECHNICAL_LITERATURE/ USER_MANUAL/CD00214614.pdf`.

[13] Demoz Gebre-Egziabher, Roger C. Hayward, and J. David Powell. ”Design of Multi-Sensor Attitude Determination Systems”. *IEEE Transactions on Aerospace and Electronic Systems*, 40(2):627 – 649, 2004.

[14] Herbert Goldstein. *”Classical Mechanics”*. Addison-Wesley, 1980.

[15] Patrice Guillet. Online GPS for PermaSense WSN. Semester thesis, 2010.

[16] Holger Harms, Oliver Amft, Rene Winkler, Johannes Schumm, Martin Kusserow, and Gerhard Troester. ETHOS: Miniature Orientation Sensor for Wearable Human Motion Analysis. In *Sensors 2010: Proceedings of IEEE Sensors conference*. IEEE, 2010.

[17] Andreas Hasler, Igor Talzi, Jan Beutel, Christian Tschudin, and Stephan Gruber. ”Wireless Sensor Networks in Permafrost Research - Concept, Requirements, Implementation and Challenges”. *Proceedings of the Ninth International Conference on Permafrost*, 2008.

[18] Robert Mahoney, Tarek Hamel, and Jean-Michel Pflimlin. ”Nonlinear Complementary Filters on the Special Orthogonal Group”. *IEEE Transactions on Automatic Control*, 53(5):1203 – 1218, 2008.

[19] Kirk Martinez, Jane K. Hart, and Royan Ong. Environmental Sensor Networks. *IEEE Computer*, pages 50–56, 2004.

[20] William Premerlani and Paul Bizard. ”Direction Cosine Matrix IMU: Theory”. 2009. `http://gentlenav.googlecode.com/files/DCMDraft2.pdf`.

[21] Kurt Seifert and Oscar Camacho. Implementing Position Algorithms Using Accelerometers. Technical report, Freescale Semiconductor Inc., 2007.

[22] Atollic TrueSTUDIO web page. `http://www.atollic.com/truestudio`.

[23] PermaSense web page. `http://www.permasense.ch`.

[24] TinyNode web page. `http://www.tinynode.com/`.