Multi-rate Sensor Fusion for GPS Navigation Using Kalman Filtering

by

David McNeil Mayhew

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING

May, 1999

Blacksburg, Virginia

Multi-rate Sensor Fusion for GPS Navigation Using Kalman Filtering

by

David McNeil Mayhew

Committee chairman: Dr. Pushkin Kachroo,
Bradley Department of Electrical Engineering

(ABSTRACT)

With the advent of the Global Position System (GPS), we now have the ability to determine absolute position anywhere on the globe. Although GPS systems work well in open environments with no overhead obstructions, they are subject to large unavoidable errors when the reception from some of the satellites is blocked. This occurs frequently in urban environments, such as downtown New York City. GPS systems require at least four satellites visible to maintain a good position ‘fix’. Tall buildings and tunnels often block several, if not all, of the satellites. Additionally, due to Selective Availability (SA), where small amounts of error are intentionally introduced, GPS errors can typically range up to 100 ft or more. This thesis proposes several methods for improving the position estimation capabilities of a system by incorporating other sensor and data technologies, including Kalman filtered inertial navigation systems, rule-based and fuzzy-based sensor fusion techniques, and a unique map-matching algorithm.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

For several years, Global Positioning Satellites have orbited the Earth to provide absolute positioning on land, on sea, and in the air. Millions of GPS receivers are in use around the planet, in applications ranging from remote desert research to underwater cartography to simple recreation. Whatever the application, the main function of the GPS receiver remains constant: to obtain absolute position measurements anywhere on the globe. One major hurdle to GPS inherent in its method of operation is blockage of satellite reception. Tall buildings, bridges, high mountains, and common foliage overhead can block satellite reception.

An alternative to GPS navigation is an inertial navigation system (INS). INS is the application of sensors such as gyroscopes and accelerometers to maintain relative position information. However, inertial navigation has its drawbacks. Over a substantial amount of time, INS errors tend to accumulate unbounded and result in position estimates that deviate from the actual position. For these reasons, methods have been devised which fuse the GPS position measurements and inertial navigation measurements to provide a best estimate of position at any given time.

The purpose of this thesis is to examine various possible solutions for this system and to present the methods by which one might implement such a system. Multiple sensor configurations are presented, along with the issues relating to each. Additionally, a detailed explanation of the Kalman filtering and rule-based sensor fusion is given. PC-based software programming for actually implementing the system is discussed in detail, as well.

1.2 Scope and Structure of Thesis

Chapter 2 provides a survey of current literature related to the topics of inertial navigation systems and algorithms, GPS systems, and other methods of sensor fusion in similar applications.

Chapter 3 covers all of the hardware that was used in this sensor fusion project. This includes the test vehicle, the data acquisition hardware, and the means by which the acquisition hardware was interfaced to the vehicle hardware. Additionally, an explanation of some of the software commands that are specific to the selected hardware is presented.

Chapter 4 provides a comprehensive list of various sensors and sensor configurations that may be used in a sensor fusion application similar to the one presented in this thesis. The dynamic equations that govern the system for each basic configuration are also covered.

Chapter 5 approaches the more advanced subject of filtering the inertial sensor outputs by means of a Kalman filter. The specific filter for the configuration used in this project is presented, which may easily be modified for other configurations. Also, the details about the rule-based sensor fusion process, and the reasoning behind it, is given. Several methods for sensor fusion parameter optimization are presented, along with a novel map-matching algorithm.

Chapter 6 covers the implementation of the entire application in software. This covers details regarding development in both a DOS and Windows 95 programming environment under C++. This chapter gets into the specifics of programming a real-time application under DOS, such as interrupt driven communications and timing, as well as some Windows graphical user interface (GUI) design considerations.

Chapter 7 presents the results of the project, based on test runs in Blacksburg, Virginia, and New York, New York.

We present conclusions of this project in Chapter 8, along with potential avenues of continued research.

Chapter 2

Literature Survey

Before the main topic of this thesis is presented, we first present the reader with a brief summary of information that we collected from various sources including books, conference papers and other theses. Presented first is information relating to systems and algorithms using INS only. Then, we summarize work relating to GPS/INS sensor fusion, in particular. Lastly, we present several developed systems that perform GPS/INS fusion.

2.1 Inertial Navigation

Billur Barshan and Hugh F. Durrant-Whyte (1995) utilize a system consisting of three gyroscopes, a tri-axial accelerometer and two tilt sensors to perform inertial navigation. They focus on careful and detailed error modeling to obtain a position drift rate of 1-8 cm/s, depending on the frequency of acceleration changes. This, like any system requires additional information from some absolute position-sensing mechanism to overcome long-term errors. However, they show that a low-cost inertial sensing system can be used to provide valuable orientation and position information particularly for outdoor mobile robot applications.

A. Svensson and J. Holst (1995) have simulated a variety of filter configurations for the purpose of submarine navigation based on several inertial sensors. They had the most success with a complex fourteen state Extended Kalman Filter (EKF), which used eight states to describe the motion of the submarine and six to describe the measurement system.

Kirill Mostov (1996) used a hybrid least-mean-squares (LMS)/Kalman filter for the purpose of maintaining stability in a system where inaccuracies in the model would otherwise cause instability. This was done by using the Kalman filter to remove noise from the system and using the LMS to compute the weight functions, which could be translated into the Kalman gain values in an iterative fashion.

2.2 GPS/INS Fusion

Allison N. Ramjattan and Paul A. Cross (1995) use only a gyroscope and an odometer encoder, along with a GPS receiver, to produce a fused output. They have used this system in the streets on central London, and have demonstrated the improved position estimate obtained from fusing data from only a few sensor inputs. They evaluated the effectiveness of their system based on the system output's deviation from the 'true' path, which was obtained by digitizing the path from an overhead map.

Ren Da and Ching-Fang Lin (1995) use the State Chi-Square Test and the ARTMAP Neural Network to perform failure diagnosis in a GPS/INS integrated navigation system. They tested their system by means of computer simulation and demonstrated the detection of soft-failures by the tests.

2.3 Contribution of This Work

Several different types of systems have been used to generate an enhanced position estimate based on data from multiple sensors. As is the case with all systems that use inertial sensors only, those systems that were mentioned in section 2.1 are limited to provide only relative position and heading information from some arbitrary starting point. The work of Ramjattan and Cross (1995) is actually very similar to the work that has been done for this thesis. However, this thesis goes further than to provide information only about how to fuse inertial and GPS data to produce an enhanced output. This thesis examines several options for methods of fusing the inertial and GPS data such that other researchers can use this work as a starting point in their research. In addition, a unique map-matching algorithm is presented that can be used to further enhance system output in areas where the roadways are known. Map matching is a technique that has been around for a long time, but has not been exploited fully in many navigation systems. However, with today's portable computers and complete maps on a single compact disc, map matching can be used more for real-time position estimation purposes.

Chapter 3

Hardware Description

The hardware chosen for this project consists of several main components. These are: the test vehicle itself, inertial and dead-reckoning sensors, a data acquisition board, a GPS receiver, and a laptop computer. All of these components are fairly common and inexpensive. Each of these hardware sub-systems will be covered in detail. In addition, some of the software issues specific to the hardware will be discussed.

3.1 Test Vehicle

The test vehicle for our experiments was a 1997 Ford Taurus 4-door sedan. No special equipment was installed on the vehicle before the research. During the course of our work, however, several items were added to aid in data collection. We mounted a small black-and-white charge-coupled device (CCD) camera behind the rear-view mirror such that it could view the road and surroundings directly ahead, but would not distract the driver. In addition, we placed a small microphone on the underside of the sun visor with a switch on the upper-left portion of the dash, so that the driver could record his voice when desired. The outputs of both of these devices were fed under trim pieces of the vehicle and into the trunk, where they input into a VHS videocassette recorder. Also, we added a simple 12-volt DC to 110-volt AC voltage inverter, so that we could operate the VCR and laptop computer for extended periods of time.

Figure 3.1 – 1997 Ford Taurus test vehicle

3.2 Dead-Reckoning Sensors

In order to collect odometry data from the vehicle, we wanted to obtain a direct measurement of distance that the vehicle had traveled. Rather than attaching an additional encoder somewhere on the drive train, we used data outputs that Ford has already provided. The method we tried was simply to use the output from the odometer directly. The output from this connection needed to be conditioned before it was input into the data acquisition board. We added a simple circuit that railed the voltage from 0 to 5 volts when it crossed a voltage threshold. This method worked well when the vehicle was moving sufficiently fast, but generally did not work below speeds of a few miles-per-hour. This is because of the nature of the technology that the sensor is based upon. The signal is produced as a magnetic field rotates (at sufficient speed) near a coil of wire, inducing a current in that wire. Because of the type of sensor used to produce the signal, it does not work well at low speeds. It was decided that this was inadequate for our purposes, so we chose another method of odometry collection.

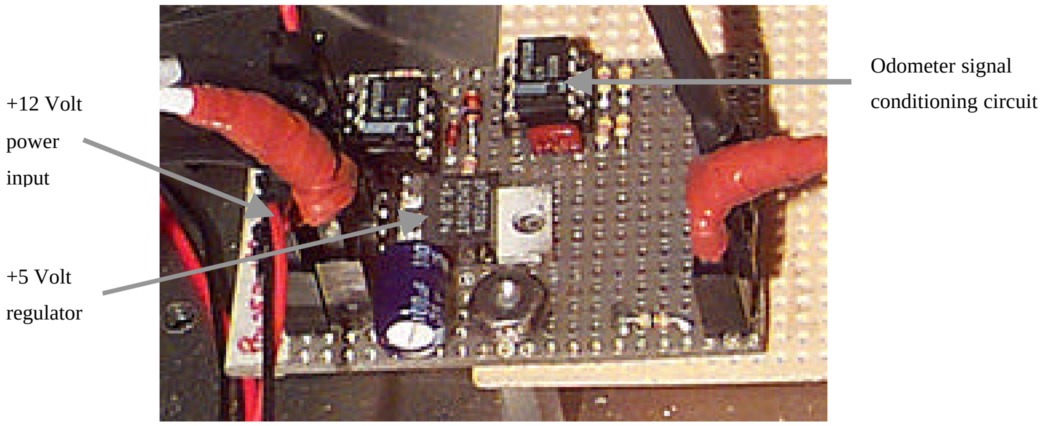


Figure 3.2 – Signal conditioning board

The second method provided better results while also being easier to implement in the vehicle. We used a signal from the anti-lock brake (ABS) on the rear-left wheel of the test vehicle. The anti-lock braking system uses a more accurate sensor for detecting wheel rotation (as compared to the odometer) which outputs a series of digital pulses. These pulses can be ‘picked off’ a wire that is part of the system and input directly into a data acquisition pulse counter. In our vehicle, a wire directly under the left rear seat carries this signal, so it was simply a matter of splicing our leads into the wires already provided. This is shown in Figure 3.3. This signal was already well suited for input into our data acquisition board, and the output was accurate even for low speeds. Each pulse from the ABS sensor indicates approximately 1/26th of a meter traveled or 0.03846 meters/pulse. This number can change slightly on a daily basis, due to changes in tire pressure or road conditions.

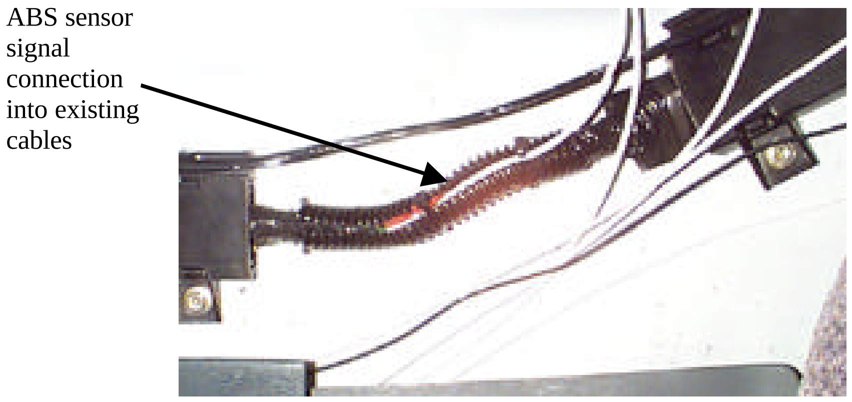


Figure 3.3 - ABS sensor connection (found under the left rear seat)

Another sensor attached directly to the hardware of the vehicle was a pull-string potentiometer wrapped around the steering column, located under the dash in front of the driver. This provided a direct measurement of the angle of the steering wheel, which corresponds directly to the angle of the vehicle’s front tires. The potentiometer was used as a simple voltage divider with 5 volts on the input, and the output signal going into the data acquisition board. We mounted the sensor such that when the steering wheel was turned all-the-way to the right it output 1.2 volts, and when the wheel was turned all-the-way to the left it output 4.5 volts. The full scale of 0 to 5 volts was not used because this would require the potentiometer to be mounted such that it was pulled to each extent of operation during use. This could potentially wear down the sensor and even break it if it were over-extended by a small amount repeatedly. The mounted potentiometer is shown in Figure 3.4.

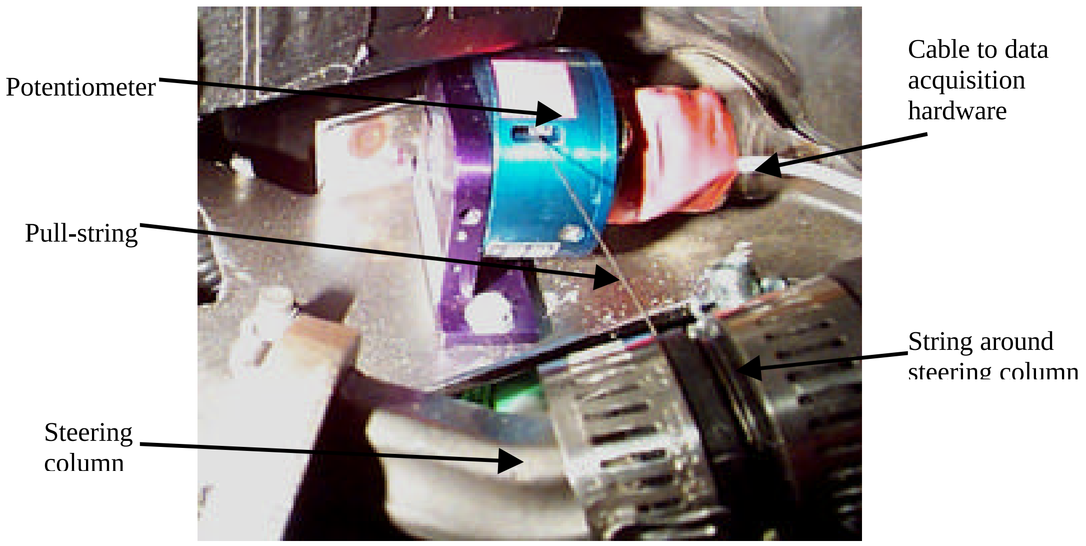

Potentiometer
Cable to data acquisition hardware
Pull-string
String around steering column
Steering column

Figure 3.4 - Mounted steering potentiometer

3.3 Inertial Sensors

We used two types of inertial sensors on this project: gyroscopes and accelerometers. Both types have been in use for navigation applications for several years. Recent advances in gyroscope technology in particular have allowed smaller, cheaper and more accurate gyroscopes to be offered, making INS solutions more practical.

Two different gyroscopes were mounted on the sensor board initially, for purposes of evaluating the performance of each. One of them broke shortly into the project (thereby failing its test) and we will not discuss it further. The gyroscope that we relied on exclusively was the Murata Gyrostar.

The Gyrostar is a vibratory piezoelectric rate sensor, which refers to the main internal component of the gyroscope that allows it to determine angular velocity. It operates on the Coriolis principle that means that a linear motion within a rotational framework will have some force that is perpendicular to that linear motion (Miyazaki, 1994). This simply means that, in the case of automobile navigation, the gyroscope is designed to measure the force perpendicular to the vehicle’s forward motion, which is proportional to angular velocity. The Gyrostar is capable of a measurement range of roughly ±80 deg/sec and has a linearity 0.5% full scale, which is sufficient for automobile applications (“Gyrostar,” 1994).

The accelerometer that we used in the project is the Single Chip Accelerometer with Signal Conditioning. The ADXL05 has an adjustable measurement range from ±1*g* to ±5*g* and an adjustable output scale from 200 mV/g to 1 V/g. The entire sensor is encased in a single 10-pin TO-100 case, and requires only a small circuit to set the adjustable range and scale and to filter the output as desired ("+/-1g to +/-5g Single Chip," 1996). For our purposes, we wish to reduce high-frequency output (such as that due to vibration) and use only the lower frequency output (such as that due to inertial effects of turning and acceleration). This is accomplished by using the manufacturer recommended DC-coupled connection, which has a frequency response from dc (0 Hz) to 1000Hz and measures +/- 2g full scale. In this application, sensor data is collected at 100Hz, so filtering out frequencies above 1000Hz removes high frequency signal components which cannot be removed in software. The sensor outputs approximately 2 to 5 mg (thousandths of a gravity) of noise in the frequency range, which must be removed during sensor filtering.

Two accelerometers were employed on our sensor board: One that measured longitudinal acceleration, and one that measured lateral acceleration. The accelerometers' relative orientations are shown in Figure 3.5.

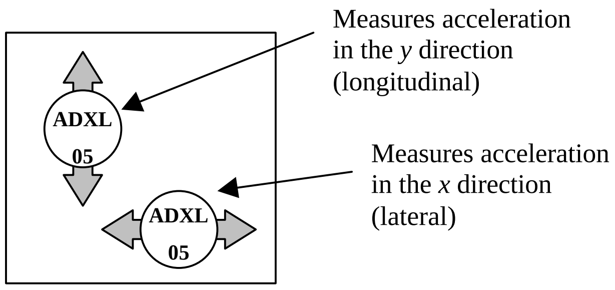

Measures acceleration in the *y* direction (longitudinal)
ADXL 05
Measures acceleration in the *x* direction (lateral)
ADXL 05

Figure 3.5 - Diagram of accelerometer orientations on sensor board

From this data, we can integrate to find relative speed and heading. This is discussed in more detail in the chapter on system modeling. An example plot of the inertial sensor outputs for a 10-second time frame is shown in Figure 3.6.

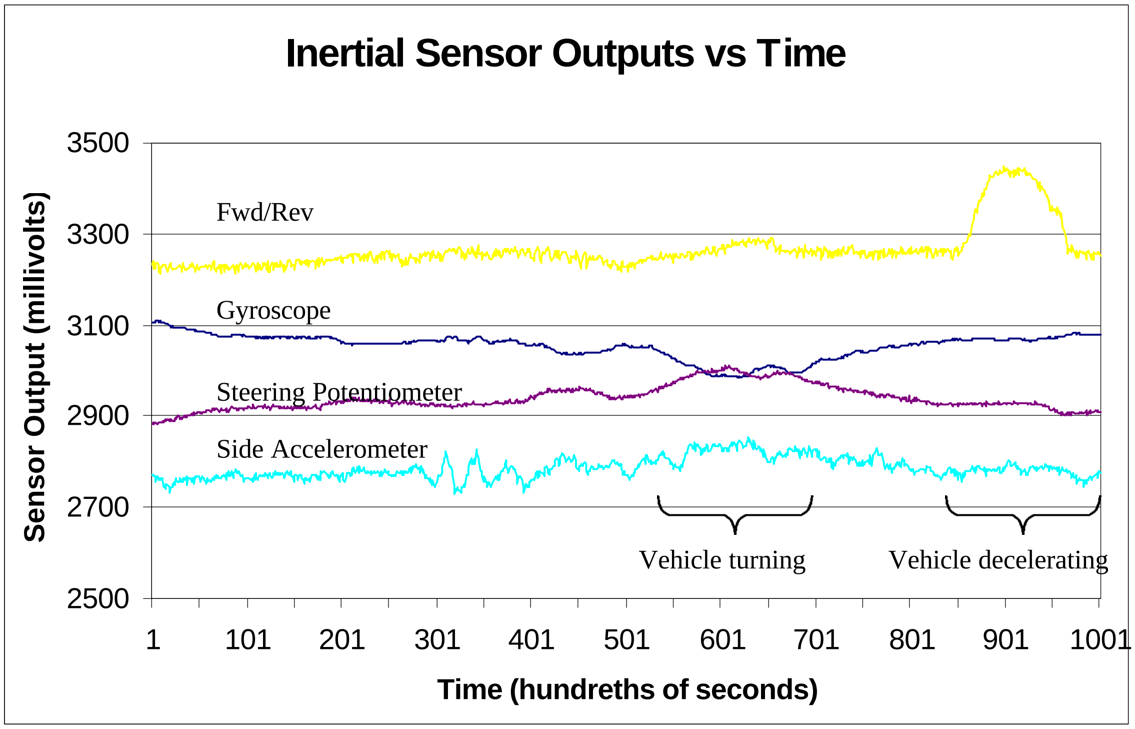


Figure 3.6 – Example plot of inertial sensor outputs

3.4 Data Acquisition Board

The data acquisition board we used on this project was a Iotech DaqBook 100. It is a small box that sits outside the PC, and attaches to the PC via a parallel cable. All of our sensor inputs are connected into the three connectors on the back of the DaqBook – one analog I/O, one digital I/O, and one pulse/frequency/high-speed digital input. Our two gyroscopes (even though one was not operational), two accelerometers and steering potentiometer all input to the analog I/O port. The single distance encoder output (we used either the odometer input or the ABS input at any one time) fed into the pulse digital input. The DaqBook analog inputs can

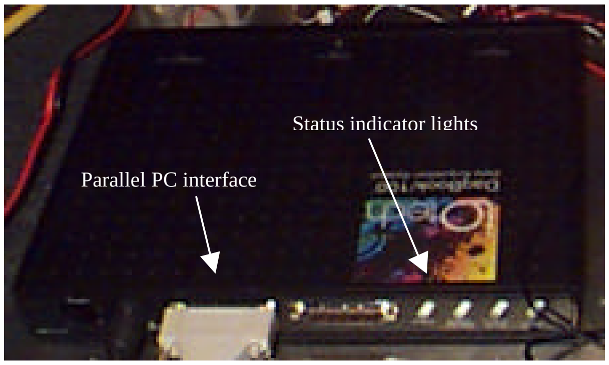


Figure 3.7 – Close-up picture of the DaqBook

be set to operate in *differential* or *single-ended* mode. This means that analog signals can be measured based on the potential between the single pin's input and the DaqBook's ground (single-ended) or the signals can be measured based on the potential between adjacent pins (differential). In addition, the board can be set to measure analog inputs as *unipolar* or *bipolar*. In unipolar mode, input voltages from 0 to +10 volts can be applied, and in bipolar mode, input voltages of up to 5 volts in magnitude in either polarity can be applied. Each of the analog pins we set to operate in differential, unipolar mode.

When the DaqBook is connected to a standard parallel port, it supports up to 170 Kbytes/sec of bi-directional communication. We are sampling data at 100 Hz in this project, and each sample is 14 bytes of data, resulting in 11.2 Kbytes/sec of data transferred. Any commands sent to the board must also be considered, but we can see that the board can easily handle the data rate we desire. More about the data samples will be presented in the software chapter of this thesis. (*DaqBook*, 1994)

The data acquisition board can be used in several different modes, but we essentially used only two: polled output, and timed output. In DOS based programming, we are less encumbered by delays introduced by the operating system (OS) on I/O operations, such as parallel port reads and writes. For this reason, we are able to use polled output from the DaqBook, which is substantially easier to implement. A 100 Hz loop is implemented using interrupt based timing in the host computer, and in each loop we send a request and receive a response with negligible delay. When programming under Windows 95, which is not truly a real-time OS, we found that we could not use the simple polled-response method that worked with DOS. After a little work, we found that the DaqBook could be set to automatically sample data at a specified rate. This rate is maintained by internal timing circuitry, so our program did not need to initiate each sample. The program merely 'grabbed' the data sample from the parallel port at the appropriate time. More details about the implementation in the programs are presented in the chapter on software.

3.5 GPS Receiver

Perhaps the single most important piece of hardware in this system is the GPS receiver. The output from the receiver is the only way we have any notion of our absolute position on the globe. For information regarding some basic GPS fundamentals, refer to Appendix C.

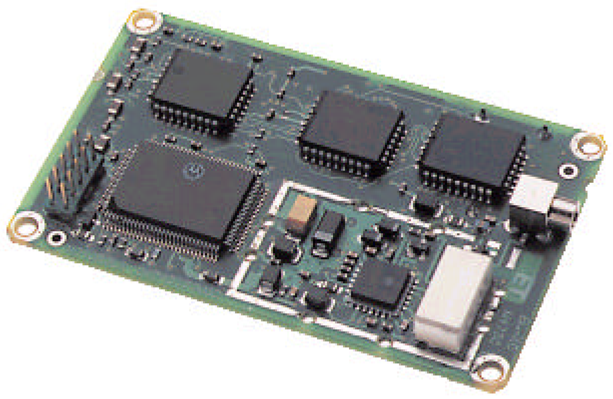


Figure 3.8 – Motorola VP Oncore GPS Receiver (from Motorola Oncore User's Guide)

The GPS receiver used in this project was a Motorola VP Oncore 6-channel receiver. This version of the Oncore receiver communicates serially via a transistor-transistor logic (TTL) interface with the host computer at 9600 baud. The receiver has the capability to perform differential GPS (DGPS) given the appropriate input from a source such as a Coast Guard DGPS station. We did not use this feature since it required additional hardware, a Coast Guard beacon receiver, and because differential correction signals are not available everywhere.

We used the GPS receiver in a very simple manner. The receiver can be set such that it automatically sends its complete message once every second. It is then the job of the host computer to detect the message and interpret it correctly. From this data, we know our current position, time, and status information for the receiver. The receiver is capable of receiving and processing dozens of user commands, but for our purposes, the automatic once-per-second output is adequate. For more information about the Motorola Binary Format messages, refer to Appendix C. When processing the sensor data, we would like to know whether the GPS data being returned is likely to be 'good' or not. A simply method of determining this is based on the status information returned from the receiver with each data message. The GPS receiver returns a status byte with the following information:

```
Bit 7: Position propagate mode
Bit 6: Poor geometry (DOP > 20)
Bit 5: 3D fix
Bit 4: Altitude hold (2D fix)
Bit 3: Acquiring satellites/position hold
Bit 2: Differential
Bit 1: Insufficient visible satellites (<3)
Bit 0: Bad almanac
```

A given GPS sample is considered ‘bad’ if any of bits 0, 1, or 6 is set or if either of bits 4 and 5 is not set. That is, the almanac data (information about the location of the GPS satellites in space) must be accurate and a sufficient number of satellites must be visible (3 or more). Also, the receiver must have both a 2D and a 3D fix (location estimate) to be considered ‘good’. Checking the status byte does not guarantee that the data returned from the receiver is accurate, but it does quickly eliminate many bad GPS data points from consideration.

The gyroscopes, accelerometers, DaqBook, and GPS receiver all are mounted securely on a single piece of aluminum in the trunk of the test vehicle, shown in Figure 3.9.

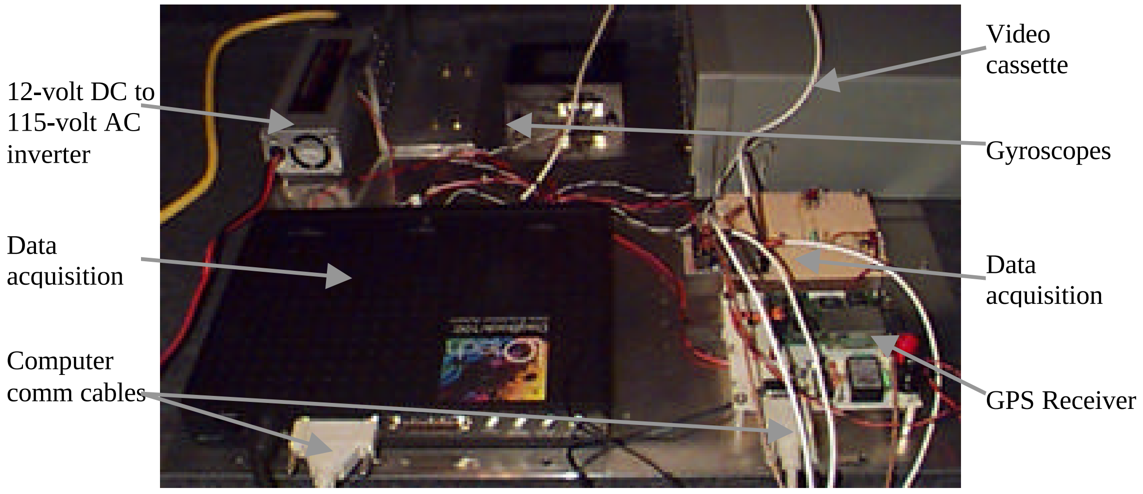


Figure 3.9 - The entire data collection assembly

3.6 Host Computer

The computer we used for data collection was a Fujitsu Pentium 133 with 16 megabytes RAM running Windows 95 – a typical laptop computer. During operation, the only external connections required were to the GPS receiver, which used a serial port, and to the DaqBook, which used the parallel port. Using a laptop made the data collection easy. By using long serial and parallel cables from the trunk to the front seat, the operator could sit in the passenger seat and watch all of the data collection take place in real-time. This proved very useful in the few occasions when something would happen, such as a cable coming loose, because we could detect and correct this problem immediately, rather than waste time on a bad data collection.

Chapter 4

Modeling and Sensor Fusion Algorithms

In order to produce a good filtering algorithm with which to work, we must start with accurate dynamical models describing the system. This chapter covers many alternatives and groups of alternatives, such as various sensor choices, that may be presented to the designer of a similar navigation system. For each alternative, we will present the dynamical equations that relate it to the system's state variables. In addition, we will discuss the viability of actually implementing the given alternative in a system.

4.1 System Variables

Regardless of which sensor configuration we choose the desired output is the same. At a minimum, we want the best possible estimation of our current position at all times. Additionally, we may desire such information as speed, heading and acceleration. A simple dynamic model of the vehicle is shown in Figure 4.1.

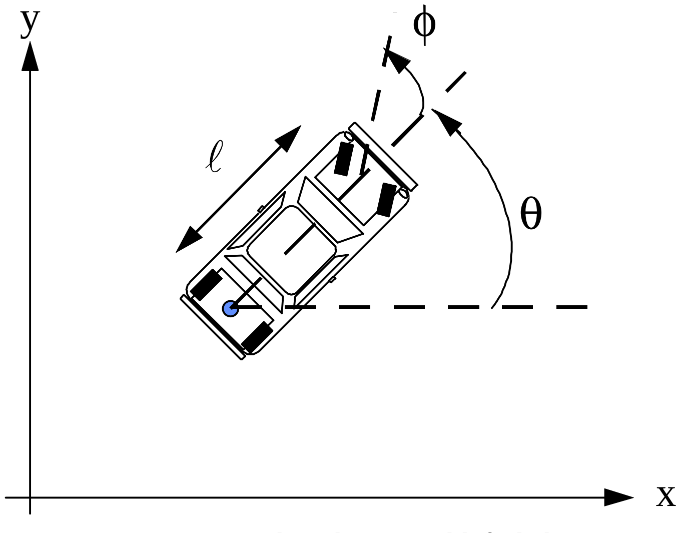


Figure 4.1 – Simple 2D dynamic model of vehicle

4.2 Sensor Types

At a bare minimum, we desire lateral and longitudinal information about our vehicle. Thus, we need both lateral sensors and longitudinal sensors. This section reviews a variety of dead-reckoning sensor types, and the information that they can provide to a navigation system.

4.2.1 Distance Encoders

Distance encoders are very useful for automobile navigation because they are digital in nature. A single rotation of the shaft on which an encoder is mounted produces a fixed number of pulses, which can easily be counted. By keeping a running count of all of the encoder pulses, a navigation algorithm can know exactly what total distance has been traversed. A first derivative results in (roughly) instantaneous speed, and a second derivative results in (roughly) instantaneous acceleration. However, since we typically are interested in vehicle position, which is a function of distance from the starting point, the exact distance output from the encoder is very useful.

Figure 4.2 – A typical small quadrature encoder

A small drawback to using some encoders is the inability to determine direction from the output. Many encoders, however, have two sensors that output signals 90^o out of phase from each other. These output signals could be used to determine both speed and direction. These types of encoders are known as quadrature encoders. Quadrature encoders can be used to determine speed and direction from the two sensor outputs by examining the relative phase of each of the outputs.

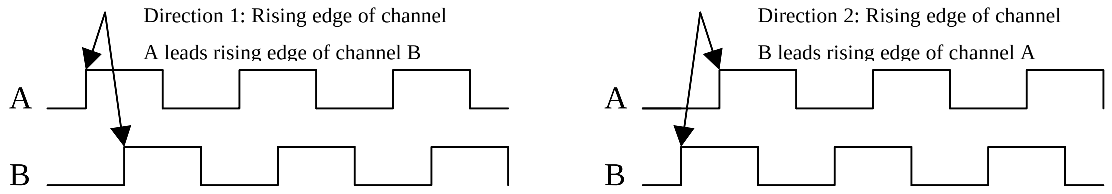

Direction 1: Rising edge of channel A leads rising edge of channel B
A
B
Direction 2: Rising edge of channel B leads rising edge of channel A
A
B

Figure 4.3 – Obtaining direction from quadrature encoder outputs

As was noted in the chapter on hardware, the signal output reliability can affect the overall acceptability of the encoder as a distance measured. In the case of this project, the signal running to the odometer was initially used as a distance encoder substitute. This proved unsuitable because of the unreliability of the signal at low speeds. We then switched to using the output from the anti-lock braking system, which produces a digital signal like that of a normal encoder (non-quadrature). We could therefore determine speed of travel with a high amount of accuracy.

4.2.2 Tachometers

Tachometers differ from encoders in that they provide a direct measure of speed instead of distance. A tachometer typically outputs a voltage level that is proportional to speed of the shaft on which it is mounted. In an automobile, this shaft must be located on the drive side of the transmission (as opposed to the engine side). This is because a shaft on the drive side of the transmission will always be spinning at a rate proportional to the speed of the vehicle.

Tachometers are generally cheaper than encoders are, but they also have drawbacks that are more significant. Because a tachometer outputs a voltage level, the navigation scheme (and subsequent filtering algorithm) must take into account the possibility of variations in offset and magnitude due to slight differences in manufacturing from unit to unit. Additionally, environmental changes can alter the operating parameters of such a device somewhat. In addition, to get distance from a tachometer output, the algorithm must integrate the signal over time. This causes the small errors in the output signal to accumulate.

4.2.3 Accelerometers

Accelerometers can be used for both speed and heading estimation. Accelerometers provide a direct measurement of acceleration, or force, in one direction. An accelerometer typically measures the acceleration of the car due to forces acting on the car. Ideally, by using only the two accelerometers, one may determine 2-dimensional position at any given time.

However, several problems arise when trying to do this. Accelerometers usually measure more than what we actually want to measure. The output signal is composed of a zero-frequency (DC) component, a low-frequency component and a high-frequency component. The zero-frequency component of the output signal is due to any offset in the signal which is intended in the design of the accelerometer signal plus an offset due to any slight misalignment in mounting or tilt of the vehicle. The high-frequency component is due to vibrations in the vehicle, which is undesired in the signal. Fortunately, passive analog filtering can usually take out the high-frequency component, once a cut-off frequency has been determined. The zero-frequency component must be taken care of in the navigation algorithm as a center point of the accelerometer. In the ideal case, we want only the component of the signal that is due to the movement of the vehicle on the road.

Another consideration in using accelerometers is that for determining heading, one must first know angular velocity. Acceleration normal (perpendicular) to the direction of motion relates speed and angular velocity. Thus, speed must be known to determine angular velocity and heading.

There are a couple substantial benefits to using accelerometers, however. First, accelerometers can be purchased in a single chip package, making them very small and cheap. Second, accelerometers are capable of measuring distance without any attachments to the vehicle itself. In the design of a self-contained INS package, with no external encoder or tachometer attachment, accelerometers are the only choice.

4.2.4 Tilt Sensors

Tilt sensors are used to measure the pitch or roll of a vehicle. Pitch and roll data may or may not be useful, depending on the intended application. For example, if we are concerned that the vehicle may traverse severe inclines while driving, then we may need to use tilt information to compensate for accelerometer errors. Suppose we have one of the following exaggerated cases:

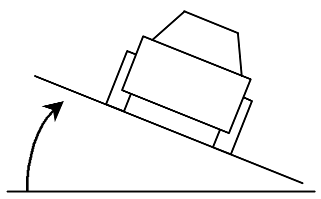


Figure 4.4a - Vehicle with side tilt

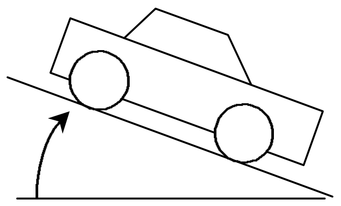


Figure 4.4b - Vehicle with front tilt

In Figure 4.4a, the vehicle is shown with a severe side tilt, due to a tilted road surface. In this case, an onboard accelerometer will read a significant side-to-side acceleration, indicating that the vehicle is continuously turning in one direction. Similarly, in Figure 4.4b, the vehicle is shown with a forward tilt, due to a sloped road surface. In such a situation, an accelerometer would read a continuous acceleration as if the vehicle were continually decelerating.

Figure 4.5 – A typical liquid-filled tilt sensor

In either case, the accelerometer readings are misleading about the vehicle's motion because of an inclined road surface. An additional sensor, such as a tilt sensor can be used to indicate if a given force is due to acceleration of the vehicle in a particular direction, or if it is due to a tilt in the vehicle body.

4.2.5 Gyroscopes

Gyroscopes have advanced significantly in the last several years, and are a favorite for inertial navigation systems. Gyroscopes, like accelerometers, require no signal attachments directly to the vehicle. Unlike accelerometers, however, gyroscopes always output a signal proportional to angular velocity, independent of the speed of the vehicle. In the case of using an accelerometer, speed was needed to determine angular velocity. This meant that any error in the speed measurement would be magnified in the overall position estimate. This problem does not occur when using gyroscopes.

Some problems are apparent with gyroscopes, like other INS devices. Changes in temperature or humidity can cause the operating parameters of a gyroscope to change slightly, which will introduce heading and position errors over time. This can be corrected somewhat by use of adaptive parameter estimation, which will be covered in the chapter on advanced algorithm development.

4.2.6 Steering Position

Steering position information is useful because it provides us with a direct measurement of a physical condition of the vehicle. We assume that a measurement of steering shaft position is directly proportional to the front tire angle of the vehicle. This, in turn, provides a way to calculate angular velocity given the speed of the vehicle. As with accelerometers, because the angular velocity calculation depends on vehicle speed, errors in the speed measurement are magnified in the final position estimate.

There are several other drawbacks to using a steering shaft measurement, despite it initially seeming to be a good alternative. First, like a distance encoder or tachometer, a steering measurement device, such as a potentiometer, must be mounted to the shaft of the steering wheel. This mounting process is generally not trivial and must be calibrated and adjusted for each particular vehicle. Additionally, in using a measurement of the front wheel angle to calculate angular velocity, we must know the length of the vehicle. Using a steering shaft measurement requires substantial calibration for the vehicle on which it is used. However, decent results can be obtained from a steering measurement when it is properly used.

4.3 Sensor Configurations

For this project, several sensor configurations were considered when designing the filtering algorithm. Four different configurations are presented here, chosen mainly because they each include a subset of the sensors with which we began the project. Any of the configurations may be modified easily to include additional sensors.

Note that every configuration has a GPS receiver, because it is our only means of absolute positioning. The only difference, then, is amongst the variety of dead-reckoning sensors that were chosen for the particular configuration. In this section, we are focusing on the configurations of sensors, rather than the sensors themselves. Thus, for modeling purposes, we will assume that every sensor has accurate, noiseless signal outputs.

4.3.1 Configuration 1

GPS Receiver, Steering Position, and Odometer

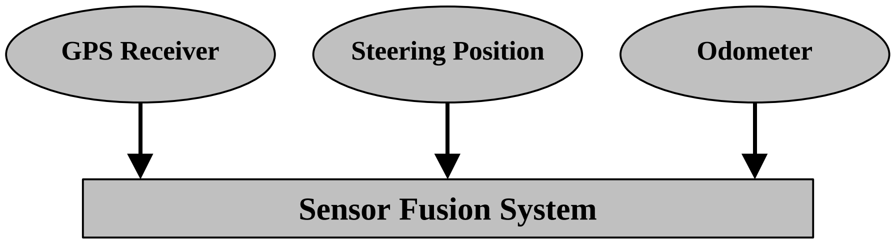


Figure 4.6 – Configuration 1 block diagram

In this configuration, the odometer signal is a series of pulses, where each pulse represents some fixed distance covered. Thus, the odometer provides us with a direct measurement of distance covered. Either an absolute encoder or a potentiometer can measure the steering position. In this case, the position of the front wheels is of interest, so we assume a bicycle model where the front and rear wheels are each considered a single wheel at the midpoint of the two axles. Let the state of the vehicle be represented by (x, y, q, f, v). x and y give the location of the rear axle midpoint, q gives the angle of the vehicle body with the horizontal, f gives the steering angle with respect to the vehicle's body, and v is the forward velocity of the rear wheels of the vehicle. (See Figure 4.1)

The dynamic equations for this vehicle model are:

$$\begin{aligned}
\dot{x} &= v\cos(q) \\
\dot{y} &= v\sin(q) \\
\dot{q} &= \frac{1}{l} v\tan(q) \\
v &= \dot{a} \times METERS\_PER\_TICK
\end{aligned}$$

where a is the odometer measurement (in ticks) and METERS_PER_TICK is a conversion factor which changes ticks to meters. The value of METERS_PER_TICK depends on the number of ticks per wheel revolution and the circumference of the wheel, which may change, but we will assume to be a constant. We must use the discrete form of these equations to implement them in a processor. The discrete form of the above equations is:

$$\begin{aligned}
x(k+1) &= x(k) + Tv(k)\cos(q(k)) \\
y(k+1) &= y(k) + Tv(k)\sin(q(k)) \\
q(k+1) &= q(k) + T\frac{1}{l}v(k)\tan(f(k)) \\
v(k+1) &= [a(k) - a(k-1)] \times METERS\_PER\_TICK
\end{aligned}$$

Here, T is the sample time (0.01 seconds, in our case) and *k* is the discrete time step index.

4.3.2 Configuration 2

GPS Receiver, Gyroscope, and Odometer

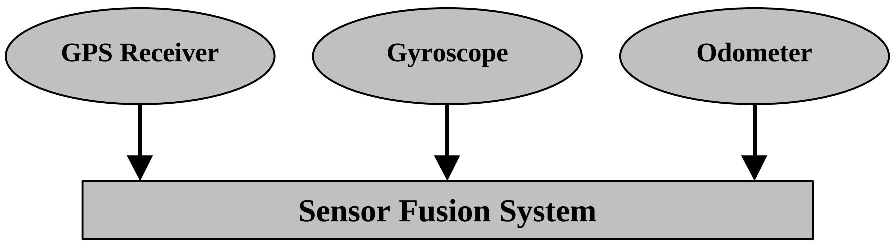


Figure 4.7 – Configuration 2 block diagram

Our second configuration involved using a gyroscope for heading information instead of using a steering measurement. This simplifies the model somewhat, because we can essentially model the vehicle as a point. This is because a gyroscope provides a direct measurement of angular velocity ($\dot{q}$), so we can ignore the front wheels and the length of the vehicle. The dynamic equations for this model are:

$$\begin{aligned}
\dot{x} &= v\cos(q) \\
\dot{y} &= v\sin(q) \\
\dot{q} &= U_1 \\
v &= \dot{a} \times METERS\_PER\_TICK
\end{aligned}$$

where U_1 is the gyroscope measurement in degrees/second of angular change. The discrete form of the dynamic equations is:

$$\begin{aligned}
x(k+1) &= x(k) + Tv(k)\cos(q(k)) \\
y(k+1) &= y(k) + Tv(k)\sin(q(k)) \\
q(k+1) &= q(k) + TU_1(k) \\
v(k+1) &= [a(k) - a(k-1)] \times METERS\_PER\_TICK
\end{aligned}$$

Note that the value of q at time k+1 is dependent upon the current measurement and the previous value of q. This means that q must be initialized at some point to a know value.

4.3.3 Configuration 3

GPS Receiver, Gyroscope, and Accelerometer

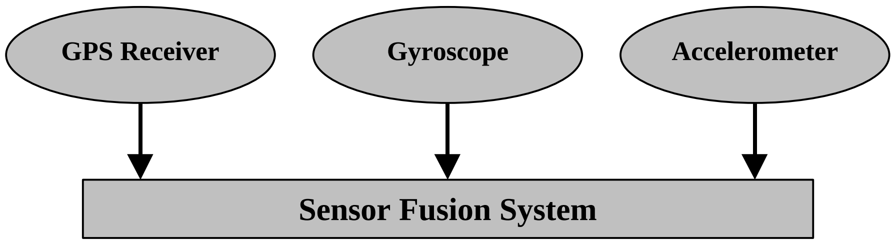


Figure 4.8 – Configuration 3 block diagram

The third configuration considered involves the use of an accelerometer for acceleration measurement, which can be used to determine relative speed and distance, and a gyroscope again for heading information. The dynamic equations for this configuration are shown here:

$$\begin{aligned}
\dot{x} &= v\cos(q) \\
\dot{y} &= v\sin(q) \\
\dot{q} &= U_1 \\
\dot{v} &= U_2
\end{aligned}$$

where U_1 is again the gyroscope measurement in degrees/second and U_2 is the accelerometer reading in meters/second^2. The discrete form of these equations is shown here:

$$\begin{aligned}
x(k+1) &= x(k) + Tv(k)\cos(q(k)) \\
y(k+1) &= y(k) + Tv(k)\sin(q(k)) \\
q(k+1) &= q(k) + TU_1(k) \\
v(k+1) &= v(k) + TU_2(k)
\end{aligned}$$

As in configuration 3, this configuration has q dependent on the measurement and the previous value of q. The value of v is calculated in a similar manner in this case. Thus, both q and v must be initialized to known values at some point.

4.3.4 Configuration 4

GPS Receiver, Gyroscope, and Two Orthogonal Accelerometers

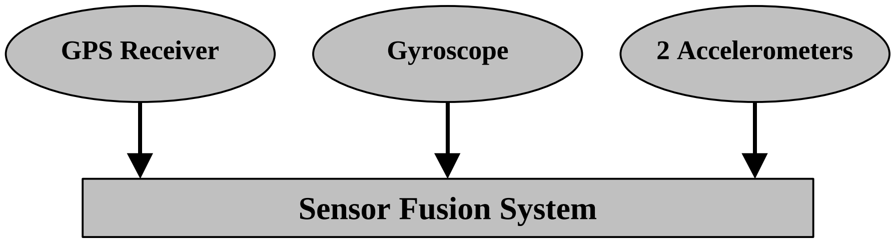


Figure 4.9 – Configuration 4 block diagram

This configuration is the only one in which redundant sensor data is present. The gyroscope and one of the accelerometers both provide heading information in the model. The dynamic equations that relate the measurements to the system variables are shown here:

$$\begin{aligned}
\dot{x} &= v\cos(q) \\
\dot{y} &= v\sin(q) \\
\dot{q} &= U_1 \\
\dot{q} &= \frac{U_3}{v} \\
\dot{v} &= U_2
\end{aligned}$$

Where U_1 is the gyroscope measurement in degrees/second, U_2 is the first accelerometer reading in meters/second^2, and U_3 is the second accelerometer reading in g. The discrete form of these equations is shown here:

$$
\begin{aligned}
x(k+1) &= x(k) + Tv(k)\cos(q(k)) \\
y(k+1) &= y(k) + Tv(k)\sin(q(k)) \\
q(k+1) &= q(k) + TU_1(k) \\
q(k+1) &= q(k) + T\frac{1}{v \times 1g}U_3(k) \\
v(k+1) &= v(k) + TU_2(k)
\end{aligned}
$$

4.3.5 Configuration Choice

In our project, based on the quality of the available sensors, we chose to use the second configuration. In the test we performed, we found that the best performance was obtained by using only the gyroscope and the odometer. Depending on the application or available sensors, a different configuration can be chosen.

Chapter 5

Algorithm Extensions

When designing the overall fusion algorithm, we must take into consideration the multi-rate sensor collection, and design our algorithm appropriately. That is, we need to fuse inertial and GPS data, when we are sampling our inertial sensors at 100 Hz and receiving GPS data at 1 Hz. Here, the term ‘fusion’ is refers generally to the process of combining two sets of data to produce a better output. Sensor fusion can be accomplished with a filter, or it can be done by weighting each set of data based on a set of rules or fuzzy logic. We considered a couple different methods for fusing these sets of data. For example, we could extrapolate GPS data at 100 Hz based on previous GPS data trends and perform sensor fusion at 100 Hz. Conversely, we could perform sensor fusion of GPS and inertial data at 1 Hz (whenever GPS data is received) and rely on inertial data alone to carry us between GPS samples. It is the second method we chose to implement, primarily because we felt the simpler approach was better and fusion at 1 Hz would be adequate for our purposes.

This chapter will cover the general approach of our algorithm, as well as a description of our Kalman filter for inertial reckoning. Then we describe two methods of determining the sensor fusion weights – a simple rule-based method, and a more robust fuzzy logic-based method. Then a simple method for map-matching is described which may be implemented if accurate latitude/longitude road data is available for a region.

5.1 General Approach

There are two basic types of methods for integration of GPS and INS data in a system. These are *tightly coupled* and *loosely coupled*. Internal to the GPS receiver, a set of pseudorange measurements is obtained for position estimation. A pseudorange is a distance estimate from the GPS receiver to one satellite before being error corrected. The straightforward (and most optimal) approach for integrating GPS with an INS is to directly utilize GPS pseudorange measurements from the GPS receiver to correct INS error growth with a specially designed Kalman filter that models the INS errors and the

GPS measurement geometry. This is a tightly coupled integration technique, a block diagram of which is shown in Figure 5.1. With a loosely coupled integration, the GPS pseudorange measurements are preprocessed by a Kalman filter internal to the GPS receiver, which produces "GPS derived" geographic position and velocity as the receiver outputs. (Weiss, 1996) The block diagram for a loosely coupled integration is shown in Figure 5.2. Because the Motorola Oncore GPS receiver has a built in Kalman filter and outputs the "GPS derived" position and velocity outputs, we use the loosely coupled configuration in our experiments.

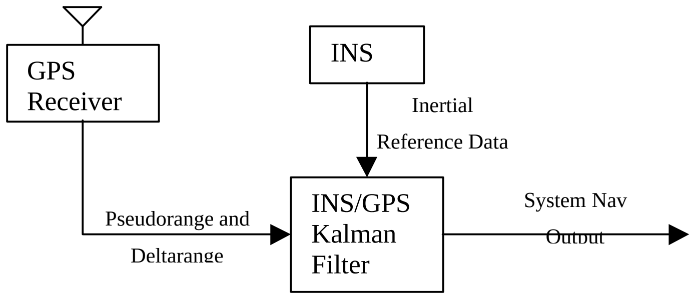


Figure 5.1 – Tightly coupled GPS/INS System

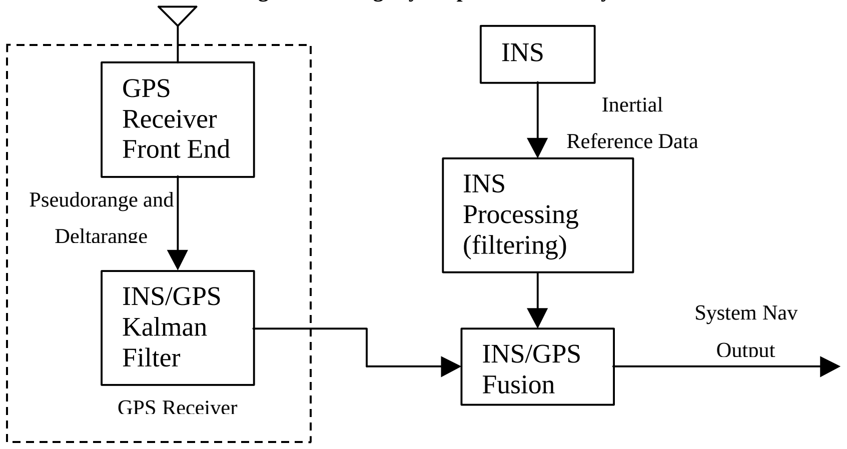


Figure 5.2 – Loosely coupled GPS/INS System

A simple block diagram of our implemented algorithm is shown in Figure 5.3.

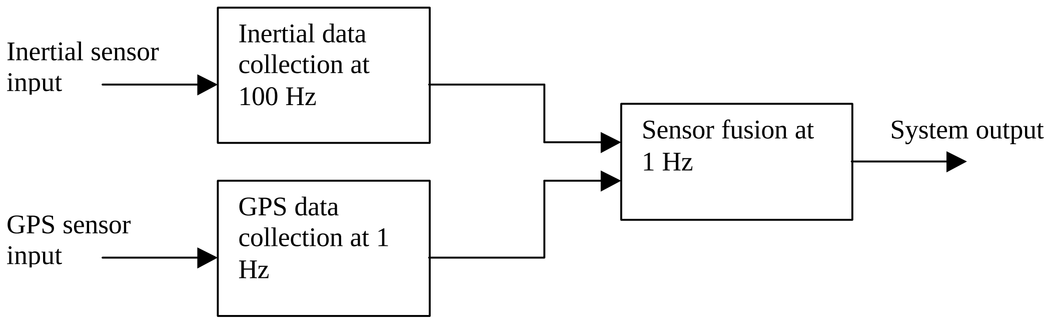


Figure 5.3 – Basic sensor fusion algorithm flow

We perform inertial navigation filtering at 100 Hz, then fuse the inertial data with the GPS data at 1 Hz. That is, our Kalman filtered state vector is not directly dependent on the GPS receiver output. Instead, we use some sort of fusion algorithm to combine the output from the Kalman filtered inertial data and the GPS data. Inertial/GPS sensor fusion amounts to dynamically determining weights with which we combine the position determined by the INS data and the position output from the GPS receiver. Thus, we would like to determine the best values for Ψ_{POS} and Ψ_{HEAD} for the weightings of GPS position and heading, respectively. In our case, we do not determine a value to fuse GPS and INS velocities, because we assume that the INS velocity is the best measurement available to the system.

After determining the weights, we fuse the inertial and GPS data using the following equations:

$$Lat_{est} = \Psi_{POS} Lat_{GPS} + (1 - \Psi_{POS}) Lat_{INS}$$
$$Long_{est} = \Psi_{POS} Long_{GPS} + (1 - \Psi_{POS}) Long_{INS}$$
$$Heading_{est} = \Psi_{HEAD} Heading_{GPS} + (1 - \Psi_{HEAD}) Heading_{INS}$$

where Lat_{est}, $Long_{est}$, and $Heading_{est}$ are the estimated latitude, longitude, and heading, respectively. Note that these weights will be 1 when initializing the system and they with be 0 when GPS data is bad. When the GPS data is good, the weights will lie somewhere in the range [0,1].

5.2 Geodetic to Ground Conversion

In order to relate the geodetic coordinates (latitude and longitude) to local ground coordinates (X and Y in meters), a conversion equation is required. The reason that this relation is needed is so that the data obtained from the GPS receiver, which is in geodetic coordinates (latitude and longitude), can be processed along with data obtained

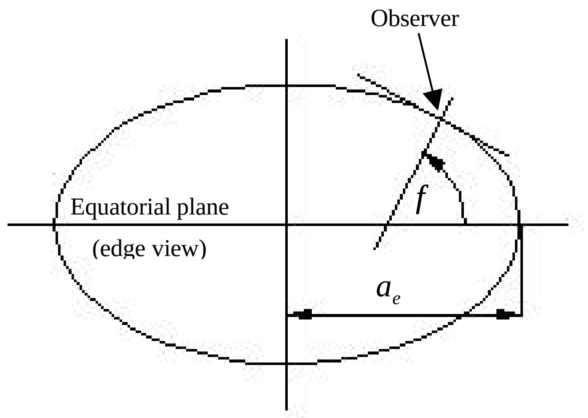


Figure 5.4 – Cross-section view of elliptical Earth

from the odometer and inertial sensors, which is most easily converted into relative X and Y meters. Given a particular location on the surface of the Earth, we would like to know how many meters of north-south or east-west travel correspond to how many degrees of latitude or longitude, respectively. Note that, assuming the Earth to be a perfect ellipsoid with a fixed equatorial radius a_e and eccentricity e, these conversion rates are dependent only upon the current latitude measurement f.

The conversion rates that were used in our software and algorithms were taken from calculations performed by Gene Felis, Electronics Engineer, NUWC Div Keyport, in 1976. He based his calculations on the elliptical world model with geodetic latitude as presented on pages 26-29 of *Methods of Orbit Determination* by P. R. Escobal (1965). The formula derivations are rather complex and have been included in Appendix B of this thesis. The end results of the derivations are shown here:

The constant C_1, which is the number of meters north to south corresponding to the change of 1 degree in latitude, is calculated as follows:

$$C_1 = \frac{2p * a_e * \left(1 - e^2\right)}{360 * 3600 * (1 - e^2 \sin^2 f)^{3/2}}$$

The constant C_2, which is the number of meters east corresponding to a change of 1 degree of longitude, is calculated as:

$$C_2 = \frac{2p * a_e * \cos f}{360 * 3600 * \left(1 - e^2 \sin^2 f\right)^{\frac{1}{2}}}$$

a_e is the equatorial radius of the earth ≈ 6,378,150 meters

e is the eccentricity of the elliptical cross section of the earth ≈ 0.08181333

f is the geodetic latitude of the observer (the vehicle)

To summarize, assume that the vehicle is on the surface of the Earth at latitude ϕ. Then, we have the following two conversion rates to convert meters to milliseconds of latitude and longitude (and vice versa):

$$1 \text{ millisecond latitude} = \frac{1}{32.55720\left(1 - 0.0066934 \sin^2 f\right)^{\frac{3}{2}}} \text{ meters}$$

$$1 \text{ millisecond longitude} = \frac{\cos f}{33.3392843\left(1 - 0.0066934 \sin^2 f\right)^{\frac{1}{2}}} \text{ meters}$$

5.3 Kalman filter for INS

We use a Kalman filter in this project only to determine the (relative) position output from the inertial sensors. The internal details of the Kalman filter are not the focus of this thesis. For Kalman filtering background, we refer the reader to *An Introduction to Kalman Filtering with Applications* (Miller and Leskiw, 1987). We are primarily interested in the parameters that affect the performance of the filter. Sensor inputs are typically converted into meaningful values such as meters, degrees/sec, etc. The parameters that are used for these conversions are subject to errors and require correction for effective filtering operation.

A Kalman filter can be used to produce the optimal state estimate of a system given a set of measurements and relationships between the measurements and the state vector. It can be applied when the relationship between the state vector and the measurement vector can be written as:

$$X(k+1) = A(k)X(k) + w(k)$$
$$Y(k) = CX(k) + v(k)$$

where *X(k)* is the state vector at time step *k* and *Y(k)* is the measurement vector at time step *k*. *A(k)* is the state transfer matrix and *C* is the observation matrix. *w* and *v* are zero mean, white gaussian noise variables.

In our project, we use an eight state Kalman filter. The two values in the measurement vector are velocity from the ABS wheel encoder and angular velocity from a gyroscope. The state vector, X, and the measurement vector, Y, are shown here:

$$X(k) = \begin{bmatrix} x[m\sec] \\ y[m\sec] \\ q\,[\deg] \\ \dot{q}\left[\frac{\deg}{\sec}\right] \\ \ddot{q}\left[\frac{\deg}{\sec^2}\right] \\ \Delta O\left[\frac{meters}{\sec}\right] \\ \Delta\dot{O}\left[\frac{meters}{\sec^2}\right] \\ \Delta\ddot{O}\left[\frac{meters}{\sec^3}\right] \end{bmatrix} \qquad Y(k) = \begin{bmatrix} q\left[\frac{\deg}{\sec}\right] \\ \Delta O\left[\frac{meters}{\sec}\right] \end{bmatrix}$$

where *x* and *y* are the longitude and latitude position estimates, respectively. *q* is the current vehicle heading estimate and *DO* is the current vehicle velocity estimate. *O* stands for *odometer*, so *DO* represents a change in odometer value (distance), which is velocity.

We are using two measurement values – velocity based on a wheel encoder output (*DO*) and angular velocity based on a gyroscope output (*q*). The encoder tick/meters-per-second conversion factor is a parameter which may be affected by tire pressure, sensor error, etc. The gyroscope output/angular velocity conversion requires both a center point (the sensor output value which indicates zero angular velocity) and a conversion ratio (a number which converts millivolts of sensor output to degrees/sec). These values vary as the physical properties of the gyroscope change due to environmental changes. The methods by which we make these adjustments will be covered in section 5.6.

Each time that new inertial data is received in the system (at approximately 100Hz) the Kalman filter is run to produce an optimal state estimate given the measurement information. During each iteration, updating the filter is accomplished in four main steps. These are:

- Prediction of covariance matrix of states

$$P_1(k) = A(k-1)P(k-1)A^T(k-1) + Q$$

- Calculation of Kalman gain matrix

$$K(k) = P_1(k)C^T\left(CP_1(k)C^T + R\right)^{-1}$$

- Update of the state estimation

$$\hat{X}(k) = A(k-1)\hat{X}(k-1) + K(k)\left[Y(k) - CA(k-1)\hat{X}(k-1)\right]$$

- Update of covariance matrix of states

$$P(k) = P_1(k) - K(k)CP_1(k)$$

where *$P_1(k)$* is the predicted error covariance matrix at time *k*, *K(k)* is the Kalman gain matrix at time *k*, and *P(k)* is the updated error covariance matrix at time *k*. *R* is the measurement noise matrix, which contains the expected covariance of the measurement data. *Q* is the state noise matrix, which contains the expected covariance of the state matrix values. Essentially, *R* is a measure of confidence in the measurements and *Q* is a measure of confidence in the state estimate. That is, if an extremely precise instrument were being used to quantify a particular measurement variable, the corresponding value in the measurement noise matrix, *R*, would be very low. Conversely, if a very noisy instrument were being used, the corresponding value in *R* would be very high.

In our particular application of the Kalman filter, we used the following matrices for the state transfer matrix, *A(k)*, and the observation matrix, *C*:

$$A(k) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cos(q(k)) & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \sin(q(k)) & 0 & 0 \\ 0 & 0 & 1 & \Delta t & \frac{(\Delta t)^2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{(\Delta t)^2}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Based on our collected data, we were able to determine appropriate values for the measurement noise matrix, *R*, and the state noise matrix, *Q*. The values in *R* were obtained by finding the variance in the measurement data that was collected. Likewise, the values in *Q* were obtained by finding the variance in the state variable data that was collected and processed. We have assumed that the state variables and measurement variables are uncorrelated, which is why there are only diagonal terms in *R* and *Q*.

$$R = \begin{bmatrix} 2.618 & 0 \\ 0 & 0.00251 \end{bmatrix}$$

$$Q = \begin{bmatrix} 7000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 30 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.856 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.0463 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0533 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0304 \end{bmatrix}$$

Before the recursive Kalman filter can operate, we first needed to initialize it appropriately. This consisted of assigning an initial state vector *X(0)* and an initial state covariance matrix *P(0)*. The covariance matrix is initialized to be the same as the state noise matrix, *Q*. The initial state vector must be initialized with some data values that represent absolute measurements, such as position and heading. These data could be known and placed into the state vector in advance (if the vehicle's starting point is know, for example) or the data could be obtained during run time and placed in the matrix before the filter begins operation. In our case, we chose to obtain the data during run time because we did not know our exact starting point. This initial information (initial latitude, longitude and heading) was obtained from the GPS received once it had gotten an initial position fix.

5.4 Rule-based fusion of INS and GPS

Rule-based sensor fusion refers to the integration of Kalman filtered INS data and the raw GPS position and heading information based on a set of rules, thresholds, and weights. These rules are designed to compensate for errors in the GPS data, such as drift and multi-path, while still using the GPS data in an effective manner to update our position estimate.

For example, we found that when the vehicle is stopped, the GPS receiver tends to drift around unpredictably and widely vary the heading value, while remaining close to the current position. If we were doing simple weighting of GPS and INS data, our position estimate would follow the GPS output around and end up with a completely wrong heading value. Thus we added a rule:

If we are nearly stopped (based on GPS or INS velocity) we completely ignore the GPS heading value, that is, we set Ψ_{HEAD} *equal to zero.*

Another simple rule is this:

If the GPS data is 'bad' (too far from current position, too few satellites, bad fix, etc.) then we set Ψ_{POS} *and* Ψ_{HEAD} *equal to 0, effectively running on inertial navigation only.*

However, we must allow for the unlikely event that we have made an error such that we think the GPS data is bad when it is actually good. Thus:

If GPS has had a good satellite fix for a long time, but we have assumed it to be bad, we must have made a mistake, so we set Ψ_{POS} *and* Ψ_{HEAD} *equal to 1.*

The phrase 'a long time' refers to an amount of time that must be determined by experimentation with the system. Obviously, these rules may vary depending on the internal operations of a particular GPS receiver, and parameters must be altered appropriately.

A brief summary of the rules which we found useful, and why they were included, is shown below. The phrases *GPS Good* and *GPS Bad* indicate only whether the GPS data should be considered good or bad based on the status information from the receiver itself. This depends on number of satellites in view, signal strength, etc.

If GPS is good and we have not initialized our position estimate, then initialize the estimated position and heading equal to the GPS position and heading.

Note that this might result in an initial position that is not very accurate due to GPS problems. We assume that the filtering algorithm will correct this in a short amount of time.

Otherwise, if GPS is good, the GPS position is close to the current position estimate, and the GPS velocity is over a specified minimum, then we do a weighted sum of the current estimate and the GPS position and heading.

In general, we want this to be the rule that is always executed, indicating that the current estimate and the GPS positions agree. The GPS velocity is required to have a minimum value because the position output from the GPS receiver tends to drift backwards when the vehicle is not moving.

If GPS has been good for an amount of time, but the GPS position has been too far from our estimated position to fuse the data, we extend the position threshold a small amount and check again.

This rule is to cover the unlikely event that our estimated position has been thrown off for some reason. Although it is not very likely, it is possible for our estimate to get out of position just far enough so that the estimate and the GPS positions are separated by a distance too great to consider valid. In this case, we detect that the GPS data has been 'good' for a significant amount of time, so we assume that our estimate is off and converge our estimate to the GPS position.

If GPS has been good for a very long time, but the GPS position has drifted much too far from the estimate position to fuse in either of the above cases, we reinitialize the estimate position to the GPS position.

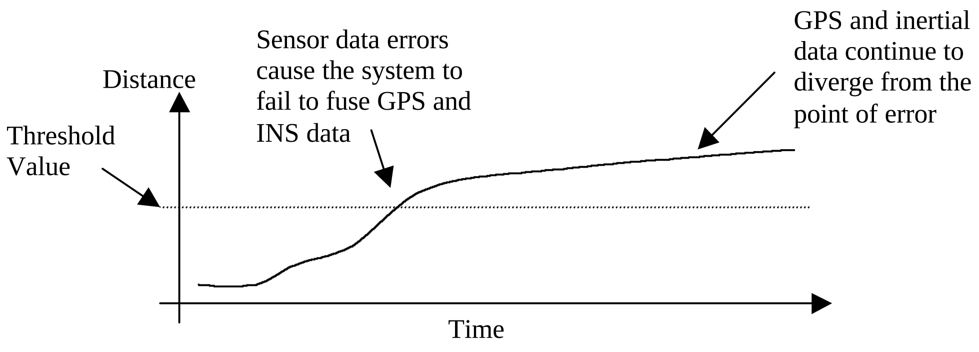
In this case, we assume that some serious (but temporary) failure has occurred in the GPS receiver or the sensors such to cause our estimate to diverge so far from the true position that we can not fuse data. Thus, we decide to simply reinitialize our estimate with the GPS data and start the fusion process over again. After adding the previous rules to our algorithm, this rule never was executed in our sample data. It is possible, however, that this event would occur so this last-resort rule should be in place.

Note that all of the rules use terms such as 'very long time', 'close', and 'too far'. These values must be determined by experimentation and tweaking of parameters until the system obtains satisfactory results. In a rule-based system, such as the one described above, problems arise relating to the use of hard limits in place of terms such as 'close' and 'too far'. If the data just barely crosses from one set, such as 'close', into another set, such as 'far', the behavior of the fusion process can suddenly jump. Another alternative, which replaces hard thresholds with 'fuzzy' thresholds, is the use of fuzzy logic for the fusion process. We describe a fuzzy logic system for sensor fusion in the following section.

5.5 Fuzzy fusion of INS and GPS

As mentioned in the previous section, problems can arise from using 'hard' (or 'crisp') thresholds to define boundaries between data sets such as 'close' and 'far'. Instead, the use of fuzzy logic specifically allows for the ambiguity of these data set boundaries in its operation. Fuzzy logic allows a system to respond with a mixture of behaviors depending on the membership of the input data in various fuzzy sets. Also, rather than using a very complex set of nested if-then-else statements, we can perform sensor fusion in a more structured and easily understood manner.

Consider the case involving the distance between the current position estimate and the GPS position, as described in the rules listed in section 5.3. Using rule based fusion, when the distance is 'small', we assume the GPS position is accurate and we fuse the data appropriately. However, when the distance is 'large' we assume the GPS receiver is suffering from some significant amount of error and we ignore it completely. In general this works, but there may be cases when valid GPS data is considered to be far from the current estimate position and is ignored.



If we use fuzzy logic, however, this will not happen. Suppose we use a simple membership function as shown here:

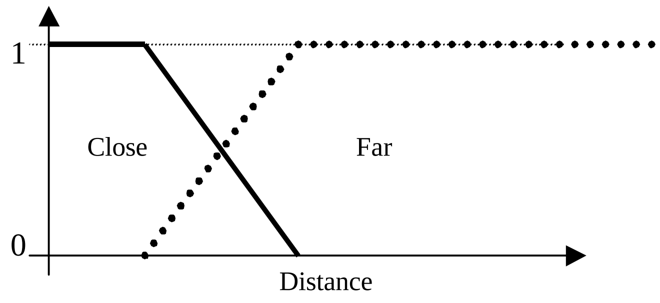


Figure 5.6 – Fuzzy membership functions for distance

Then, the system will make decisions based on the *membership* of a crisp data value in a fuzzy set. In this case, the fuzzy sets are defined to be **Close** and **Far**. Suppose that we have a distance value, d_1. The membership of d_1 in the sets **Close** and **Far** can be determined as shown in Figure 5.7. (Jang et al, 1997; Rao, 1995)

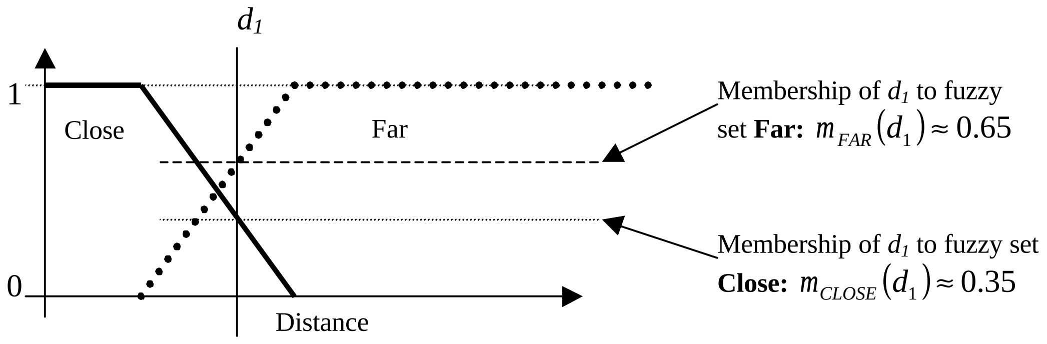


Figure 5.7 – 'Fuzzification' of a crisp data value

Now suppose we also have fuzzy sets **Fast** and **Slow**, which are sets based on the speed value from the GPS receiver. Then we now have a set of fuzzy rules, such as:

If Distance is Close and Speed is Fast then Ψ_{POS} *is Large*

This states that if the GPS distance is close to our estimated position, and the GPS receiver measures a significant vehicle speed, then we assume that the GPS data is 'good' and set the GPS position weight to a large value. Likewise, we may have the following fuzzy rule:

If Distance is Far and Speed is Slow then Ψ_{POS} *is Small*

This statement is essentially the converse of the previous rule. That is, if the GPS distance is far from our estimate position and the GPS receiver measures a low vehicle speed, then we assume the GPS data is 'bad' and set the GPS position weight to a small value. Several more variables must be added to make the system robust, however. For example, a variable counting the length of time that Ψ_{POS} has been small is needed to determine if our estimate is in error.

Once all of the variables have been fuzzified, they must then be processed and defuzzified. A common method of processing fuzzy data is simply to use the MIN function. That is, the combination (fuzzy AND) of two or more membership values is the minimum of the values. Consider the above example where $m_{FAR}(d_1) = 0.65$ and $m_{CLOSE}(d_1) = 0.35$. In addition, suppose that the fuzzification of the current vehicle speed s$_1$ resulted in $m_{FAST}(s_1) = 0.55$ and $m_{SLOW}(s_1) = 0.50$. Consider the previously mentioned rule:

If Distance is Close and Speed is Fast then Ψ_{POS} is Large.

When processing this rule, we consider the membership values of each of the conditionals to determine the output. In this case, where $m_{CLOSE}(d_1) = 0.35$ and $m_{FAST}(s_1) = 0.55$ the output for the rule is the minimum of the two membership functions, resulting in a value of 0.35. The value 0.35 represents this rule's effect on making Ψ_{POS} to be 'Large'. Similarly, consider the rule:

If Distance is Far and Speed is Slow then Ψ_{POS} is Small

Because $m_{FAR}(d_1) = 0.65$ and $m_{SLOW}(s_1) = 0.50$, the output for this rule is 0.50. In this case, the value of 0.50 represents this rule's effect on making Ψ_{POS} to be 'Small'. One can easily see that given these two rules and these membership values, the fuzzy output is going to be more 'Small' than it is 'Large'.

Once the outputs from all such rules have been computed, they must be combined and then defuzzified. One method that is commonly used for defuzzification is the centroid method. Consider the simple membership function for Ψ_{POS} shown below:

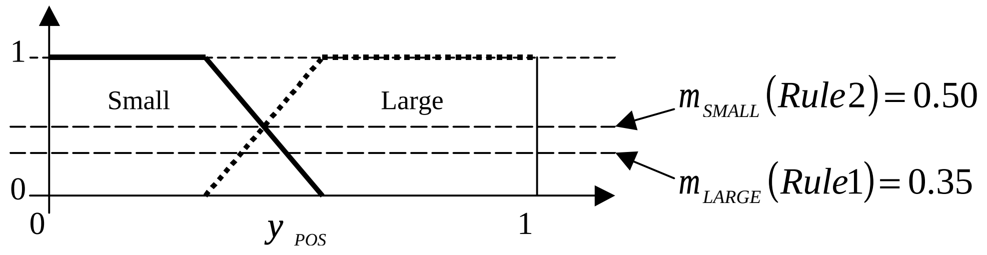


Figure 5.8 – Output membership function for position weighting value

If there were multiple rules that produced a value for m_{SMALL} these would need to be combined at this point using a fuzzy OR operation. In this method of fuzzy processing and defuzzification, the fuzzy OR operation is implemented simply by finding the maximum of all membership values being combined. In this example, however, there is only one rule contributing a m_{SMALL} and only one rule contributing a m_{LARGE}.

Combining the value for m_{SMALL} and the value for m_{LARGE} to produce an output membership function for y_{POS} is done by truncating each of the membership sets at the corresponding membership value, and merging the sets into a single fuzzy output set. This set is the fuzzy output for the variable y_{POS} based on all of the fuzzy rules in the system. For example, the above membership function for y_{POS}, based the given values for m_{SMALL} and m_{LARGE} would become:

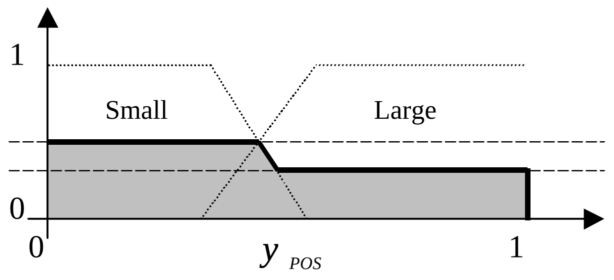


Figure 5.9 – Fuzzy output of the system

After the fuzzy output of the system has been determined, the only remaining task is to 'defuzzify' the output to produce a crisp number that can be used as a weight to fuse the inertial and GPS data values. For our project, we chose to perform defuzzification using the centroid method, which is a method that is very commonly used in fizzy logic applications. The centroid method involved finding the centroid of the shaded area in Figure 5.9. The crisp output of the system is actually the x-axis value corresponding to the centroid of the shaded area.

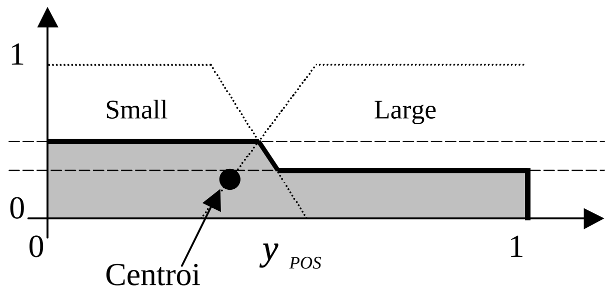


Figure 5.10 – Centroid representing crisp output of the system

In our implementation of fuzzy logic for sensor fusion, we chose to keep the set of variables small and the rules simple. The intention was to explore the possibility of enhancing fusion performance using fuzzy logic techniques. The fuzzy logic system we created used three variables as inputs and produced one output. The three inputs were:

Distance – The distance from the current position estimate to the GPS position

Velocity – The value of the vehicle velocity returned by the GPS receiver

NumNoGood – The number of fusion iterations in which the GPS data has been determined to be 'bad' because it was too far from the current position estimate. (This variable is used for error recovery in the event that our position estimate becomes seriously skewed from the actual vehicle position)

The single output of the fuzzy fusion was *GPSWeight*, the weight of the GPS data for fusion, which ranges from 0 to 1. (The weighting of the inertial data is *1-GPSWeight*). In our approach to creating fuzzy rules, we chose to simply make fuzzy versions of the rules that we had determined using rule-based fusion. All of the rules used in our application are shown here:

if (Distance is SMALL) and (Velocity is FAST) then (GPSWeight is LARGE)

if (Distance is MED) and (Velocity is FAST) then (GPSWeight is MED)

if (Distance is SMALL) and (Velocity is MED) then (GPSWeight is MED)

if (Distance is MED) and (Velocity is MED) then (GPSWeight is SMALL)

if (Distance is SMALL)and (Velocity is SLOW) then (GPSWeight is SMALL)

if (Distance is MED) and (Velocity is SLOW) then (GPSWeight is VERY SMALL)

if (Distance is LARGE) and (NumNoGood is SMALL) then (GPSWeight is VERY SMALL)

if (Distance is LARGE) and (NumNoGood is MED) then (GPSWeight is VERY SMALL)

if (Distance is LARGE) and (NumNoGood is LARGE) then (GPSWeight is VERY LARGE)

5.6 Fusion Parameter Optimization

For an effective and reliable filter implementation, the filter parameters are equally important as the governing system of equations. Without good filter and fusion parameters, the system will never be able to operate as expected. Fusion parameters include values such as those that define the threshold values for the rule-based fusion implementation and those that define the fuzzy boundaries in the fuzzy fusion implementation. Appropriate or inappropriate choice of decision parameters will cause the system to perform better or worse, as measured by some relevant objective or fitness function. (Chambers, 1995) In the case of our system, the fitness function is a measurement of how closely the fused system output matches the true vehicle position. However, since it is nearly impossible to precisely measure the true vehicle position during data collection, the fitness function is actually a measurement of how closely the fused system output is to the road segments on which the vehicle traveled.

There are several choices for possible fitness functions measuring position estimate deviation from the actual roadway positions. Some of these are:

- Largest perpendicular deviation from road segment throughout entire run
- Average perpendicular deviation from road segments for entire run
- Mean-Square perpendicular deviation from road segments for entire run

We chose the last method, the mean-square deviation from road segments for the entire run. This fitness function takes into account all of the data points for the entire run and places more weight on larger error values. That is, one set of parameters may produce a larger average position error, but it may still be consider 'more fit' than another set of parameters that has more large singular position errors.

Here we examine a few methods by which these parameters may be updated to produce a more optimal system behavior. These methods include manual optimization, gradient descent methods, and genetic algorithms.

5.6.1 Manual Optimization

Manual parameter optimization refers to the process of updating the fusion parameters 'by hand'. This is the easiest method of optimization to implement (because there is actually no implementation) but it is also the most tedious method to use for the update process. This method consists of trying a given set of parameters, evaluating the results of that set, and then using some method to update a particular parameter. Typically, the update method involves some sort of intuition about the nature of the system and the type of error evident in the output. This method is often used to arrive at a crude starting point for another more precise method, such as those listed below.

5.6.2 Iterative Hill Climbing

The iterative hill-climbing optimization method is an algorithmic implementation of what manual optimization often involves. Manual optimization has the benefit of human intuition in choosing parameter values, but the iterative hill-climbing algorithm has the benefit of being easily implemented on a computer. Thus, the iterative hill-climbing method can be run automatically for an extended period of time, producing a result with relative ease.

In general, the hill-climbing algorithm starts with an initial guess about the parameter values of the optimum solution. Then, one of the parameters is changed by a suitably chosen (or guessed) increment. If the evaluation function gets better, we keep moving in the same direction by the same increment. If the function gets worse, we undo the last increment and start changing one of the other parameters. This process continues through all of the coordinates until all the coordinates have been tested. We then halve the increment amount, reverse its sign, and start again with the newest set of parameters. The entire process continues until the increments have been halved enough times that the parameters have been determined with the desired accuracy. (Horst et al., 1995) For example, supposing we had only two parameter values to optimize, we might end of with the sequence of parameter values indicated in Figure 5.11 until a suitable goal is achieved.

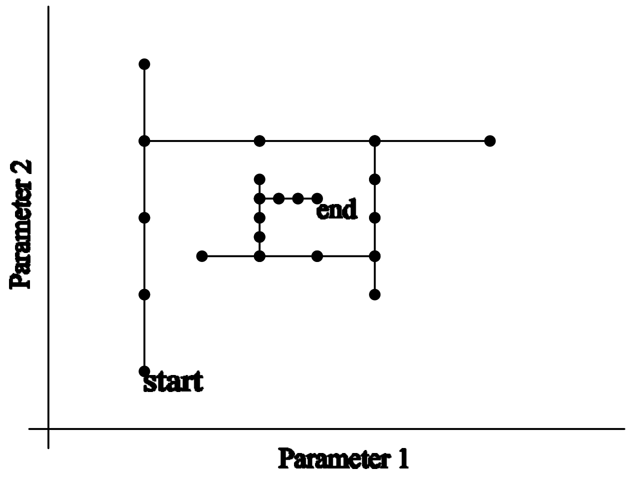


Figure 5.11 – Iterative hill climbing optimization method

5.6.3 Genetic Algorithms

Note that in both the manual and the hill-climbing optimization techniques, the parameter optimization is easily subject to getting stuck in local maxima (or local minima when minimizing the fitness function). This is a problem for many optimization problems, and genetic algorithms have often been used to generate more optimal solutions than those that may be obtained by standard hill-climbing or gradient-descent methods.

Genetic algorithms apply the rules of reproduction, gene crossover, and mutation to ‘pseudo-organisms’ so those organisms can pass beneficial and survival-enhancing traits to new generations. In our case, the pseudo-organisms would represent sets of sensor fusion parameters. Just as a real organism’s characteristics are encoded in DNA, the pseudo-organisms characteristics are encoded in an *electronic genotype*, which mimics the DNA of natural life. The electronic genotype is merely a string of bits that represents a given sequence of sensor fusion parameters.

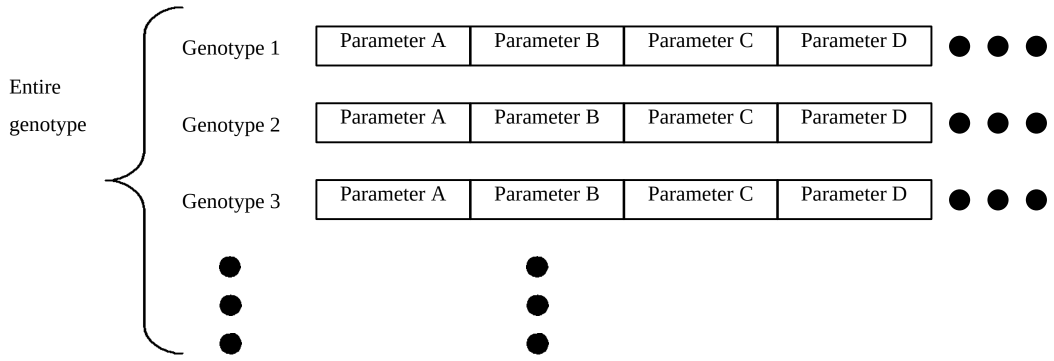


Figure 5.12 – Electronic genotype representation of parameters

Just as reproduction, genetic crossover, and mutation alter a natural DNA sequence, similar operations can be used to alter the electronic genotypes. Additionally, the production of new genotypes and the elimination of unfit genotypes can be used to modify the set of genotypes.

Reproduction is the process of producing one ‘child’ genotype as a result of merging the characteristics of two ‘parent’ genotypes:

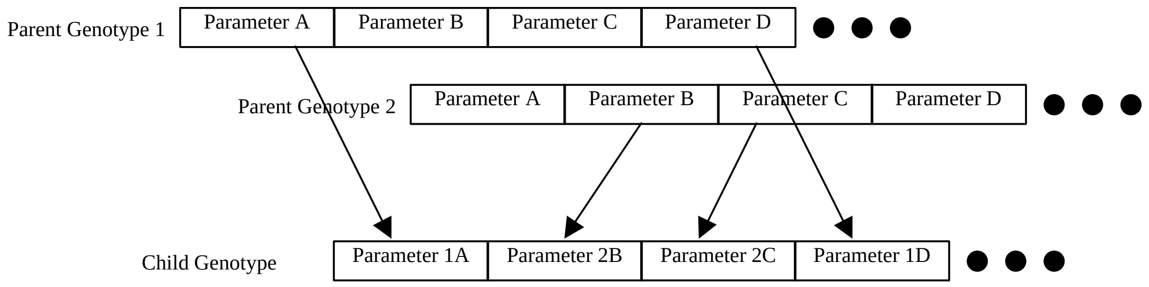


Figure 5.13 – Reproduction of electronic genotypes

Genetic crossover is accomplished by crossing the ‘genes’ of two genotypes, replacing the original two with the altered versions:

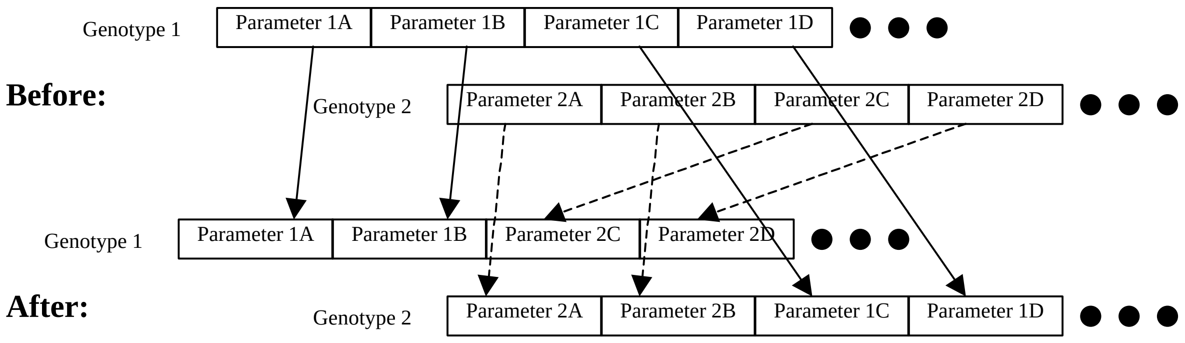


Figure 5.14 – Genetic crossover of electronic genotypes

Mutation simply involves replacing a genotype with a slightly altered version of itself. This process induces small variations in a random manner for the purpose of finding a better parameter set.

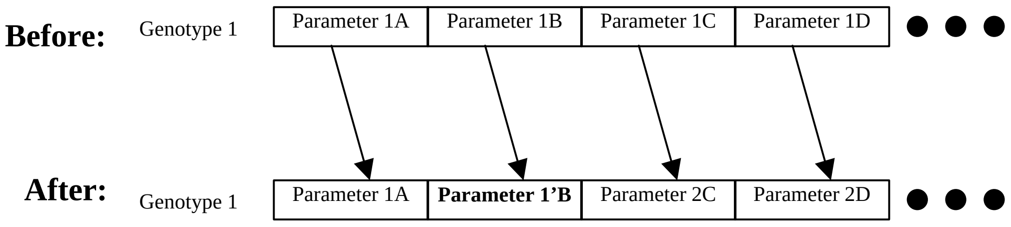


Figure 5.15 – Genetic mutation of electronic genotypes

By using these genetic algorithm techniques, a more ideal parameter set can be obtained, at the expense of the additional time required to test numerous parameter sets that are very ‘unfit’. The genetic algorithm optimization method is not subject to getting stuck in local minima or maxima, however, it is not guaranteed to ever converge to a good solution. (Chambers, 1995)

5.7 Map Matching

Although the fuzzy fusion of GPS and INS data results in very accurate position estimates in most environments, we are often interested in the performance of the system in an urban environment such as downtown New York City. In this type of environment, the GPS signal is often blocked or significant errors are introduced such that even the most robust of filtering algorithms will output an inaccurate position. In addition, the need for accurate position estimates is even greater in a city environment due to the close proximity of adjacent roads. Thus, we introduce a method of increasing the ability of the system by using prior knowledge of a given area.

Map matching is simply a method of using stored information about a region to improve the ability of a position estimation system to handle errors. Essentially, a known map reduces the possible space that a vehicle could occupy, assuming that the vehicle is actually on or near a road. Additionally, by knowing the exact locations of intersections, we can determine if a turn detected by GPS or INS sensors is 'legal'. Simply stated, we use knowledge of a region's roads to confine the position and motion of our vehicle.

There are only a few situations that need to be handled by our algorithm. These situations are: startup/initialization, position between intersections (Figure 5.16), position near intersection and no turn detected (Figure 5.17), position near intersection and turn detected (Figure 5.18), and error/re-initialization. The two cases where an initialization is required (which should be very rare) are the only cases where a search to find the nearest road is required. In these cases, our algorithm needs to determine what our starting point is. This generally involves an exhaustive search of every road in the database, in order to find the road of minimum perpendicular distance to our current position. There are some shortcuts that can be used, but this is still a relatively time consuming process. In general, we want our map-matching algorithm to operate without doing any time consuming processing.

Performing map matching after a valid starting position is known does not need to involve a search of all possible roads. We only have to handle three situations if a valid position is already known – the vehicle could be on a road between intersections, or near an intersection when no turn is detected, or near an intersection when a turn is detected.

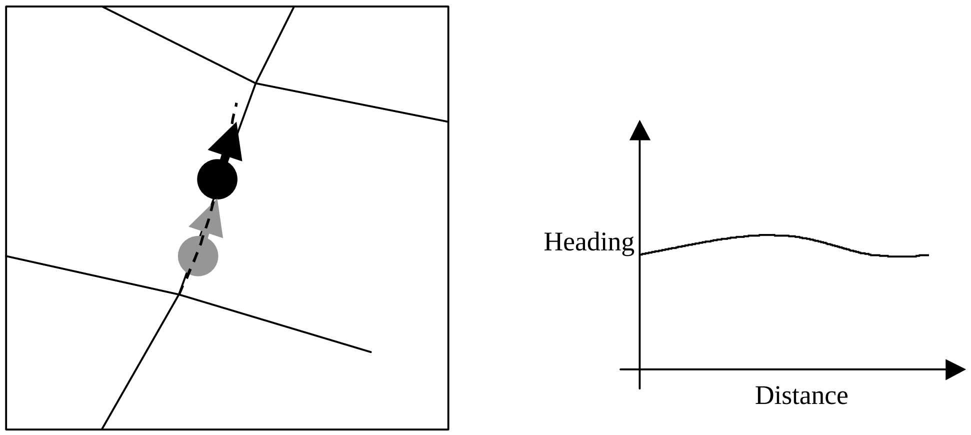


Figure 5.16 – Vehicle heading versus distance between intersections

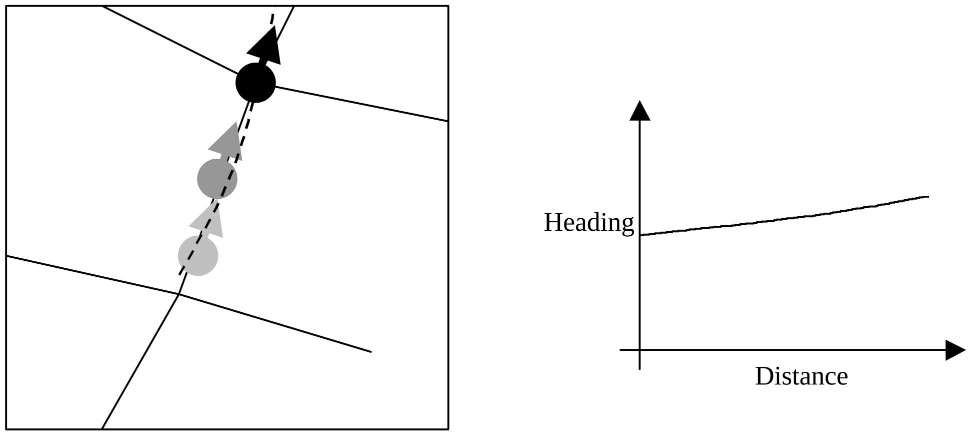


Figure 5.17 – Vehicle heading versus distance when traveling straight through an intersection

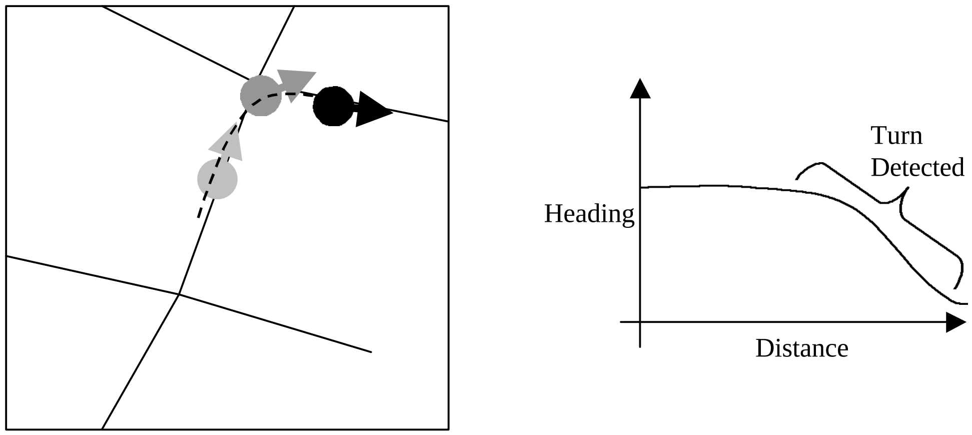


Figure 5.18 – Vehicle heading versus distance when turning in an intersection

To allow for efficient traversal of the roadways, the map must be stored in a particular format in memory. The format developed for this project uses a linked mesh of nodes in memory. Each node represents a specific intersection and contains latitude and longitude information about the intersection. Additionally, each node points to every adjacent node to which there is a connecting roadway. Each node can be represented in computer memory by the following structure:

```
typedef struct _Intersection
{
       int NumConnections;                  // number of adjacent intersections
       _Intersection **Connections;         // pointers to adjacent intersections
       int LatitudeMsec, LongitudeMsec;     // lat/long of this intersection
} Intersection;
```

Where `LatitudeMsec` and `LongitudeMsec` represent the position of the intersection in milliseconds (1 msec longitude equals approximately 1.15 inches at the equator). `Connections` points to a list of pointers to adjacent nodes, of length `NumConnections`. Upon initialization of the map structure, `Connections` can be treated simply as an array of pointers to other nodes.

For general navigation, after the initial position is known, maintaining a position estimate is simply a matter of traversing the linked mesh of nodes. For example, suppose node *A* is connected to nodes *B*, *C*, and *D* (shown in Figure 5.19 below). Then if we estimate our current position to be near node *A*, then we know that we only need to consider movement towards nodes *B*, *C*, and *D*.

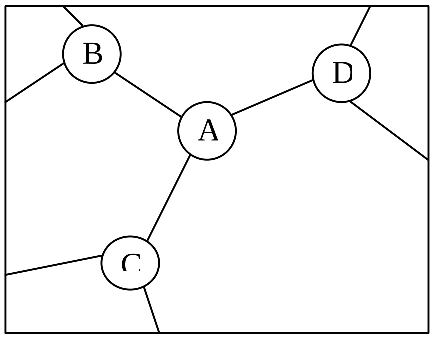


Figure 5.19 – Mesh of nodes (intersections) map representation

Given that the algorithm needs to consider only the adjacent intersections, the problem is reduced to that of choosing the correct one. Suppose that we are traveling from intersection *C* through intersection *A* and on to intersection *B*. The vehicle's position estimate can be maintained from intersection *C* to intersection *A* using any combination of the previously mentioned techniques, utilizing inertial and dead-reckoning sensors, and GPS data where available.

As the vehicle moves near intersection *A*, the algorithm begins to consider which of the adjacent nodes will be visited next. The vehicle's position estimate from intersection *C* to intersection *A* is performed by inertial, dead-reckoning and GPS sensors. These sensors may be prone to drift errors, such that there is an ambiguous region, of radius r_{ar}, near intersection *A*. In this ambiguous region, the map-matching algorithm is not able to determine on which road segment the vehicle is currently moving. The value for r_{ar} is determined by the expected drift in the position estimate between the most distant adjacent nodes in the given map. For example, if the expected drift in position estimate is 4% of the distance traveled, and the most distant adjacent nodes in the map are .1 miles apart, than the ambiguous region should have a radius slightly larger than 4% of .1 miles, or .004 miles (roughly 21 feet).

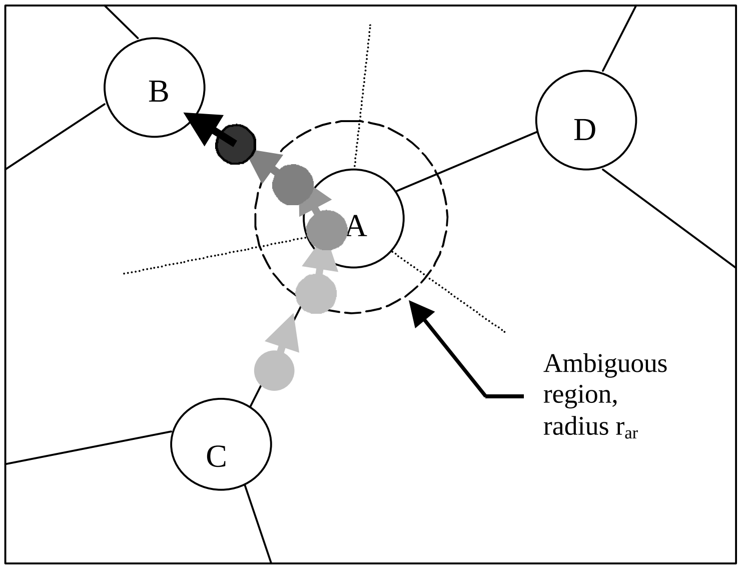


Figure 5.20 – Vehicle traveling through ambiguous region surrounding an intersection

When the vehicle moves into the imaginary circle surrounding an intersection, the algorithm does not know on which road segment the vehicle is moving, but rather it is able to estimate a probability that the vehicle is on any given road segment. The probability that the vehicle is on a given road segment can be determined based on how close the vehicle is to the road segment, relative to how close it is to every other road segment. If the current position of the vehicle is requested, then the algorithm would return the position of the vehicle projected onto the road segment with the highest probability. However, a position estimate for the vehicle is always maintained independent of the road segments while in the ambiguous region. Upon exiting the region, the algorithm would then consider the vehicle to be on the segment with the highest probability at that time instant.

Suppose the vehicle traveled along the path indicated by the heavy dashed line in Figure 5.21. The position estimate of the vehicle as determined without using map matching is indicated by the dotted line. Notice that near intersection *A* the position estimate drifts towards intersection *D*, while the vehicle actually moves on to intersection *B*. This drift may be due to GPS blockage, sensor errors, etc. In this case, the map-matched position estimate would be correctly projected along the line segment from *C* to *A*. Near intersection *A*, the position estimate would briefly snap to the road segment joining intersections *A* and *D*. However, the algorithm would correct this mistake as the vehicle moves away from node *A*, and the position estimate would then correctly snap to the road segment connecting intesections *A* and *B*.

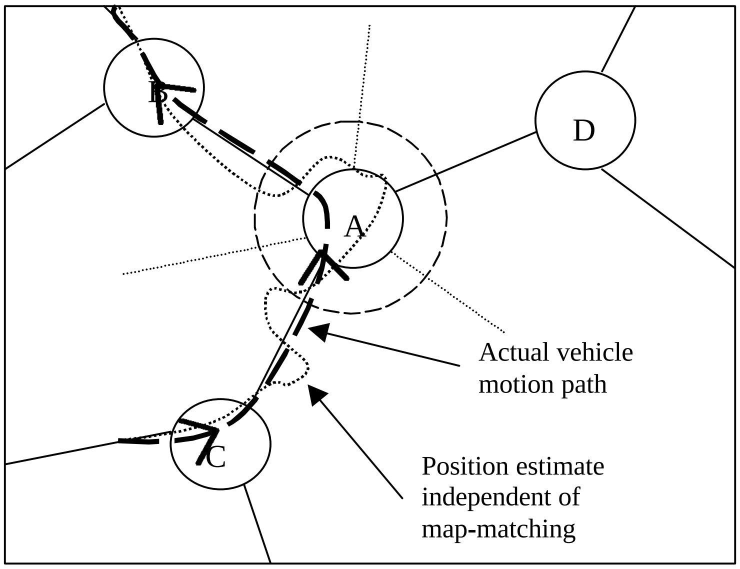

B
D
A
C
Actual vehicle motion path
Position estimate independent of map-matching

Figure 5.21 – Error reduction using ambiguous region in map-matching algorithm

Chapter 6

Software Design

This chapter presents the software development for the sensor fusion project. Without proceeding through all of the intermediate problems and corrections that occurred throughout the development cycle, we will demonstrate how the data was collected and processed and why the code was written the way it was. First, we will present the overall structure and data flow of the software. Then, we will cover both DOS-based and Windows-based programming, with the various considerations for each type of programming.

6.1 Software Structure and Data Flow

The general data flow in the software should closely resemble the flow in the filtering algorithm. Therefore, if we are filtering at both 100 Hz and 1 Hz, our software should reflect this in its design. The core of the filtering algorithm flow is shown here:

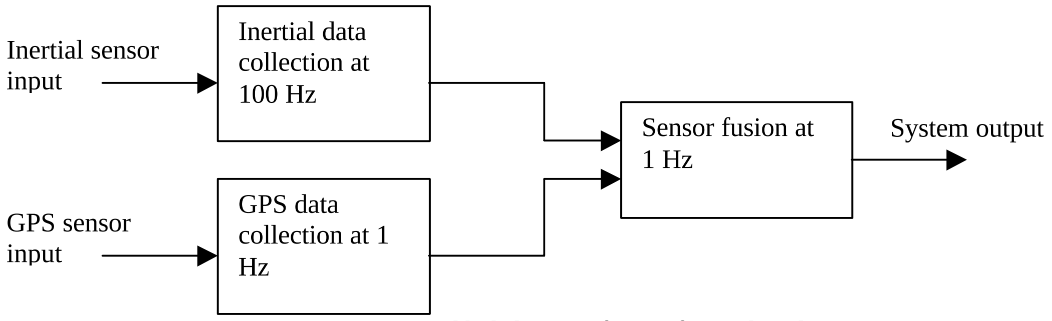


Figure 6.1 – Basic block diagram of sensor fusion algorithm

We desire some more functionality in the software than simple collecting and processing on-line. For example, we desire the ability to dump all of the collected data to the PC's hard drive for later analysis. Similarly, we desire the ability to read the collected data from the hard drive and run our program just as if the collection were happening in real-time. In addition, we want several methods of viewing the output from the software, such as real-time viewing of the raw GPS data, as well as the filtered position data for comparison. These additions complicate our software design, as shown in Figure 6.1.

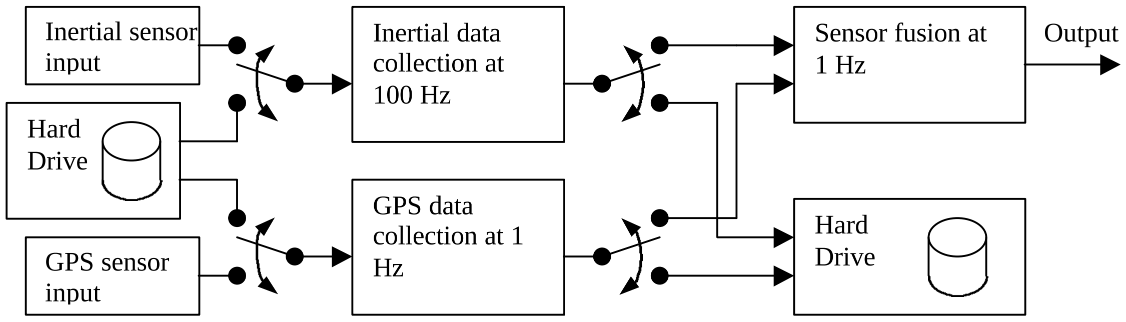


Figure 6.2 – Block diagram of algorithm with alternate input and output locations

Although great effort was spent in making both the DOS and Windows programs capable of all tasks, each operating system is more suited for either real-time data collection or post-collection analysis. DOS does not use the graphical user interface (GUI) that Windows uses, and it does not directly support multi-tasking, so system resources are used on one application exclusively. In addition, DOS gives the programmer easier access to low-level functionality of the system, such as communications ports and direct VGA screen output. This makes DOS much more suited to real-time applications, because we can use the system in a much ‘leaner’ manner by telling the system exactly what to do and when to do it. Conversely, Windows provides high-level abstractions of the system, providing the programmer with many useful functions at the cost of a large a amount of overhead. In addition, Windows ‘protects’ direct access to low-level system devices, such as communications ports, making them more difficult and time-consuming to work with. For these reasons, Windows is more suited for post-collection analysis and visualization of the data. Either system can do both collection and analysis, but DOS is better at real-time collection, and Windows is better at graphical visualization. Thus, this is how we primarily used the software – we collected data with the DOS software, and did post-collection analysis with the Windows software.

6.2 DOS Software Development

For those who have never had the good fortune to do some old-fashioned low-level DOS programming, this section may prove quite interesting. We wish to present the methods by which a DOS program can be made to collect, process, and display sensor data in exactly the manner we desire. We need to make special allowances for a few parts of our program. We wish to collect sensor data at precisely 100 Hz and to collect GPS data at precisely 1 Hz.

After some thought, we realize that the GPS data should not be 'collected' at 1 Hz, but instead we want to 'listen' at the serial communications port (com port) for the GPS data. This is because the GPS receiver is sending data to the PC whenever it wants to, essentially making it an asynchronous process to our program. For this reason, we cannot have our program wait on the com port, because we never know how long we will be waiting. Instead, we would like to ignore the com port, and only service it when new data arrives. Thus, we make use of interrupts. Even if we ignore the com port completely, we are faced with one serious problem. How do we make sure that the main collection/processing loop of the program takes place at exactly 100 Hz? Again, interrupts come to the rescue. The following section covers the implementation of an interrupt driven program in DOS. All programming was done using Borland C++ version 4.5. However, most of the code presented here can be used with other 16-bit compilers with little or no modification.

6.2.1 Programmable Interval Timer

This section covers details regarding the implementation of a very precise method of timing the main loop of our DOS-based program. We explain some nitty-gritty details here, not to be read by the faint-of-heart. Note: the following discussion is based on information that is several years old. The actual chip-level implementation may be different in modern PCs, but they still function in accordance with this discussion.

Every PC has, on its motherboard, an Intel 8253 Programmable Interval Timer (PIT). The PIT chip has three channels, each of which is responsible for a different task. Channel 0 is responsible for updating the system clock. It is usually programmed to generate about 18.2 'ticks' a second. Interrupt 8, which is serviced by the PC, is generated for every tick. Typically, the operating system services the interrupt in order to maintain the current time-of-day estimate. Channel 1 controls DRAM memory refreshing. DRAM memory requires periodic refreshing to prevent information loss, for which this channel is responsible. Channel 2 is connected to the PC speaker and is typically used to generate a square wave so a continuous tone is heard. (Brey, 1987; Roden, 1992; Mazidi and Mazidi, 1995)

The channel of interest for us is channel 0, the channel used to update the system timer. As mentioned above, channel 0 is typically set to tick at roughly 18.2 Hz – much slower than our desired 100 Hz. Therefore, we must reprogram the PIT chip to generate ticks at 100 Hz. The following four statements are all that are required to set the PIT chip frequency:

```
counter=(long)PIT_FREQ/frequency;     // calculate new counter value
outp(0x43,0x34);                      // send command to set new value to the PIT
outp(0x40,counter%256);               // send low byte of new counter value
outp(0x40,counter/256);               // send high byte of new counter value
```

`counter` is the value which is loaded into the PIT chip and serves as a divisor of the PIT frequency, which is `1234DD` hex, or `1193181` decimal. We desire a frequency of 100 Hz, so `counter` is `11931` decimal, or `2E9B` hex. (The function `outp(portid, value)` simply outputs the byte `value` to the hardware port `portid`)

Yet another problem arises when we do this. If we reprogram the PIT chip to generate interrupts at 100 Hz and then make our own interrupt service routine (ISR) to handle the interrupts, we will prevent the system timer from being updated. This is resolved by calling the system timer interrupt manually, at roughly 18.2 Hz. The following code segment shows our entire timer ISR. Note that we do not do any data collection, screen writes, or file I/O in the ISR. We only increment a number, set a flag, and call the system timer routine, if necessary. This is because interrupt service routines should be kept short and simple whenever possible to avoid problems.

```
void __interrupt __far Handler(void){
        hsec++;                       /* increment global 100th second count */
        new_sense=1;                  /* 100th second elapsed, set global flag */
        clock_ticks+=counter;
        if(clock_ticks>=0x10000L){            /* roughly 1/18.2 sec has elapsed */
                clock_ticks-=0x10000L;
                (*BIOSTimerHandler)();        /* call the system timer handler */
        }
        else outp(0x20,0x20);                 /* clear interrupt and continue */
}
```

The keywords `__interrupt` and `__far` tell the compiler explicitly how to handle the function. The keyword `__interrupt` tells the compiler that this function is meant to be called as an ISR, because the compiler must add special assembly calls to properly return from an ISR. The keyword `__far` tells the compiler that this function may be called from outside of the current code segment. These keywords are specific to Borland C++ version 4.5, and may need to be changed for other compilers.

Following are the code segments that initialize and clean up the system timer ISR, along with the global declaration for the variable `BIOSTimerHandler`, which is a pointer to hold the address of the system timer ISR. `SetTimer` is called at the beginning of the program to setup the PIT chip and new timer ISR jump vector, and `CleanUpTimer` is called to reset the PIT chip and old timer ISR *jump vector*. A jump vector refers to a place in memory that holds the address of an ISR. Every time any interrupt is generated, the system looks at a predefined place in memory to find the address of the ISR, or jump vector.

```
#define TIMERINTR      0x08           /* timer interrupt jump vector location */
#define PIT_FREQ       0x1234DDL      /* frequency of PIT chip */

void __interrupt(__far *BIOSTimerHandler)(void);  /* old timer ISR pointer */

void SetTimer(void interrupt(__far *TimerHandler)(void), int frequency){
 clock_ticks=0;                       /* initialize clock counter */
 counter=(long)PIT_FREQ/frequency;    /* determine PIT divisor */
 BIOSTimerHandler=getvect(TIMERINTR); /* get address of old timer handler */
 setvect(TIMERINTR,TimerHandler);     /* set 100 Hz timer handler jump vector */
 outp(0x43,0x34);                     /* set PIT chip frequency */
 outp(0x40,counter%256);              /* output low byte of counter value */
 outp(0x40,counter/256);              /* output high byte of counter value */
}

void CleanUpTimer(void){
 outp(0x43,0x34);                     /* reset PIT chip frequency */
 /* outputting 0x0000 to the PIT chip sets the maximum counter value of 0x10000 */
 outp(0x40,0x00);                     /* send low byte of counter value */
 outp(0x40,0x00);                     /* send high byte of counter value */
 setvect(TIMERINTR,BIOSTimerHandler); /* set old timer handler jump vector */
}
```

The following table contains a list of many of the important system interrupts that can be called, as listed in *Programmer's Problem Solver* (Jourdain, 1992).

Vector	Function	Vector	Function
00h	Divide by zero error	14h	Com port driver
01h	Processor single step	15h	Network & miscellaneous services
02h	Nonmaskable interrupt	16h	Keyboard buffer access
03h	Processor break point	17h	Printer access
04h	Processor overflow	18h	ROM BASIC
05h	Print screen	19h	System restart
06h	Unused	1Ah	Timer & real-time clock access
07h	Unused	1Bh	Ctrl-Break handler
08h	Timer (time-of-day count)	1Ch	User defined timer tick routine
09h	Keyboard	1Dh	Video parameter table
0Ah	Reserved	1Eh	Disk parameter table
0Bh	COM2	1Fh	Graphics character table
0Ch	COM1	20h	Program terminate
0Dh	Hard disk drive controller	21h	DOS functions
0Eh	Diskette drive controller	22h	Terminate vector
0Fh	Printer controller	23h	Ctrl-C vector
10h	Video driver	24h	Critical-error vector
11h	Equipment configuration check	25h	Absolute disk sector read
12h	Memory size check	26h	Absolute disk sector write
13h	Disk I/O (PC/XT)	27h	Terminate and stay resident

6.2.2 Com Port Interrupt Programming

Activating the ISR for com port programming is quite similar to activating the ISR for the system timer. The functions `SetCom1` and `CleanUpCom1` are analogous to `SetTimer` and `CleanUpTimer`, described in the section above. These functions are shown here:

```
#define COM1INTR              0x0C          /* com port ISR jump vector number */
#define GPS_PORT_DATA         0x3F8         /* port where data arrives */
#define GPS_PORT_STATUS       0x3FD         /* line status register */
#define GPS_PORT_INTR         0x3F9         /* interrupt enable register */
#define GPS_PORT_CONT         0x3FC         /* modem control register */

void SetCom1(void interrupt(__far *ComHandler)(void)){
  BIOSCom1Handler=getvect(COM1INTR);  /* get old com ISR handler */
  setvect(COM1INTR,ComHandler);              /* set new com ISR handler */
  gps_port_intr_set=inportb(GPS_PORT_INTR);  /* get initial port settings */
  outportb(GPS_PORT_CONT,11);         /* assert DTR and RTS signals */
  outportb(GPS_PORT_INTR,1);          /* set to interrupt on data received */
  inportb(GPS_PORT_DATA);             /* clear any pending data on com port */
  outp(0x21,inportb(0x21)&0xEF);
}

void CleanUpCom1(void){
  outportb(GPS_PORT_INTR,gps_port_intr_set); /* return initial port settings */
  setvect(COM1INTR,BIOSCom1Handler);         /* set old com ISR handler */
}
```

Note that `SetCom1` does more than just set the jump vector for the ISR. It also must do some setup specifically for the com port. Most importantly, it sets the com port to generate an interrupt (which is then handled by our ISR) whenever a data byte has arrived on the port. Likewise, the ISR for the com port must do more than set a flag. At any given time, our program is unaware of where we are in the message from the GPS receiver. For this reason, the interrupt handler must be able to synchronize itself with the GPS message.

Synchronization is accomplished by finding a known sequence of characters in the message, then orienting the rest of the message by this sequence. We implement this by means of input states. Initially, we are in state 0 waiting for the first character in the sequence, which is a `@` character. Once we get the character, we advance to state 1, where we again wait for a `@` character. Then we wait for a `B` in the same way. If the sequence fails to match the expected `@@B` at any point, we return to state 0. After the start sequence has been detected, we assume that we are synchronized with the GPS receiver and collect the remainder of the fixed-length message. Once the entire message has been collected, we set a flag indicating that the main program should interpret the message. The state diagram for the com input is shown in Figure 6.3.

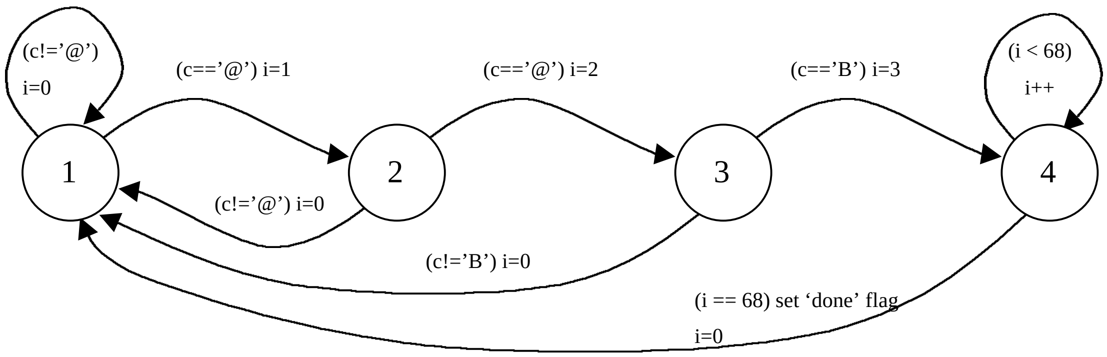


Figure 6.3 – Serial com state diagram for GPS data input

Note that the sequence detection and message gathering must be done in the com port ISR, instead of simply setting a flag each time a character is received and having the main program handle it. This is because our main program is running at 100 Hz due to the timer interrupt, while we receive data bytes at nearly 10 times that rate. However, the com port input is in bursts of 68 characters at 1 Hz. This means that our program has time to run at least one 100 Hz loop and interpret the received message before the next message is begun.

Despite this long discussion of the ISR, the actual implementation is quite simple. The com port ISR code is shown here:

```
void __interrupt __far Com1Handler(void){
  gps_char=inportb(GPS_PORT_DATA);    /* retrieve data from com port */
  if(!new_gps){
    switch(gps_report_idx){   /* depending on what state we’re in... */
    case 0:
    case 1:                           /* waiting for message beginning (@@) */
      if(gps_char=='@'){
        gps_report[gps_report_idx]='@';
        gps_report_idx++;
      }
      break;
    case 2:                           /* waiting for ‘B’ character */
      if(gps_char!='B') gps_report_idx=0;
      else{
        gps_report[gps_report_idx]='B';
        gps_report_idx++;
      }
      break;
    default:                  /* waiting for remainder of message */
      gps_report[gps_report_idx]=gps_char;
      gps_report_idx++;
      break;
    }
    if(gps_report_idx==68){   /* if message complete... */
      gps_report_idx=0;       /* reset message index for next message */
      new_gps=1;              /* set flag to indicate complete message */
    }
  }
  outp(0x20,0x20);                    /* clear interrupt */
}
```

6.2.3 VGA Output

In the interest of keeping the software as simple and fast as possible, while still providing some graphical feedback, we used the video mode known as Mode 13. Many older DOS based video games use Mode 13 because it is very simple and easy to implement on any PC. We implemented a very low-level pixel plot routine, and from it built routines to plot lines, boxes, and text. This section will cover only the basics of Mode 13 operation.

Mode 13 is a 320x200 resolution graphics video mode which every graphics card supports. Video memory for this mode is represented as a single contiguous segment of memory starting at location `A0000` hex. Each pixel is a single byte in the array, so the entire screen is 64000 (320x200) bytes in size. Colors are produced by maintaining a palette of 256 color values. The palette is a series of 256 three-byte triplets defining the red, green, and blue components for each entry. Thus, the mode is capable of representing 16.8 million (256^3) different colors, but only 256 at a time. (Mazidi and Mazidi, 1995)

Initializing the video mode to Mode 13 is very simple. We need only to put the value `13` hex into the `ax` CPU register and generate the BIOS interrupt `10` hex. This is shown in the function `set_vga`:

```
void set_vga(void){
        asm{
                pusha           // store the A register so we don’t mess it up
                mov ax,0x0013   // load 0x13 into the A register (VGA mode)
                int 0x10        // call the BIOS interrupt to set mode 0x13
                popa            // restore the A register
        }
}
```

Resetting the mode to the text mode, which we are used to, is just as simple. We just put `03` hex into the `ax` register and generate the same interrupt. This is shown in the function `set_text`:

```
void set_text(void){
        asm{
                pusha           // store the A register so we don’t mess it up
                mov ax,0x0003   // load 0x03 into the A register (text mode)
                int 0x10        // call the BIOS interrupt to set mode 0x03
                popa            // restore the A register
        }
}
```

After the mode has been initialized, plotting a pixel is as simple as writing the color byte to the appropriate position in memory. The video memory array starts at the upper-left corner of the screen, runs across the screen line-by-line, and ends at the lower-right corner of the screen. (LaMothe et al., 1994)

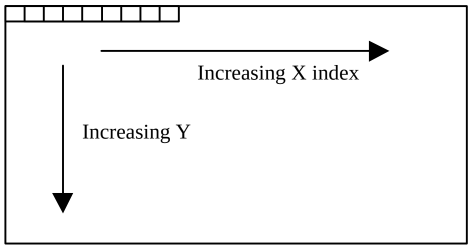


Figure 6.4 – VGA screen memory orientation

So we orient our coordinate system such that the x-axis starts on the left and points right, while our y-axis starts at the top and points down. Thus, if we want to plot a point at the position `(x,y)`, we simply write to the array at offset `[y*320 + x]`. The routine `blit_bit` plots a pixel of value `color` to the screen at position `(x,y)`. A pointer to the screen memory is passed in `vscreen`, which is always `A0000` hex. A common way to calculate the offset into the array is to use bit-shifts, rather than a multiplication. Note that `y*320` is equal to `y*256+y*64`, which may also be written as `y<<8+y<<6`. The rationale for this is that two bit-shifts and an addition are faster than a multiplication. The code for `blit_bit` is shown here:

```
/* draw a single colored pixel into the video memory at location vscreen */
void blit_bit(int x, int y, unsigned int color, unsigned char far *vscreen)
{
       vscreen[(y<<8)+(y<<6)+x]=color;      /* set pixel color value in memory */
}
```

Based on this simple function for setting a pixel value in video memory, we can easily make functions that efficiently draw horizontal or vertical lines into video memory.

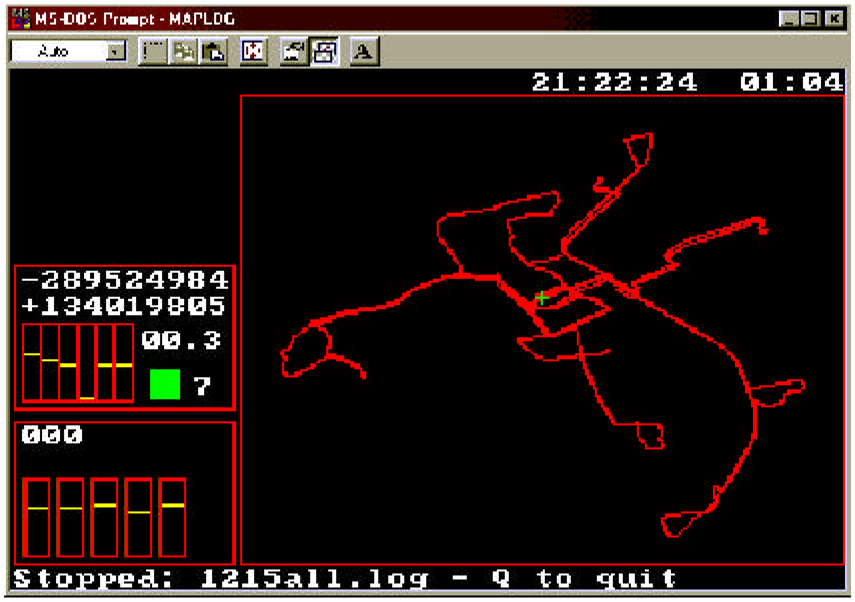
```
/* draw a colored horizontal line into the video memory location vscreen */
void h_line(int x1,int x2,int y,unsigned int color,unsigned char far *vscreen){
       _fmemset((char*)(vscreen+((y<<8)+(y<<6))+x1),color,x2-x1+1);
}

/* draw a colored vertical line into the video memory location vscreen */
void v_line(int y1,int y2,int x,unsigned int color,unsigned char far *vscreen){
       unsigned int line_offset;
       int index;
       line_offset=((y1<<8)+(y1<<6))+x;      /* pixel location in memory */
       for(index=0;index<=(y2-y1);index++){  /* for each row in the line */
              vscreen[line_offset]=color;   /* set pixel value in memory */
              line_offset+=320;             /* increment to next line */
       }
}
```

Drawing lines with arbitrary slopes in an efficient manner is a little more complicated. As with the bit_blit function, we would like our line-drawing function to make no multiplies or divides, for efficiency purposes. To accomplish this, we use a line-drawing technique known as Bresenham's Algorithm (Abrash, 1997).

```
/* draw a colored line from point p1 to p2 into video memory location vscreen */
void line(int x1,int y1,int x2,int y2,
                      unsigned int color,unsigned char far *vscreen){
 int i,d1x,d1y,d2x,d2y,u,s,v,m,n;
 u=x2-x1;      /* overall 'run' of the line */
 v=y2-y1;      /* overall 'rise' of the line */
 d1x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d1y=sgn(v);   /* line left-to-right or right-to-left */
 d2x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d2y=0;
 m=abs(u);
 n=abs(v);
 if(m<=n){     /* if |slope| < 1 */
  d2x=0;
  d2y=sgn(v);
  m=abs(v);
  n=abs(u);
 }
 s=m/2;
 for(i=0;i<(m+1);i++){
  /* plot single pixel on the screen (only if within screen bounds) */
  if(y1>=0 && y1<=199 && x1>=0 && x1 <= 319) vscreen[(y1<<8)+(y1<<6)+x1]=color;
  s += n;
  /* increment to next pixel location */
  if(s>=m){
        s -= m;
        x1 += d1x;
        y1 += d1y;
  }
  else{
        x1 += d2x;
        y1 += d2y;
  }
 }
}
```

6.2.4 Data Visualization

By building several functions based on the simple `blit_bit` function, we are able to produce a somewhat crude, but useful, graphical interface for our program. We output some status information, such as BIOS time, running time, and file name, as well as the sensor outputs and much of the GPS message in graphical form. If we properly define a viewing window in latitude/longitude milliseconds, we can view the raw GPS message position output in real-time. We can see the number of satellites visible, the signal strengths, the GPS position, etc. This allows us to be immediately aware of an error, such as loss of satellite fix, so that we can try to find the source of the problem. A screen shot of the data acquisition program, Maplog, is shown below in Figure 6.5. An overlay of the raw collected GPS data onto a map of Blacksburg, VA is shown in Figure 6.6.



```
21:22:24  01:04
-289524984
+134019805
00.3
7
000
Stopped: 1215all.log - Q to quit
```

Figure 6.5 – Maplog data acquisition program showing plot of GPS data

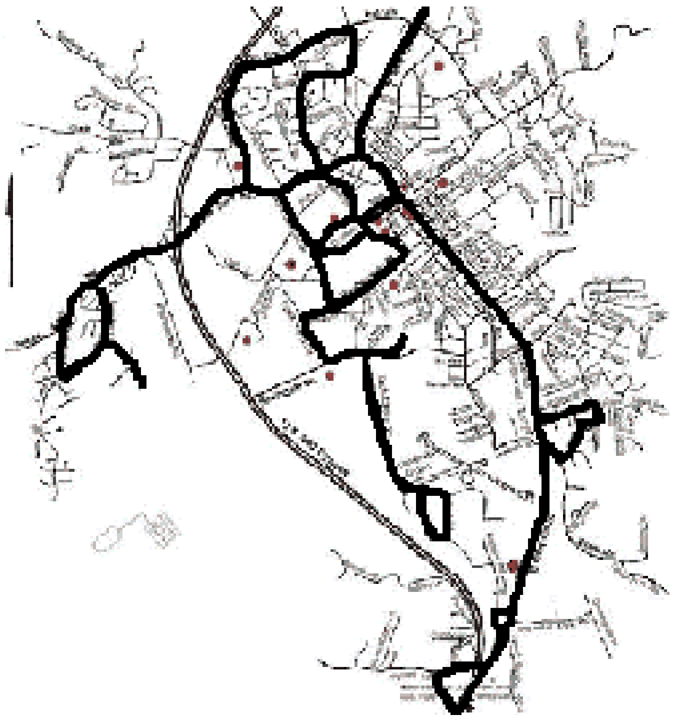


Figure 6.6 – Collected raw GPS data overlaid onto map of Blacksburg, VA

6.3 Windows Software Development

After we developed the DOS based program, we desired a better way to do post-processing and visualization of the data. We chose the Windows 95 platform because of the native graphical user interface (GUI) and multi-threading capabilities. Also, by using a programming application such as Microsoft Visual C++, creating an application's structure and interface is relatively simple. In this chapter, we will cover some of the basics of Windows 95 (Win95) programming, with particular emphasis on multi-threading and utilizing the GUI for data visualization. In addition, the multimedia timer and serial communications, which apply to a data-collection system, will be covered in detail. All programming was done with Microsoft Visual C++ 5.0 (MSVC) and utilizes the Microsoft Foundation Class (MFC), so any examples shown may be specific to this programming environment.

6.3.1 Multi-Threading

An important difference between DOS-based and Win95-based programming is the native support of multi-threading under Win95. Multi-threading is the ability of the operating system to run two pieces of code simultaneously. In most PC systems, the central processing unit (CPU) is not capable of running two pieces of code at exactly the same time. Instead, the operating system performs *time slicing*, where CPU time is divided up and each section of code (sometimes called a *thread* or *process*, depending on the context) is given a portion of the CPU's time. Thus, each thread thinks it has the whole machine to itself, when it is really sharing the CPU time with several other threads or processes. There are many details of multi-threading that occur on the operating system level, such as virtual address spaces and register swapping, that are beyond the scope of this thesis and therefore will not be discussed.

Multi-threading not only allows other applications to run on a system at the same time our application is running, but it also allows us to use multiple threads within a single application. In Win95 programming, it is quite common to use a single *user-interface* thread and a *worker* thread. The main purpose of this is to allow the worker thread to function even while the user-interface thread is busy. (Bennett et al, 1997) For example, whenever a window is 'dragged' across the screen, the thread that controls that window ‘blocks’ until the window is released. Of course, the thread does not halt execution; it is instead busily executing the code responsible for dragging the window. However, this means that the thread is not able to do anything else while the window is being moved. In many applications, such as a simple word processor, this is not a problem; the program only does anything in response to a user action anyway. If we are doing real-time I/O or processing, however, we do not want our processing thread to halt when the user does something. Hence, we separate the tasks into the user-interface portion, and the data collecting and processing portions. Thus, when the user manipulates the user-interface, the processing thread can continue as if it had complete control of the CPU.

6.3.2 Multimedia Timer and Real-time I/O

There are two methods of generating timed function calls in Win95. One method involves using a simple message handler provided by the MFC. The `CWnd` class provides a function called `OnTimer()`, which can be used by the programmer to generate function calls at fixed intervals. Many MFC classes inherit the `CWnd` class, including windows, dialog boxes, buttons and many more. This makes it very easy and convenient to make use of this timing capability. However, after working with this timing method for quite a while, we found that it when set to run at 1 Hz, it actually runs slightly slower than 1 Hz. It runs even slower as the desired frequency is increased. Since we wanted to run at 100 Hz, this problem was unacceptable. Attempting to run at 100 Hz resulted in running at a mere 17 Hz! Although this timer is simple to implement, we turned to something a little more satisfactory – the multimedia timer.

The multimedia timer is not as simple to use as the `CWnd` timer is, but it provides higher accuracy at much higher speeds. The multimedia timer uses a separate thread to generate timed function calls in the application. The handling of the thread is done internal to the multimedia function calls, but the programmer must properly set-up and shut-down the timer, using the multimedia functions `timeSetEvent()` and `timeKillEvent()`, respectively. Also, the multimedia library, `mmsystem.lib`, must be linked into the application and the header file, `mmsystem.h`, included in the source code. Despite the additional trouble, the multimedia timer can generate function calls at a steady 100 Hz, so we were able to capture data at the desired sampling rate. (For more information about the multimedia timer functions, see ‘timeSetEvent’ in the MSVC help files)

6.3.3 Serial Communications

For one used to serial communications in a DOS environment, serial communications under Windows 95 is a very tricky thing. The designers of Windows 95 have managed to abstract serial I/O to be a form of file I/O, where the serial port is a sort of file. A very simple explanation of the process is as follows: When properly set up, Windows 95 maintains input and output serial buffers. From the application programmer’s point-of-view, writing to the serial port is easy. As long as we do not overrun the output buffer, we can send any amount of data to the port (abstracted as a file-write process) at any point in the program. Serial input, however, occurs asynchronously to the program. That is, data is received at unknown times and at unknown rates.

In Win95, unlike DOS, the program does not need to be alerted every time a byte is received on the serial port. Low level handlers in the operating system fill the input buffers with a limited amount of data automatically. Our program is then alerted at some point, and we read some or all of the data from the input buffer (abstracted as a file read). Because an unknown amount of data is arriving and our program may have to wait an indeterminate amount of time for a read or write operation, we make use of a mode of I/O known as *overlapped* I/O. In this mode, operations that take a long time will return with a specific error code indicating that the operation is incomplete. If our program wants to read data from the com port, we use overlapped I/O, rather than waiting for that amount of data to arrive at the port. This allows the program to continue executing while an I/O operation is completing.

Implementing overlapped I/O is very complex, and the details of implementation are beyond the scope of this thesis. In order to get overlapped I/O working well, we ended up converting some of Microsoft’s example code to suit our purpose. It would have been very difficult otherwise. If you are trying to figure out overlapped I/O for

Windows 95/NT, we suggest referring to the Microsoft examples or our code in the appendix to this thesis.

6.3.4 File I/O

File I/O under Win95 using the MFC is very similar to old-fashioned C-style library functions such as `fread` and `fwrite`. Using the `CFile` class from the MFC provides the member functions `Read` and `Write` for performing binary file reads and writes, almost exactly like `fread` and `fwrite`. The problem for us occurred when we tried to take data collected from our DOS program and read it into our Win95 program. The problem has to do the *packing alignment* for our data structures. Packing alignment refers to the alignment of structure data members in memory. Under the DOS programming environment, compilers typically pack data members on the *1-byte (one-byte) boundary*. Under a 32-bit OS, such as Win95 or UNIX, compilers typically pack data members on the *4-byte boundary*. For example, note the following structure:

```
struct mystructure
{
        char a;
        long b;
};
```

Using a typical DOS compiler, this structure would occupy five bytes of memory – 1 byte for the character and 4 for the long integer. Using a Win95 compiler, however, this structure would occupy eight bytes – 4 for the character and 4 for the long integer. The character itself is only one of the four memory bytes, but the compiler decided to position the 32-bit long integer evenly on a 4-byte chunk of memory.

Usually, a programmer does not notice this difference when porting code from a 16-bit to a 32-bit platform. In this case however, a problem was introduced because of the cross-use of data files written on one platform and read on the other. When we wrote the data structures to disk, we would call a command such as:

```
fwrite(&mydata, sizeof(mystructure), 1, myfile);
```

where `mydata` is of type `mystructure` and `myfile` is a file stream pointer. Then we would like to read the data in the Win95 program using a function call like:

```
myfile.Read(&mydata, sizeof(mystructure));
```

where `myfile` is an instantiation of the class `CFile`. Since the `sizeof()` operator returns the number of actual bytes of memory consumed by the structure, we would write 5 bytes, and read 8 bytes. In addition, the alignment within the structure would be off, and the data would be garbage.

The solution to all of this turned out to be quite simple. Using a preprocessor directive called `#pragma pack`, we are able to force MSVC to pack the structures to the 1-byte boundary. The usage is as follows:

```
#pragma pack(push, identifier, 1)
typedef struct _mystructure
{
        char a;
        int b;
} mystructure;
#pragma pack(pop, identifier)
```

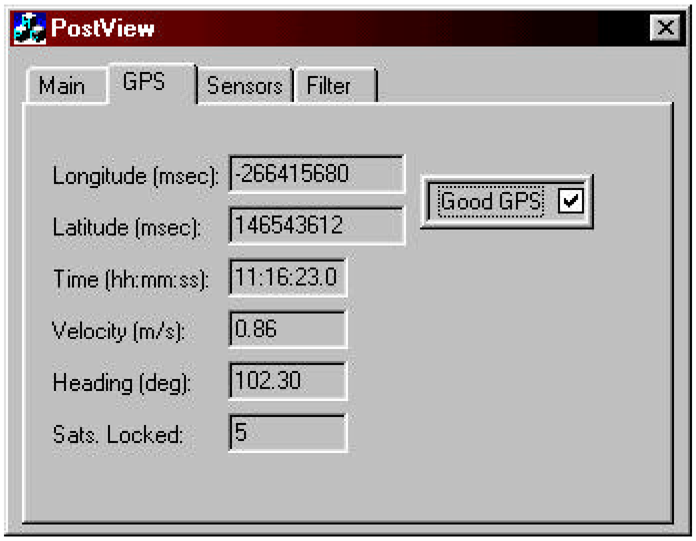
where *identifier* is any unique identifier so that the *push* and *pop* statements can be matched. This causes the compiler to pack only the `mystructure` structure to the 1-byte boundary, allowing us to read older files with no substantial code modification.

6.3.5 Filter Implementation

The actual implementation of the filter in the program was done using a set of matrix functions built on standard ANSI C routines, which were taken primarily from *Numerical Recipes in C* (Press et al., 1988). We used ANSI C to aid in portability in case the need should ever arise to use the software on another platform. For example, one application may involve using the filter in a DSP or embedded microprocessor in a portable system, which would most likely require ANSI C language compliance to compile.

6.3.6 Data Visualization

Data visualization refers to the ability to view the data streams going into and out of the program. In this case, we would like to be able to see the raw input data as it is processed, which includes the sensor values and GPS data values. Relevant GPS data includes the current unfiltered latitude and longitude, heading, velocity, time and the number of satellites locked. The simple graphical display for this data is shown in Figure 6.7.


PostView

Main | GPS | Sensors | Filter

Longitude (msec): -266415680

Latitude (msec): 146543612

Time (hh:mm:ss): 11:16:23.0

Velocity (m/s): 0.86

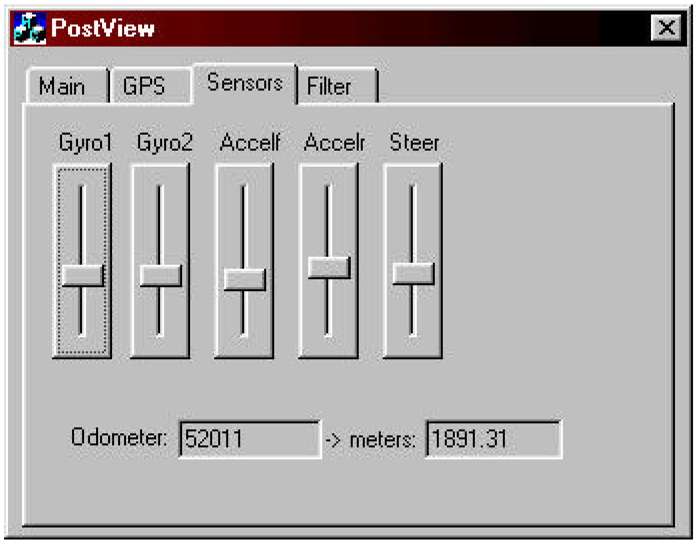
Heading (deg): 102.30

Sats. Locked: 5

Good GPS ✓

Figure 6.7 – GPS information view of post-processing application

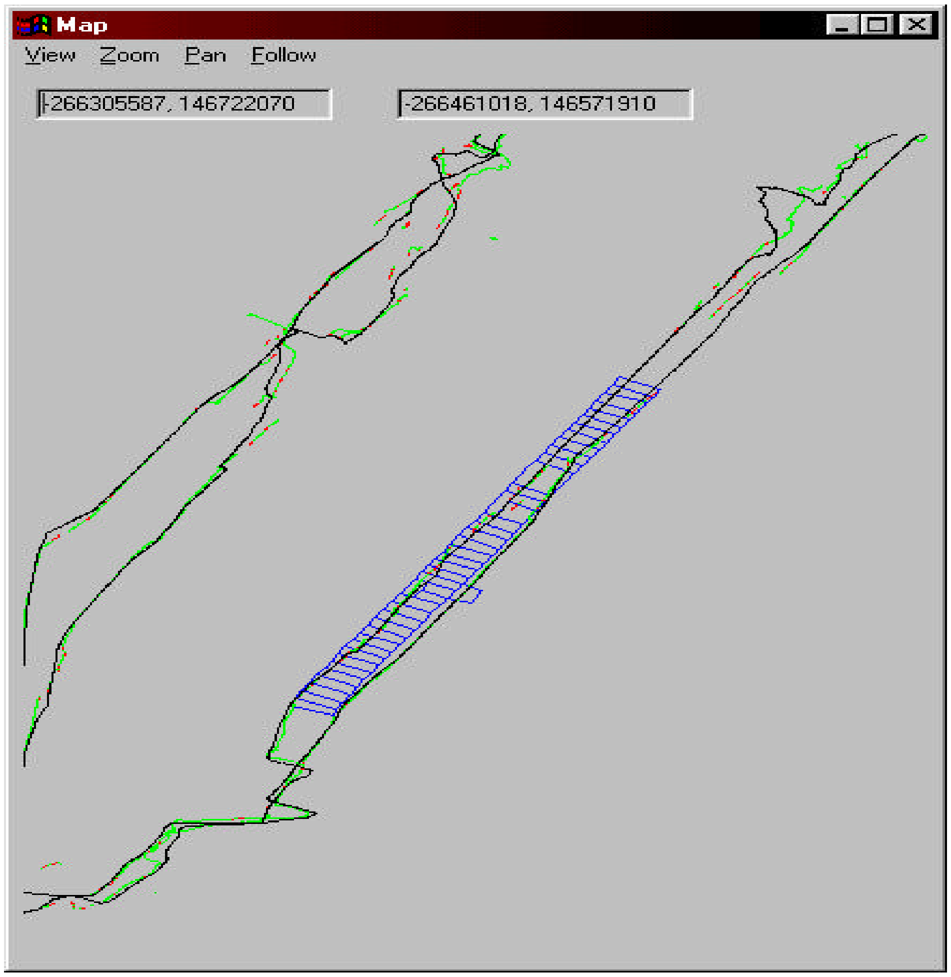
The raw sensor values include the measurement values for the steering potentiometer, both accelerometers and both gyroscopes. A screen shot of this display window is in Figure 6.8.



Odometer: 52011 -> meters: 1891.31

Figure 6.8 – Sensor data view of post-processing application

As with the DOS data collection program, we would like to be able to see the latitude and longitude plotted on some sort of display graphically. Additionally, we would like to see both the raw GPS input position and the filtered output position estimate from the system. All of this data is shown in a separate window, which can be resized as desired. Additionally, the post processing program can read in a file which defines the road segments as discussed in the map-matching section of this thesis, and it can display these road segments, over which the vehicle position is plotted. Although difficult to distinguish in printed form, both the collected raw GPS data and the filtered output position are plotted simultaneously for comparison. A screen capture of this window is shown in Figure 6.9.



-266305587, 146722070

-266461018, 146571910

Figure 6.9 – Graphical output showing raw GPS data, filtered output and current map

Chapter 7

Results

The results for this project have been very promising. We have shown that with a system using only a few inexpensive inertial sensors along with a common GPS receiver, we are able to produce a much better position estimate output than is output from the GPS receiver alone. This is true both for areas of good GPS coverage, where the inertial data augments the position estimate capabilities, and for areas of bad or intermittent GPS coverage, where the inertial data serves as the only means of position estimation for a period of time. For this project, the improvement that was shown was primarily qualitative. That is, due to the difficult nature of doing so, precise methods of quantifying relative performance were not developed. However, the intent of this project was to show that integrated inertial and GPS data can produce a significantly better system output than GPS alone. Some methods for precisely evaluating the system output performance are presented in the following chapter, and methods for fine-tuning system parameters were presented in the chapter on algorithm extensions.

As shown in Figure 7.1, the system has been demonstrated to be able to compensate for complete and partial GPS losses for over 1 full city block. The worst problems arose when the GPS signal was considered to be 'good' by our simple validity test, when it was actually significantly off the correct position. This could happen as a result of multi-path errors, or some other error that is not easily detectable. This problem can be addressed by either improving the validity test for the GPS data, or by fine-tuning the sensor fusion parameters.

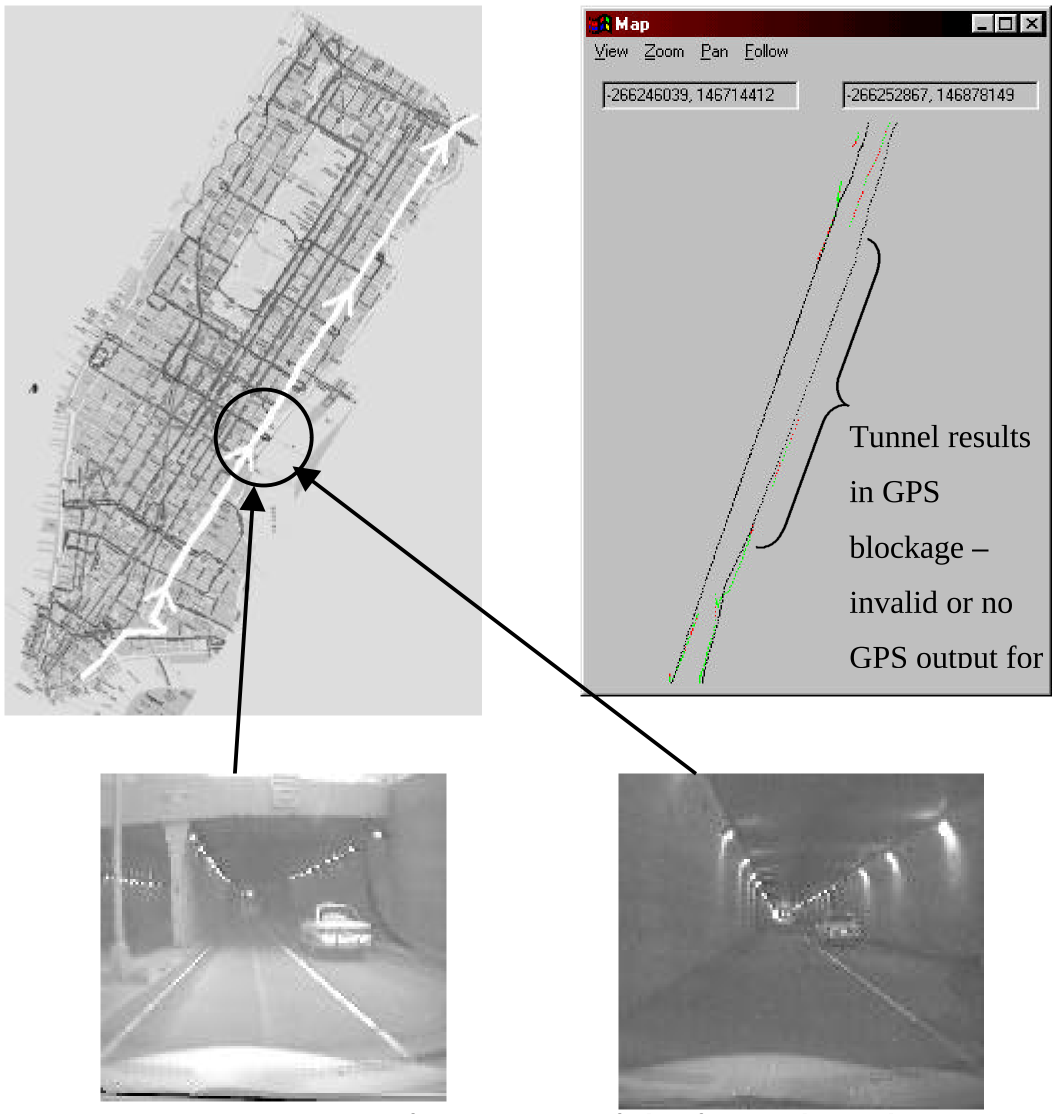


Figure 7.1 – Sensor fusion system corrects for loss of GPS signal in tunnel

Figure 7.1 shows one instance where the sensor fusion algorithm worked very well to compensate for a total GPS signal loss. The white line on the upper-left image indicates that path that was taken during one data collection run. The upper-right image is a screenshot of the map output view of the post-processing application, which shows where the GPS signal is completely lost and the sensor fusion algorithm switches to dead-reckoning using only inertial data. (Note that this is difficult to see in printed form because the colors become indistinguishable.) In addition, when the GPS signal is reestablished, the current position estimate and the GPS output data are integrated in a smooth transition. The lower left and right images are snapshots from the onboard VCR while entering the tunnel and inside the tunnel, respectively.

Chapter 8

Conclusions/Further Research

8.1 Further Research

The results from this project are encouraging and show that these techniques do work to improve position estimates in urban environments. However, there is much room for improvement in the overall techniques, as well as the particular implementation discussed. Some of these suggested improvements will be discussed here.

In order to evaluate the proposed system more precisely, improvements must be made in the area of output analysis. For this project, output analysis was done primarily by visual inspection and comparison of the generated outputs. To effectively fine tune the system and optimize the numerous fusion parameters, a more precise evaluation system must be implemented. This is not an easy task, because it requires knowledge of the desired output, to which the actual system output must be compared.

Performing this analysis task could be aided by the use of the map-matching techniques discussed earlier in this thesis. However, evaluating the output from the map-matching system would still require the beforehand knowledge of the actual path traversed by the vehicle. Ideally, the process of evaluating the entire system would proceed as follows:

1 – Generate a intersection/road segment based map for a well known region, such as downtown New York City, as discussed in the map-matching section of this thesis. This could be done using a GIS package such as ArcInfo or Etak. Alternately, this map could be generated by collecting 'near perfect' GPS data on the roadway. This is usually done by collecting samples in one location over a long period of time and averaging the data.

2 – Collect a large amount of inertial sensor and GPS data within that region

3 – Store the list of road segments traversed (or the list of intersections traversed) while collecting the GPS and inertial sensor data

4 – Post process the GPS and inertial sensor data using only the Kalman filter and sensor fusion algorithms, as discussed earlier in this thesis. Compare the fused output with the ideal output from the stored list of road segments. Use this information to fine-tune the sensor filtering and fusion parameters to produce a good output. The iterative hill-climbing or genetic algorithm methods could be implemented to automate the process of parameter optimization.

5 – Post process the GPS and inertial sensor data, producing a map-matched output from the system. The output from the system can be evaluated by comparing the percentage of correctly traversed intersections on the map with the list of actual intersections traversed. This system of evaluation can be used to fine-tune the map-matching algorithm to generate a reliable output based on the map-matched position estimate. If the output from the GPS/inertial sensor fusion portion of the system has a large amount of error, it may be required to manually reset the position estimate occasionally.

6 – After both the sensor fusion and the map-matching parts of the system have been optimized, the system could be set to evaluate its own performance during runtime. Assuming that the map-matched output is correct for some time interval while the output from the sensor fusion subsystem indicates that some sort of inertial drift is occurring, this information could be used to dynamically adjust filtering and fusion parameters to account for this drift. Such drift is known to occur as vibrations or temperature changes alter some sensor characteristics.

For this project, a small amount of the road segment/intersection map for downtown New York City was generated manually, using data from a Geographic Information System (GIS) software package. This was a very tedious and time-consuming process, which could be automated in future work. One of two options could be used to make the process much easier and more general in future use. The first option is to write a program that interfaces into a GIS database and outputs the intersection/road-segment format map for the map-matching program to read. The second option is to incorporate the capability to read the GIS database directly into the sensor fusion/map-matching program. The second option is the more desirable one, as it is more generalized and can be made to work any place in the world by simply using a different GIS database.

Another area of improvement for the project is the required hardware for the sensor collection and integration. This project used a variety of sensors, a large data acquisition board, and a laptop computer. For this type of solution to be viable in a consumer-type product, unneeded sensors should be removed along with the expensive data acquisition board. The laptop computer could also be replaced with a cheaper embedded system, provided a way to enter data into the system and to remove data from the system.

8.2 Conclusion

This project has shown that significant vehicle position estimate results may be obtained by integrating inertial sensor data and GPS sensor data using relatively simple sensor fusion techniques. The techniques presented here focus around a loosely coupled configuration of GPS and inertial navigation sensors, where the ultimate goal was to find the optimal weighting of the input from each sensor. An eight-state, two-input Kalman filter running at 100 Hertz was used to estimate the relative position of the vehicle between consecutive GPS samples. As GPS samples arrived at approximately 1 Hz, one of several sensor fusion techniques was used to determine the relative weightings for the GPS and inertial position estimates. This project has demonstrated that good results are obtainable by using only a few relatively inexpensive inertial sensors and an inexpensive GPS receiver.

References

1. "+/- 1 g to +/- 5 g Single Chip Accelerometer with Signal Conditioning," Technical Paper, Analog Devices, 1996.

2. Abrash, M., *Michael Abrash's Graphics Programming Black Book*, Coriolis Group Books, Boston, 1997.

3. Barshan, B. and Durrant-Whyte, H., "Inertial Navigation Systems for Mobile Robots," *IEEE Transactions on Mobile Robotics and Automation*, Vol. 11, No. 3, June, 1995.

4. Bennett, David, et al., *Visual C++ 5 Developer's Guide*, Sams Publishing, Indianapolis, 1997.

5. Brey, B., *8086/8088 Microprocessor Architecture, Programming and Interfacing*, Merrill Publishing Company, Columbus, 1987.

6. Chambers, L., ed., *Practical Handbook of Genetic Algorithms, Volume I*, CRC Press, New York, 1995.

7. Da, Ren, and Ching-Fang Lin, "Failure Diagnosis Using the State Chi-Square Test and the ARTMAP Neural Networks," *Proceedings of the American Control Conference*, June 1995.

8. *DaqBook User's Manual*, Rev. 3, IOtech, Inc., April 1994.

9. Escobal, Pedro Ramon, *Methods of Orbit Determination*, John Wiley & Sons, Inc., New York, 1965.

10. "Gyrostar Piezoelectric Vibrating Gyroscope," Technical Paper, Murata Manufacturing Co., Ltd., Kyoto, 1994.

11. Hofmann-Wellenhof, B., Lichtenegger, H., and Collins, J., *GPS Theory and Practice*, Springer-Verlag/Wien, New York, 1997.

12. Horst, R., Pardalos, P., and Thoai, N., *Introduction to Global Optimization*, Kluwer Academic Publishers, Boston, 1995.

13. Jang J.-S. R., Sun C.-T., and Mizutani, E., *Neuro-Fuzzy and Soft Computing*, Prentice Hall, Inc., Upper Saddle River, 1997.

14. Jourdain, R., *Programmer's Problem Solver*, Brady Publishing, New York, 1992.

15. Kaplan, Elliot D., ed., *Understanding GPS Principles and Applications*, Artech House Publishers, Boston, 1996.

16. LaMothe, A., Ratcliff, J., Seminatore, M., and Tyler, D., *Tricks of the Game-Programming Gurus*, Sams Publishing, Indianapolis, 1994.

17. Logsdon, T., *The Navstar Global Positioning System*, Van Nostrand Reinhold, New York, 1992.

18. Logsdon, T., *Understanding the Navstar GPS, GIS, and IVHS*, Van Nostrand Reinhold, New York, 1995.

19. Mazidi, M. and Mazidi, J., *The 80x86 IBM PC & Compatible Computers, Assembly Language, Design and Interfacing*, Prentice Hall, Englewood Cliffs, 1995.

20. Miller, Kenneth S. and Donald M. Leskiw, *An Introduction to Kalman Filtering with Applications*, Robert E. Krieger Publishing Company, Malabar, 1987.

21. Miyazaki, J., Higashimae, R., Kurihara, T., and Ishino, S., “Vibratory Gyroscope and Various Applications,” Technical Paper, Murata Manufacturing Co., Ltd., Kyoto, 1994.

22. Mostov, K., “Fuzzy Adaptive Stabilization of Higher Order Kalman Filters,” Research Report, University of California at Berkeley, 1996.

23. *Motorola Oncore User’s Guide*, Motorola, Inc., 1996.

24. Press, W., Flannery, B., Teukolsky, S., and Vetterling, W., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1988.

25. Ramjattan, A. and Cross, P., “A Kalman Filter for an Integrated Land Vehicle Navigation System,” *Journal of Navigation*, Vol. 48, May, 1995.

26. Rao, V. and Rao, H., *C++ Neural Networks & Fuzzy Logic*, MIS: Press, New York, 1995.

27. Roden, T., “High-Resolution Timing,” *Dr. Dobb’s Journal*, September, 1992.

28. Svensson, A. and Holst, J., “Integration of Navigation Data,” *Journal of Navigation*, Vol. 48, May, 1995.

29. Weiss, J., “Analysis of Upgraded GPS Internal Kalman Filter,” *IEEE AES Systems Magazine*, January, 1996

Appendix A. GPS Basics

A.1 History

The immediate predecessor of today's modern GPS is the Navy Navigation Satellite System (NNSS), also called TRANSIT system. This system included six satellites orbiting at altitudes of about 100 km with nearly circular polar orbits. This system was developed by the US military to determine the coordinates of vessels and aircraft. Civilian use of this satellite system was eventually authorized and the system became widely used worldwide both for navigation and surveying. Some of the early TRANSIT experiments showed that accuracy of about one meter could be obtained by occupying a point for several days.

The Global Positioning System (GPS) was developed to replace the TRANSIT system because of two major shortcomings in the system. Large time gaps in coverage were the main problem with TRANSIT. Since a satellite would typically pass overhead every 90 minutes, users had to interpolate their position between passes. The second problem was its relatively low navigation accuracy. The GPS system was designed to address these problems, answering the questions "what time, what position, and what velocity" quickly, accurately and inexpensively anywhere on the globe at any time. (Hofmann-Wellenhof et al., 1997)

A.2 Overview

The basic principle of GPS relies on the ability to measure distances from several points in space and performing triangulation based on these distances. A Navstar receiver anywhere on or near the surface of the Earth picks up the signals from four or more Navstar satellites. A string of precisely timed binary pulses travels from each satellite to the receiver, taking about one-eleventh of a second. The receiver estimates the signal travel time by subtracting the time from its internal clock from the time indicated by the satellite when it transmitted the pulse. This signal is then multiplied by the speed of light to obtain the estimated range to the first satellite.

If the clock in the receiver were perfectly synchronized with the clocks onboard the satellites, three ranging measurements of this type would be required to obtain an accurate three-dimensional position estimate. However, most Navstar receivers use inexpensive quartz crystal oscillators to measure the current time. Consequently, the receiver actually estimates the *pseudo-range* (false range) to each Navstar satellite. Since each range measurement is corrupted by the same timing error in the receiver's clock, this error can be removed mathematically with the measurement of a fourth satellite pseudo-range. In addition to a three-dimensional position estimate, a Navstar receiver can calculate its velocity and heading, along with the time of day and the date. (Logsdon, 1992)

A.3 System Segments

The Navstar Global Positioning System is typically divided logically into three main pieces, or segments. These segments are the space segment, the user segment, and the control segment. The *space segment* consists of 21 satellites plus 3 active on-orbit spares arranged in six 55-degree orbit planes 10,898 nautical miles above the earth. The *user segment* consists of the hundreds of thousands of Navstar receivers located on the ground, in the air, and aboard ships, together with a few aboard orbiting satellites. The user segment is completely passive, that is, the Navstar receivers only detect the signals emitted from the space segment. Because the Navstar satellites tend to lose track of their precise position and the exact time, a computer-driven *control segment* is required to correct for this subtle drift. The control segment consists of a group of unmanned monitor stations that track each Navstar satellite, calculating the precise position and timing errors for the satellite, which transmit to the satellites for error corrections. (Logsdon, 1995)

A.4 Differential GPS

Differential positioning with GPS, abbreviated DGPS, is a technique for improving GPS performance where two or more receivers are used. One receiver, usually at rest, is located at the reference site A with known coordinates and the remote receiver B is usually roving. The reference or base station calculates pseudorange corrections (PRC) and range rate corrections (RRC) which are transmitted to the remote receiver in near real time. The remote receiver applies the corrections to the measured pseudoranges and performs point positioning with the corrected pseudoranges. The use of the corrected pseudoranges improves positional accuracy. (Kaplan ed., 1996)

Appendix B. Geodetic to Ground Conversion

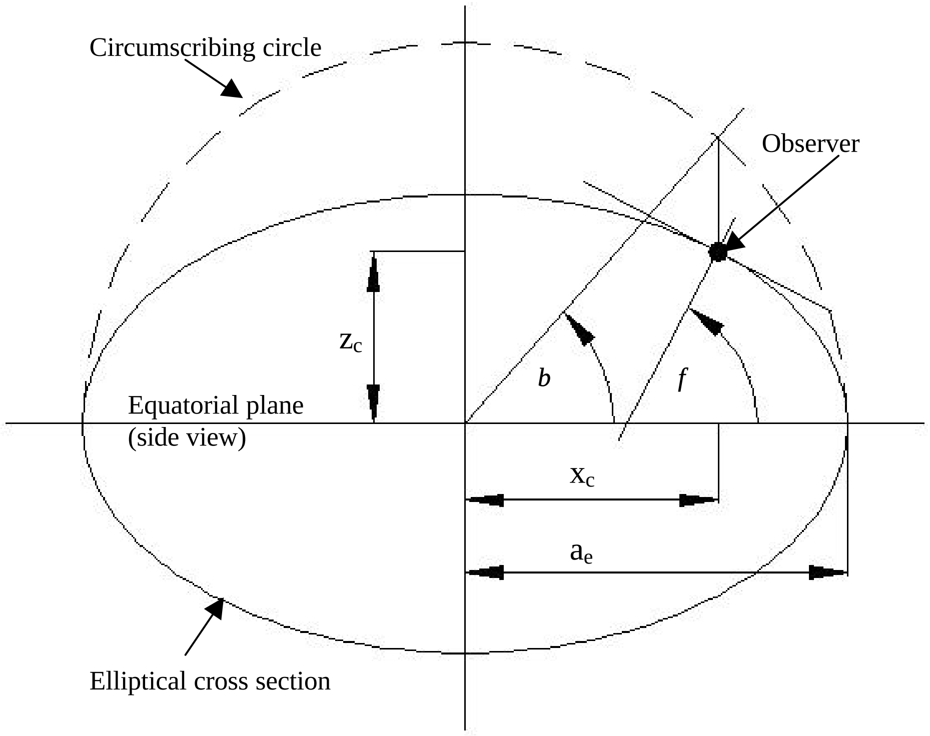


Figure B.1 – Detailed cross-section view of earth for geodetic to ground conversion

The GPS receiver returns data in the form of geodetic latitude/longitude coordinates, while the dead-reckoning system returns relative ground coordinates with units of meters. Therefore, it is desirable to calculate values C_1 and C_2 such that:

1 meter = C_1 * (1 degree latitude), and

1 meter = C_2 * (1 degree longitude)

This allows us to convert the relative meter measurements from the dead-reckoning system into degrees/minutes/seconds of latitude and longitude so that the data can be fused with the GPS data. Note that, assuming the Earth to be a perfect ellipsoid with a fixed equatorial radius a_e and eccentricity e, these conversion rates are dependent only upon the current latitude measurement f . The first part of the derivation consists of calculating the coordinates (x_c, z_c) of a point on the earth's surface, as indicated on Figure

B.1, and is taken from pages 26-29 from *Methods of Orbit Determination* by P.R.Escobal (1965). It is included here as a reference for the reader.

By inspection of Figure B.1, we have the following two relations for x_c and z_c:

$$x_c = a_e \cos b \qquad z_c = a_e\sqrt{1-e^2}\sin b \tag{1}$$

By differentiating each of the equations, we get:

$$-\mathbf{d}x_c = a_e \sin b\,\mathbf{d}b \qquad \mathbf{d}z_c = a_e\sqrt{1-e^2}\cos b\,\mathbf{d}b \tag{2}$$

Therefore, it follows that:

$$\mathbf{d}s \equiv \sqrt{(-\mathbf{d}x_c)^2 + (\mathbf{d}z_c)^2} = a_e\sqrt{1-e^2\cos^2 b}\,\mathbf{d}b \tag{3}$$

and

$$\sin f = -\frac{\mathbf{d}x_c}{\mathbf{d}s} = \frac{\sin b}{\sqrt{1-e^2\cos^2 b}} \tag{4}$$

$$\cos f = \frac{\mathbf{d}z_c}{\mathbf{d}s} = \frac{\sqrt{1-e^2}\cos b}{\sqrt{1-e^2\cos^2 b}} \tag{5}$$

By multiplying Eq. 4 by $\sqrt{1-e^2}$ *and squaring Eq. 5, we get:*

$$(1-e^2)\sin^2 f = \frac{(1-e^2)\sin^2 b}{1-e^2\cos^2 b} \tag{6}$$

$$\cos^2 f = \frac{(1-e^2)\cos^2 b}{1-e^2\cos^2 b} \tag{7}$$

By adding Eq. 6 and Eq. 7 and taking the square root of each side:

$$\sqrt{1-e^2\cos^2 b} = \frac{\sqrt{1-e^2}}{\sqrt{1-e^2\sin^2 f}} \tag{8}$$

By using equations 4 and 5 with equation 8, we get:

$$\sin b = \sin f\sqrt{1-e^2\cos^2 b} = \frac{\sqrt{1-e^2}\sin f}{\sqrt{1-e^2\sin^2 f}} \tag{9}$$

$$\cos b = \frac{\cos f\sqrt{1-e^2\cos^2 b}}{\sqrt{1-e^2}} = \frac{\cos f}{\sqrt{1-e^2\sin^2 f}} \tag{10}$$

By combining equations 9 and 10 with equations 1:

$$x_c = \frac{a_e \cos f}{\sqrt{1 - e^2 \sin^2 f}} \tag{11}$$

$$z_c = \frac{a_e \left(1 - e^2\right) \sin f}{\sqrt{1 - e^2 \sin^2 f}} \tag{12}$$

After the rectangular coordinates of the point on the surface of the earth have been calculated, we are now able to calculate the values C$_1$ and C$_2$. The following calculations based on the above formulae were performed by Gene Felis, Electronics Engineer, NUWC Div Keyport, in 1976:

The constant C$_1$ is the derivative of the ellipse at the position (x_c,z_c):

$$C_1 = \frac{\mathbf{d}s}{\mathbf{d}f} = \sqrt{\left(\frac{\mathbf{d}x_c}{\mathbf{d}f}\right)^2 + \left(\frac{\mathbf{d}z_c}{\mathbf{d}f}\right)^2} \tag{13}$$

Therefore, we must first calculate $\frac{\mathbf{d}x_c}{\mathbf{d}f}$ *and* $\frac{\mathbf{d}z_c}{\mathbf{d}f}$.

$$\mathbf{d}\left(\frac{f}{g}\right) = \frac{g * \mathbf{d}f - f * \mathbf{d}g}{g^2} \tag{14}$$

for x_c: $f = a_e \cos f \qquad g = \sqrt{1 - e^2 \sin^2 f}$ (15)

$$\mathbf{d}f = -a_e \cos f \qquad \mathbf{d}g = \frac{1}{2}\left(1 - e^2 \sin^2 f\right)^{-\frac{1}{2}}\left(-e^2\right) 2 \sin f \cos f \, \mathbf{d}f \tag{16}$$

$$= \frac{\left(-e^2\right) \sin f \cos f \, \mathbf{d}f}{\sqrt{1 - e^2 \sin^2 f}} \tag{17}$$

Putting equations 15, 16, 17 into equation 14, we get:

$$\frac{\mathbf{d}x_c}{\mathbf{d}f} = \frac{\sqrt{\left(1 - e^2 \sin^2 f\right)}\sqrt{\left(-a_e \sin f\right)} - \left(a_e \cos f\right)\left[\frac{\left(-e^2\right) \sin f \cos f}{\sqrt{\left(1 - e^2 \sin^2 f\right)}}\right]}{1 - e^2 \sin^2 f} \tag{18}$$

$$= \frac{\frac{\sqrt{1 - e^2 \sin^2 f}}{\sqrt{1 - e^2 \sin^2 f}}\sqrt{1 - e^2 \sin^2 f}\left(-a_e \sin f\right) - \left(-a_e \cos f\right)\left[\frac{\left(-e^2\right) \sin f \cos f}{\sqrt{1 - e^2 \sin^2 f}}\right]}{1 - e^2 \sin^2 f}$$

$$= \frac{\dfrac{(1-e^2\sin^2 f)(-a_e \sin f) - (a_e \cos f)[(-e^2)\sin f \cos f]}{\sqrt{1-e^2\sin^2 f}}}{1-e^2\sin^2 f}$$

$$= \frac{(-a_e)[((1-e^2\sin^2 f)\sin f - e^2 \sin f \cos^2 f)]}{\sqrt{(1-e^2\sin^2 f)^3}}$$

$$= \frac{-a_e \sin f\,[1-e^2(1-\cos^2 f) - e^2\cos^2 f]}{\sqrt{(1-e^2\sin^2 f)^3}}$$

$$= \frac{-a_e \sin f\,(1-e^2+e^2\cos^2 f - e^2\cos^2 f)}{\sqrt{(1-e^2\sin^2 f)^3}}$$

Finally, we reduce to:

$$\frac{\mathbf{d}x_c}{\mathbf{d}f} = \frac{-a_e \sin f\,(1-e^2)}{\sqrt{(1-e^2\sin^2 f)^3}} \tag{19}$$

for z$_c$: $\quad f = a_e(1-e^2)\sin f \qquad\qquad g = (1-e^2\sin^2 f)^{1/2}$ (20)

$$\mathbf{d}f = a_e(1-e^2)\cos f\,\mathbf{d}f \qquad\qquad \mathbf{d}g = \frac{(-e^2)\sin f \cos f\,\mathbf{d}f}{\sqrt{1-e^2\sin^2 f}} \tag{21}$$

Again, combining equations 20 and 21 into equation 14:

$$\frac{\mathbf{d}z_c}{\mathbf{d}f} = \frac{(1-e^2\sin^2 f)^{\frac{1}{2}} a_e(1-e^2)\cos f - a_e(1-e^2)\dfrac{\sin f\,(-e^2)\sin f \cos f}{\sqrt{1-e^2\sin^2 f}}}{1-e^2\sin^2 f} \tag{22}$$

$$= \frac{a_e(1-e^2)\left[\dfrac{(1-e^2\sin^2 f)^{\frac{1}{2}}(1-e^2\sin^2 f)^{\frac{1}{2}}\cos f + e^2\sin^2 f \cos f}{(1-e^2\sin^2 f)^{\frac{1}{2}}(1-e^2\sin^2 f)^{\frac{1}{2}}}\right]}{1-e^2\sin^2 f}$$

$$= \frac{a_e(1-e^2)\cos f\,[1-e^2\sin^2 f + e^2\sin^2 f]}{(1-e^2\sin^2 f)^{\frac{3}{2}}}$$

Which reduces to:

$$= \frac{a_e(1-e^2)\cos f}{(1-e^2\sin^2 f)^{\frac{3}{2}}} \tag{23}$$

Substituting equations 19 and 23 into equation 13, we get:

$$\frac{\mathbf{d}s}{\mathbf{d}f} = \sqrt{\left(\frac{\mathbf{d}x_c}{\mathbf{d}f}\right)^2 + \left(\frac{\mathbf{d}z_c}{\mathbf{d}f}\right)^2}$$

$$= \sqrt{\frac{a_e{}^2 \sin^2 f \left(1 - e^2\right)^2 + a_e{}^2 \cos^2 f \left(1 - e^2\right)^2}{\left(1 - e^2 \sin^2 f\right)^3}}$$

Reducing to:

$$= \frac{a_e\left(1 - e^2\right)}{\left(1 - e^2 \sin^2 f\right)^{\frac{3}{2}}} \qquad (24)$$

Equation 24 represents $\frac{\mathbf{d}s}{\mathbf{d}f}$ *in units of* a_e *per radian. To get into meters per degree of latitude, we use* $2p$ radians $= 360^\circ$, *which results in the following:*

$$\mathrm{C}_1 = \frac{\mathbf{d}s}{\mathbf{d}f} = \frac{2p\left(1 - e^2\right)a_e}{360\left(1 - e^2 \sin^2 f\right)^{\frac{3}{2}}} \qquad (25)$$

To obtain the value for C_2*, we use equation 11 to compute the distance d around the earth along a line of latitude* f *:*

$$d = 2p x_c = \frac{2p a_e \cos f}{\left(1 - e^2 \sin^2 f\right)^{\frac{1}{2}}} \qquad (26)$$

Because d represents the distance for a full 360° *around the earth, we divide equation 26 by 360 to get* C_2*:*

$$C_2 = \frac{d}{360} = \frac{2p a_e \cos f}{360\left(1 - e^2 \sin^2 f\right)^{\frac{1}{2}}} \qquad (27)$$

where $\quad$ *a_e » 6,378,150 meters*

e » 0.08181333

For our purposes, we desire C_1 and C_2 in units of meters per *millisecond* instead of meters per *degree*. Therefore, we simply divide equations 25 and 27 by 3,600,000 to convert each from degrees to milliseconds. Substituting the values for a_e and e and converting to meters per millisecond, we have:

$$C_1 = \frac{1}{32.55720\left(1 - 0.0066934\sin^2 f\right)^{\frac{3}{2}}} \tag{28}$$

$$C_2 = \frac{\cos f}{33.3392843\left(1 - 0.0066934\sin^2 f\right)^{\frac{1}{2}}} \tag{29}$$

Appendix C. Motorola GPS Receiver Messages

What follows is a breakdown of one input command and the resulting response from the Motorola Oncore GPS receiver, as taken from the *Motorola Oncore User's Guide* (1996).

To set the response message rate of the GPS receiver, the following message is sent from the user to the GPS receiver:

@@BamC<CR><LF>

m – mode	0 – output response message once (polled) 1 .. 255 – response message output at indicated rate (continuous) 1 – once per second 2 – once every two seconds 255 – once every 255 seconds
C – checksum	

The following response would be sent from the GPS receiver to the user at the specified rate:

@@Bamdyyhmsffffaaaaoooohhhhmmmmvvhhddtntims
dimsdimsdimsdimsdimsdsC<CR><LF>

Where we can interpret is as follows:

Date:	
m – month	1 .. 12
d – day	1 .. 31
yy – year	1980 .. 2079
Time:	
h – hours	0 .. 23
m – minutes	0 .. 59
s – seconds	0 .. 60
ffff – fractional seconds	0 .. 999,999,999 (0.0 to 0.999999999)
Position:	
aaaa – latitude in msec	-324,000,000 .. 324,000,000 (-90^o to +90^o)
oooo – longitude in msec	-648,000,000 .. 648,000,000 (-180^o to +180^o)
hhhh – height in cm (GPS, ref ellipsoid)	-100,000 .. +1,800,000 (-1000.00 to +18,000.00 meters)
mmmm – height in cm (MSL ref)	-100,000 .. +1,800,000 (-1000.00 to +18,000.00 meters)
Velocity:	
vv – velocity in cm/sec	0 .. 51400 (0 to 514.00 m/sec)
hh – heading	0 .. 3599 (0.0 to 359.9 deg)

Geometry:	
dd – current DOP	0 .. 999 (0.0 to 99.9 DOP)
t – DOP type	0 – PDOP (in 3D)
	1 – HDOP (in 2D)
Satellite visibility and tracking status:	
n – num of visible sats	0 .. 12
t – num of satellites tracked	0 .. 8
For each of six receiver channels:	
i – satellite ID	0 .. 37
m – channel tracking mode	0 .. 8
0 – Code Search	5 – Message Sync Detect
1 – Code Acquire	6 – Satellite Time Avail
2 – AGC Set	7 – Ephemeris Acquire
3 – Freq Acquire	8 – Avail for Position
4 – Bit Sync Detect	
s – signal strength	0 .. 255
(number proportional to signal-to-noise ratio)	
d – channel status flag	
Each bit represents one of the following:	
(msb)	Bit 7: Using for position fix
	Bit 6: Satellite momentum alert flag
	Bit 5: Satellite anti-spoof flag set
	Bit 4: Satellite reported unhealthy
	Bit 3: Satellite reported inaccurate
	Bit 2: Spare
	Bit 1: Spare
(lsb)	Bit 0: Parity Error
(End of channel dependant data)	
s – receiver status message	
(msb)	Bit 7: Position propagate mode
	Bit 6: Poor geometry (DOP > 20)
	Bit 5: 3D fix
	Bit 4: Altitude hold (2D fix)
	Bit 3: Acquiring satellites/position hold
	Bit 2: Differential
	Bit 1: Insufficient visible satellites (<3)
(lsb)	Bit 0: Bad almanac
C - checksum	

Appendix D. Program Listings

D.1 DOS Program Listings

The DOS program that was written for this thesis was used primarily to collect GPS and sensor data and store it to disk in real-time. This program also used simple VGA graphics output to display information about GPS readings and sensor values, and to display the current GPS position graphically on a local area map on the screen. The program could also be used to review the collected data after it had been collected to disk. The block diagram shown in Figure D.1 shows the basic flow of the software.

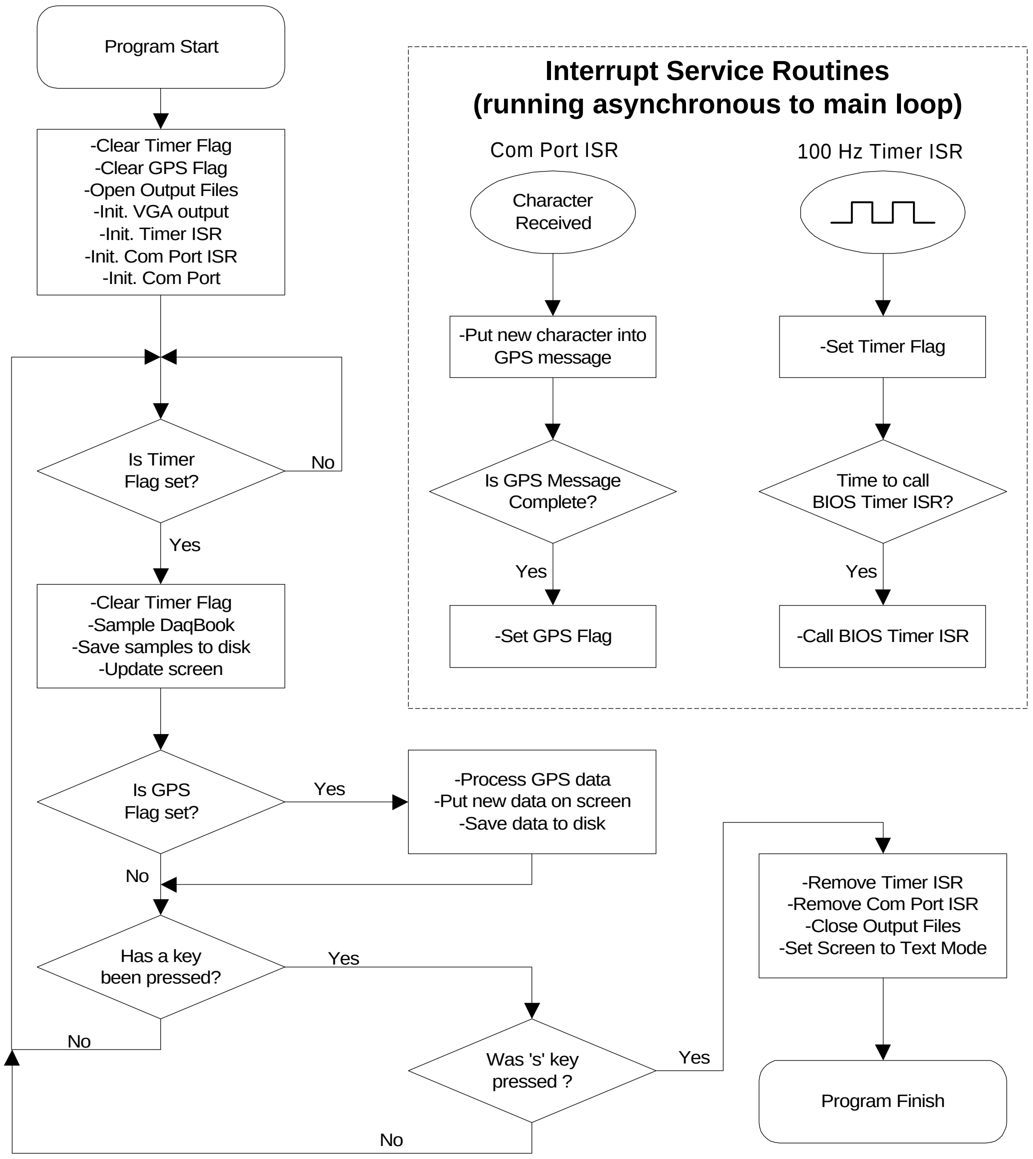


Figure D.1 – DOS data collection program software flow

Listed below is a fragment of the header file `maplog.h`. This file contains a number of define statements for I/O port addresses, screen size information, etc. This file also contains the type declarations for the sensor data that is collected and stored to disk.

Maplog.h

```
#ifndef _MAPLOG_H
#define _MAPLOG_H

#define PALETTE_MASK                    0x3C6           /* video port addr to set mask */
#define PALETTE_REGISTER_RD             0x3C7           /* video port addr to read palette */
#define PALETTE_REGISTER_WR             0x3C8           /* video port addr to write palette */
#define PALETTE_DATA                    0x3C9           /* video port addr to send data to */
#define VGA256                          0x13            /* VGA screen mode value */
#define TEXT_MODE                       0x03            /* Text mode value */
#define CHAR_WIDTH                      8               /* Character width (pixels) */
#define CHAR_HEIGHT                     8               /* Character height (pixels) */
#define ROM_CHAR_SET_SEG                0xF000          /* Character segment in ROM */
#define ROM_CHAR_SET_OFF                0xFA6E          /* Character offset in segment */
#define SCREEN_WIDTH                    320             /* Default screen width (pixels) */
#define SCREEN_HEIGHT                   200             /* Default screen height (pixels) */

typedef struct RGB_color_typ{                   /* structure containing palette color info */
       unsigned char red;
       unsigned char green;
       unsigned char blue;
} RGB_color,*RGB_color_ptr;

typedef struct log_data_st{                     /* structure containing inertial sensor info */
       unsigned long ticks;
unsigned long odometer_total;
       unsigned int gyro1;
       unsigned int gyro2;
       unsigned int accelf;
       unsigned int accelr;
   unsigned int steer;
} T_LOG_DATA;

typedef struct point_st{                        /* structure representing a single point */
 long x,y;
} POINT;

typedef struct road_segment_st{                 /* structure representing a road segment */
 POINT p1,p2;                                   /*  for map drawing */
 long road;
} ROAD_SEG;

#define GPS_PORT_DATA         0x3F8          /* COM1 I/O character buffer */
#define GPS_PORT_STATUS       0x3FD          /* Line status register */
#define GPS_PORT_INTR         0x3F9          /* Interrupt enable register */
#define GPS_PORT_CONT         0x3FC          /* Control register */

#define TIMERINTR             0x08           /* Timer ISR jump vector */
#define COM1INTR              0x0C           /* Com1 ISR jump vector */
#define PIT_FREQ              0x1234DDL      /* Default PIT frequency */

#endif
```

Maplog.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <bios.h>
#include <dos.h>
#include <mem.h>
#include <conio.h>
#include <alloc.h>
#include <math.h>
#include <string.h>
#include "drgps.h"            /* structures for storing GPS info */
#include "daqbook.h"          /* functions for accessing the DaqBook */
#include "maplog.h"           /* structures and defines for this program */

// #define     _NOBOOK          /* define for testing when no daqbook connected */

/* Function prototypes */
void SetTimer(void interrupt(__far *)(void),int);
void SetCom1(void interrupt(__far *)(void));
void CleanUpTimer(void);
void CleanUpCom1(void);
void __interrupt Handler(void);
void __interrupt Com1Handler(void);
void set_pallette(void);
void GrabPallette(unsigned char Pall[256][3]);
void RestorePallette(unsigned char Pall[256][3]);
void Set_Palette_Register(int,RGB_color_ptr);
void h_line(int,int,int,unsigned int,unsigned char far *);
void v_line(int,int,int,unsigned int,unsigned char far *);
void line(int,int,int,int,unsigned int,unsigned char far *);
void map_line(POINT,POINT,unsigned int,unsigned char far *);
void fill_box(int,int,int,int,unsigned int,unsigned char far *);
void set_vga(void);
void set_text(void);
void set_palette(void);
void fill_screen(unsigned int,unsigned char far *);
void blit_char(int,int,char,int,unsigned char far *);
void blit_bit(int,int,unsigned int,unsigned char far*);
void init_daqbook(void);
void _far _pascal daq_error(int error_code);
void pc_set_to_gps(void);
void get_pc_time(void);
void get_odometer_count(void);
void get_user_key(void);
int gps_serial_wait_for_next(void);
int get_gps_data(void);
void clear_gps_serial(void);
void Pos_Status_Data_Decode(unsigned char*,T_POS_CHAN_STATUS*,char);
void draw_map_border(unsigned int,unsigned char far*);
void draw_sensor_border(unsigned int,unsigned char far*);
void draw_gps_border(unsigned int,unsigned char far*);
void draw_guage_border(unsigned int,unsigned char far*);
void draw_sensor_vals(unsigned int,unsigned char far*);
void draw_map(unsigned int,unsigned char far*);
void blit_string(int,int,char*,unsigned int,unsigned char far*);
void draw_strength_border(unsigned int,unsigned char far*);
void draw_strength_vals(unsigned int,unsigned char far*);
void put_box(int,int,int,int,unsigned char*,unsigned char far*);
void get_box(int,int,int,int,unsigned char*,unsigned char far*);
void new_gps_stuff(void);
int sgn(int);
POINT ll_to_screen(POINT);
int in_view(POINT);
void goodbye(int);

/* Global data */
void __interrupt(__far *BIOSTimerHandler)(void);     /* old timer ISR pointer */
void __interrupt(__far *BIOSCom1Handler)(void);      /* old com1 ISR pointer */
```

```
volatile long counter, clock_ticks;                 /* timer ISR counter variables */
unsigned char far * vscreen;          /* pointer to the video screen in memory */
unsigned char far *rom_char_set;      /* pointer to characters in memory */
FILE *log_file;                       /* output .log file */
FILE *gps_file;                       /* output .gps file */
FILE *map_file;                       /* input map file */
FILE *cfg_file;                       /* input configuration file */
FILE *key_file;                       /* output key press file */
FILE *debug;                          /* output debug file */
int logging,postview,mapping,running; /* indicator flags for program flow */
volatile int new_sense;               /* .01 second elapsed flag */
struct time gps_time;                 /* time of a GPS capture */
struct time pc_time;                  /* time of a sensor capture */
unsigned long hour,min,sec,pc_seconds,start_hsec;
volatile unsigned long hsec;          /* timer ISR counter variables */
volatile int new_gps;                 /* new GPS data flag */
unsigned long pct_val,old_pct_val,last_gps_hsec;
unsigned long odometer_total,prev_odometer_total,odometer_sample;
long run_sec,run_min,prev_sec;        /* elapsed times for output screen */
volatile char user_key;               /* user input key */
T_LOG_DATA log_data;                  /* sensor input data */
T_LOG_DATA *old_data;                 /* previous sensor input data */
int curr_old_data;
int old_speed,curr_speed,curr_guage;
char old_sats_tracked;                /* number of satellites tracked for screen output */
T_POS_CHAN_STATUS GPS_chan,GPS_chan2; /* processes GPS input data */
unsigned char gps_report[100];        /* GPS raw input data string */
long gps_latitude,gps_longitude;      /* GPS lat/long from receiver */
long calc_latitude,calc_longitude;    /* lat/long for screen output */
long old_calc_lat,old_calc_long;      /* old lat/long for screen output */
unsigned long gps_seconds,gps_hsec;
int old_gps_valid,gps_valid,gps_found;
/* several strings for output to screen */
char rtstr[6],speed_str[5],pct_str[9],tsgps_str[5];
char status_line[41];
char lat_str[15],long_str[15];
int sen_high[5],sen_low[5],sen_val[5];
unsigned int strength[6];
/* several numbers indicating map bounds, position, etc */
long mapullong,mapullat,maplrlong,maplrlat;
int mapposx,mapposy;
/* several temporary variables for various things */
long tlat,tlong;
unsigned long nexttime;
unsigned char cursor_back_box[100];
char map_name[25];
unsigned char stmp[20];
long tempnum[5];
double templong;
char gps_char;                        /* GPS input character (from com port) */
/* map information */
long num_map_roads,num_map_segments;
char *map_roads;
ROAD_SEG *map_segments;
int quickview;                        /* flag set to no delay on replay of data */
int novga;                            /* flag set to not display data graphically */
int tsec_since_gps;                   /* count of time since last GPS message received */
int gps_report_idx;
unsigned char gps_port_intr_set;

void main(int argc,char *argv[]){
 /* declare and initialize local data */
 char log_file_name[40];              /* file names */
 char gps_file_name[40];
 char key_file_name[40];
 unsigned char OldPallette[256][3];   /* location to store original palette */
 int i;

 /* initialize global data */
 vscreen=(unsigned char far*)0xA0000000L;           /* location of video memory */
 rom_char_set=(unsigned char far *)0xF000FA6EL;     /* rom character set */
```

```
get_pc_time();
hsec=start_hsec=pc_seconds*100+pc_time.ti_hund;     /* initial hsec count */
new_sense=quickview=mapping=logging=postview=0;     /* initialize all flags to false */
/* initialize all data to default values */
curr_guage=curr_speed=0;
mapposx=mapposy=0;
novga=gps_report_idx=0;
calc_latitude=calc_longitude=0;
prev_sec=old_speed=old_gps_valid=-1;
old_sats_tracked=-1;
/* initialize output strings appropriately */
for(i=0;i<4;i++) rtstr[i]=tsgps_str[i]=' ';
for(i=0;i<8;i++) pct_str[i]=' ';
rtstr[2]=pct_str[2]=pct_str[5]=':';
tsgps_str[2]='.';
rtstr[5]=speed_str[3]=lat_str[10]=long_str[10]=pct_str[8]=tsgps_str[4]=0;
for(i=0;i<6;i++) strength[i]=0;
for(i=0;i<5;i++){
 sen_high[i]=5000;
 sen_low[i]=0;
 sen_val[i]=0;
}
for(i=0;i<10;i++) lat_str[i]=long_str[i]='0';
/* initialize map border values */
mapullong = STMAPULLONG;
mapullat = STMAPULLAT;
maplrlong = STMAPLRLONG;
maplrlat = STMAPLRLAT;
nexttime = 0;
log_file = gps_file = map_file = cfg_file = debug = NULL;
old_data = NULL;
map_roads=NULL;
map_segments=NULL;
debug=fopen("debug.dat","w");               /* extra debug info to file */
old_data=farmalloc(sizeof(T_LOG_DATA)*OLD_DATA_SIZE);
if(old_data==NULL){
 printf("Error allocating memory!\n");
 printf("Exiting . . .\n");
 goodbye(0);
}
for(i=0;i<OLD_DATA_SIZE;i++) old_data[i]=log_data;  /* init data to 0's */

/* parse command line */
for(i=1;i<argc;i++){
 if(!strcmp(argv[i],"-l") || !strcmp(argv[i],"-L")){
       if(!postview){
        logging=1;           /* logging (data acquisition) mode */
        i++;
        strcpy(log_file_name, argv[i]);
        strcpy(gps_file_name, argv[i]);
        strcpy(key_file_name, argv[i]);
        strcat(log_file_name, ".log");
        strcat(gps_file_name, ".gps");
        strcat(key_file_name, ".key");
        /* open log output file */
        if((log_file = fopen(log_file_name, "wb"))==NULL){
         printf("Error opening log file (%s) for output!\n", log_file_name);
         printf("Exiting . . .\n");
         goodbye(0);
        }
        /* open GPS output file */
        if((gps_file = fopen(gps_file_name, "wb"))==NULL){
         printf("Error opening gps file (%s) for output!\n", gps_file_name);
         printf("Exiting . . .\n");
         goodbye(0);
        }
        /* open key press output file */
        if((key_file = fopen(key_file_name, "wb"))==NULL){
         printf("Error opening key file (%s) for output!\n", key_file_name);
         printf("Exiting . . .\n");
         goodbye(0);
```

```
                 }
                }
                else goodbye(1);
          }
          else if(!strcmp(argv[i],"-p") || !strcmp(argv[i],"-P")){
                if(!logging){
                 postview=1;          /* postview (replay) mode */
                 i++;
                 strcpy(log_file_name, argv[2]);
                 strcpy(gps_file_name, argv[2]);
                 strcat(log_file_name, ".log");
                 strcat(gps_file_name, ".gps");
                 /* open log input file */
                 if((log_file = fopen(log_file_name, "rb"))==NULL){
                  printf("Error opening log file (%s) for input!\n", log_file_name);
                  printf("Exiting . . .\n");
                  goodbye(0);
                 }
                 /* open GPS input file */
                 if((gps_file = fopen(gps_file_name, "rb"))==NULL){
                  printf("Error opening gps file (%s) for input!\n", gps_file_name);
                  printf("Exiting . . .\n");
                  goodbye(0);
                 }
                }
                else goodbye(1);
          }
          else if(!strcmp(argv[i],"-m") || !strcmp(argv[i],"-M")){
                char mapfilename[20];
                i++;
                strcpy(mapfilename, argv[i]);
                strcat(mapfilename, ".map");
                /* open map input file */
                if((map_file = fopen(mapfilename, "rb"))==NULL){
                 printf("Error opening map file (%s) for input!\n", mapfilename);
                 printf("Exiting . . .\n");
                 goodbye(0);
                }
                fread(map_name,24,1,map_file);
                fread(&num_map_roads, sizeof(long),1,map_file);
                fread(&num_map_segments, sizeof(long),1,map_file);
                /* allocate memory for roads and segments */
                map_roads = farmalloc(24*num_map_roads);
                map_segments = farmalloc(sizeof(ROAD_SEG)*num_map_segments);
                if(map_roads == NULL || map_segments == NULL){
                       printf("Error allocating memory!\n");
                       printf("Exiting . . .\n");
                       goodbye(0);
                }
                /* read map roads and segments */
                for(i=0;i<num_map_roads;i++){
                       fread(&map_roads[i*24],24,1,map_file);
                }
                for(i=0;i<num_map_segments;i++){
                       fread(&map_segments[i],sizeof(ROAD_SEG),1,map_file);
                }
                fclose(map_file);
                map_file = NULL;
                mapping=1;
          }
          else if(!strcmp(argv[i],"-novga") || !strcmp(argv[i],"-NOVGA")){
                novga=1;              /* disable VGA (graphics) output */
          }
          else if(!strcmp(argv[i],"-c")){
                char cfgfilename[20];
                char var[20];
                int j;
                i++;
                strcpy(cfgfilename, argv[i]);
                strcat(cfgfilename, ".cfg");
                /* open configuration input file */
```

```
      if((cfg_file = fopen(cfgfilename, "r"))==NULL){
       printf("Error opening configuration file (%s) for input!\n", cfgfilename);
       printf("Exiting . . .\n");
       goodbye(0);
      }
      /* read configuration parameters */
      while(fscanf(cfg_file, "%s",var)!=EOF){
       for(j=0;j<strlen(var);j++) var[j] |= 0x20; /* make lowercase */
       if(!strcmp(var, "mapullong")) fscanf(cfg_file, "%lu", &mapullong);
       else if(!strcmp(var, "mapullat")) fscanf(cfg_file, "%lu", &mapullat);
       else if(!strcmp(var, "maplrlong")) fscanf(cfg_file, "%lu", &maplrlong);
       else if(!strcmp(var, "maplrlat")) fscanf(cfg_file, "%lu", &maplrlat);
       else if(!strcmp(var, "sensor")){
        fscanf(cfg_file, "%d", &j);
        if(j>=0 && j<=5) fscanf(cfg_file, "%d %d", &sen_low[j],&sen_high[j]);
       }
       else fgets(var,20,cfg_file); /* eat remainder of unknown config line */
      }
      fclose(cfg_file);
      cfg_file = NULL;
 }
 else goodbye(1);     /* unknown command line parameter -> exit */
}
if(!postview && !logging) goodbye(1);       /* doing nothing -> exit */
bioscom(0,0xE3,0);                   /* set COM1 to 9600,N,8,1 for GPS receiver */
if(logging){
 sprintf(status_line, "Waiting: %s - G to go", log_file_name);
 #ifndef _NOBOOK
      init_daqbook();               /* initialize DaqBook */
      bioscom(0,0xE3,0);            /* set COM1 to 9600,N,8,1 for GPS receiver */
      pc_set_to_gps();              /* synchronize trigger times */
      get_odometer_count();         /* initialize odometer counter */
      daqCtrMultCtrl(DmccLoad,1,1,0,0,0);
 #endif
 odometer_total=prev_odometer_total=0;
}
else{
 sprintf(status_line, "Waiting: %s - G to go", log_file_name);
}

curr_old_data=0;
if(!novga){
 set_vga();                     /* switch to vga 320x200 graphics mode */
 GrabPallette(OldPallette);     /* save the old palette */
 set_pallette();                /* set new palette */

 /* do initial screen setup here */
 draw_map_border(BORDERCOL, vscreen);
 draw_sensor_border(BORDERCOL, vscreen);
 draw_gps_border(BORDERCOL, vscreen);
 draw_guage_border(BORDERCOL, vscreen);
 draw_strength_border(BORDERCOL, vscreen);
 blit_string(STATUSX, STATUSY, status_line, TEXTCOL, vscreen);
 if(mapping) draw_map(BLUE, vscreen);
}
else printf("%s\n",status_line);

SetTimer(Handler,100);               /* set timer interrupt to 100 Hz */
SetCom1(Com1Handler);                /* set com interrupt handler */

for(user_key=0;user_key!='g' && user_key!='G';get_user_key());    /* wait for 'G' */

/* clear old message string and draw a new one */
if(!novga) blit_string(STATUSX,STATUSY,status_line,BACKGNDCOL,vscreen);
if(logging) sprintf(status_line,"Logging: %s - S to stop",log_file_name);
else sprintf(status_line,"Reading: %s - S to stop",log_file_name);
if(!novga) blit_string(STATUSX,STATUSY,status_line,TEXTCOL,vscreen);
else printf("%s\n",status_line);

/* main loop for data collection */
running=1;
```

```
 while(running){      /* loop until running == 0 */
  while(!new_sense && !quickview){}   /* wait until .01 sec if not quickviewing */
  /* .01 second elapsed - collect new sensor data */
  new_sense=0;                       /* reset flag */
  if(!start_hsec) start_hsec=hsec;
  if(logging){                       /* logging - acquire data from daqbook */
       #ifndef _NOBOOK
        get_odometer_count();
        daqAdcRd(0,&(log_data.gyro1),DgainX1);
        daqAdcRd(1,&(log_data.gyro2),DgainX1);
        daqAdcRd(2,&(log_data.accelf),DgainX1);
        daqAdcRd(3,&(log_data.accelr),DgainX1);
        daqAdcRd(4,&(log_data.steer),DgainX1);
       #endif
       log_data.ticks=hsec;
       log_data.odometer_total=odometer_total;
fwrite(&log_data,sizeof(T_LOG_DATA),1,log_file); /* write to file */
  }
  else{                              /* post view – read data from file */
       if(fread(&log_data,sizeof(T_LOG_DATA),1,log_file)==NULL){
        /* done input from log file - set flags */
        running=0;
       }
  }
  if(running){
       draw_sensor_vals(BACKGNDCOL,vscreen);  /* draw over old sensor values */
       sen_val[0]=log_data.gyro1;
       sen_val[1]=log_data.gyro2;
       sen_val[2]=log_data.accelf;
       sen_val[3]=log_data.accelr;
       sen_val[4]=log_data.steer;
       draw_sensor_vals(SENSORCOL,vscreen);   /* draw new sensor values */
       old_data[curr_old_data]=log_data;      /* store log data in array */
       curr_old_data++;
       if(curr_old_data==OLD_DATA_SIZE) curr_old_data=0;
       curr_speed=                            /* estimate speed from delta positions */
(old_data[(curr_old_data+OLD_DATA_SIZE-1)%OLD_DATA_SIZE].odometer_total-
old_data[(curr_old_data+OLD_DATA_SIZE-1-SPEED_DELTA) %OLD_DATA_SIZE].odometer_total)*SPEED_FACT;
  }
  if(logging){
       if(new_gps){     /* whole gps report received */
        Pos_Status_Data_Decode(gps_report, &GPS_chan, 0);   /* decode GPS message */
        gps_time.ti_hour = GPS_chan.hours;
        gps_time.ti_min = GPS_chan.minutes;
        gps_time.ti_sec = floor(GPS_chan.seconds);
        gps_time.ti_hund = floor(GPS_chan.seconds * 100.0);
        gps_time.ti_hund %= 100;
        hour = gps_time.ti_hour;
        min = gps_time.ti_min;
        sec = gps_time.ti_sec;
        gps_seconds = hour * 3600 + min * 60 + sec;
        gps_hsec = gps_time.ti_hund;
        gps_hsec += gps_seconds * 100;
        gps_valid = 1;                              /* check receiver status */
        if(GPS_chan.rcvr_status & 0x43) gps_valid = 0;
        if(!(GPS_chan.rcvr_status & 0x30)) gps_valid = 0;
        fwrite(&(log_data.ticks),sizeof(unsigned long),1,gps_file);
        last_gps_hsec=log_data.ticks;
        fwrite(&GPS_chan,sizeof(T_POS_CHAN_STATUS),1,gps_file);    /* output to file */
        new_gps_stuff();
        new_gps=0;
       }
  }
  else{                          /* postview - read GPS data from file */
       if(nexttime<=log_data.ticks){ /* synchronize with inertial sensor data */
        if(nexttime){
         GPS_chan=GPS_chan2;
         last_gps_hsec=nexttime;
         gps_latitude=(long)((((abs(GPS_chan.latitude.degrees)*60.0)+
                GPS_chan.latitude.minutes)*60.0+GPS_chan.latitude.seconds)*1000.0);
         gps_longitude=(long)((((abs(GPS_chan.longitude.degrees)*60.0)+
                GPS_chan.longitude.minutes)*60.0+GPS_chan.longitude.seconds)*1000.0);
         if(GPS_chan.latitude.degrees<0) gps_latitude=0-gps_latitude;
         if(GPS_chan.longitude.degrees<0) gps_longitude=0-gps_longitude;
         gps_valid = 1;
         if(GPS_chan.rcvr_status & 0x43) gps_valid = 0;
         if(!(GPS_chan.rcvr_status & 0x30)) gps_valid = 0;
         new_gps_stuff();
         /* override INS calculations if good GPS */
        }
        fread(&nexttime,sizeof(unsigned long),1,gps_file);
        fread(&GPS_chan2,sizeof(T_POS_CHAN_STATUS),1,gps_file);
       }
  }
  /* calculate new long, lat here */
  calc_latitude=gps_latitude;
  calc_longitude=gps_longitude;

  if(!novga){                 /* update screen portions at staggered intervals */
       if(!(hsec%10) || quickview){
        if(mapposx || mapposy){       /* put box back at mapposx, mapposy */
         put_box(mapposx-2,mapposy-2,CURSBACKBOXWIDTH,CURSBACKBOXHEIGHT,
cursor_back_box,vscreen);
         blit_bit(mapposx,mapposy,RED,vscreen);
        }
        /* recalc mapposx,mapposy */
        if(calc_longitude>=mapullong && calc_longitude<=maplrlong &&
              calc_latitude<=mapullat && calc_latitude>=maplrlat){
         POINT p1,p2;
         p1.x=calc_longitude;
         p1.y=calc_latitude;
         p2=ll_to_screen(p1);
         mapposx=(int)p2.x;
         mapposy=(int)p2.y;
        }
        else{
         mapposx=mapposy=0;
        }
        if(mapposx || mapposy){           /* get box at mapposx,mapposy */
         get_box(mapposx-2,mapposy-2,CURSBACKBOXWIDTH,CURSBACKBOXHEIGHT,
                              cursor_back_box,vscreen);
         /* draw marker */
         v_line(mapposy-2,mapposy+2,mapposx,MAPPOSCOL,vscreen);
         h_line(mapposx-2,mapposx+2,mapposy,MAPPOSCOL,vscreen);
        }
       }
       switch((int)(hsec%10)){
       case 2:        /* time = xxxxx.2 seconds -> update runtime strings */
        run_sec=((hsec-start_hsec)%6000)/100;
        run_min=((hsec-start_hsec)%360000L)/6000;
        if(run_sec!=prev_sec) {
         /* output new time to screen */
         if(prev_sec!=-1) blit_string(RTPOSX,RTPOSY,rtstr,BACKGNDCOL,vscreen);
         rtstr[0]=(char)((run_min%100)/10)+'0';
         rtstr[1]=(char)(run_min%10)+'0';
         rtstr[3]=(char)((run_sec%100)/10)+'0';
         rtstr[4]=(char)(run_sec%10)+'0';
         blit_string(RTPOSX,RTPOSY,rtstr,TEXTCOL,vscreen);
         prev_sec=run_sec;
        }
        break;
       case 4:        /* time = xxxxx.4 seconds -> update current time string */
        if(logging) pct_val=hsec;
        else pct_val=log_data.ticks;
        if(old_pct_val/100!=pct_val/100){   /* new pc second to print*/
         blit_string(PCTSTRPOSX,PCTSTRPOSY,pct_str,BACKGNDCOL,vscreen);
         pct_str[0]=(char)(pct_val/3600000L)+'0';
         pct_str[1]=(char)((pct_val%3600000L)/360000L)+'0';
         pct_str[3]=(char)((pct_val%360000L)/60000L)+'0';
         pct_str[4]=(char)((pct_val%60000L)/6000)+'0';
         pct_str[6]=(char)((pct_val%6000)/1000)+'0';
         pct_str[7]=(char)((pct_val%1000)/100)+'0';
         blit_string(PCTSTRPOSX,PCTSTRPOSY,pct_str,TEXTCOL,vscreen);
        }
        blit_string(TSGPSSTRPOSX,TSGPSSTRPOSY,tsgps_str,BACKGNDCOL,vscreen);
        tsec_since_gps=(int)(log_data.ticks-last_gps_hsec)/10;
        tsgps_str[0]=(char)((tsec_since_gps%1000)/100)+'0';
        tsgps_str[1]=(char)((tsec_since_gps%100)/10)+'0';
        tsgps_str[3]=(char)(tsec_since_gps%10)+'0';
        blit_string(TSGPSSTRPOSX,TSGPSSTRPOSY,tsgps_str,TEXTCOL,vscreen);
        break;
       case 6:        /* time = xxxxx.6 seconds -> update speed string */
        if(old_speed!=curr_speed){
         blit_string(SPDSTRPOSX,SPDSTRPOSY,speed_str,BACKGNDCOL,vscreen);
         old_speed=curr_speed;
         speed_str[0]=((old_speed%1000)/100)+'0';
         speed_str[1]=((old_speed%100)/10)+'0';
         speed_str[2]=(old_speed%10)+'0';
         blit_string(SPDSTRPOSX,SPDSTRPOSY,speed_str,TEXTCOL,vscreen);
        }
        break;
       case 8:        /* time = xxxxx.8 seconds -> update lat/long strings */
        if(old_calc_long!=calc_longitude || old_calc_lat!=calc_latitude){
         blit_string(LATSTRPOSX,LATSTRPOSY,lat_str,BACKGNDCOL,vscreen);
         blit_string(LONGSTRPOSX,LONGSTRPOSY,long_str,BACKGNDCOL,vscreen);
         tlat=calc_latitude;
         tlong=calc_longitude;
         if(tlat<0){
               lat_str[0]='-';
               tlat=0-tlat;
         }
         else lat_str[0]='+';
         if(tlong<0){
               long_str[0]='-';
               tlong=0-tlong;
         }
         else long_str[0]='+';
         for(i=0;i<9;i++){
               lat_str[9-i]=(char)(tlat%10)+'0';
               long_str[9-i]=(char)(tlong%10)+'0';
               tlat=tlat/10;
               tlong=tlong/10;
         }
         blit_string(LATSTRPOSX,LATSTRPOSY,lat_str,TEXTCOL,vscreen);
         blit_string(LONGSTRPOSX,LONGSTRPOSY,long_str,TEXTCOL,vscreen);
        }
        break;
       default:
        break;
       }
}
get_user_key();
switch(user_key){                  /* process run-time keypresses */
case 's':           /* stop running */
case 'S':
      running=0;
      break;
case '0':           /* select first sensor as current */
      curr_guage=0;
      draw_guage_border(BORDERCOL,vscreen);
      break;
case '1':           /* select second sensor as current */
      curr_guage=1;
      draw_guage_border(BORDERCOL,vscreen);
      break;
case '2':           /* select third sensor as current */
      curr_guage=2;
      draw_guage_border(BORDERCOL,vscreen);
      break;         /* select fourth sensor as current */
case '3':
      curr_guage=3;
      draw_guage_border(BORDERCOL,vscreen);
      break;
```

```
case '4':           /* select fifth sensor as current */
     curr_guage=4;
     draw_guage_border(BORDERCOL,vscreen);
     break;
case '5':           /* select sixth sensor as current */
     curr_guage=5;
     draw_guage_border(BORDERCOL,vscreen);
     break;
case 'i':           /* decrease low range value for current sensor */
     if(curr_guage){
      draw_guage_border(BACKGNDCOL,vscreen);
      draw_sensor_vals(BACKGNDCOL,vscreen);
      sen_low[curr_guage-1]-=GUAGE_DELTA;
      if(sen_low[curr_guage-1]<0) sen_low[curr_guage-1]=0;
      draw_guage_border(BORDERCOL,vscreen);
      draw_sensor_vals(SENSORCOL,vscreen);
     }
     break;
case 'I':           /* increase low range value for current sensor */
     if(curr_guage){
      draw_guage_border(BACKGNDCOL,vscreen);
      draw_sensor_vals(BACKGNDCOL,vscreen);
      sen_low[curr_guage-1]+=GUAGE_DELTA;
      if(sen_low[curr_guage-1]>sen_high[curr_guage-1])
       sen_low[curr_guage-1]=sen_high[curr_guage-1]-10;
      draw_guage_border(BORDERCOL,vscreen);
      draw_sensor_vals(SENSORCOL,vscreen);
     }
     break;
case 'u':           /* decrease high range value for current sensor */
     if(curr_guage){
      draw_guage_border(BACKGNDCOL,vscreen);
      draw_sensor_vals(BACKGNDCOL,vscreen);
      sen_high[curr_guage-1]-=GUAGE_DELTA;
      if(sen_high[curr_guage-1]<sen_low[curr_guage-1])
       sen_high[curr_guage-1]=sen_low[curr_guage-1]+10;
      draw_guage_border(BORDERCOL,vscreen);
      draw_sensor_vals(SENSORCOL,vscreen);
     }
     break;
case 'U':           /* increase high range value for current sensor */
     if(curr_guage){
      draw_guage_border(BACKGNDCOL,vscreen);
      draw_sensor_vals(BACKGNDCOL,vscreen);
      sen_high[curr_guage-1]+=GUAGE_DELTA;
      if(sen_high[curr_guage-1]>5000) sen_high[curr_guage-1]=5000;
      draw_guage_border(BORDERCOL,vscreen);
      draw_sensor_vals(SENSORCOL,vscreen);
     }
     break;
case 'q':           /* toggle quickview mode */
     if(postview){
      if(!quickview) quickview=1;
      else quickview=0;
     }
     break;
case 't':           /* move cursor up */
     calc_latitude+=5000L;
     break;
case 'v':           /* move cursor down */
     calc_latitude-=5000L;
     break;
case 'f':           /* move cursor left */
     calc_longitude-=5000L;
     break;
case 'g':           /* move cursor right */
     calc_longitude+=5000L;
     break;
case ' ':           /* put time into key file */
     if(logging) fwrite(&(log_data.ticks),sizeof(unsigned long),1,key_file);
     break;
```

```
  default:            /* no key or invalid key */
        break;
  }
 }
 CleanUpTimer();                      /* reset timer interrupt to old rate */
 CleanUpCom1();                       /* reset com port interrupt ISR */
 fclose(key_file);                    /* close files */
 fclose(log_file);
 fclose(gps_file);
 log_file=gps_file=key_file=NULL;

 /* output status message */
 if(!novga) blit_string(STATUSX,STATUSY,status_line,BACKGNDCOL,vscreen);
 sprintf(status_line,"Stopped: %s - Q to quit",log_file_name);
 if(!novga) blit_string(STATUSX,STATUSY,status_line,TEXTCOL,vscreen); \
 else printf("%s\n",status_line);

 for(user_key=0;user_key!='q' && user_key!='Q';get_user_key());     /* wait for ‘q’ */

 if(!novga){
  RestorePallette(OldPallette);       /* restore old pallette */
  set_text();                         /* reset to text mode */
 }
 goodbye(0);
}

/* interrupt handler for timer ISR */
void __interrupt __far Handler(void){
 hsec++;
 new_sense=1;                 /* 100th second elapsed, set new data flag */
 clock_ticks+=counter;
 if(clock_ticks>=0x10000L){
  clock_ticks-=0x10000L;
  (*BIOSTimerHandler)();
 }
 else outp(0x20,0x20);
}

/* interrupt handler for com port ISR */
void __interrupt __far Com1Handler(void){
 gps_char=inportb(GPS_PORT_DATA);
 if(!new_gps){
  switch(gps_report_idx){
  case 0:
  case 1:
        if(gps_char=='@'){
         gps_report[gps_report_idx]='@';
         gps_report_idx++;
        }
        break;
  case 2:
        if(gps_char!='B') gps_report_idx=0;
        else{
         gps_report[gps_report_idx]='B';
         gps_report_idx++;
        }
        break;
  default:
        gps_report[gps_report_idx]=gps_char;
        gps_report_idx++;
        break;
  }
  if(gps_report_idx==68){
        gps_report_idx=0;
        new_gps=1;
  }
 }
 outp(0x20,0x20);
}

/* set the timer rate and ISR function */
```

```
void SetTimer(void interrupt(__far *TimerHandler)(void), int frequency){
 clock_ticks=0;
 counter=(long)PIT_FREQ/frequency;
 BIOSTimerHandler=getvect(TIMERINTR);
 setvect(TIMERINTR,TimerHandler);
 outp(0x43,0x34);
 outp(0x40,counter%256);
 outp(0x40,counter/256);
}

/* set the com port mode and ISR function */
void SetCom1(void interrupt(__far *ComHandler)(void)){
 BIOSCom1Handler=getvect(COM1INTR);
 setvect(COM1INTR,ComHandler);
 gps_port_intr_set=inportb(GPS_PORT_INTR);
 outportb(GPS_PORT_CONT,11);
 outportb(GPS_PORT_INTR,1);
 inportb(GPS_PORT_DATA);
 outp(0x21,inportb(0x21)&0xEF);
}

/* reset timer rate and ISR function */
void CleanUpTimer(void){
 outp(0x43,0x34);
 outp(0x40,0x00);
 outp(0x40,0x00);
 setvect(TIMERINTR,BIOSTimerHandler);
}

/* reset com port ISR function */
void CleanUpCom1(void){
 outportb(GPS_PORT_INTR,gps_port_intr_set);
 setvect(COM1INTR,BIOSCom1Handler);
}

/* take the current palette and put it into memory */
void GrabPallette(unsigned char Pall[256][3]) {
	int loop1;
	for(loop1=0;loop1<256;loop1++){
		outp (0x03C7,loop1);
		Pall[loop1][0] = inp (0x03C9);
		Pall[loop1][1] = inp (0x03C9);
		Pall[loop1][2] = inp (0x03C9);
	}
}

/* take the palette from memory and make it current */
void RestorePallette(unsigned char Pall[256][3]) {
	int loop1;
	for(loop1=0; loop1<255; loop1++){
		outp(0x03C8,loop1);
		outp (0x03C9,Pall[loop1][0]);
		outp (0x03C9,Pall[loop1][1]);
		outp (0x03C9,Pall[loop1][2]);
	}
}

/* set a single palette member */
void Set_Palette_Register(int index,RGB_color_ptr color){
	outp(PALETTE_MASK,0xff);
	outp(PALETTE_REGISTER_WR,index);
	outp(PALETTE_DATA,color->red);
	outp(PALETTE_DATA,color->green);
	outp(PALETTE_DATA,color->blue);
}

/* draw a colored horizontal line into the video memory location vscreen */
void h_line(int x1,int x2,int y,unsigned int color,unsigned char far *vscreen){
	_fmemset((char*)(vscreen+((y<<8)+(y<<6))+x1),color,x2-x1+1);
}
```

```
/* draw a colored vertical line into the video memory location vscreen */
void v_line(int y1,int y2,int x,unsigned int color,unsigned char far *vscreen){
       unsigned int line_offset;
       int index;
       line_offset=((y1<<8)+(y1<<6))+x;      /* pixel location in memory */
       for(index=0;index<=(y2-y1);index++){  /* for each row in the line */
              vscreen[line_offset]=color;   /* set pixel value in memory */
              line_offset+=320;             /* increment to next line */
       }
}

/* draw a colored line from point p1 to p2 into video memory location vscreen */
void line(int x1,int y1,int x2,int y2,unsigned int color,unsigned char far *vscreen){
 int i,d1x,d1y,d2x,d2y,u,s,v,m,n;
 u=x2-x1;      /* overall ‘run’ of the line */
 v=y2-y1;      /* overall ‘rise’ of the line */
 d1x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d1y=sgn(v);   /* line left-to-right or right-to-left */
 d2x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d2y=0;
 m=abs(u);
 n=abs(v);
 if(m<=n){     /* if |slope| < 1 */
  d2x=0;
  d2y=sgn(v);
  m=abs(v);
  n=abs(u);
 }
 s=m/2;
 for(i=0;i<(m+1);i++){
  /* plot single pixel on the screen (only if within screen bounds) */
  if(y1>=0 && y1<=199 && x1>=0 && x1 <= 319) vscreen[(y1<<8)+(y1<<6)+x1]=color;
  s += n;
  /* increment to next pixel location */
  if(s>=m){
       s -= m;
       x1 += d1x;
       y1 += d1y;
  }
  else{
       x1 += d2x;
       y1 += d2y;
  }
 }
}

/* draw a colored line from point p1 to p2 into video memory location vscreen */
void map_line(POINT p1,POINT p2,unsigned int color,unsigned char far *vscreen){
 int i,d1x,d1y,d2x,d2y,u,s,v,m,n,x1,y1,x2,y2;
 x1=(int)p1.x; /* starting x point */
 y1=(int)p1.y; /* starting y point */
 x2=(int)p2.x; /* stopping x point */
 y2=(int)p2.y; /* stopping y point */
 u=x2-x1;      /* overall ‘run’ of the line */
 v=y2-y1;      /* overall ‘rise’ of the line */
 d1x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d1y=sgn(v);   /* line left-to-right or right-to-left */
 d2x=sgn(u);   /* line top-to-bottom or bottom-to-top */
 d2y=0;
 m=abs(u);
 n=abs(v);
 if(m<=n){     /* if |slope| < 1 */
  d2x=0;
  d2y=sgn(v);
  m=abs(v);
  n=abs(u);
 }
 s=m/2;
 for(i=0;i<(m+1);i++){
  /* plot single pixel on the screen (only if within map bounds) */
  if(y1 > MBULY && y1 < MBLRY && x1 > MBULX && x1 < MBLRX){
        vscreen[(y1<<8)+(y1<<6)+x1]=color;
  }
  s += n;
  /* increment to next pixel location */
  if(s>=m){
        s -= m;
        x1 += d1x;
        y1 += d1y;
  }
  else{
        x1 += d2x;
        y1 += d2y;
  }
 }
}

/* return the sign of a */
int sgn(int a){
 if(a>0) return 1;
 if(a<0) return -1;
 return 0;
}

/* draw a solid colored box */
void fill_box(int sx,int sy,int width,int height,unsigned int color,unsigned char far *vscreen){
 int offset,i;
 offset=(sy<<8) + (sy<<6) + sx;
 for(i=sy;i<sy+height;i++){
  _fmemset((char*)(vscreen+offset),color,width);
  offset+=320;
 }
}

/* fill the entire screen with a given color */
void fill_screen(unsigned int color, unsigned char far *vscreen){
        int i;
        for(i=0;i<200;i++) _fmemset((char*)(vscreen+(i<<8)+(i<<6)),color,320);
}

/* put a single pixel on the screen */
void blit_bit(int x,int y,unsigned int color,unsigned char far *vscreen){
 vscreen[(y<<8)+(y<<6)+x]=color;
}

/* set the screen to vga mode */
void set_vga(void){
        asm{
                pusha
                mov ax,0x0013
                int 0x10
                popa
        }
}

/* set the screen to text mode */
void set_text(void){
        asm{
                pusha
                mov ax,0x0003
                int 0x10
                popa
        }
}

/* setup the palette colors for this application */
void set_pallette(void){
        RGB_color color;
        color.red=0;
        color.green=0;
        color.blue=0;
        Set_Palette_Register(BLACK,&color);
        color.red=255;
        Set_Palette_Register(RED,&color);
        color.green=255;
        Set_Palette_Register(YELLOW,&color);
        color.red=0;
        Set_Palette_Register(GREEN,&color);
        color.blue=255;
        Set_Palette_Register(CYAN,&color);
        color.green=0;
        Set_Palette_Register(BLUE,&color);
        color.red=255;
        Set_Palette_Register(MAGENTA,&color);
        color.green=255;
        Set_Palette_Register(WHITE,&color);
}

/* put a single colored character on the screen an location (xc, yc) */
void blit_char(int xc,int yc,char c,int color,unsigned char far *vscreen){
        int offset,x,y;
        unsigned char far *work_char;
        unsigned char bit_mask=0x80;
        work_char=rom_char_set+c*CHAR_HEIGHT; /* get offset position into rom */
        offset=(yc<<8)+(yc<<6)+xc;            /* get offset position on the screen */
        for(y=0;y<CHAR_HEIGHT;y++){
                bit_mask=0x80;
                for(x=0;x<CHAR_WIDTH;x++){
                        /* put the character on the screen bit-by-bit */
                        if((*work_char & bit_mask)) vscreen[offset+x]=(unsigned char)(color);
                        bit_mask=(bit_mask>>1);
                }
                offset += SCREEN_WIDTH;
                work_char++;
        }
}

/* initialize the DaqBook to the desired mode */
void init_daqbook(void){
        daqSetErrHandler(daq_error);          /* establish error handler */
        daqInit(LPT1, 7);                     /* connect to daqboook */
        daqAdcSetTag(0);                      /* disable tagged ADC data */
        daqCtrSetMasterMode(1,DcsF5,0,0,0);   /* set counter master mode */
        /* string counter 1 and 2 for 32-bit event counting */
        daqCtrSetCtrMode(1,DgcNoGating,1,DcsSrc1,0,0,1,0,1,DocInactiveLow);
        daqCtrSetCtrMode(2,DgcNoGating,1,0,0,0,1,0,1,DocInactiveLow);
        daqCtrSetLoad(1,0);
        daqCtrSetLoad(2,0);
        daqCtrMultCtrl(DmccLoad, 1, 1, 0, 0, 0);
        daqCtrMultCtrl(DmccArm, 1, 1, 0, 0, 0);
}

/* error callback function for the DaqBook functions */
void _far _pascal daq_error(int error_code){
        clrscr();
        printf("\nError! Program aborted\nDaqBook/100 Error: 0x%x\n",error_code);
        farfree(old_data);
        exit(1);
}

/* detect GPS receiver and synchronize PC time to it */
void pc_set_to_gps(void){
        int i=0;
        gps_found=1;
        for(i=0;i<5000 && !get_gps_data();i++) delay(1);
        if(i<5000) settime(&gps_time);
        else{           /* no gps receiver found - do not collect data from it */
                gps_found=0;
        }
}

/* get the time from the PC BIOS */
void get_pc_time(void){
        gettime(&pc_time);
        hour = pc_time.ti_hour;
        min = pc_time.ti_min;
        sec = pc_time.ti_sec;
        pc_seconds = hour * 3600 + min * 60 + sec;
}

/* get the odometer reading from the DaqBook and compute relative distance */
void get_odometer_count(void){
        unsigned int count;
        unsigned long new_total;
        daqCtrMultCtrl(DmccSave, 1, 1, 0, 0, 0);
        daqCtrGetHold(2, &count);
        new_total = count;
        new_total *= 0x10000L;
        daqCtrGetHold(1, &count);
        new_total += count;
        odometer_sample = new_total - odometer_total;
        odometer_total = new_total;
}

/* detect a user key press */
void get_user_key(void){
        if (!kbhit()){
                user_key=0;
                return;
        }
        user_key=getch();
}

/* decode the raw GPS message into useable data */
void Pos_Status_Data_Decode(unsigned char *Status_Message,
T_POS_CHAN_STATUS *pos_chan, char scan_mode){
  UNSIGNED_ONEBYTE i;
  UNSIGNED_ONEBYTE tempchar;
  UNSIGNED_FOURBYTE tempu4byte;
  FOURBYTE temps4byte;
  double degrees, minutes;
  int message_posn = 0;

  /* skip first 4 bytes (@@Ba) */
          message_posn = 4;

  /* read and scale the rest of the data */
          pos_chan->month = Status_Message[message_posn++];
          pos_chan->day   = Status_Message[message_posn++];
          tempchar = Status_Message[message_posn++];
          pos_chan->year = ( tempchar << 8 ) + Status_Message[message_posn++];
          pos_chan->hours    = Status_Message[message_posn++];
          pos_chan->minutes  = Status_Message[message_posn++];
          tempchar   = Status_Message[message_posn++];  /* integer seconds */

          PACK8(    Status_Message[message_posn],
               Status_Message[message_posn+1],
               Status_Message[message_posn+2],
               Status_Message[message_posn+3],
               tempu4byte);
          message_posn += 4;
          pos_chan->seconds = (double) tempchar + ( ( (double) tempu4byte ) / 1.0E+9 );
          if (scan_mode != 0) return;    /* do not include position data unless asked */

          PACK8(    Status_Message[message_posn],
               Status_Message[message_posn+1],
               Status_Message[message_posn+2],
               Status_Message[message_posn+3],
               temps4byte);
          message_posn += 4;
          gps_latitude = temps4byte;
          degrees = (double) temps4byte * MSECS_TO_DEGREES ;
          pos_chan->latitude.degrees = (TWOBYTE) degrees ;
          if ( degrees < 0 )
```

```
         degrees = fabs ( degrees ) ;
         minutes =  ( degrees - (TWOBYTE) degrees ) * 60.0 ;
         pos_chan->latitude.minutes = (TWOBYTE) ( minutes ) ;
         pos_chan->latitude.seconds = ( minutes - (TWOBYTE) minutes ) * 60.0 ;

         PACK8(    Status_Message[message_posn],
              Status_Message[message_posn+1],
              Status_Message[message_posn+2],
              Status_Message[message_posn+3],
              temps4byte);
         message_posn += 4;
         gps_longitude = temps4byte;
         degrees = (double) temps4byte * MSECS_TO_DEGREES ;
         pos_chan->longitude.degrees = (TWOBYTE) degrees ;
         if ( degrees < 0 ) degrees = fabs ( degrees ) ;
         minutes = ( degrees - (TWOBYTE) degrees ) * 60.0 ;
         pos_chan->longitude.minutes = (TWOBYTE) ( minutes ) ;
         pos_chan->longitude.seconds = ( minutes - (TWOBYTE) minutes ) * 60.0 ;
         templong=pos_chan->longitude.seconds+pos_chan->longitude.minutes*60.0+
                      abs(pos_chan->longitude.degrees)*3600.0;
         gps_longitude=(long)(templong*1000.0);
         if(pos_chan->longitude.degrees<0.0) gps_longitude=0-gps_longitude;

         PACK8(Status_Message[message_posn],
              Status_Message[message_posn+1],
              Status_Message[message_posn+2],
              Status_Message[message_posn+3],
              temps4byte);
         message_posn += 4;
         pos_chan->datum_height = (double) temps4byte / 100.0 ;

         PACK8(Status_Message[message_posn],
              Status_Message[message_posn+1],
              Status_Message[message_posn+2],
              Status_Message[message_posn+3],
              temps4byte);
         message_posn += 4;
         pos_chan->msl_height = (double) temps4byte / 100.0 ;
         tempchar = Status_Message[message_posn++];
         pos_chan->velocity = (double)((tempchar<<8)+Status_Message[message_posn++])/100.0;
         tempchar = Status_Message[message_posn++];
         pos_chan->heading = (double)((tempchar<<8)+Status_Message[message_posn++])/10.0;
         tempchar = Status_Message[message_posn++];
         pos_chan->current_dop = (double)((tempchar<<8)+Status_Message[message_posn++])/10.0;
         pos_chan->dop_type     = Status_Message[message_posn++];
         pos_chan->visible_sats = Status_Message[message_posn++];
         pos_chan->sats_tracked = Status_Message[message_posn++];
         for (i = 0; i < NUM_CHANNELS; i++) {
               pos_chan->channel[i].svid     = Status_Message[message_posn++];
               pos_chan->channel[i].mode     = Status_Message[message_posn++];
               pos_chan->channel[i].strength = Status_Message[message_posn++];
               pos_chan->channel[i].flags    = Status_Message[message_posn++];
         }
         pos_chan->rcvr_status = Status_Message[message_posn++];
}

/* draw the border lines around the main map view */
void draw_map_border(unsigned int color,unsigned char far *vscreen){
 h_line(MBULX,MBLRX,MBULY,color,vscreen);
 h_line(MBULX,MBLRX,MBLRY,color,vscreen);
 v_line(MBULY,MBLRY,MBULX,color,vscreen);
 v_line(MBULY,MBLRY,MBLRX,color,vscreen);
}

/* draw the border lines around the sensor boxes */
void draw_sensor_border(unsigned int color,unsigned char far *vscreen){
 h_line(SBULX,SBLRX,SBULY,color,vscreen);
 h_line(SBULX,SBLRX,SBLRY,color,vscreen);
 v_line(SBULY,SBLRY,SBULX,color,vscreen);
 v_line(SBULY,SBLRY,SBLRX,color,vscreen);
}
```

```
/* draw the border lines around the GPS info region */
void draw_gps_border(unsigned int color,unsigned char far *vscreen){
 h_line(GBULX,GBLRX,GBULY,color,vscreen);
 h_line(GBULX,GBLRX,GBLRY,color,vscreen);
 v_line(GBULY,GBLRY,GBULX,color,vscreen);
 v_line(GBULY,GBLRY,GBLRX,color,vscreen);
}

/* draw the border around each of the individual sensor boxes */
void draw_guage_border(unsigned int color,unsigned char far *vscreen){
 int i,tcol,ticol;
 if(color==BACKGNDCOL) ticol=BACKGNDCOL;
 else ticol=BROWN;
 for(i=0;i<5;i++){
  if(i==(curr_guage-1) && color!=BACKGNDCOL) tcol=BORDERSELECTCOL;
  else tcol=color;
  h_line(SBULX+SENOFFX+(i*SENSPACE)+1,SBULX+SENOFFX+SENWIDTH+(i*SENSPACE),
                      SBULY+SENOFFY,tcol,vscreen);
  h_line(SBULX+SENOFFX+(i*SENSPACE)+1,SBULX+SENOFFX+SENWIDTH+(i*SENSPACE),
                      SBULY+SENOFFY+SENHEIGHT,tcol,vscreen);
  v_line(SBULY+SENOFFY,SBULY+SENOFFY+SENHEIGHT,SBULX+SENOFFX+(i*SENSPACE)+1,
                      tcol,vscreen);
  v_line(SBULY+SENOFFY,SBULY+SENOFFY+SENHEIGHT,
                      SBULX+SENOFFX+(i*SENSPACE)+SENWIDTH,tcol,vscreen);
  v_line(SBULY+SENOFFY+(int)(((5000.0-sen_high[i])*SENHEIGHT)/5000.0),
                      SBULY+SENOFFY+(int)(((5000.0-sen_low[i])*SENHEIGHT)/5000.0),
                      SBULX+SENOFFX+(i*SENSPACE),ticol,vscreen);
 }
}

/* plot a string to the output window starting at position sx, sy */
void blit_string(int sx,int sy,char* str,unsigned int color,unsigned char far* vscreen){
 int i=0;
 while(str[i]!=0){
  blit_char(sx,sy,str[i],color,vscreen);
  sx+=8;
  i++;
 }
}

/* draw a bar representing one of the sensor values */
void draw_sensor_vals(unsigned int color,unsigned char far *vscreen){
 int i,offset;
 for(i=0;i<5;i++){
  if(sen_val[i]<=sen_high[i] && sen_val[i]>=sen_low[i]){
       offset=(int)(((float)(sen_high[i]-sen_val[i])/
                       (float)(sen_high[i]-sen_low[i]))*(float)(SENHEIGHT-2))+1;
       h_line(SBULX+SENOFFX+(i*SENSPACE)+2,SBULX+SENOFFX+SENWIDTH+(i*SENSPACE)-1,
                       SBULY+SENOFFY+offset,color,vscreen);
  }
 }
}

/* draw border around satellite signal strength indicators */
void draw_strength_border(unsigned int color,unsigned char far *vscreen){
 int i;
 for(i=0;i<6;i++){
  h_line(GBULX+STROFFX+(i*STRSPACE),GBULX+STROFFX+STRWIDTH+(i*STRSPACE),
                      GBULY+STROFFY,color,vscreen);
  h_line(GBULX+STROFFX+(i*STRSPACE),GBULX+STROFFX+STRWIDTH+(i*STRSPACE),
                      GBULY+STROFFY+STRHEIGHT,color,vscreen);
  v_line(GBULY+STROFFY,GBULY+STROFFY+STRHEIGHT,GBULX+STROFFX+(i*STRSPACE),
                      color,vscreen);
  v_line(GBULY+STROFFY,GBULY+STROFFY+STRHEIGHT,
                      GBULX+STROFFX+(i*STRSPACE)+STRWIDTH,color,vscreen);
 }
}

/* draw current satellite strength bar */
void draw_strength_vals(unsigned int color,unsigned char far *vscreen){
 int i,offset;
 for(i=0;i<6;i++){
   offset=(int)(((float)(STRUPPER-strength[i])/
                (float)(STRUPPER-STRLOWER))*(float)(STRHEIGHT-2))+1;
   h_line(GBULX+STROFFX+(i*STRSPACE)+1,GBULX+STROFFX+STRWIDTH+(i*STRSPACE)-1,
                GBULY+STROFFY+offset,color,vscreen);
 }
}

/* put a bitmap with width, height at position sx, sy */
void put_box(int sx,int sy,int width,int height,unsigned char* box, unsigned char far* vscreen){
 int vsoff,i,j,boxoff;
 boxoff=0;
 vsoff=(sy<<6) + (sy<<8) + sx;
 for(i=0;i<height;i++){
  for(j=0;j<width;j++) vscreen[vsoff+j]=box[boxoff++];
  vsoff+=320;
 }
}

/* get a bitmap with width, height at position sx, sy */
void get_box(int sx,int sy,int width,int height,unsigned char* box, unsigned char far* vscreen){
 int vsoff,i,j,boxoff;
 boxoff=0;
 vsoff=(sy<<6) + (sy<<8) + sx;
 for(i=0;i<height;i++){
  for(j=0;j<width;j++) box[boxoff++]=vscreen[vsoff+j];
  vsoff+=320;
 }
}

/* new GPS data collected – draw new values and indicators on the screen */
void new_gps_stuff(void){
 int i;
 draw_strength_vals(BACKGNDCOL,vscreen);
 for(i=0;i<6;i++) strength[i]=GPS_chan.channel[i].strength;
 draw_strength_vals(YELLOW,vscreen);
 if(old_gps_valid!=gps_valid){
  if(gps_valid) fill_box(GPSVALPOSX,GPSVALPOSY,GPSVALWIDTH,GPSVALHEIGHT,GREEN,vscreen);
  else fill_box(GPSVALPOSX,GPSVALPOSY,GPSVALWIDTH,GPSVALHEIGHT,RED,vscreen);
  old_gps_valid=gps_valid;
 }
 if(old_sats_tracked!=GPS_chan.sats_tracked){
  blit_char(SATTRKPOSX,SATTRKPOSY,old_sats_tracked+'0',BACKGNDCOL,vscreen);
  old_sats_tracked=GPS_chan.sats_tracked;
  blit_char(SATTRKPOSX,SATTRKPOSY,old_sats_tracked+'0',TEXTCOL,vscreen);
 }
}

/* convert a lat/long point to screen coordinates */
POINT ll_to_screen(POINT llp){
 int offx,offy;
 POINT retval;
 offx=(int)(((float)(MBLRX-MBULX)/(float)(maplrlong-mapullong))*(float)(llp.x-mapullong));
 offy=(int)(((float)(MBLRY-MBULY)/(float)(mapullat-maplrlat))*(float)(mapullat-llp.y));
 retval.x=MBULX+offx;
 retval.y=MBULY+offy;
 return retval;
}
```

```
/* return 1 if point p is in the map view, else return 0 */
int in_view(POINT p){
 if(p.x>=mapullong && p.x<=maplrlong &&p.y<=mapullat && p.y>=maplrlat) return 1;
 return 0;
}

/* close all files, free all allocated data, and exit the program */
void goodbye(int usage){
 if(log_file!=NULL) fclose(log_file);
 if(gps_file!=NULL) fclose(gps_file);
 if(map_file!=NULL) fclose(map_file);
 if(cfg_file!=NULL) fclose(cfg_file);
 if(key_file!=NULL) fclose(key_file);
 if(debug!=NULL) fclose(map_file);
 if(old_data!=NULL) farfree(old_data);
 if(map_roads!=NULL) farfree(map_roads);
 if(map_segments!=NULL) farfree(map_segments);
 /* print a little message if usage flag is set */
 if(usage){
  printf("Usage:\n");
  printf("   'maplog {-p/-l} filename [-m mapfile] [-novga]' (no extensions)\n");
  printf("Exiting . . .\n");
 }
 exit(0);
}

/* draw all of the map lines in the map window */
void draw_map(unsigned int color,unsigned char far* vscreen){
 int i;
 for(i=0;i<num_map_segments;i++){
  if(in_view(map_segments[i].p1) || in_view(map_segments[i].p2))
        map_line(ll_to_screen(map_segments[i].p1),ll_to_screen(map_segments[i].p2),
                                              color,vscreen);
 }
}
```

D.2 Windows Program Listings

The Windows 95/98/NT program that was written for this thesis was used primarily to process the previously collected data using the implemented filter, and to generate and display the filtered output. This program used a simple dialog based application structure to view the data from the GPS and inertial sensors, and displayed the map and processed output in a separate graphical display window.

The program follows the basic flow as shown in Figure 6.2 on page 43, with the inputs being from the hard drive (pre-collected data) and the output to the screen. The methods which operate at 1 Hz and 100 Hz are `Filter1Hz` and `Filter100Hz`, respectively. The `OnIdle` method is called by Windows and performs the necessary synchronization of `Filter1Hz` and `Filter100Hz`. Listed below are several of these functions that actually did the filtering and fusion in the program. The program contains a large amount of graphical interface code that is not relevant to the operation of the filter, and has therefore been omitted from this listing.

```
// OnIdle is called by Windows when it is not doing anything else
BOOL CPostViewApp::OnIdle(LONG lCount)
{
if(m_pMainPage->m_RunCheckVar)       // if running
        {
                m_RunCount++;
                // read next sensor value from file -> time data is global time
                if(m_pLogFile->Read(&m_CurrSensorData,sizeof(T_LOG_DATA))
!=sizeof(T_LOG_DATA))
                {
                        m_pMainPage->SetRunState(FALSE);
                        return TRUE;
                }
                // first time execution: read GPS value no matter what
                if(!m_NextGPSTime)
                {
                        // set the current GPS data to the first GPS data in the file,
                        //  even if it isn't quite time yet
                        if(m_pGPSFile->Read(&m_NextGPSTime,sizeof(unsigned int))!=
sizeof(unsigned int))
                        {
                                m_pMainPage->SetRunState(FALSE);
                                return TRUE;
                        }
                        m_pGPSFile->Read(&m_NextGPSData,sizeof(T_POS_CHAN_STATUS));
                        memcpy(&m_CurrGPSData,&m_NextGPSData,sizeof(T_POS_CHAN_STATUS));
                        m_CurrGPSTime = m_NextGPSTime;
                        m_GPSLatitude=(int)((m_CurrGPSData.latitude.seconds+
                                m_CurrGPSData.latitude.minutes*60.0+
                                abs(m_CurrGPSData.latitude.degrees)*3600.0)*1000.0)*
                                SGN((m_CurrGPSData.latitude.degrees));
                        m_GPSLongitude=(int)((m_CurrGPSData.longitude.seconds+
                                m_CurrGPSData.longitude.minutes*60.0+
                                abs(m_CurrGPSData.longitude.degrees)*3600.0)*1000.0)*
                                SGN((m_CurrGPSData.longitude.degrees));
                        if((m_CurrGPSData.rcvr_status & 0x43) ||
                                !(m_CurrGPSData.rcvr_status & 0x30))
                                        m_GPSGood = FALSE;
                        else m_GPSGood = TRUE;
                        m_pGPSPage->m_GoodGPSCheckVar = m_GPSGood;
                        m_pGPSPage->UpdateContents();
                }
                // run Filter100Hz (for sensor data) -> update sensor page
                if(m_pMainPage->m_HertzEditVar &&
!(m_RunCount%(m_pMainPage->m_HertzEditVar/10)))
                        m_pSensorPage->UpdateContents();
                else if(!(m_RunCount%50)) m_pSensorPage->UpdateContents();

                Filter100Hz();          // execute the inertial sensor filter

                // if time on sensor>=next GPS time (synchronize GPS and inertial data)
                if(m_CurrSensorData.ticks>=m_NextGPSTime)
                {
                        //    copy m_NextGPSData to m_CurrGPSData
                        memcpy(&m_CurrGPSData,&m_NextGPSData,sizeof(T_POS_CHAN_STATUS));
                        m_CurrGPSTime = m_NextGPSTime;
                        //    update m_GPSLatitude, m_GPSLongitude, and m_GPSGood
                        m_GPSLatitude=(int)((m_CurrGPSData.latitude.seconds+
                                m_CurrGPSData.latitude.minutes*60.0+
                                abs(m_CurrGPSData.latitude.degrees)*3600.0)*1000.0)*
                                SGN((m_CurrGPSData.latitude.degrees));
                        m_GPSLongitude=(int)((m_CurrGPSData.longitude.seconds+
                                m_CurrGPSData.longitude.minutes*60.0+
                                abs(m_CurrGPSData.longitude.degrees)*3600.0)*1000.0)*
                                SGN((m_CurrGPSData.longitude.degrees));
                        if((m_CurrGPSData.rcvr_status & 0x43) ||
!(m_CurrGPSData.rcvr_status & 0x30))
                                m_GPSGood = FALSE;
                        else m_GPSGood = TRUE;
                        //    update GPS page (w/ new current GPS data)
                        m_pGPSPage->m_GoodGPSCheckVar = m_GPSGood;
                        m_pMapDlg->
PlotRawGPS(CPoint(m_GPSLongitude,m_GPSLatitude),m_GPSGood);
                        // -> update output window
                        m_pGPSPage->UpdateContents();
                        //    Filter1Hz (for GPS data)
                        if(!m_Fuzzy) Filter1Hz();       // use rule-based fusion
                        else Fuzzy1Hz();                // use fuzzy fusion
                        //    read next GPS data into m_NextGPSData
                        if(m_pGPSFile->Read(&m_NextGPSTime,sizeof(unsigned int))
!=sizeof(unsigned int))
                        {
                                m_pMainPage->SetRunState(FALSE);
                                return TRUE;
                        }
                        m_pGPSFile->Read(&m_NextGPSData,sizeof(T_POS_CHAN_STATUS));
                }
                // sleep to effect a pseudo-accurate timing
                if(m_pMainPage->m_HertzEditVar) Sleep(1000/(m_pMainPage->m_HertzEditVar));
        }
        else Sleep(100);        // keep from bogging down system in tight loop

        return TRUE;    // return TRUE so that OnIdle is called again
}
```

```
void CPostViewApp::InitFilter(void)
{
	int i,j;
	// do anything related to initializing the filter here
	// (just to distinguish filter related initialization)

	// allocate vectors and matrices
	xk = vector(1,8);
	xkm1 = vector(1,8);
	yk = vector(1,2);
	Ak = matrix(1,8,1,8);
	Kk = matrix(1,8,1,2);
	C = matrix(1,2,1,8);
	Pk = matrix(1,8,1,8);
	Pk1 = matrix(1,8,1,8);
	Pkm1 = matrix(1,8,1,8);
	R = matrix(1,2,1,2);
	Q = matrix(1,8,1,8);
	mtemp1=matrix(1,8,1,8);
	mtemp2=matrix(1,8,1,8);
	mtemp3=matrix(1,2,1,8);
	mtemp4=matrix(1,2,1,2);
	mtemp5=matrix(1,2,1,2);
	mtemp6=matrix(1,2,1,2);
	mtemp7=matrix(1,8,1,2);
	mtemp8=matrix(1,2,1,8);
	mtemp9=vector(1,2);
	mtemp10=vector(1,2);
	mtemp11=vector(1,8);
	mtemp12=vector(1,8);
	mtemp13=matrix(1,8,1,8);
	mtemp14=matrix(1,8,1,8);

	// fill in intial values for Ak,C,Pkm1,Q,R
	// leave xkm1 all 0's until good GPS

	// initialize state transition matrix
	for(i=1;i<=8;i++) for(j=1;j<=8;j++) Ak[i][j]=Pkm1[i][j]=Q[i][j]=0;
	for(i=1;i<=2;i++)
	{
		for(j=1;j<=2;j++) R[i][j]=0;
		for(j=1;j<=8;j++) C[i][j]=0;
	}
	for(i=1;i<=8;i++) Ak[i][i]=1;
	Ak[3][4] = DELTA_TIME;
	Ak[3][5] = SQR(DELTA_TIME)/2;
	Ak[4][5] = DELTA_TIME;
	Ak[6][7] = DELTA_TIME;
	Ak[6][8] = SQR(DELTA_TIME)/2;
	Ak[7][8] = DELTA_TIME;
	// initialize observation matrix
	C[1][4] = 1;
	C[2][6] = 1;
	// initialize variance of initial errors in Pkm1
	for(i=1;i<=8;i++) Pkm1[i][i] = 0.02;
	// initialize measurement noise matrix (R)
	R[1][1] = 2.618;
	R[2][2] = 0.00251;
	// initialize state noise matrix (Q)
	//for(i=1;i<=8;i++) Q[i][i] = 0.05;
	Q[1][1] = 7000;
	Q[2][2] = 7000;
	Q[3][3] = 30;
	Q[4][4] = 2;
	Q[5][5] = 0.856;
	Q[6][6] = 0.0463;
	Q[7][7] = 0.0533;
	Q[8][8] = 0.0304;

	m_CurrHeading = 0;
	for(i=0;i<HEADING_FILT_SIZE;i++) m_HeadingFilt[i]=0;
```

```
        // initialize variables representing physical constants, conversions, etc
        m_fInitGPS = FALSE;
        m_fxkValid = FALSE;
        m_NextGPSTime = 0;
        m_RunCount = 0;
        old_odo_total = 0;
        m_MvoltPerDegree = 9.5;
        m_TicksPerMeter = 27.3;
        m_GyroCenter = 3069.0;
        m_SteerCenter = 2919.8;
        m_GPSVelocityThreshold=2.8;
        m_GPSPosWeight=0.04;
        m_GPSHeadingWeight=0.6;
        m_GPSPosThreshold=(3.0E8);
        m_MaxDistPer100 = 0.45;
        m_NumGoodNoWeight=0;
        odo_zero_count = 0;
        m_GPSNoUpdateThresh=30;
        m_GPSNoUpdateThresh2=3000;
        meters_to_msec_long=meters_to_msec_lat=0;
        filter_count=0;

        // Initialize fuzzy variables
        GPSToCurrDist.SetNumMembers(3);
        GPSToCurrDist.SetMemberFunc(SMALL_3,0,0,1.5E7,2.0E7);       // SMALL_3
        GPSToCurrDist.SetMemberFunc(MED_3,1.0E7,5.0E7,1.0E8,3.5E8); // MED_3
        GPSToCurrDist.SetMemberFunc(LARGE_3,2.0E8,3.0E8,10.0E12,10.0E12);  // LARGE_3
        GPSCurrVelocity.SetNumMembers(3);
        GPSCurrVelocity.SetMemberFunc(SLOW_3,0,0,2.0,2.5);          // SLOW_3
        GPSCurrVelocity.SetMemberFunc(MED_3,2.0,2.5,3.0,3.5);       // MED_3
        GPSCurrVelocity.SetMemberFunc(FAST_3,2.5,3.0,500.0,500.0);  // FAST_3
        GPSHeadWeight.SetNumMembers(5);
        GPSHeadWeight.SetMemberFunc(ZERO_5,0,0,0.03,0.05);          // ZERO_5
        GPSHeadWeight.SetMemberFunc(SMALL_5,0.02,0.05,0.25,0.3);    // SMALL_5
        GPSHeadWeight.SetMemberFunc(MED_5,0.25,0.3,0.4,0.5);        // MED_5
        GPSHeadWeight.SetMemberFunc(LARGE_5,0.4,0.5,0.8,0.95);      // LARGE_5
        GPSHeadWeight.SetMemberFunc(ONE_5,0.9,0.95,1,1);            // ONE_5
        GPSPosWeight.SetNumMembers(5);
        GPSPosWeight.SetMemberFunc(ZERO_5,0,0,0.001,0.001);         // ZERO_5
        GPSPosWeight.SetMemberFunc(SMALL_5,0,0.002,0.015,0.02);     // SMALL_5
        GPSPosWeight.SetMemberFunc(MED_5,0.015,0.02,0.02,0.03);     // MED_5
        GPSPosWeight.SetMemberFunc(LARGE_5,0.02,0.03,0.05,0.10);    // LARGE_5
        GPSPosWeight.SetMemberFunc(ONE_5,0.999,0.999,1,1);          // ONE_5
        NumNoGood.SetNumMembers(3);
        NumNoGood.SetMemberFunc(SMALL_3,0,0,25,35);                 // SMALL_3
        NumNoGood.SetMemberFunc(MED_3,25,35,2500,3500);             // MED_3
        NumNoGood.SetMemberFunc(LARGE_3,2500,3500,1000000,1000000); // LARGE_3
        DistRatio.SetNumMembers(2);
        DistRatio.SetMemberFunc(SMALL_2,0,0,1.0,1.5);               // SMALL_2
        DistRatio.SetMemberFunc(LARGE_2,1.0,1.5,1000000,1000000);   // LARGE_2

        m_FilterInitialized = TRUE;
}
```

```
void CPostViewApp::StopFilter(void)
{
	// free all of the allocated matrices
	free_vector(xk,1,8);
	free_vector(xkm1,1,8);
	free_vector(yk,1,2);
	free_matrix(Ak,1,8,1,8);
	free_matrix(Kk,1,8,1,2);
	free_matrix(C,1,2,1,8);
	free_matrix(Pk,1,8,1,8);
	free_matrix(Pk1,1,8,1,8);
	free_matrix(Pkm1,1,8,1,8);
	free_matrix(R,1,2,1,2);
	free_matrix(Q,1,8,1,8);
	free_matrix(mtemp1,1,8,1,8);
	free_matrix(mtemp2,1,8,1,8);
	free_matrix(mtemp3,1,2,1,8);
	free_matrix(mtemp4,1,2,1,2);
	free_matrix(mtemp5,1,2,1,2);
	free_matrix(mtemp6,1,2,1,2);
	free_matrix(mtemp7,1,8,1,2);
	free_matrix(mtemp8,1,2,1,8);
	free_vector(mtemp9,1,2);
	free_vector(mtemp10,1,2);
	free_vector(mtemp11,1,8);
	free_vector(mtemp12,1,8);
	free_matrix(mtemp13,1,8,1,8);
	free_matrix(mtemp14,1,8,1,8);

	m_FilterInitialized = FALSE;
}


// Filter1Hz is called to perform the older 'rule-based' sensor fusion when new GPS data
//  is received. Either Filter1Hz or Fuzzy1Hz is called, depending on the user selection
void CPostViewApp::Filter1Hz(void)
{
	// we just 'received' new GPS data and wish to use rule-based fusion
//  (roughly 1 Hz)
	if(!m_fInitGPS && m_GPSGood)
	{
		m_fInitGPS = TRUE;
		// just got first good GPS value -> initialize position, etc.
//  in state matrix
		double newhead = (90.0-m_CurrGPSData.heading);
		newhead=fmod(newhead+360.0,360.0);
		xkm1[1] = m_GPSLongitude;
		xkm1[2] = m_GPSLatitude;
		xkm1[3] = newhead;
		// calculate local meters -> msec lat/long here
		// first, convert latitude msec to radians
		double tlat = DEG_TO_RAD(m_GPSLatitude/3600000.0);
		double tsin = sin(tlat);
		meters_to_msec_long = pow(1-E2*tsin*tsin,0.5)/(0.03083070*cos(tlat));
		meters_to_msec_lat = pow(1-E2*tsin*tsin,1.5)/0.030715169;
		m_LastGoodGPSLat = xkm1[2];
		m_LastGoodGPSLong = xkm1[1];
		m_LastGoodGPSDLat = 0;
		m_LastGoodGPSDLong = 0;
	}
	else if(m_GPSGood && m_FuseData)
	{
		double d1 = m_GPSLongitude-xkm1[1];
		double d2 = m_GPSLatitude-xkm1[2];
		if((SQR(d1)+SQR(d2))<=(m_GPSPosThreshold))
		{
			if(m_CurrGPSData.velocity>=(m_GPSVelocityThreshold))
			{
				// perform median filtering on the GPS heading outputs
//  to eliminate glitches
				m_HeadingFilt[m_CurrHeading]=m_CurrGPSData.heading;
```

```
                              m_CurrHeading = (m_CurrHeading+1)%HEADING_FILT_SIZE;
                              double SortHeading[HEADING_FILT_SIZE];
                              for(int i=0;i<HEADING_FILT_SIZE;i++) SortHeading[i] =
m_HeadingFilt[i];
                              qsort(SortHeading,HEADING_FILT_SIZE,sizeof(double),dcomp);
                              m_GPSHeading = SortHeading[(HEADING_FILT_SIZE-1)/2];
                              // weighted sum of GPS position and current position
//  using resultant GPSPosWeight
                              xkm1[1]=(m_GPSPosWeight*m_GPSLongitude)+((1.0-
(m_GPSPosWeight))*xkm1[1]);
                              xkm1[2]=(m_GPSPosWeight*m_GPSLatitude)+((1.0-
(m_GPSPosWeight))*xkm1[2]);
                              double newhead;
                              // assumed heading and GPS sensor heading are
//  90 degrees out of phase and inverted
                              double gpshead = (90.0-m_GPSHeading);
                              while((xkm1[3]-gpshead)>180.0) gpshead+=360.0;
                              while((gpshead-xkm1[3])>180.0) gpshead-=360.0;
                              // weighted sum of GPS heading and current heading
//  using resultant GPSHeadingWeight
                              newhead = m_GPSHeadingWeight*gpshead+(1-
(m_GPSHeadingWeight))*xkm1[3];
                              xkm1[3] = newhead;
                      }
                      m_LastGoodGPSLat = xkm1[2];
                      m_LastGoodGPSLong = xkm1[1];
                      m_LastGoodGPSDLat = 0;
                      m_LastGoodGPSDLong = 0;
                      m_NumGoodNoWeight=0;
              }
              else if((++m_NumGoodNoWeight)>m_GPSNoUpdateThresh)
              {
                      // GPS data has been valid for some time, but is far enough
//  away such that is considered inaccurate
                      double LastGoodGPSGPSDist = SQR(m_GPSLongitude-m_LastGoodGPSLong) +
                              SQR(m_GPSLatitude-m_LastGoodGPSLat);
                      double LastGoodGPSOdoDist =   SQR(m_LastGoodGPSDLat) +
SQR(m_LastGoodGPSDLong);
                      if(LastGoodGPSGPSDist<=(1.25*LastGoodGPSOdoDist) ||
                              m_NumGoodNoWeight>m_GPSNoUpdateThresh2)
                      {
                              // we think we messed up... reinitialize position and
//  heading as if new beginning
                              double newhead = (90.0-m_GPSHeading);
                              newhead=fmod(newhead+360.0,360.0);
                              xkm1[1] = m_GPSLongitude;
                              xkm1[2] = m_GPSLatitude;
                              xkm1[3] = newhead;
                              m_NumGoodNoWeight=0;
                      }
              }
      }
      // else GPS data is invalid, so ignore it altogeter

      // plot the filtered position in the output window
      if(m_fInitGPS && m_fxkValid) m_pMapDlg->
PlotFiltPos(CPoint(int(xk[1]),int(xk[2])));
}
```

```
// Fuzzy1Hz is called to perform the fuzzy-based sensor fusion when new GPS data
//  is received. Either Filter1Hz or Fuzzy1Hz is called, depending on the user selection
void CPostViewApp::Fuzzy1Hz(void)
{
        // we just 'received' new GPS data and wish to 'Fuzzy' fuse GPS and sensor data
//  (roughly 1 Hz)
        if(!m_fInitGPS && m_GPSGood)
        {
                m_fInitGPS = TRUE;
                // just got first good GPS value -> initialize position, etc.
//  in state matrix
                double newhead = (90.0-m_CurrGPSData.heading);
                newhead=fmod(newhead+360.0,360.0);
                xkm1[1] = m_GPSLongitude;
                xkm1[2] = m_GPSLatitude;
                xkm1[3] = newhead;
                // calculate local meters -> msec lat/long here
                // first, convert latitude msec to radians
                double tlat = DEG_TO_RAD(m_GPSLatitude/3600000.0);
                double tsin = sin(tlat);
                meters_to_msec_long = pow(1-E2*tsin*tsin,0.5)/(0.03083070*cos(tlat));
                meters_to_msec_lat = pow(1-E2*tsin*tsin,1.5)/0.030715169;
                m_LastGoodGPSLat = xkm1[2];
                m_LastGoodGPSLong = xkm1[1];
                m_LastGoodGPSDLat = 0;
                m_LastGoodGPSDLong = 0;
        }
        else if(m_GPSGood && m_FuseData)
        {
                // fuzzy rules
                // if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is FAST)
//  then (GPSPosWeight is LARGE)
                // if (GPSToCurrDist is MED) and (GPSCurrVelocity is FAST)
//  then (GPSPosWeight is MED)
                // if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is MED)
//  then (GPSPosWeight is MED)
                // if (GPSToCurrDist is MED) and (GPSCurrVelocity is MED)
//  then (GPSPosWeight is SMALL)
                // if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is SLOW)
//  then (GPSPosWeight is SMALL)
                // if (GPSToCurrDist is MED) and (GPSCurrVelocity is SLOW)
//  then (GPSPosWeight is ZERO)
                // if (GPSToCurrDist is LARGE) and (NumNoGood is SMALL)
//  then (GPSPosWeight is ZERO)
                // if (GPSToCurrDist is LARGE) and (NumNoGood is MED)
//  then (GPSPosWeight is ZERO)
                // if (GPSToCurrDist is LARGE) and (NumNoGood is LARGE)
//  then (GPSPosWeight is ONE)
                // GPSHeadWeight similar to GPSPosWeight, but with
//  different membership values
                GPSToCurrDist.SetValue(SQR(m_GPSLongitude-xkm1[1])+
SQR(m_GPSLatitude-xkm1[2]));
                GPSCurrVelocity.SetValue(m_CurrGPSData.velocity);
                double LastGoodGPSGPSDist = SQR(m_GPSLongitude-m_LastGoodGPSLong) +
SQR(m_GPSLatitude-m_LastGoodGPSLat);
                double LastGoodGPSOdoDist =   SQR(m_LastGoodGPSDLat) +
SQR(m_LastGoodGPSDLong);
                DistRatio.SetValue(LastGoodGPSGPSDist/LastGoodGPSOdoDist);
                NumNoGood.SetValue(m_NumGoodNoWeight);
                // determine m_GPSHeadingWeight and m_GPSPosWeight based on
//  fuzzy membership values

                double memval;
                // if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is FAST)
//  then (GPSPosWeight is LARGE)
                memval=MIN(GPSToCurrDist.GetMembership(SMALL_3),
GPSCurrVelocity.GetMembership(FAST_3));
                GPSPosWeight.SetMembership(LARGE_5,memval);
                GPSHeadWeight.SetMembership(LARGE_5,memval);
                // if (GPSToCurrDist is MED) and (GPSCurrVelocity is FAST)
//  then (GPSPosWeight is MED)
```

```
               // if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is MED)
//  then (GPSPosWeight is MED)
               memval=MAX(
MIN(GPSToCurrDist.GetMembership(MED_3),
GPSCurrVelocity.GetMembership(FAST_3)),
MIN(GPSToCurrDist.GetMembership(SMALL_3),
GPSCurrVelocity.GetMembership(MED_3)));
               GPSPosWeight.SetMembership(MED_5,memval);
               GPSHeadWeight.SetMembership(MED_5,memval);
               // if (GPSToCurrDist is MED) and (GPSCurrVelocity is MED)
//  then (GPSPosWeight is SMALL)
// if (GPSToCurrDist is SMALL) and (GPSCurrVelocity is SLOW)
//  then (GPSPosWeight is SMALL)
memval=MAX(
MIN(GPSToCurrDist.GetMembership(MED_3),
GPSCurrVelocity.GetMembership(MED_3)),
MIN(GPSToCurrDist.GetMembership(SMALL_3),
GPSCurrVelocity.GetMembership(SLOW_3)));
               GPSPosWeight.SetMembership(SMALL_5,memval);
               GPSHeadWeight.SetMembership(SMALL_5,memval);
               // if (GPSToCurrDist is MED) and (GPSCurrVelocity is SLOW)
//  then (GPSPosWeight is ZERO)
               // if (GPSToCurrDist is LARGE) and (NumNoGood is SMALL)
//  then (GPSPosWeight is ZERO)
               // if (GPSToCurrDist is LARGE) and (NumNoGood is MED)
//  then (GPSPosWeight is ZERO)
               memval=MAX3(
       MIN(GPSToCurrDist.GetMembership(LARGE_3),
NumNoGood.GetMembership(MED_3)),
MIN(GPSToCurrDist.GetMembership(LARGE_3),
NumNoGood.GetMembership(SMALL_3)),
       MIN(GPSToCurrDist.GetMembership(MED_3),
GPSCurrVelocity.GetMembership(SLOW_3)));
               GPSPosWeight.SetMembership(ZERO_5,memval);
               GPSHeadWeight.SetMembership(ZERO_5,memval);
               // if (GPSToCurrDist is LARGE) and (NumNoGood is LARGE)
//  then (GPSPosWeight is ONE)
memval=
MIN(GPSToCurrDist.GetMembership(LARGE_3),
NumNoGood.GetMembership(LARGE_3));
               GPSPosWeight.SetMembership(ONE_5,memval);
               GPSHeadWeight.SetMembership(ONE_5,memval);

               // get de-fuzzified output weighting values
               m_GPSHeadingWeight = GPSHeadWeight.GetValue();
               m_GPSPosWeight = GPSPosWeight.GetValue();

               // use threshold to determine if the data was fused significantly
               if(m_GPSPosWeight<0.025) m_NumGoodNoWeight++;
               else m_NumGoodNoWeight=0;

               // perform median filtering on the GPS heading
//  outputs to eliminate glitches
               m_HeadingFilt[m_CurrHeading]=m_CurrGPSData.heading;
               m_CurrHeading = (m_CurrHeading+1)%HEADING_FILT_SIZE;
               double SortHeading[HEADING_FILT_SIZE];
               for(int i=0;i<HEADING_FILT_SIZE;i++) SortHeading[i] = m_HeadingFilt[i];
               qsort(SortHeading,HEADING_FILT_SIZE,sizeof(double),dcomp);
               m_GPSHeading = SortHeading[(HEADING_FILT_SIZE-1)/2];
               // weighted sum of GPS position and current position
//  using resultant GPSPosWeight
               xkm1[1]=(m_GPSPosWeight*m_GPSLongitude)+((1.0-m_GPSPosWeight)*xkm1[1]);
               xkm1[2]=(m_GPSPosWeight*m_GPSLatitude)+((1.0-m_GPSPosWeight)*xkm1[2]);
               // assumed heading and GPS sensor heading are 90 degrees
//  out of phase and inverted
               double gpshead = (90.0-m_GPSHeading);
               // get GPS heading and current heading within 180 degrees
//  and both positive
               //  so the average does not suffer from wrap-around errors
               while((xkm1[3]-gpshead)>180.0) gpshead+=360.0;
               while((gpshead-xkm1[3])>180.0) gpshead-=360.0;
               // weighted sum of GPS heading and current heading
//  using resultant GPSHeadingWeight
               xkm1[3] = m_GPSHeadingWeight*gpshead+(1-m_GPSHeadingWeight)*xkm1[3];
        }
        // else GPS data is invalid, so ignore it altogether

        // plot the filtered position in the output window
        if(m_fInitGPS && m_fxkValid) m_pMapDlg->
PlotFiltPos(CPoint(int(xk[1]),int(xk[2])));
}

// Filter100Hz is called whenever a new inertial sensor value is
//  acquired. This is independent
//  of the method of GPS/INS fusion being used. This is primarily just a Kalman filter
void CPostViewApp::Filter100Hz(void)
{
        // we just 'sampled' our sensors (at 100 Hz)
        if(m_fInitGPS)
        {
               // perform extended kalman filter interation
               // -> put measurement variables into yk
               //    (yk[1] = dtheta and yk[2] = odo_measure)
               double dtheta1 = ((m_GyroCenter-m_CurrSensorData.gyro1)/m_MvoltPerDegree);
               double odo_measure = (m_CurrSensorData.odometer_total-
old_odo_total)/m_TicksPerMeter;
               if(odo_measure<0 || odo_measure>m_MaxDistPer100) odo_measure=0;
               old_odo_total=m_CurrSensorData.odometer_total;
               if(odo_measure==0) odo_zero_count++;
               else odo_zero_count=0;
               if(odo_zero_count>20) m_GyroCenter=
m_GyroCenter*0.98+m_CurrSensorData.gyro1*0.02;
               if(ABS(m_CurrSensorData.steer-m_SteerCenter)<50)
                      dtheta1=dtheta1*ABS(m_CurrSensorData.steer-m_SteerCenter)/100.0;
               yk[1] = dtheta1;
               yk[2] = odo_measure;

               // -> generate system transfer by linearizing non-linear functions (Ak)
               Ak[1][6]=cos(DEG_TO_RAD(xkm1[3]))*meters_to_msec_long;
               Ak[2][6]=sin(DEG_TO_RAD(xkm1[3]))*meters_to_msec_lat;

               m_LastGoodGPSDLat += Ak[2][6]*odo_measure;
               m_LastGoodGPSDLong += Ak[1][6]*odo_measure;
               // -> generate measurement transfer by linearizing
//  non-linear functions (Ck)
               // -> predict error covariance
               //       (Pk1 = Ak * Pkm1 * AkT + Q)
               mat_mult(Ak,Pkm1,mtemp1,8,8,8);
               mat_mult_transpose(mtemp1,Ak,mtemp2,8,8,8);
               mat_add(mtemp2,Q,Pk1,8,8);

               // -> find Kalman gain based on predicted error covariance
               //       (Kk = Pk1 * CT * [C * Pk1 * CT + R]^-1)
               mat_mult(C,Pk1,mtemp3,2,8,8);
               mat_mult_transpose(mtemp3,C,mtemp4,2,8,2);
               mat_add(mtemp4,R,mtemp5,2,2);
               mat_inverse(mtemp5,mtemp6,2);
               mat_mult_transpose(Pk1,C,mtemp7,8,8,2);
               mat_mult(mtemp7,mtemp6,Kk,8,2,2);

               // -> estimate state based on old state, and difference
               //       between predicted observations and actual observations
               //       (xk = Ak * xkm1 + Kk * (yk - C * Ak * xkm1)
               mat_mult(C,Ak,mtemp8,2,8,8);
               mat_mult_vector(mtemp8,xkm1,mtemp9,2,8);
               vec_sub(yk,mtemp9,mtemp10,2);
               mat_mult_vector(Kk,mtemp10,mtemp11,8,2);
               mat_mult_vector(Ak,xkm1,mtemp12,8,8);
               vec_add(mtemp12,mtemp11,xk,8);

               // -> get error covariance
               //       (Pk = Pk1 - Kk * C * Pk1)
```

```
                mat_mult(Kk,C,mtemp13,8,2,8);
                mat_mult(mtemp13,Pk1,mtemp14,8,8,8);
                mat_sub(Pk1,mtemp14,Pk,8,8);

                // -> current state values are in xk
                //    (longitude = xk[1] and latitude = xk[2])
                if(m_DebugOut)
                {
                        // print a bunch of stuff to a file for debug puposes
                        char str[150];
                        int i;
                        for(i=0;i<8;i++)
                        {
                                sprintf(str,"%.5lf ",xk[i+1]);
                                if(m_DebugFile!=NULL) m_DebugFile->Write(str,strlen(str));
                        }
                        sprintf(str,"\n");
                        if(m_DebugFile!=NULL) m_DebugFile->Write(str,strlen(str));
                }

                // -> update for next time
                //       (xkm1 = xk)
                //       (Pkm1 = Pk)
                mat_copy(Pk,Pkm1,8,8);
                vec_copy(xk,xkm1,8);

                filter_count++;

                m_fxkValid = TRUE;
        }
}
```

Vita

David McNeil Mayhew was born on September 15, 1976 in Dale City, Virginia. He attended North Stafford High School and graduated in 1994. David entered Virginia Polytechnic Institute and State University as an undergraduate in the Engineering program in fall of 1994. David graduated with his Bachelor of Science in Computer Engineering in December of 1997. David then remained at Virginia Tech and completed his Masters of Science Degree in Electrical Engineering in the summer of 1999. David has taken an engineering position with Intelligent Automation Inc., a robotics and artificial intelligence research and development company located in Rockville, Maryland.