TSPI Data Reduction

Current Draft (using LaTeX)

V0.7.18 (2003-01-15 13:44:42)
Built (2003-01-15 14:23:39)
Original Word Version V0.6.4 (2000-July-03)

Peter P. Eiserloh
Code 525300D
Data Production Branch
Electronic Combat Range
Naval Air Warfare Center
China Lake, CA 93555

ABSTRACT

Flight testing of aircraft, and the accuracy of airborne sensors requires a source of truth data which can be compared against the on-board data. This Time Space and Position Information (TSPI) data needs to be post-processed and merged with the aircraft data.
This paper describes some of the issues and techniques used to perform this data merging. In particular, details of the coordinate transfer from the TSPI reference point to the aircraft is discussed, along with the format of various TSPI files.

Contents

PREFACE

This document started out as a collection of notes of the algorithms that I use for TSPI data reduction, and file formats. Over time people have seen it and have asked for copies. Now I am in the process (time permitting) of polishing it for greater distribution.

These are DRAFT copies, and I am sure that problems still exist. So, please send any comments to me, so that I can improve it.

Chapter-1 introduces the problem of a curved earth, and the requirement for doing coordinate transformations. The World Geodetic System 84 (WGS–84) is discussed.

Chapter-2 discusses various coordinate transformations.

Chapter-3 discusses how to use those coordinate transformations in a set of sequences with direct application towards TSPI data reduction.

Chapter-4 describes some subtle problems that have occurred to both myself, and others over time.

Chapter-5 discusses some of the file formats used to store TSPI information, including the Common TSPI Data Format (CTDF).

Chapter-6 discusses manipulating the data, such as smoothing, interpolation, and similar topics.

Appendices provide derivations of some of the equations, background data on coordinate transformations, the geoid height tables, and source code.

I many respects, this document will never be finished, there will always be something new to learn.

Any source code you see here is part of the data reduction tools available at http://eiserloh.chinalake.navy.mil/. This site is within a firewall and is only visible within China Lake. Alternatively, you can get this from http://www.eiserloh.org/.

I am in the process of moving the document from a proprietary word processing document format to LaTeX. LaTeX is a set of macros for the TeX typesetting software, which is both open source, and free.

ACKNOWLEDGMENTS

I thank the people who have helped me find the information collected here, and those who reviewed earlier versions of this document.

Dr. Dan Price the first person other than myself to read this document. He also supplied the original source code for earlier versions of the software that we use to analyze our flight data.

Andy Borman for pointing out certain topics that were covered in a confusing manner, so that they are now discussed much clearer. Andy also reminded me that commonly used terms in the literature are used incorrectly. I now explicitly state that I use the mathematical terminology vice common usage (spheroid vice ellipsoid).

Dr. Floyd Hall for finding the original documentation on the Bowring method, in addition to being my source for many of the algorithms, documentation, and other data.

A number of other people have helped to review this document, and I appreciate those inputs.

Thanks, Peter P. Eiserloh

Chapter 1

Introduction

1.1 The Basic Problem

We perform flight tests with our aircraft to determine system accuracy in targeting and weapons deployment. This requires that we know the position of the aircraft during the period of the test. This is provided by the ranges in the form of Time Space & Position Information (TSPI) data.

The ranges collect their TSPI data from various sensors such as NIKE radars, Alrite lasers, and now GPS/ARDS pods. Each of these sensors' data is processed into a common format, while possibly being filtered, and smoothed. It is then provided to their customer (us). In special cases we may also request the raw data from the sensor.

The TSPI sensor usually measures the aircraft's position in terms of slant range, azimuth, and elevation (RAE) from the sensors' position. These sensors will be located at different positions. When we are analyzing the flight data we do not want to handle all these different coordinate system origins, rather we only want one.

For air to air missions the ranges provide a single origin (China Lake uses L20). Air-to-Ground missions usually pick the ground target as the origin.

Note: A ground moving target test is similar to an air-to-air test. Both the shooter aircraft and the target vector use a common fixed reference point.

After the coordinate transformation to the common origin the TSPI target's (our aircraft's) position is normally represented in a Cartesian (orthogonal) coordinate system vice the RAE system. China Lake's Range Command Center (RCC) uses the axes X, Y, and Z, which can be CDU or ENU (if the flight line rotation is zero).

The EAST, NORTH, and UP (ENU) coordinate system is referenced to true north and the gravitational down vector. East simply happens to be orthogonal to North, and Down in the right handed sense. The aircraft uses a similar coordinate system NED, which is also a right handed local coordinate system, but it uses a down axis rather than an up axis, and the first two are reordered to make it right handed.

Optionally the positions may be in the CDU coordinate system composed of the axes Cross Range, Down Range, and Up. CDU is simply ENU rotated around to the flight line. This is used many times with air-to-ground missions, because this is referenced to the flight line along which the aircraft flies, and the bombs will fall.

While the earth really is a round surface, TSPI data is reported in an orthogonal coordinate system, which is tangent to the surface of the earth at the TSPI origin with the ENU axes. Because the earth is round, another origin will have a different set of axes, even if they are called ENU. This is rotated by the difference in both latitude and longitude.

The aircraft avionics internally uses a North, East, Down (NED) coordinate system, with its origin at the aircraft's current position. This origin changes over time as the aircraft changes position. While we try to explicitly state the coordinate system in use, we do prefer the NED coordinate system, so if not explicitly

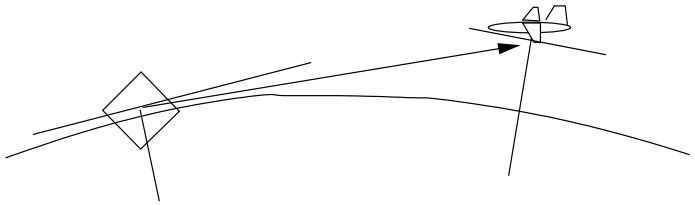

Figure 1.1: The TSPI vice the aircraft coordinate system.

stated then assume a NED system.

The aircraft will detect and track the target with a sensor of some kind. These may be a RADAR, TV, HUD, or a Forward Looking Infra-Red (FLIR)sensor. Since our job is to test the sensor's and the system performance, we will need to compare the targeting information from the sensor against a truth value. This is why we use the TSPI data as measured with by a TSPI sensor. The source of the TSPI data should be at least 4 times more accurate than the error in the aircraft sensor, and hopefully 10 times better.

The basic problem is that the aircraft coordinate system is different than the TSPI coordinate system. They differ in both the origin and the orientation of the axes. In figure 1.1, the TSPI reference point is on the surface of the earth (at some altitude above mean sea level) with an orthogonal coordinate system in a NED system at it's location. At the current position of the aircraft the NED system is rotated with respect to the TSPI reference point. This rotation is in both latitude and longitude. The third axis only needs to be considered if there is a flight line rotation (see section 2.2, on page 7).

We need to change coordinate systems from the TSPI reference to a reference point centered on the aircraft. The basic process involves the following steps:

1. Use the TSPI reference point's geodetic coordinates (latitude, longitude, & altitude.) along with the target vector to get the geodetic coordinates of the target aircraft. If necessary the target vector may have to be transformed from (Range, Azimuth, Elevation) to (North, East, Down).

2. Given the TSPI origin and a vector from that origin, we calculate the geodetic coordinates of the aircraft.

3. Using the geodetic coordinates of the target aircraft we rotate the target vector into the NED coordinate system of the target aircraft.

4. If required, a further rotation to the aircraft's Forward, Right, Down (FRD) coordinate system can be performed if the orientation of the aircraft is known. This orientation has to be extracted from on-board data (from the Inertial Navigation Set (INS)), since TSPI does not measure it.

5. Once the TSPI data is in the same coordinate system as the aircraft, then we can compare the on-board data against the "truth" data from TSPI.

Some flights will have two (or more) aircraft, one is the shooter, and the other(s) is (are) the target. We need to generate the vector from the shooter to the target. The two aircraft scenario will use the same tools we develop for the one aircraft scenario. In chapter 2, we develop the tools we will need, these are the coordinate transformations.

Note: China Lake is located in the western hemisphere which is in the negative longitudes. We tend to just drop the sign, but we need to reintroduce it or our software will place us in the eastern hemisphere near Nanjing China ($\phi = +35^\circ$ 40', $\lambda = +117^\circ$ 40', h = 2260 ft) which is in the eastern hemisphere.

1.2 The World Geodetic System - 84 (WGS–84)

In 1987 the Department of Defense adopted the World Geodetic System 84 datum. A datum defines a coordinate system, including its origin, axes, and spheroid. It superseded the previous datum WGS–72,

constant	*value*	*units*	*comment*
a	6,378,137	meters	semi-major axis, the radius at the equator
	20,925,646.3255	feet	
b	6,356,752.3142	meters	semi–minor axis, the radius at the poles
	20,855,486.5952	feet	
e^2	0.00669437999013		square of eccentricity
e	0.0818191908426		eccentricity
f	1/298.257223563		flattening

Table 1.1: The WGS–84 Earth Model

WGS–66, and the original WGS–60. WGS60 was the first datum to define a spheroid with its origin at the earth's center; previous datum were tailored such that the spheroid's surface was a better match to the geoid (see Appendix C) over a particular area. The tailored spheroid used in North America was NADS–27. Since these datum were tailored for specific locations they worked very poorly in other locations. The WGS series of datum are designed to be used anywhere in the world.

It is important to ensure that everyone uses the proper datum, or else we could end up placing bombs 600 feet short of our intended target. This happened over Libya in the late 1980s where the intelligence sources used one datum, but the aircraft used a different one.

If the earth were flat comparing TSPI data to aircraft data would be very easy, but the earth is round. Even if it were a sphere it would not be very difficult, but the earth is shaped like a squashed ball. The rotation of the earth produces inertial forces (centrifugal) so the equator bulges out, and the poles are sucked in toward the center. This produces an oblate spheroid.

A spheroid is a surface of revolution defined by rotating an ellipse about one of it's axes (Note this is not the same as an ellipsoid, which has three separate axes). There are two kinds of spheroids, oblate, and prolate. An oblate spheroid is wider than it is tall (the rotated ellipse's semi–major axis is the equator). A prolate spheroid is taller than it is wide (the rotated ellipse's semi–minor axis is the equator).

The World Geodetic System 84 (WGS–84) defines the earth model as an oblate spheroid. Common usage in the literature calls this the WGS–84 ellipsoid, but in standard mathematical terminology it is a spheroid. The parameters of the WGS–84 spheroid are specified in table 1.1.

The geodetic coordinates (latitude ϕ, longitude λ, and geodetic height h) are based upon gravitational potentials, and are not referenced to the center of the earth. This means that the down vector does not in general intersect the center of the earth (see Figure 1.4).

The latitude is not the angle from the center of the earth, but rather the angle of the normal line from the point on the surface to the intersection with the plane of the equator. The longitude is the angle along the equator (about the earth's axis of rotation) from the prime meridian (through Greenwich England). The geodetic height is usually referenced to mean sea level (i.e., orthometric height). Many times it will be referenced to the spheroid (the spheroidal height).

The earth centered, earth fixed coordinate (ECEF) system is defined by the three axes EFG (see figure 1.2). The EFG geocentric coordinate system is the master coordinate system that we use for most coordinate transformations. This is a right-handed orthogonal coordinate system whose origin is the center of the earth, and whose axes are fixed to points on the earth. This is not an inertial coordinate system, since the earth rotates about its axis.

The G–axis points from the center of the earth to the north pole (along the axis of rotation). The E-axis points from the center of the earth to the equator at the prime meridian (the zero longitude line which goes through Greenwich England). The F–axis is derived from the E, and G axes to be orthogonal, it ends up pointing into the Indian Ocean at the equator.

If we cut a slice through the spheroid along a meridian line we form an ellipse (see Figure 1.3). At any point on this ellipse is a tangent line (horizon), and a perpendicular (down vector). The down vector (see Figure 1.4) does not intersect the center of the ellipse, but it does intersect the equatorial plane at an angle ϕ

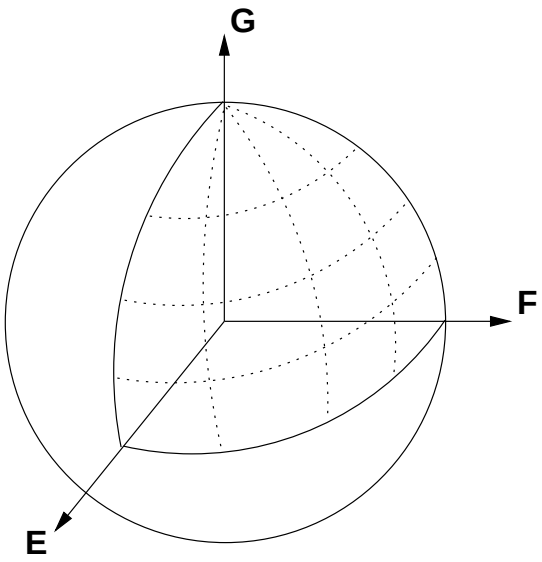

Figure 1.2: Geocentric System (EFG)

(the latitude).

Okay, we simplified things somewhat. Actually, the earth is bumpy, caused by geologic structures within the surface of the earth. The geoid is the gravitational equi-potential surface corresponding to the mean sea level (MSL). There are differences in the composition of the crust of the earth. This bumpiness causes mean sea level to deviate from a spheroid. The WGS–84 spheroid was calculated to best approximate the geoid.

Some locations have heavy deposits and the water tends to collect over these, raising the mean sea level above the spheroid over these areas. In other locations the deposits within the crust is not so heavy, so the mean sea level is lower. These areas of the globe usually contain land masses such as continents. When averaged over the entire earth, the geoidal separation from the WGS–84 spheroid is zero. At any particular point, the geoid will in general be either a positive or a negative offset from the spheroid. The WGS–84 earth model attempts to give the best approximation of its spheroid's surface to the geoid. The difference between the geoid and the WGS–84 spheroid (the geodetic separation) is empirically described by tables. This geodetic separation of mean sea level from the spheroid has to be empirically measured. This has been measured by the defense mapping agency, and is available in appendix C (The Geoid).

If you are given a height referenced to mean sea level (MSL), then you need to determine the geoid separation at that position, and convert the height to a spheroidal height before you do any coordinate transformations into the EFG coordinate system. In practice, the geoidal separation is small enough compared to the size of the earth that one can ignore it.

Here at China Lake the geoid separation is about -94 feet (see page 52). Sea Level is 94 feet below the spheroid at this latitude/longitude.

1.2.1 Heights

- Geoidal Height
- Orthometric Height
- Geocentric Height
- Spheroidal Height
- Topocentric Height

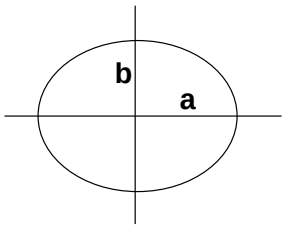

Figure 1.3: Ellipse

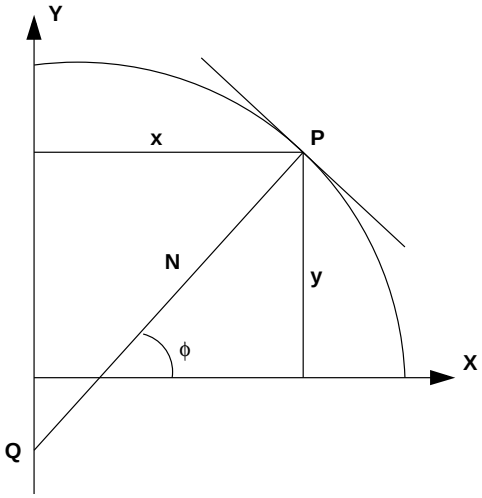

Figure 1.4: The Down Line PQ

Chapter 2

Coordinate Transforms

2.1 ENU –VICE– NED

The East/North/Up (ENU) coordinate system is commonly used by the range department, and the TSPI data is usually reported in this coordinate system. The aircraft avionics package uses the North/East/Down (NED) system. Both of these are right handed coordinate systems. There is a very simple transformation between them. The same matrix is used for going both directions (it is it's own inverse).

$$\begin{bmatrix} Y_N \\ Y_E \\ Y_D \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} X_E \\ X_N \\ X_D \end{bmatrix} \tag{2.1}$$

$$\begin{bmatrix} X_E \\ X_N \\ X_D \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} Y_N \\ Y_E \\ Y_D \end{bmatrix} \tag{2.2}$$

2.2 CDU2NED

When an aircraft makes a bombing run, it is on a particular flight line (usually not true north). The flight line is described as an angle from true north towards the flight line. The resulting coordinates will be rotated from true north by this rotation angle, and it defines a new coordinate system. This coordinate system is in CROSS–RANGE, DOWN–RANGE, and UP (CDU). A simple two dimensional rotation will transform from CDU to ENU coordinates, and then from ENU to NED. The inverse can also be performed, (NED to CDU).

In the example below the tip of the bomb symbol is +100 feet off range, and +200 feet down range. The flight line angle is about 40 degrees counter–clockwise (or negative 40 degrees).

We rotate the vector (100, 200, 0) by a rotation matrix of an angle equal to the negative of the flight line.

$$\begin{bmatrix} X_E \\ X_N \\ X_D \end{bmatrix} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y_C \\ Y_D \\ Y_U \end{bmatrix} \tag{2.3}$$

$$= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y_C \\ Y_D \\ Y_U \end{bmatrix} \tag{2.4}$$

We can modify this matrix to create a rotation matrix which includes the ENU to NED transformation. This looks a little different from a normal rotation matrix because it is also switching the order of the axis (see above).

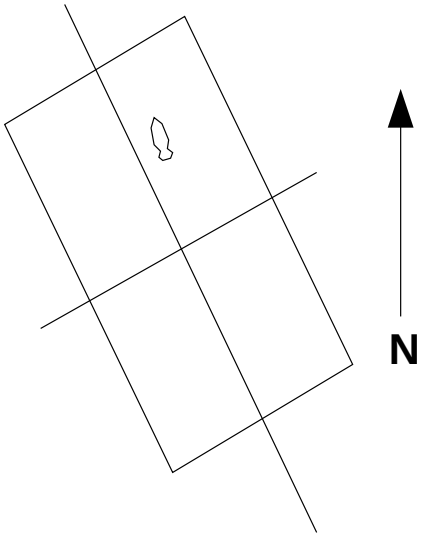

Figure 2.1: FLIGHT LINE ROTATED FROM NORTH

$$\begin{bmatrix} X_N \\ X_E \\ X_D \end{bmatrix} = \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 \\ \cos(\theta) & \sin(\theta) & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} Y_C \\ Y_D \\ Y_U \end{bmatrix} \tag{2.5}$$

Which when applied to our example (100, 200, 0) at $\theta = -40$ degrees, we get:

$$\begin{bmatrix} X_N \\ X_E \\ X_D \end{bmatrix} = \begin{bmatrix} -\sin(-40) & \cos(-40) & 0 \\ \cos(-40) & \sin(-40) & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 100 \\ 200 \\ 0 \end{bmatrix} = \begin{bmatrix} 217 \\ -52 \\ 0 \end{bmatrix}$$

2.3 RAE2NED

When a sensor such as the Alrite Laser measures the position of it's target, it measures the slant range (R), azimuth angle(δ), and elevation angle (ϵ) to the target. This coordinate system is called *RAE*. Here we define δ as the azimuth angle from the north axis towards the east axis, and ϵ as the elevation angle from the horizontal up towards the line of sight (i.e., down is negative).

We want the target vector in the right-handed orthogonal NED coordinate system. Depending upon the source of TSPI this may already be done, if not then we will have to do it.

$$X_N = R\cos(\epsilon)\cos(\delta) \tag{2.6}$$
$$X_E = R\cos(\epsilon)\sin(\delta) \tag{2.7}$$
$$X_D = -R\sin(\epsilon) \tag{2.8}$$

We also define the reverse transform from NED to RAE.

$$R = \sqrt{X_N^2 + X_E^2 + X_D^2} \tag{2.9}$$
$$\delta = \arctan(X_E, X_N) \tag{2.10}$$
$$\epsilon = \arcsin(-X_D/R) \tag{2.11}$$

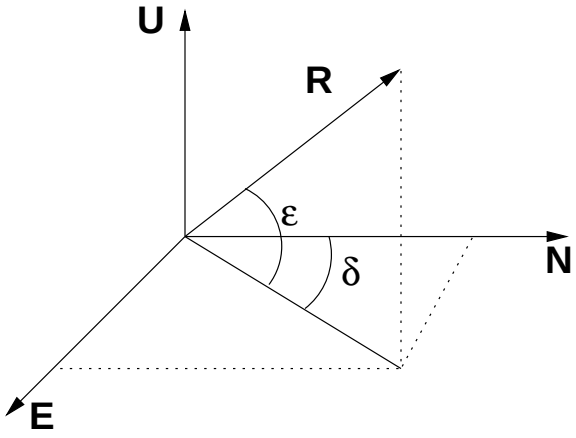

Figure 2.2: An RAE vector (Polar coordinates)

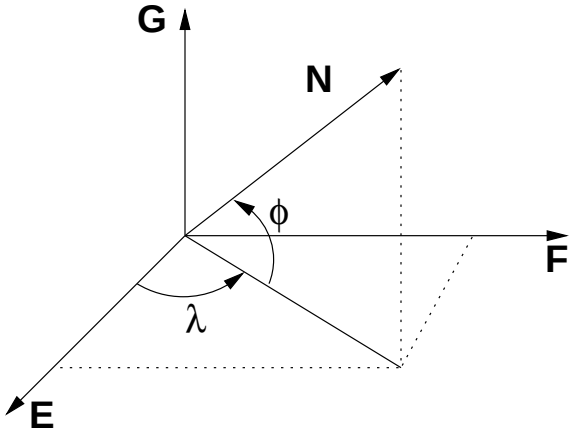

Figure 2.3: A Simplified Polar Coordinate System

Here, I use the calculated value of R while calculating the elevation angle. I could have calculated the ground distance, $\sqrt{X_N^2 + X_E^2}$ and used arctan2() instead of arcsin(), but the above method saves a couple steps.

2.4 LLH2EFG

The Geodetic to Geocentric coordinate transform is called *llh2efg*. A location specified in geodetic coordinates (latitude ϕ, longitude λ, and height h) can be converted to geocentric coordinates (EFG).

The latitude ϕ, longitude λ, and height h, of the TSPI reference point is converted to the EFG geocentric coordinate system as follows (see appendix A.1 on page 37).

$$X_E = (N + h) \cos\phi \cos\lambda \quad (2.12)$$

$$X_F = (N + h) \cos\phi \sin\lambda \quad (2.13)$$

$$X_G = \left[N\left(1 - e^2\right) + h\right] \sin\phi \quad (2.14)$$

Where:

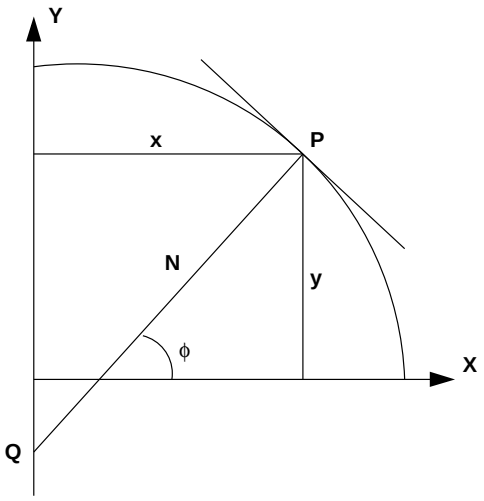

Figure 2.4: PLANE TANGENT TO THE EARTH SURFACE

ϕ is the latitude,

λ is the longitude,

N is the radius of curvature in the prime vertical (see appendix A.2.1 on page 38, for the derivation)

$$\boxed{N = \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}}} \tag{2.15}$$

2.5 EFG2LLH

The transformation *efg2llh* which goes from geocentric (EFG) to geodetic (LLH) coordinates is probably the most complex of the coordinate system transformations. In particular, determining the latitude is the most difficult one of all. The longitude is the easiest.

In the past we used a few ad-hoc methods of calculating the latitude, usually iterative approaches. The modern method is called the "Bowring Method", which was first published in 1976 [BOWRING-1976]. Bowring's method is independent of the particular earth model. We of course will be using WGS–84, although the algorithm could use any other such as NADS–27.

In the following discussion we will first determine the longitude, then latitude, and finally the height above the spheroid. Since the orthometric height is referenced to MSL, we will need to apply the geoidal separation at that latitude/longitude (see Appendix C).

2.5.1 LONGITUDE

The simplest of the geodetic coordinates to calculate is the longitude, since it is not affected by the eccentricity of the earth.

The longitude shows how far East/West we are from the prime meridian. The tangent of the longitude is directly formed using the geocentric components.

$$\tan(\lambda) = X_F / X_E$$

$$\boxed{\lambda = \arctan 2\,(X_F, X_E)} \tag{2.16}$$

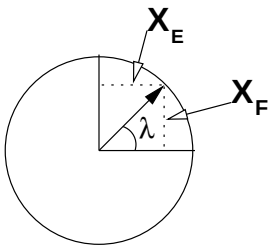

Figure 2.5: Longitude formed by E and F components

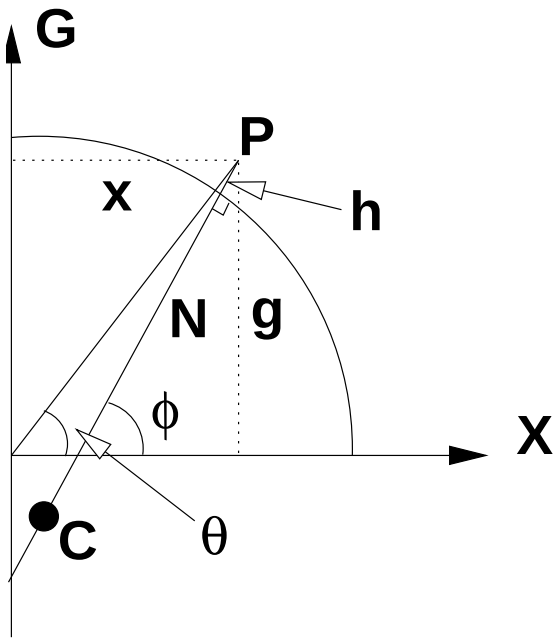

Figure 2.6: Bowring's Center of Curvature "C"

Where Arctan2(y, x) will generate the correct quadrant. If the function arctan2() is not available but arctan() is then you have to determine the correct quadrant yourself.

2.5.2 LATITUDE

The Bowring method can be considered to be a direct solution, but is in reality an iterative one. It is simply that the first iteration is so good, that a second one is superfluous. The second iteration causes such a small correction that we can usually ignore it.

According to Bowring, the worst error would be $0.0018''$ of latitude (or about 2 inches). This largest error occurs at a spheroidal height twice the distance of the semi-major axis (H=2a), or about 8 thousand miles of altitude. At earth-bound altitudes (-5 to +10 K meters) this error is much smaller.

Bowring's method uses the North-South radius of curvature R instead of the usual East-West radius of curvature N. This uses a point not on the polar axis (whereas the older iterative method's radius of curvature terminates on the polar axis).

First, we introduce a new variable X to indicate the distance from the axis of rotation to the point in question. This effectively bisects the spheroid forming an ellipse along the meridian (longitudinal) line through the point.

$$X = \sqrt{E^2 + F^2} \tag{2.17}$$

We provide an initial estimate α. This corresponds to rescaling the ellipse back into a circle, then using the tangent function.

$$\tan(\alpha) = \frac{G}{X}\left(\frac{a}{b}\right)$$

$$\alpha = \arctan\left[\frac{G}{X}\left(\frac{a}{b}\right)\right] \tag{2.18}$$

The normal line with the length of the North-South radius of curvature R terminates at the point "C": (see figure 2.6).

The angle from this point to the point (X, G) has the tangent:

$$\tan(\phi) = \left[\frac{G + b\left(\frac{e^2}{1-e^2}\right)\sin^3\alpha}{X - ae^2\cos^3\alpha}\right] \tag{2.19}$$

Giving a latitude of:

$$\boxed{\phi = \arctan\left[\frac{G + b\left(\frac{e^2}{1-e^2}\right)\sin^3\alpha}{X - ae^2\cos^3\alpha}\right]} \tag{2.20}$$

If you wanted to iterate this you would feed the first iterations ϕ back into the α.

2.5.3 HEIGHT

We will attack the height problem via two routes. Both start from the original transformation (see equations 2.12, 2.13, and 2.14), and the distance from the axis of rotation X (eq. 2.17). We then merge the two approaches into one equation.

For the first route, we remove eq's 2.12, and 2.13 dependency on the longitude by using the distance from the axis of rotation to the point (collapsing the E, and F components into X). Then simply solve for the height h.

$$\begin{aligned}
X &= \sqrt{X_E^2 + X_F^2} && (2.21)\\
X &= (N+h)\cos\phi && (2.22)\\
X &= N\cos\phi + h\cos\phi && (2.23)\\
h\cos\phi &= X - N\cos\phi && (2.24)\\
h &= \tfrac{X}{\cos\phi} - N \quad ; \text{where } \phi \neq 90 \text{ (fails at the poles)} && (2.25)
\end{aligned}$$

The second route uses the G component (eq. 2.14).

$$\begin{aligned}
G &= N\left(1-e^2\right)\sin\phi + h\sin\phi && (2.26)\\
h\sin\phi &= G - N\left(1-e^2\right)\sin\phi && (2.27)\\
h &= \left(\tfrac{G}{\sin\phi}\right) - N\left(1-e^2\right) \quad ; \text{where } \phi \neq 0 \text{ (fails on the equator)} && (2.28)
\end{aligned}$$

These two approaches fail at different places. We can merge them into one equation which works for all latitudes. Starting with eq. (2.25) we multiply by $\cos^2\phi$. We similarly multiply eq. (2.28) by $\sin^2\phi$.

$$h\cos^2\phi = X\cos\phi - N\cos^2\phi \tag{2.29}$$
$$h\sin^2\phi = G\sin\phi - N\left(1-e^2\right)\sin^2\phi \tag{2.30}$$

Now we add the two equations and simplify.

$$\begin{aligned} h\cos^2\phi + h\sin^2\phi &= X\cos\phi + G\sin\phi \\ &\quad - N\cos^2\phi - N\left(1-e^2\right)\sin^2\phi \\ h\left(cos^2\phi + \sin^2\phi\right) &= X\cos\phi + G\sin\phi \\ &\quad - N\left[\cos^2\phi + \sin^2\phi - e^2\sin^2\phi\right] \\ h &= X\cos\phi + G\sin\phi - N\left(1-e^2\sin^2\phi\right) \end{aligned}$$

And, using eq. (2.15) we can remove N by reintroducing the semi-major axis of the earth.

$$\boxed{h = X\cos\phi + G\sin\phi - a\sqrt{1-e^2 sin^2\phi}} \tag{2.31}$$

This equation will work everywhere on the earth, without limitations.

2.6 NED2EFG Rotation

A vector can be rotated from a local NED coordinate system at location (ϕ, λ, h) to the geocentric coordinate system by a rotation matrix.

$$X_{EFG} = A\left(\phi,\lambda\right)\cdot X_{NED} \tag{2.32}$$

Where:

$$\boxed{A\left(\phi,\lambda\right) = \begin{bmatrix} -\sin\phi\cos\lambda & -\sin\lambda & -\cos\phi\cos\lambda \\ -\sin\phi\sin\lambda & \cos\lambda & -\cos\phi\sin\lambda \\ \cos\phi & 0 & -\sin\phi \end{bmatrix}} \tag{2.33}$$

2.6.1 Deriving the NED TO EFG Rotation Matrix

The rotation matrix (eq. 2.33) is a composite rotation matrix formed from three primitive rotation matrices. First we rotate the coordinate system about it's east axis to align its north-axis to the G-axis (by the negative of the latitude $-\phi$). Then we rotate about the north-axis to align the east-axis to the F-axis (by the longitude λ). Finally we rotate about the east-axis by -90 degrees ($-\pi/2$) to match all the axes. All these rotations can be combined into one composite rotation matrix. We follow the derivation with test examples.

It is very important to apply these rotation matrices in the correct order. Each time we apply one rotation, the coordinate axes are redefined. Once all the rotations are defined then we can combine them into one composite rotation matrix. We only need to use two of the three primitive rotation matrices R_x, and R_y. When we start the X, Y, and Z axes are aligned to the North, East, and Down axes respectively.

We will use X, Y, and Z coordinates in the process of these transformations. Initially the NED axes are aligned to the XYZ. The final transform will align them to the EFG axes.

Of particular concern are the signs of the rotation angles. Remember that this is a right handed coordinate system where positive angles adjust vectors from North $\Rightarrow$ East, from East $\Rightarrow$ Down, and from Down $\Rightarrow$ North.

Because we are not using translations within this composite matrix, we do not need homogeneous coordinates (we can use a 3 x 3 matrix instead of a 4 x 4 matrix).

1. The first rotation is about the east–axis (y–axis), and rotates the north axis to align with the G–axis (it also aligns the down–axis to the equatorial plane). This is effectively moving the vector from the north towards the down–axis. This first rotation is a negative rotation with magnitude of the latitude (negative of the latitude).

$$R_y(-\phi) = \begin{bmatrix} \cos(-\phi) & 0 & \sin(-\phi) \\ 0 & 1 & 0 \\ -\sin(-\phi) & 0 & \cos(-\phi) \end{bmatrix} \tag{2.34}$$

$$= \begin{bmatrix} \cos(\phi) & 0 & -\sin(\phi) \\ 0 & 1 & 0 \\ \sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \tag{2.35}$$

$$\tag{2.36}$$

2. The second rotation is about the north–axis (x–axis), and moves the east axis to align with the F–axis. This is effectively moving the vector from the east–axis towards the down–axis (this is a positive rotation by an amount of the longitude).

$$R_x(\lambda) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\lambda & -\sin\lambda \\ 0 & \sin\lambda & \cos\lambda \end{bmatrix} \tag{2.37}$$

3. The third and final rotation aligns all three axes N, E, and D to the geocentric axes E, F, and G respectively. This rotation is about the current east axis (the y–axis). This is equivalent to moving a vector from the north–axis towards the down axis. This is a negative 90 degree rotation (i.e. - $\pi/2$).

$$R_y(-90) = \begin{bmatrix} \cos(-90) & 0 & \sin(-90) \\ 0 & 1 & 0 \\ -\sin(-90) & 0 & \cos(-90) \end{bmatrix} \tag{2.38}$$

$$= \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{2.39}$$

We ensure that each rotation matrix is applied to the vector in the order specified above. Each rotation matrix is prepended to the previous matrices.

$$X_{EFG} = A(\phi, \lambda) \cdot X_{NED} \tag{2.40}$$

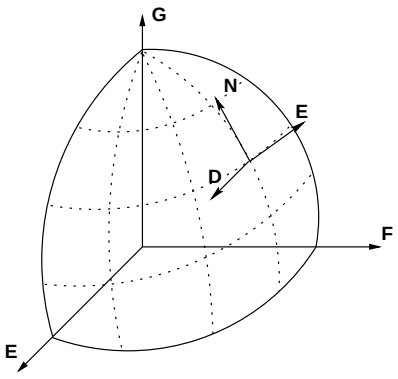

Figure 2.7: Original NED Coordinate System

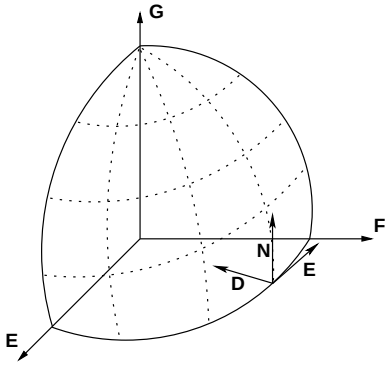

Figure 2.8: Rotated about the East Axis by its Latitude

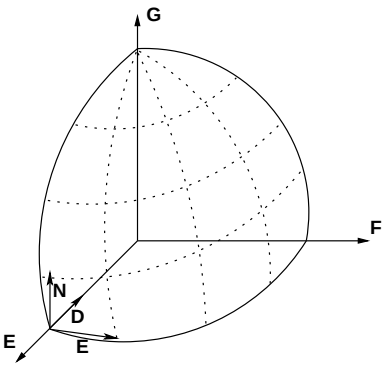

Figure 2.9: Rotated about the North by its Longitude

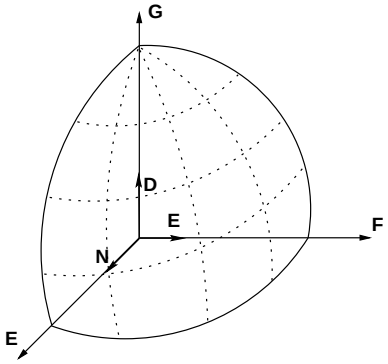

Figure 2.10: Final Rotation to the EFG Coordinate System

where:

$$A(\phi,\lambda) = R_y(-90)\cdot R_x(\lambda)\cdot R_y(-\phi) \tag{2.41}$$

Which are expanded and matrix multiplied to give:

$$A(\phi,\lambda) = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\lambda & -\sin\lambda \\ 0 & \sin\lambda & \cos\lambda \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix} \tag{2.42}$$

$$A(\phi,\lambda) = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ -\sin\phi\sin\lambda & \cos\lambda & -\cos\phi\sin\lambda \\ \sin\phi\cos\lambda & \sin\lambda & \cos\phi\cos\lambda \end{bmatrix} \tag{2.43}$$

And, finally:

$$A(\phi,\lambda) = \begin{bmatrix} -\sin\phi\cos\lambda & -\sin\lambda & -\cos\phi\cos\lambda \\ -\sin\phi\sin\lambda & \cos\lambda & -\cos\phi\sin\lambda \\ \cos\phi & 0 & -\sin\phi \end{bmatrix} \tag{2.44}$$

2.6.2 Testing the NED TO EFG Rotation Matrix

We generate a few simple tests.

ϕ=0, λ=0

A local NED vector [1, 2, 3] at (ϕ = 0, λ= 0) should map to EFG [-3, 2, 1].

$$X_{EFG} = A(\phi=0,\lambda=0) = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -3 \\ 2 \\ 1 \end{bmatrix} \tag{2.45}$$

ϕ=0, λ=90

A local NED vector [1, 2, 3] at (ϕ = 0, λ= 90) should map to EFG [-2, -3, 1].

$$X_{EFG} = A(\phi=0,\lambda=90) = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -2 \\ -3 \\ 1 \end{bmatrix} \tag{2.46}$$

ϕ=90, λ=0

A local NED vector [1, 2, 3] at (ϕ = 90, λ= 0) should map to EFG [-1, 2, -3].

$$X_{EFG} = A\left(\phi = 90, \lambda = 0\right) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ -3 \end{bmatrix} \tag{2.47}$$

2.6.3 The RIPS Rotation Matrix

The rotation matrix used by the RIPS software is different than the above matrix because RIPS uses vectors in ENU (east, north, up) rather than NED. This rotation matrix is documented in (JAMES-1990). We show that this corresponds to the rotation matrix we derived earlier, by multiplying eq. (2.33) by the NED to ENU matrix eq. (2.1).

$$A_{RIPS}\left(\phi, \lambda\right) = \begin{bmatrix} -\sin\lambda & -\sin\phi\cos\lambda & \cos\phi\cos\lambda \\ \cos\lambda & -\sin\phi\sin\lambda & \cos\phi\sin\lambda \\ 0 & \cos\phi & \sin\phi \end{bmatrix} \tag{2.48}$$

2.7 EFG2NED Rotation

When we rotate a vector from the EFG coordinate system to a local NED system we use the geodetic coordinates of the destination. We can use the inverse of the rotation matrix of before.

$$X_{NED} = B\left(\phi, \lambda\right) \cdot X_{EFG} \tag{2.49}$$

$$B\left(\phi, \lambda\right) = A\left(\phi, \lambda\right)' = A\left(\phi, \lambda\right)^T \tag{2.50}$$

In general an inverse matrix is not the same as it's transpose, but the primitive rotation matrices are anti-symmetric. The inverse of an anti-symmetric matrix is its transpose. If two anti-symmetric matrices are multiplied then their inverse will also be it's transpose.

Note: We can't just use the negative angle in these matrices, because negative angles mean the other hemisphere. This becomes especially important when you include the geoid separation in calculating the height.

$$B\left(\phi, \lambda\right) \neq A\left(-\phi, -\lambda\right)' \tag{2.51}$$

Chapter 3

Using the Coordinate Transforms

3.1 Find the Vector Between 2 Geodetic Points

To find the vector between two points we first transform both sets of coordinates into a common Cartesian coordinate system (i.e., EFG), then we subtract the first point from the second. This is the vector from the first point to the second point in the EFG coordinate system. Then the vector is rotated to the first point's local NED coordinate system.

$$\begin{aligned} X_E &= (N_1 + h_1)\cos\phi_1\cos\lambda_1 && (3.1)\\ X_F &= (N_1 + h_1)\cos\phi_1\sin\lambda_1 && (3.2)\\ X_G &= \left[N_1\left(1 - e^2\right) + h_1\right]\sin\phi_1 && (3.3) \end{aligned}$$

$$\begin{aligned} Y_E &= (N_2 + h_2)\cos\phi_2\cos\lambda_2 && (3.4)\\ Y_F &= (N_2 + h_2)\cos\phi_2\sin\lambda_2 && (3.5)\\ Y_G &= \left[N_2\left(1 - e^2\right) + h_2\right]\sin\phi_2 && (3.6) \end{aligned}$$

$$V_{EFG} = Y_{EFG} - X_{EFG} \tag{3.7}$$

$$V_{NED} = B\left(\phi_1, \lambda_1\right) \cdot V_{EFG} \tag{3.8}$$

Alternatively we could have rotated the vector into the second point local NED system.

3.2 Rotating a Vector from One Point to Another

When we have the latitude, and longitudes of both points we can rotate any vector in a local NED to the other points NED system.

We could generate a big rotation matrix that does it all, or we can do it the easy way. We use the tools already built. We use the rotation matrix eq. (2.33) to rotate the vector to EFG, then eq. (2.49) to rotate it back into NED, but this time at the second point.

3.3 Vector from the Shooter Aircraft

We are given a vector from the TSPI reference point in its local NED coordinate system to each of the TSPI targets. We want vectors from the shooter aircraft to it's targets.

3.3.1 Vector from Aircraft to TSPI Origin

We need the vector in the aircraft's NED from the aircraft to the TSPI origin, but we are only given the TSPI origin's geodetic coordinates, and the vector from the origin to the aircraft in the origin's NED system.

1. Calculate the aircraft's geodetic coordinates using one of the techniques outlined above.
2. Rotate the target vector from the origins NED into the aircraft's NED.
3. Negate the vector to get the vector from aircraft to the origin.

3.3.2 Vector from Aircraft to Target Other Than TSPI Origin

We are given the vector from the TSPI origin to the shooter aircraft, and we also have the coordinates of the target (which is not the same as the TSPI reference point). We need the vector from the aircraft to the target, in the aircraft's NED coordinate system.

1. Calculate the target's geocentric (EFG) coordinates from it's geodetic coordinates using one of the techniques outlined above.
2. Calculate the aircraft's geocentric (EFG) coordinates using one of the techniques outlined above.
3. Subtract the A/C EFG vector from the target's EFG vector to give the A/C to target EFG vector.
4. Rotate the target EFG vector into the aircraft's NED.

3.3.3 Vector from Shooter to Target Aircraft

We are given vectors to two aircraft from the TSPI origin, but we want the vector from the first (shooter) aircraft to the second (target) aircraft. We use the tools already developed to get the geodetic coordinates of the shooter. We get the EFG vectors from the origin to the two aircraft. We form the EFG vector from the shooter to the target by subtracting the shooter's EFG vector from the target's vector. Then we can rotate this vector from EFG to the local NED coordinates of the shooter.

1. Calculate the shooter's geodetic coordinates using one of the techniques outlined above.
2. Rotate both aircraft vectors into EFG.
3. Subtract the shooter's EFG vector from the target's EFG vector to get the EFG vector from the shooter to the target.
4. Rotate this vector from EFG the shooter's NED.

3.4 Uses of TSPI Data

3.4.1 Comparisons With Truth Data

Now that the vector to the aircraft is in the aircraft's NED system we can form a vector to the origin. In A/G scenarios the TSPI origin is usually the target of the aircraft. In A/A (and ground moving A/G) scenarios there will be two TSPI target vectors. From this we can compare the aircraft sensor data to truth data. The deltas are almost always sensor minus TSPI.

LIST OF DELTAS

1. Aircraft Position (latitude, longitude, altitude)
2. Aircraft Velocities
3. Target Slant Range
4. Target Velocities
5. Target Accelerations
6. Target Position [Cross range, Down range, Altitude (CDU)]
7. Angles to Target (LOS, north, east, down)
8. Roll Compensated Angles to Target (Elevation, Azimuth)

The first set of deltas requires simple subtraction. The interesting delta is the position in CDU.

3.4.2 BOMBING RESULTS ANALYSIS

When collecting a database of bombing results, we want to know the release conditions of that weapon. This includes items such as:

1. time
2. altitude,
3. airspeed,
4. ground track,
5. flight path angle,
6. cross range, and
7. down range.

Chapter 4

Issues and Gotchas

4.1 RCCS/2 FILTERED TIME

China Lake's new range TSPI data system (RCCS/2) uses CTDF, but it has some unique "features" that need to be addressed. When a track's parameters are filtered, there is a corresponding time tag. This time tag is the time of the center of the filter.

Although each data record has a time associated with it, the filtered data is associated with its own time tag. Each track within the data record will have both a device time, and a time tag in addition to the event time. This is an artifact of how RCCS/2 performs the filtering and how it generates the resulting output file. It looks like RCCS/2 is a real time system, even though TSPI reports are generated during post processing.

As RCCS/2 develops, these artifacts may disappear.

4.2 Echo Range's Height

China Lake's Echo Range can provide two heights. The geodetic (or topocentric) height is the height normal to the spheroid (i.e., down is determined by the gravitational down vector), the same height that we have been using throughout this paper.

Whereas the geocentric height (h' in the figure below) is defined on the line from the target to the center of the earth. This geocentric height is not the same as the gravitational down vector.

The flight test engineer should ensure that the data products provided by Echo range use topocentric coordinates.

4.3 TSPI Sensor Accuracy

The various sensors used to measure the position of the aircraft (or other TSPI target) have different performance characteristics. A sensor's accuracy is really dependent on the slant range and is measured in angle, but at nominal distances we can give the accuracies in distances. RCC advertises accuracies for its sensors (see table 4.1).

In addition, the sensors must be calibrated correctly to achieve its specified accuracy. For instance, the Baker Range NIKE Radars may not be calibrated correctly, in which case we can see two NIKEs tracking the same aircraft disagree by as much as 600 feet.

In many cases, we mount two ARDS pods on the same aircraft. We see that the ARDS pods disagree by as much as 5 feet in the normal case. On occasion, during aircraft maneuvers which block one of the pod's line of sight to the satellites we can see a jump in indicated position by as much as 4000 feet.

The NAV solution simply takes the real-time calculated position. In addition to the calculated position the ARDS pod sends a large data stream to both the ground station and the internal recorder which contains

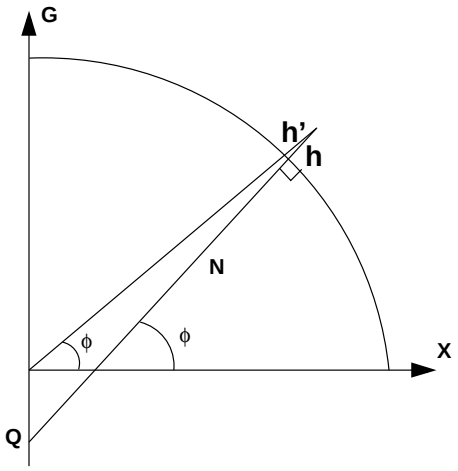

Figure 4.1: Geocentric vice Geodetic Heights

Sensor	*Accuracy*
NIKE	± 50 feet
ALRITE	± 10 feet
ARDS-GPS (NAV)	± 5 feet
ARDS-GPS (Method 3)	± 2 feet

Table 4.1: Some TSPI Sensor Accuracies

among other things the level of quality (LOQ) of its data. The Method–3 solution uses post processing to calculate the position using filtering, and taking into account the LOQ, in addition it can merge the positions of both ARDS pods. The Method–3 solution allows a particular pod to have a position jump detected and handled by using either the other pods data, or filtered position used.

In addition, on some aircraft which have a GPS sensor such as a MAGR integrated with the aircraft's Inertial Navigation Set (INS) , we can use the aircraft's own idea of its position from the INS. This may seem like a circular argument, but not really. We want to compare the TSPI vice aircraft sensor's idea of the vector from the aircraft's position to the target.

For example, if the sensor is the airborne radar in the aircraft and the target is a surveyed point on the ground, then the radar will measure a vector to the target. We can then calculate a vector using the GPS aided INS position and the coordinates of the target. This is obviously not as conceptually simple as using an external sensor to measure the aircraft's position, but in a pinch this may work.

Note: The GPS aided INS accuracy is not going to be as good as the Method–3 solution. I have heard people say the position can be as bad as 50 feet, but is usually better.

Note: The AV–8B aircraft has wings that have pronounced wing flexure and vibration. This makes it very difficult for the GPS receiver in the ARDS pod to achieve lock–on. In addition, the AV–8B does not have wingtip stations, so a pod would have to be mounted on one of the pylons (under the wing). The wing will mask the reception GPS satellite signals.

Chapter 5

FILE FORMATS

5.1 INTRODUCTION

This chapter will discuss some of the various file formats used to contain TSPI data. First we discuss a simple the format that was used in the past by us internally, then we discuss the format of the data we receive as raw data from various sources. Lastly we speculate on a format, that could be used in the future, but is today simply a pipe dream.

The RCC Integration and Processing System (RIPS) data format is an obsolete format, that has been superseded by the Common Test Data Format (CTDF). If the user needs to use this file format, please request the document "Range Control Center Integration and Processing System (RIPS) Data Product Definition" from the Range Control Complex. Alternatively, you can ask us for some general information. The RIPS format is no longer used, and will not be discussed any further.

5.2 COMMON TEST DATA FORMAT (CTDF)

5.2.1 RATIONAL & OVERVIEW

The Range Commanders Council (coordinating with McDonnell Douglas Aircraft Corp., now Boeing Corp.) have defined a new TSPI data format the Common Test Data Format (CTDF) currently at version 1. It was originally developed at NAWC-AD PAX River with the name Common TSPI Data Format. It is now called Common Test data Format in order to be more generic. CTDF superseded the old RIPS format here at China Lake in May of 1997. This file format was designed be used by all the Navy ranges including PAX, Pt. Mugu, and China Lake. This format is designed to be as portable as possible. It is also the TSPI format produced by the new TSPI data Range Control System (RCCS/2) .

All data fields in a CTDF file are ASCII text.

The tape format uses a physical block size of 24000 bytes. Inside each physical block are numerous logical blocks. As many logical blocks reside within the physical block as possible. Any remaining space is padded with spaces. Various different logical blocks are defined to contain different types of data.

Each logical block starts with 5 bytes containing digits which encodes the length of the logical block. The next character (the sixth character) is a capital letter which identifies the type of this logical block. The remaining $N - 6$ characters contain the data of the logical block.

Ex. This logical block contains the string "sample" within a bogus block type "A".

```
00012Asample
```

Each logical block is identified with a type character in the sixth byte. This allows logical blocks to contain different records of data. Version 1 of the CTDF defines 6 different records.

G	General Information Record
P	Parameter Identification Record
D	Data Record
E	End of Run Record
X	End of Data Record
S	Status Record

5.2.2 General Information Record (G)

The general information record contains things like the flight number, flight date, the TSPI reference point coordinates, a list TSPI sensors, ... etc. The general record contains a few generic items, and two tables. Each TSPI source is assigned a unique "track" number here in the general record, and an entry in the table of source tracks. Every reference point, has an entry in the reference point table. The following is a pseudo-code definition of the record structure.

```
TYPE
STRING = ARRAY OF CHAR;
G_Rec = RECORD (* variable length *)
len           : STRING[5];
type          : "G";
gh_version       : STRING[12];
gh_test_range    : STRING[30];
gh_flt_date      : STRING[8];
gh_num_tspi_srcs : STRING[3];
gh_tspi_srcs     : ARRAY [N] OF TspiSrcRec;
gh_num_ref_pts   : STRING[2];
gh_ref_pts       : ARRAY [M] OF RefPtRec;
END;

TspiSrcRec = RECORD (* 178 bytes fixed length *)
trk_num       : STRING[3];
trk_ac_bureau : STRING[11];
trk_flt_number: STRING[8];
trk_name      : STRING[8];
trk_desc      : STRING[48];
trk_smooth    : STRING[100];
END;

RefPtRec = RECORD (* 114 bytes fixed length *)
rp_type   : STRING[3;
rp_name   : STRING[8]
rp_desc   : STRING[48];
rp_lat    : STRING[15];  (* F15.6 +/-DDDMMSS.XXXXXX *)
rp_long   : STRING[15];  (* F15.6 +/-DDDMMSS.XXXXXX *)
rp_heigth : STRING[10];  (* right justified, zero padded *)
rp_rotate : STRING[15];  (* F15.6 +/-DDDMMSS.XXXXXX *)
END;
```

5.2.3 Parameter Identification Record (P)

The parameter identification record allocates the fields of the data records to the parameters found in the P record. Up to 9999 parameters can be defined. Each parameter has a name, engineering units, source track

number (from the G record), and inter–track number. Parameters also support inter-track data to measure the difference between two different tracks (normally set to zero). A reference point type is also defined to identify if this is a primary or secondary. Each Range facility may have different parameter names, and so data reduction software will still have to be tailored to the particular range activity.

```
P_Rec = RECORD (* variable length *)
len        : STRING[5];
type       : "P";
num_params : STRING[4];
params     : ARRAY [N] OF ParamRec;
END;

ParamRec = RECORD (* 53 bytes fixed length *)
param_name          : STRING[24];
param_eng_units     : STRING[20];
param_src_track_num : STRING[3];
param_inter_trk_num : STRING[3];
param_ref_pt_type   : STRING[3];
END;
```

5.2.4 Data Record (D)

The data record contains the time, and a 15 byte data field for each parameter defined in the P record. The time field has the following format: “DDD:HH:MM:SS.XXXXXXb”. The time is referenced to GMT. Here DDD is the Julian day, HH is the number of hours from midnight, MM the number of minutes from the start of the hour, SS is the number of seconds from the start of the minute, and XXXXXX is the number of microseconds from the start of the second. The “b” is a blank character. The event field is a left justified text field, padded with blanks on the right.

Sometimes a parameter has questionable quality. In this case a convention has been defined where the last character of the field will be overwritten with a “?”.

```
D_Rec = RECORD (* variable length *)
len         : STRING[5];
type        : "D";
data_time   : STRING[20];
data_event  : STRING[8];
data_fields : ARRAY [N] OF STRING[15];
END;
```

5.2.5 End of Run Record (E)

The end of run record delineates the end of a data run. There is usually a break in the data between passes. This also allows for differing setups (i.e. changes in targets, ... etc.).

```
E_Rec = RECORD (* 16 byte fixed length *)
len     : "00016";
type    : "E";
end_str : "END OF RUN";
END;
```

5.2.6 End of Data Record (X)

The end of data record indicates that there will be no more data records in the file (i.e. end of test).

```
X_Rec = RECORD (* 17 byte fixed length *)
len     : "00017";
type    : "X";
end_str : "END OF DATA";
END;
```

5.2.7 Status Record (S)

The status record provides up to 400 characters of comments to be entered. These may include comments about tracking instruments, data quality, event descriptions, . . . etc. Padded on the right with blanks.

```
S_Rec = RECORD (* 406 byte fixed length *)
len     : "00406";
type    : "S";
end_str : STRING[400];
END;
```

5.2.8 Parameters

The parameters defined in the parameter record describe the data that will be recorded within the data records. The test engineer may define any parameters of interest to be computed and stored in the data records. The most common will be the position:

"X_POS_FT", "Y_POS_FT", "Z_POS_FT"

Here the (X, Y, Z) position is referenced to the reference point and its flight line rotation angle. The X, and Y axis need to be rotated to give ENU coordinate system.

Alternatively the test engineer may have selected the ENU coordinate system.

"EAST_POS_FT", "NORTH_POS_FT", and "UP_POS_FT"

In addition, the test engineer may also have requested that the velocities be computed.

"X_VEL", "Y_VEL", and "Z_VEL"

A parameter that is very important, but in many cases gets overlooked is the validity of the track. When the TSPI sensor looses track of the target, we need to know.

The bottom line is that any set of parameters can be stored within the CTDF file. We need to standardize on the set of parameter names.

5.2.9 SAMPLE DATA

```
000000 :   3030 3335  3347 5665  7273 696f  6e20 312e :   00353GVersion 1.
000010 :   3020 4e41  5743 5750  4e53 2043  6869 6e61 :   0 NAWCWPNS China
000020 :   204c 616b  6520 2020  2020 2020  2020 2020 :    Lake
000030 :   3033 3331  3139 3937  3030 3130  3031 3136 :   0331199700100116
000040 :   3438 3938  2020 2020  2030 3030  3030 3036 :   4898     0000006
000050 :   354e 616d  6520 2020  2041 5244  5320 4750 :   5Name    ARDS GP
000060 :   5320 4d45  5448 4f44  2033 2054  444f 500a :   S METHOD 3 TDOP.
000070 :   2020 2020  2020 2020  2020 2020  2020 2020 :
000080 :   2020 2020  2020 2020  2041 6c67  6f72 6974 :            Algorit
000090 :   686d 2020  2020 2020  2020 2020  2020 2020 :   hm
0000a0 :   2020 2020  2020 2020  2020 2020  2020 2020 :
0000b0 :   2020 2020  2020 2020  2020 2020  2020 2020 :
0000c0 :   2020 2020  2020 2020  2020 2020  2020 2020 :
```

```
0000d0 :   2020 2020  2020 2020  2020 2020  2020 2020 :
0000e0 :   2020 2020  2020 2020  2020 2020  2030 3150 :                01P
0000f0 :   2020 4350  3220 2020  2020 4c2d  3230 2053 :     CP2     L-20 S
000100 :   5552 5645  5920 504f  494e 5420  2020 2020 :   URVEY POINT
000110 :   2020 2020  2020 2020  2020 2020  2020 2020 :
000120 :   2020 2020  2020 2020  2020 2b30  3335 3431 :             +03541
000130 :   3537 2e38  3035 3030  302b 3131  3733 3733 :   57.805000+117373
000140 :   332e 3638  3139 3939  2b30 3032  3136 322e :   3.681999+002162.
000150 :   3530 2020  2020 2020  2b30 2e30  3030 3030 :   50      +0.00000
000160 :   3030 3034  3837 5030  3030 394e  4f52 5448 :   000487P0009NORTH
000170 :   2050 4f53  4954 494f  4e20 2020  2020 2020 :    POSITION
000180 :   2020 2046  4545 5420  2020 2020  2020 2020 :      FEET
000190 :   2020 2020  2020 2030  3031 3030  3150 2020 :           001001P
0001a0 :   4541 5354  2050 4f53  4954 494f  4e20 2020 :   EAST POSITION
0001b0 :   2020 2020  2020 2020  4645 4554  2020 2020 :           FEET
0001c0 :   2020 2020  2020 2020  2020 2020  3030 3130 :               0010
0001d0 :   3031 5020  2044 4f57  4e20 504f  5349 5449 :   01P  DOWN POSITI
0001e0 :   4f4e 2020  2020 2020  2020 2020  2046 4545 :   ON           FEE
0001f0 :   5420 2020  2020 2020  2020 2020  2020 2020 :   T
000200 :   2030 3031  3030 3150  2020 4e4f  5254 4820 :    001001P  NORTH
000210 :   5645 4c4f  4349 5459  2020 2020  2020 2020 :   VELOCITY
000220 :   2020 4645  4554 2f53  4543 2020  2020 2020 :     FEET/SEC
000230 :   2020 2020  2020 3030  3130 3031  5020 2045 :          001001P  E
000240 :   4153 5420  5645 4c4f  4349 5459  2020 2020 :   AST VELOCITY
000250 :   2020 2020  2020 2046  4545 542f  5345 4320 :          FEET/SEC
000260 :   2020 2020  2020 2020  2020 2030  3031 3030 :               00100
000270 :   3150 2020  444f 574e  2056 454c  4f43 4954 :   1P  DOWN VELOCIT
000280 :   5920 2020  2020 2020  2020 2020  4645 4554 :   Y            FEET
000290 :   2f53 4543  2020 2020  2020 2020  2020 2020 :   /SEC
0002a0 :   3030 3130  3031 5020  204e 4f52  5448 2041 :   001001P  NORTH A
0002b0 :   4343 454c  2020 2020  2020 2020  2020 2020 :   CCEL
0002c0 :   2046 4545  542f 5345  432a 2a32  2020 2020 :    FEET/SEC**2
0002d0 :   2020 2020  2030 3031  3030 3150  2020 4541 :        001001P  EA
0002e0 :   5354 2041  4343 454c  2020 2020  2020 2020 :   ST ACCEL
0002f0 :   2020 2020  2020 4645  4554 2f53  4543 2a2a :         FEET/SEC**
000300 :   3220 2020  2020 2020  2020 3030  3130 3031 :   2          001001
000310 :   5020 2044  4f57 4e20  4143 4345  4c20 2020 :   P  DOWN ACCEL
000320 :   2020 2020  2020 2020  2020 2046  4545 542f :               FEET/
000330 :   5345 432a  2a32 2020  2020 2020  2020 2030 :   SEC**2          0
000340 :   3031 3030  3150 2020  3030 3136  3944 3039 :   01001P  00169D09
000350 :   303a 3233  3a34 303a  3030 2e31  3030 3030 :   0:23:40:00.10000
000360 :   3020 5020  2020 2020  2020 2d30  3030 3339 :   0 P       -00039
000370 :   3933 2e37  3838 3831  382d 3030  3136 3334 :   93.788818-001634
000380 :   392e 3733  3433 3735  2d30 3030  3030 3434 :   9.734375-0000044
000390 :   2e36 3138  3235 322b  3030 3030  3030 302e :   .618252+0000000.
0003a0 :   3932 3038  3533 2b30  3030 3030  3030 2e33 :   920853+0000000.3
0003b0 :   3539 3132  372b 3030  3030 3030  302e 3338 :   59127+0000000.38
0003c0 :   3536 3130  2b30 3030  3030 3030  2e31 3436 :   5610+0000000.146
0003d0 :   3335 312d  3030 3030  3030 302e  3035 3939 :   351-0000000.0599
0003e0 :   3637 2b30  3030 3030  3030 2e35  3534 3437 :   67+0000000.55447
0003f0 :   3330 3031  3639 4430  3930 3a32  333a 3430 :   300169D090:23:40
```

```
000400 :   3a30 302e  3135 3030  3030 2050  2020 2020 :   :00.150000 P
000410 :   2020 202d  3030 3033  3939 332e  3738 3434 :      -0003993.7844
```

5.3 TSPI on the Mux Bus

As a thought experiment we can invent new file formats.

TSPI data can be stored in many formats. One of the formats commonly used to process aircraft data is IRIG-Chapter-8 Hundred Percent Merged (HPM). We can process the TSPI data from the ranges into something that our analysis software already handles. We can create an HPM (muxall) data stream on a spare bus (bus 7).

Since our software must already accept the aircraft mux data in HPM, all we would need to provide is a data dictionary similar to a mux bus load listing. Here we could define remote terminal (RT) one (1) as administrative data. This would list the various TSPI sensors, and provide the TSPI origin. RT 2 would "transmit" the TSPI data from each sensor in messages 1-30, one message per sensor. Each message would have the same basic format.

Word Num	*Label*	*Description*
1	TIRIG1	IRIG Time Word 1
2	TIRIG2	IRIG Time Word 2
3	TIRIG3	IRIG Time Word 3
4	TLAT1	Latitude MSW
5	TLAT2	Latitude LSW
6	TLONG1	Longitude MSW
7	TLONG2	Longitude LSW
8	THEIGHT1	MSL Height MSW
9	THEIGHT2	MSL Height LSW

Chapter 6

Data Manipulation

This chapter discusses smoothing , interpolation , and data editing (elimination).

The TSPI position data, like any measurement, may not have good quality, it may be noisy and jump around. The coordinate values will need to be smoothed.

Another factor is that some data points may simply be bogus. They need to be edited out, so that they won't effect the smoothed data. Here we can evaluate the standard deviation, and anything beyond a specified multiple of that standard deviation is to be eliminated from the data set, and a new smoothed value evaluated. The smoothed values are stored in a state vector.

A state vector is a set of numbers describing various aspects of the tracked aircraft. These aspects include the time, position, and velocity. Optionally the state vector can also include the acceleration. These are most commonly are in a Cartesian coordinate system, but if needed could be in some other coordinate system.

Since, the smoothed state vector is a result of the filter, we can dis-associate from the data rate (and phase) of the original TSPI data. For example, we could change the data rate from 20 Hz to 10 Hz.

Here you use the smoothed data and interpolate to the new time line. In any case, measured data should be smoothed with a filter.

The one most commonly used filter for this purpose is the Least Squares Filter (LSF). The LSF process takes a set of measurements and fits a function of a specified type to an optimum match. Using a parabolic fit we can take a sequence of positions, and generate a state vector consisting of position, velocity, and acceleration.

6.1 TIME WINDOW

A time window of a specified size is used to select the measurements that are to be used for smoothing. Aircraft are very dynamic objects. Maneuvers mean changing the higher derivatives of their state vector (velocity, and acceleration). Maneuvers are localized in time. We do not want the aircraft state from the previous 10 minutes to effect the smoothing of the current state vector. So a time window is used to isolate the positions to be used in the smoothing from the entire data set. This time window for our purpose is usually set to plus or minus 1 second about the time being evaluated. The actual size of the window depends upon things such as how dynamic the maneuvers really are, and the date rate.

For example, Echo Range data is at a rate of 1 second. In order to get enough data points to smooth we need to expand to something like ± 5 seconds.

6.2 SMOOTHING

To filter noisy data we approximate the data with a function $f(x)$. This function $f(x)$ is a model of the dynamics of the objects whose data we are filtering, so we must choose the type of function appropriate to

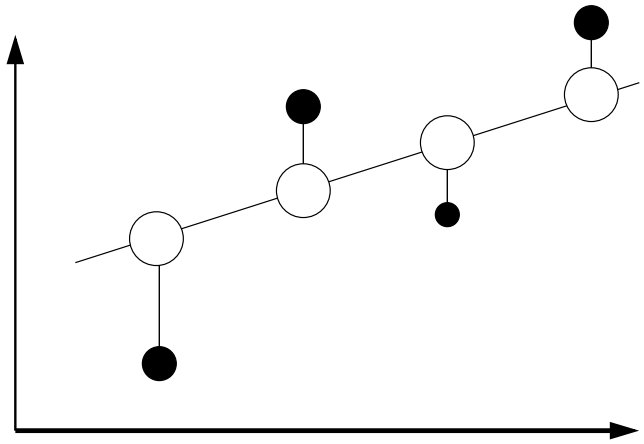

Figure 6.1: Fitting a line to the data element

that model. The coefficients to be used in the function are chosen to minimize the overall error, they are optimized for the data set being smoothed.

6.2.1 Least Squares Filtering

The Least Squares Fit (LSF) optimizes the model by minimizing the square of the distance from each data point to the modeled function. To get an extrema (such as a minimum) we take the function's derivatives and set them to zero. This gives us "normal equations", a system of linear equations. We solve this using Gauss–Jordan elimination to find the coefficients of the smoothed function f(x).

An aircraft is a maneuvering object that can accelerate. An appropriate filter function for it would be a parabolic function. Although, we will develop a parabolic least squares filter, first we will develop a simple linear filter. This allows the process to be developed with a minimum of confusion.

The actual state will change over time, so the filter is run over a window of close times. A reasonable time window is 2 seconds. This avoids problems of sharp maneuvers effecting the filtered state at a time of steady state (a maneuver will not effect the state beyond the time window).

Another benefit of using filtering is that we now can calculate a quality parameter. This quality parameter can be used similarly to the standard deviation.

A Linear Filter

One of the simplest functions is the straight line.

$$f(x) = y = a + bx \tag{6.1}$$

We need to determine the parameters a, and b so that when $f(x)$ is fitted through the given points (x_1, y_1), ..., (x_n, y_n) so that the sum of the squares of the distances of those points from the straight line is a minimum, where the distance is measured in the vertical direction (the y-direction).

Each data point (x_i, y_i) has a corresponding point on the smoothed function $f(x)$. When $f(x)$ is a line, then for each x_i we can evaluate $f(x_i) = a + bx_i$. Hence its distance from (x_i, y_i) is :

$$d_i = |y_i - a - ax_i| \tag{6.2}$$

Instead of using the absolute value, we use the square of the distance as the error function . All the errors for each point are summed. The error function is the sum of all the squared distances.

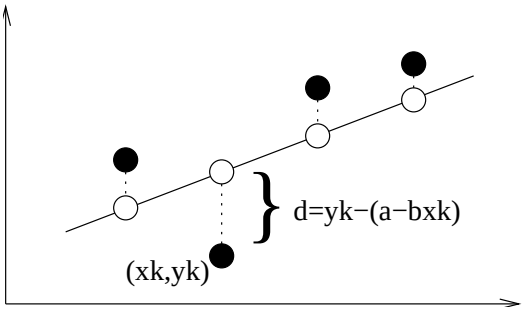

Figure 6.2: Vertical distance of a point from a straight line.

$$q = \sum_{i=1}^{n} (y_i - a - bx_i)^2 \tag{6.3}$$

Where q depends on a and b. A necessary condition for q to be an extrema (here a minimum) is that its partial derivatives with respect to a, and b be zero:

$$\frac{\partial q}{\partial a} = -2 \sum (y_i - a - bx_i) = 0 \tag{6.4}$$

$$\frac{\partial q}{\partial b} = -2 \sum x_i (y_i - a - bx_i) = 0 \tag{6.5}$$

where we sum over i from 1 to n.

Thus, we find the "normal equations" for our problem:

$$an + b \sum x_i = \sum y_i \tag{6.6}$$

$$a \sum x_i + b \sum x_i^2 = \sum x_i y_i \tag{6.7}$$

These "normal equations" are unique for each type of function $f(x)$ that we are modeling. We calculate the statistics (n, and all the $\sum$s) then solve the system of linear equations. We determine the coefficients a, and b using Gauss–Jordan elimination. The coefficients a, and b are the coefficients of $f(x)$.

The last step is to calculate the resulting error function. The smaller the better. A test to ensure that the error stays below some threshold is a good idea.

Example 1 (Linear Fit) Here is a small data set, which we wish to model with a linear function.

$$\begin{array}{l} (-1.0, 1.000) \\ (-0.1, 1.099) \\ (0.2, 0.808) \\ (1.0, 1.000) \end{array}$$

We calculate:

$$\begin{array}{rcl} n & = & 4 \\ \sum x_i & = & 0.10 \\ \sum x_i^2 & = & 2.05 \\ \sum y_i & = & 3.907 \\ \sum x_i y_i & = & 0.0517 \end{array}$$

Hence the "normal equations" for this data set are:

$$\begin{aligned} an + b\sum x_i &= \sum y_i \\ a\,(4) + b\,(0.10) &= 3.907 \end{aligned}$$

$$\begin{aligned} a\sum x_i + b\sum x_i^2 &= \sum x_i y_i \\ a\,(0.10) + b\,(2.05) &= 0.0517 \end{aligned}$$

Or when expressed as augmented matrices we solve this set of linear equations using Gauss–Jordan elimination .

$$\begin{aligned} \left[\begin{array}{cc|c} 4 & 0.10 & 3.907 \\ 0.10 & 2.05 & 0.0517 \end{array}\right] &= \left[\begin{array}{cc|c} 1 & 0.025 & 0.97675 \\ 0.10 & 2.05 & 0.0517 \end{array}\right] \\ &= \left[\begin{array}{cc|c} 1 & 0.025 & 0.97675 \\ 0 & 2.0475 & -0.045975 \end{array}\right] \\ &= \left[\begin{array}{cc|c} 1 & 0.025 & 0.97675 \\ 0 & 1 & -0.022454 \end{array}\right] \\ &= \left[\begin{array}{cc|c} 1 & 0 & 0.97731 \\ 0 & 1 & -0.022454 \end{array}\right] \end{aligned}$$

The solution is $a = 0.9773$, $b = -0.0224$, and we obtain the line:

$$y = 0.9773 - 0.0224x$$

Now to find the Root Sum Square (RSS) error of this estimate:

$$q = \sum_{i=1}^{n} (y_i - a - bx_i)^2$$

i	y_i	x_i	bx_i	$d_i = y_i - a - bx_i$	d_i^2
1	1.000	-1.0	0.02240	0.00030	9e-8
2	1.099	-0.1	0.00224	0.11946	0.01427
3	0.808	0.2	-0.04480	-0.16482	0.02717
4	1.000	1.0	-0.02240	0.04510	0.00203
Total					0.04347

So the error is the square root of q:

$$\sqrt{q} = \sqrt{0.04347} = 0.2085$$

6.2.2 A Parabolic Filter

A slightly more complicated function, but one that better matches the flight dynamics that we are trying to model, is the quadratic function which describes a parabola.

$$f(x) = y = a + bx + cx^2 \tag{6.8}$$

We need to determine the parameters a, b, and c so that when f(x) is fitted through the given points (x_1, y_1), ..., (x_n, y_n) so that the sum of the squares of the distances of those points from the parabola is a minimum.

When $f(x)$ is a quadratic, then for each x_i we can evaluate

$$y_i = a + bx_i + cx^2 \tag{6.9}$$

Hence its distance d_i from (x_i, y_i) is

$$d_i = |y_i - a - bx_i - cx_i| \tag{6.10}$$

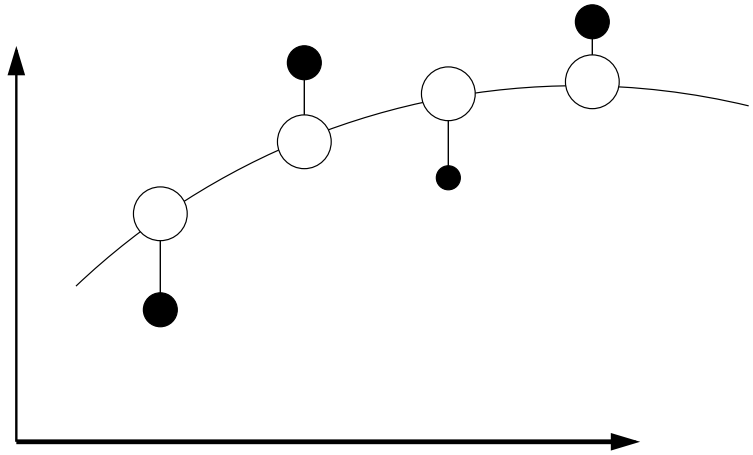

Figure 6.3: Fitting the data to a Parabola

The error function is the sum of all the squared distances.

$$q(a,b,c) = \sum_{i=1}^{n} \left(y_i - a - bx_i - cx_i^2\right)^2 \tag{6.11}$$

We minimize the error function by calculating the partial derivatives with respect to a, b, and c, then set them to zero.

$$\frac{\partial q}{\partial a} = -2\sum \left(y_i - a - bx_i - cx_i^2\right) = 0 \tag{6.12}$$

$$\frac{\partial q}{\partial b} = -2\sum x_i \left(y_i - a - bx_i - cx_i^2\right) = 0 \tag{6.13}$$

$$\frac{\partial q}{\partial c} = -2\sum x_i^2 \left(y_i - a - bx_i - cx_i^2\right) = 0 \tag{6.14}$$

Which gives the following "normal equations".

$$an + b\sum x_i + c\sum x_i^2 = \sum y_i \tag{6.15}$$

$$a\sum x_i + b\sum x_i^2 + c\sum x_i^3 = \sum x_i y_i \tag{6.16}$$

$$a\sum x_i^2 + b\sum x_i^3 + c\sum x_i^4 = \sum x_i^2 y_i \tag{6.17}$$

From here the process is the same, collect the statistics, put them into the "normal equations", solve this system of linear equations using Gauss–Jordan elimination.

Example 2 (Parabolic Fit)

We will use the same data set as the linear filter example.

i	x_i	y_i
1	-1.0	1.000
2	-0.1	1.099
3	0.2	0.808
4	1.0	1.000

We calculate the statistics:

$$\begin{aligned} n &= 4 \\ \sum x_i &= 0.1 \\ \sum x_i^2 &= 2.05 \\ \sum x_i^3 &= 0.007 \\ \sum x_i^4 &= 2.0017 \\ \sum y_i &= 3.907 \\ \sum x_i y_i &= 0.0517 \\ \sum x_i^2 y_i &= 2.0433 \end{aligned}$$

The "normal equations" for this data set then become the following system of linear equations:

$$\begin{array}{llll} a\,(4) & +b\,(0.1) & +c\,(2.05) & = \ 3.907 \\ a\,(0.1) & +b\,(2.05) & +c\,(0.007) & = \ 0.0517 \\ a\,(2.05) & +b\,(0.007) & +c\,(2.0017) & = \ 2.0433 \end{array}$$

We use Gauss–Jordan elimination to solve the above set of simultaneous equations.

$$\left[\begin{array}{lll|l} 4 & 0.1 & 2.05 & 3.907 \\ 0.1 & 2.05 & 0.007 & 0.0517 \\ 2.05 & 0.007 & 2.0017 & 2.0433 \end{array}\right] = \left[\begin{array}{lll|l} 1 & 0.025 & 0.5125 & 0.97675 \\ 0 & 2.0475 & -0.04425 & -0.4598 \\ 0 & -0.04425 & 0.95108 & 0.04097 \end{array}\right]$$

$$= 1\left[\begin{array}{lll|l} 0 & 0.51304 & & 0.97731 \\ 0 & 1 & -0.02161 & -0.02246 \\ 0 & 0 & 0.95012 & 0.03998 \end{array}\right] = 1\left[\begin{array}{lll|l} 0 & 0 & & 0.95572 \\ 0 & 1 & 0 & -0.02155 \\ 0 & 0 & 1 & 0.04208 \end{array}\right]$$

The solution is a = 0.95572, b = -0.02155, and c = 0.04208, so we obtain the parabola:

$$y = 0.95572 - (0.02155)x + (0.04208)x^2$$

Now to find the error of this estimate:

$$q\,(a,b,c) = \sum_{i=1}^{n} \left(y_i - a - bx_i - cx_i^2\right)^2 \tag{6.18}$$

i	y_i	x_i	bx_i	cx^2	$d_i = y_i - a - bx_i - cx_i^2$	d_i^2
1	1.000	-1.0	0.02155	0.04208	-0.01935	0.00037
2	1.099	-0.1	0.00216	0.00042	0.14070	0.01980
3	0.808	0.2	-0.00431	0.00168	-0.14509	0.02105
4	1.000	1.0	-0.02155	0.04208	0.02375	0.00056
Total						0.04179

So the error is the square root of q:

$$\sqrt{q} = \sqrt{0.04179} = 0.20442$$

Using the Parabolic Filter

A maneuvering aircraft has a state vector consisting of the position, velocity, and acceleration. Each of which is easily determined when using a parabolic filter.

The basic kinematic equation:

$$x = X_0 + V_0 t + \frac{1}{2} A t^2 \tag{6.19}$$

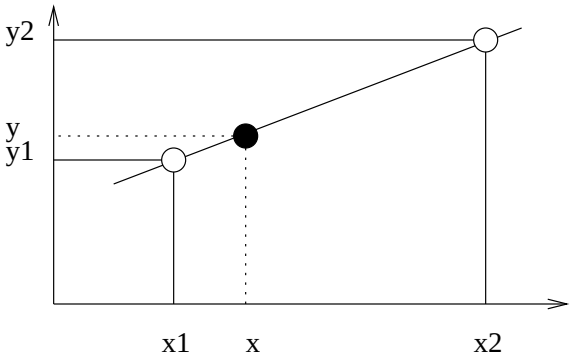

Figure 6.4: Linear Interpolation

is a quadratic function just like the parabolic filters gives us. The coefficients from the parabolic filter have the following correspondence:

$$a = X_0 \quad (6.20)$$
$$b = V_0 \quad (6.21)$$
$$c = \frac{1}{2}A \quad (6.22)$$

We differentiate the kinematic equation to get velocity and acceleration.

position	X	$a + bt + 2ct^2$
velocity	V	$b + 2ct$
acceleration	A	$2c$

6.3 Changing Time Line

When comparing aircraft data with TSPI data, the aircraft data occurs at a particular time and is not synched to occur at the same rate (or phase) as the TSPI data. The TSPI data could be recorded with a large time quantum like once a second or larger (RCC normally uses a 20 HZ rate, but Echo Range defaults to 1 second). In this case you may need to choose a different time line. So the aircraft data occurred at a time bracketed by two TSPI data elements. A simple linear interpolation can be used if the separation between samples is small enough, otherwise a parabolic interpolation may be more appropriate.

The basic process is to select the time line to which the smoothed data is to be evaluated upon. The for each discrete time on that time line, we select a set of data points within a time window around that discrete time. That data set is then used to evaluate the coefficients of the parabolic filter at that time.

6.4 Linear Interpolation

We want to evaluate the TSPI targets state (position, velocity, acceleration) at any time, but the TSPI data is only available at discrete times. The simplest (and poor) method of getting the state would be to pick the closest in time of the discrete state vectors. This does have the advantage that it is fast, and simple to implement.

A more accurate approach is to interpolate the state using the two surrounding discrete state vectors. This entails determining how far between the two bracketing events is the requested time. Then using this distance as a scaling factor.

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) \tag{6.23}$$

6.5 Data Editing

Sometimes a data value gets into the data stream, that shouldn't be there (it is an outlier, or simply bogus). If we detect such data values, they need to be edited out then the smaller data set needs to be smoothed. The first and biggest task is to detect bad data values.

The first step is to select a time line exactly corresponding to the times of the original data set. Select the data points within the time window of interest for each discrete time in the time line. We then smooth the data points within that time window, but in addition we calculate the standard deviation.

If the data point is larger than a specified multiple of the standard deviation, then mark it as deleted. Then smooth the data set again with the original time line, but this time without using the deleted data points.

The z–value is a measurement value normalized to the current standard deviation. A good z–value threshold for this purpose is 5.

6.6 Error Corrections and Biasing

TBD.

Appendix A

Derivations

A.1 Geocentric G From Latitude

Given the latitude and height above the spheroid, we want to get the value of G.

We first recognize that there are two parts to this problem.

The first contribution is from the point on the spheroid. Starting with eq. (A.17) from section A.2.1:

$$y = x(1 - e^2)\tan\phi \tag{A.1}$$

And taking x as the distance from the axis to the point on the spheroid we get:

$$x = N\cos\phi \tag{A.2}$$

Substituting eq. (A.2) into eq. (A.1) we get

$$\begin{aligned} y &= N\cos\phi(1 - e^2)\tan\phi && \text{(A.3)} \\ y &= N(1 - e^2)\sin\phi && \text{(A.4)} \\ G_1 &= N(1 - e^2)\sin\phi && \text{(A.5)} \end{aligned}$$

The second contribution is the height above the spheroid. From fig A.1 it is readily apparent that the contribution from the height is:

$$G_2 = h\sin\phi \tag{A.6}$$

Now combining the two contributions:

$$\begin{aligned} G &= G_1 + G_2 && \text{(A.7)} \\ G &= N(1 - e^2)\sin\phi + h\sin\phi && \text{(A.8)} \\ G &= \left[N(1 - e^2) + h\right]\sin\phi && \text{(A.9)} \end{aligned}$$

A.2 EARTH RADIUS OF CURVATURE DERIVATIONS

The shape of the earth can be approximated by an oblate spheroid. This looks like a ball which was squeezed down at the poles, and the equator bulged out. This assumes that the earth is symmetrical about the axis of rotation, the curvature does not change when moving east-west. The curvature does change when moving north-south. This means that at any point on the surface of the earth, the curvature depends upon the latitude and not the longitude.

If the earth were bisected from pole to pole (along a meridian line) the earth's crust would look like an ellipse. So, most of the derivations will be using the equation of an ellipse as their starting point. An ellipse is represented by the following equation:

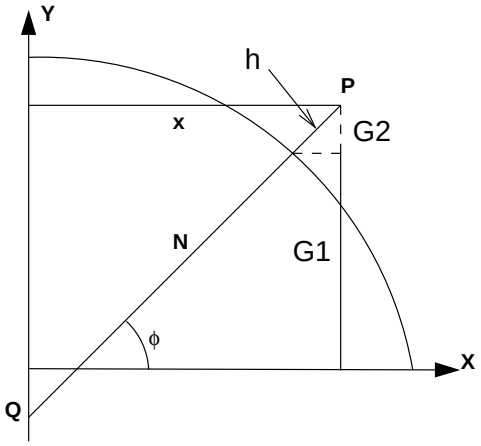

Figure A.1: Height above the spheroid

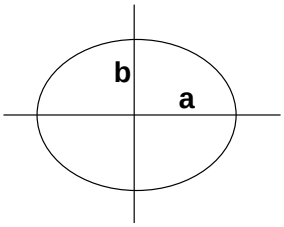

Figure A.2: A meridian line bisecting the spheroid forms an ellipse

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{A.10}$$

The deviation of an ellipse from a circle is the eccentricity "e", given by:

$$e^2 = \frac{a^2 - b^2}{a^2} \tag{A.11}$$

A.2.1 Radius of Curvature in the Prime Vertical (East-West direction)

The east–west radius of curvature in the prime vertical gives the curvature of the great circle which is tangent to the latitude line. The radius of this curvature is used as the radial distance from a point on the earth's surface downwards to the axis of rotation. The down vector depends upon the latitude, and therefore so does the radius of curvature N.

First, we define a point P on the surface of the spheroid, then we can determine its tangent plane, and corresponding down line. We then solve for x, and substitute the geometry of N.

Solving eq. (A.11) for b, and substituting into eq. (A.10) gives:

$$x^2\left(1 - e^2\right) + y^2 = a^2\left(1 - e^2\right) \tag{A.12}$$

In figure A.3, the point P is a point on the surface of the earth with a down vector which crosses the plane of the equator with an angle of ϕ. This angle ϕ is the latitude at the point P. N is the distance of the line PQ from the surface to the intersection with the axis of rotation. *Note*: that the line normal to the surface does not intersect the origin of the spheroid except when the point P is at one of the poles, or on the equator.

We get the tangent line's slope of the surface at point P using the derivative of eq. (A.12).

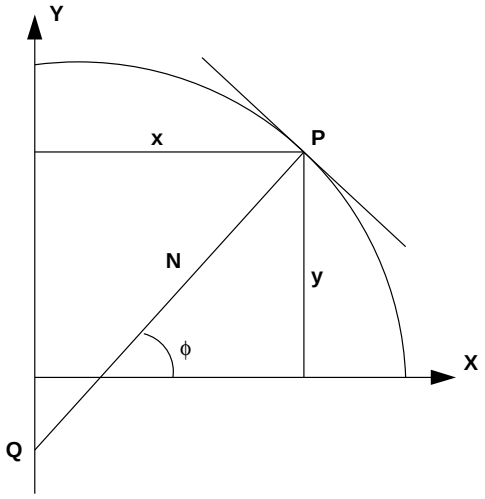

Figure A.3: N is the length of the down-line at point P

$$y^2 = a^2\left(1-e^2\right) - x^2\left(1-e^2\right) \tag{A.13}$$

$$2yy' = -2x\left(1-e^2\right) \tag{A.14}$$

$$y' = \frac{dy}{dx} = -\frac{x}{y}\left(1-e^2\right) \tag{A.15}$$

The line PQ is perpendicular to the tangent line at P. To get the slope of the line PQ we use the negative reciprocal of the tangent line's slope. the slope of the line PQ is also the tangent of the angle cutting through the x-axis.

$$-\frac{dy}{dx} = \frac{y}{x(1-e^2)} = \tan\phi \tag{A.16}$$

Solving for y yields:

$$y = x(1-e^2)\tan\phi \tag{A.17}$$

Now we substitute eq. (A.17) back into eq. (A.12) and solving for x.

$$x^2(1-e^2) + \left[x\left(1-e^2\right)\tan\phi\right]^2 = a^2\left(1-e^2\right) \tag{A.18}$$

$$x^2\left(1-e^2\right) + x^2\left(1-e^2\right)^2\tan^2\phi = a^2\left(1-e^2\right) \tag{A.19}$$

$$x^2 + x^2\left(1-e^2\right)\tan^2\phi = a^2 \tag{A.20}$$

$$x^2\left[1+\left(1-e^2\right)\tan^2\phi\right] = a^2 \tag{A.21}$$

(A.22)

$$x^2 = \frac{a^2}{1+(1-e^2)\tan^2\phi} \tag{A.23}$$

$$x^2 = \frac{a^2\cos^2\phi}{\cos^2\phi + (1-e^2)\sin^2\phi} \tag{A.24}$$

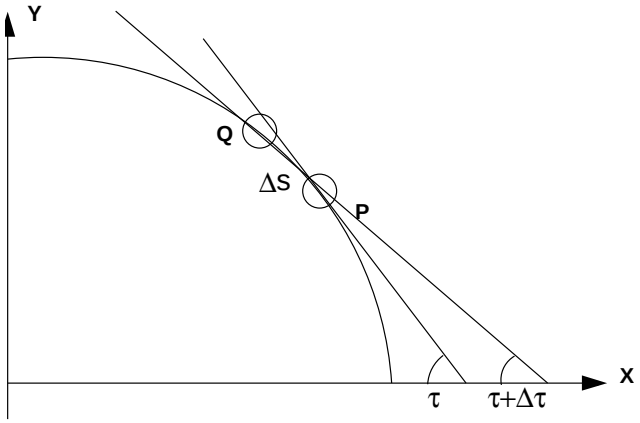

Figure A.4: The curvature is the change in slope

$$x = \frac{a\cos\phi}{\sqrt{\cos^2\phi + (1-e^2)\sin^2\phi}} \tag{A.25}$$

$$x = \frac{a\cos\phi}{\sqrt{1-e^2\sin^2\phi}} \tag{A.26}$$

Referring back to figure A.3, we find another way of representing x ($x = N\cos\phi$).
Which gives N as:

$$\boxed{N = \frac{a}{\sqrt{1-e^2\sin^2\phi}}} \tag{A.27}$$

This is the radius of curvature in the east–west direction at the latitude ϕ. It corresponds to the distance from the point P to the point Q somewhere on the axis of rotation.

A.2.2 Radius of Curvature in the Meridian (North–South direction)

The curvature of the earth in the north–south direction also depends upon the latitude. At the equator, the radius of curvature in the north–south direction R is the semi–major axis a, at the poles it is the semi–minor axis b. At the north or south pole the two radius' of curvature R, and N are equal.

The shape of a curve depends upon the rate of change of its direction (curvature K). So we need the derivative of the curve's slope in terms of the latitude.

In figure A.4, we see two points P, and Q (located at latitudes ϕ_1 and ϕ_2) separated by a distance Δs along the surface. Their tangent lines intersect the equator with angles of τ and $\tau + \Delta\tau$. These tangent lines are normal to their latitude lines N_1 and N_2 (not drawn).

The average curvature of the arc PQ is given by $\Delta\tau/\Delta s$. The curvature at P, denoted by K is the limiting value of the average curvature as Δs approaches 0 (zero).

$$K = \lim_{\Delta s\to 0}\frac{\Delta\tau}{\Delta s} = \frac{d\tau}{ds} \tag{A.28}$$

The radius of curvature is the reciprocal of the curvature:

$$R = \frac{1}{K} \tag{A.29}$$

Now since,

$$\tau = \arctan\left(\frac{dy}{dx}\right) = \arctan\left(y'\right) \tag{A.30}$$

Differentiating with respect to x yields

$$\frac{d\tau}{dx} = \frac{d\left[\arctan\left(dy/dx\right)\right]}{dx} = \frac{\frac{d^2y}{dx^2}}{1+\left(\frac{dy}{dx}\right)^2} = \frac{y''}{1+y'^2} \tag{A.31}$$

Now since,

$$\frac{ds}{dx} = \left(1+y'^2\right)^{1/2} \tag{A.32}$$

We can divide them to yield,

$$\frac{d\tau}{ds} = K = \frac{y''}{\left(1+y'^2\right)^{3/2}} \tag{A.33}$$

Now, since the radius of curvature is the reciprocal of the curvature, we can express the radius of curvature as:

$$R = \frac{1}{K} = \frac{\left(1+y'^2\right)^{3/2}}{y''} \tag{A.34}$$

Remembering eq. (A.15),

$$y' = -\frac{x}{y}\left(1-e^2\right)$$

We can differentiate this to get the second derivative:

$$y'' = -\left(1-y'^2\right) - e^2 \tag{A.35}$$

The normal line is perpendicular to the tangent line. So, we take the negative reciprocal of the slope of the normal line, the expression for the first and second derivatives of y with respect to x can also be written as:

$$y' = \frac{1}{\tan\phi} = \cot\phi \tag{A.36}$$

and

$$y'' = \frac{1+\cot^2\phi - e^2}{y} = \frac{\csc^2\phi - e^2}{y} \tag{A.37}$$

Where ϕ is the geodetic latitude as shown in figure A.3. Now we remember eqs. (A.17), and (A.26).

$$y = x(1-e^2)\tan\phi$$

$$x = \frac{a\cos\phi}{\sqrt{1-e^2\sin^2\phi}}$$

Which can be combined to yield:

$$y = \frac{a(1-e^2)\sin\phi}{\sqrt{1-e^2\sin^2\phi}} \tag{A.38}$$

Now, substituting these values for y, y', and y'' [eqs. (A.38), (A.36), and (A.37)] into the equation for R (A.34).

$$R = \left[1+\left(-\cot\phi\right)^2\right]^{3/2} \frac{\left[\frac{a(1-e^2)\sin\phi}{\sqrt{1-e^2\sin^2\phi}}\right]}{-\csc^2\phi + e^2} \tag{A.39}$$

$$R = \frac{\left[a(1-e^2)\sin\phi\right]\left(1/sin^3\phi\right)}{\sqrt{1-e^2\sin^2\phi}\left[\left(1/\sin^2\phi\right) - e^2\right]} \tag{A.40}$$

Finally,

$$\boxed{R = \frac{a(1-e^2)}{\left(1-e^2\sin^2\phi\right)^{3/2}}} \tag{A.41}$$

Or, alternatively:

$$\boxed{R = \frac{N(1-e^2)}{1-e^2\sin^2\phi}} \tag{A.42}$$

R can be visualized a the radius of a circle that best matches the earth's surface along the meridian line at that specified latitude. Whereas N, the radius of curvature in the East–West direction, always terminates on the Earth's axis of rotation, R the radius of curvature in the North–South direction usually terminates short of it (R is less than N).

Appendix B

Coordinate Transformations

B.1 Introduction

There are three basic transformations, translation, scaling, and rotations which can be applied to a vector. Although rotations are the transformation that we are primarily interested in, we will take a brief look at the other ones. Translation is just adding an offset to the vector. Scaling modifies the length of the vector. Rotation is a function which maintains the length of the vector, but changes their orientation referenced to the axes (or alternatively can be though of as rotating the axes). These are all applied to a vector by a matrix multiply. All these transformations can be thought of as changing coordinate systems.

Transformations are applied by performing a matrix multiply to the vector. Because matrix multiplies are not commutative we must specify on which side of the multiply is the matrix and which is the vector. We will then define one as the conventional method. *Note:* In the following discussion the vector X is a column vector.

$$X' = A \cdot X \tag{B.1}$$

$$X'^T = X^T \cdot B \tag{B.2}$$

We will pick eq. (B.1) as the conventional order of matrix multiplication. Here the transformation matrix occurs to the left of the vector being transformed.

Note: For the case of orthonormal matrices, A is related to B by a simple transpose. $A = B^T$

B.1.1 Viewed as Coordinate Transforms

Transforming a vector may alternately be thought of as a change from one coordinate system to another. We think of moving the coordinate system axes by the reverse of the change of the vector. In figure B.5, we have

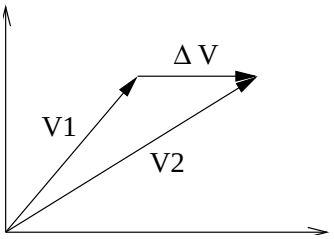

Figure B.1: Translation

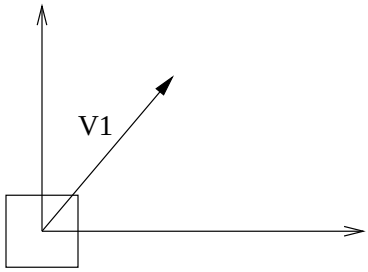

Figure B.2: Before Scaling

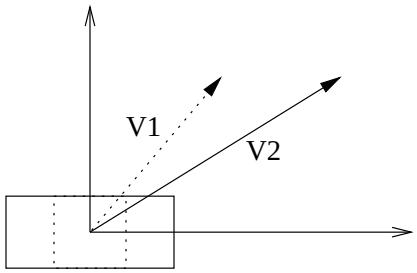

Figure B.3: After Scaling

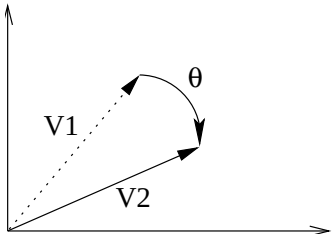

Figure B.4: Rotation

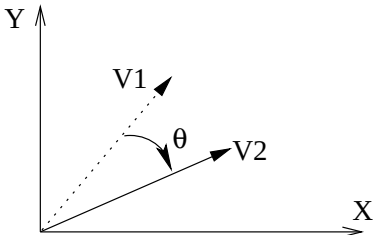

Figure B.5: Vector Rotation

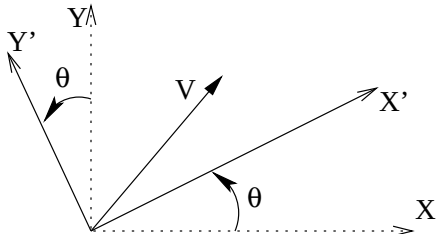

Figure B.6: Coordinate Rotation

a vector V rotated by an angle of negative theta ($-\theta$) into vector V'. This can also be thought of as changing from one coordinate system to another X', Y' (see figure B.6)

B.1.2 Composite Matrices

A composite matrix is one matrix which contains many transformations. It is formed by multiplying as many transformation matrices together as needed. Remember that matrix multiplies are not commutative, they depend upon the order in which they are multiplied. The simple rule is that the matrix closest to the vector gets applied first, then the second, and so on. For the matrix to vector multiply of (eq. B.1) this means the first transformation needs to be the rightmost one (the closest to the vector).

For Example: Lets say that we have a vector that we want to scale, then rotate, and finally translate. We set up the transformation matrices in order where the first one is closest to the vector, and the last one is the outermost one.

$$X' = T(\Delta x, \Delta y, \Delta z)R(\theta_1, \theta_2, \theta_3)S(s_x, s_y, s_z)X$$

We can form one composite matrix C that contains all the above matrices.

$$C = T(\Delta x, \Delta y, \Delta z)R(\theta_1, \theta_2, \theta_3)S(s_x, s_y, s_z)$$

Composite matrices are especially useful when a large number of vectors need the same transformation.

B.1.3 Homogeneous Coordinates

In the discussions below we use a matrix to specify the coordinate transformation, where the transformation is applied to the vector by performing a matrix multiply. Not all the transformations will work as a matrix multiply (translation is additive) with a simple matrix. If we add another component to the vector and matrices to form a homogeneous vector and matrix, then we can create a translation matrix which will work using a matrix multiply. The only reason to use homogeneous coordinates is when translations need to be built into a composite matrix.

$$X = \begin{bmatrix} x_x \\ x_y \\ x_z \\ X_w = 1 \end{bmatrix} \tag{B.3}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a13 & 0 \\ a_{21} & a_{22} & a23 & 0 \\ a_{31} & a_{32} & a33 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.4}$$

For most coordinate transformations of interest (i.e. rotations) we do NOT need to use homogeneous coordinates.

B.1.4 Orthogonality

When two vectors are orthogonal their inner products (dot products) are zero (they are perpendicular to each other). A set of basis vectors which spans a vector space is usually chosen to be orthogonal, because they are linearly independent. A coefficient in front of one basis can be changed without effecting the other basis coefficients.

A matrix can also be orthogonal. These have the special property that the transpose of a matrix is also it's inverse. A matrix is orthogonal if and only if the columns of the matrix form an orthonormal basis. See [Cullen-72] page 158. Pure rotation matrices are orthogonal. For example the following rotation matrix is orthogonal,

$$R(\theta) = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{B.5}$$

because the columns of the matrix form an orthonormal basis (inner-products are zero).

$$C1 = \begin{bmatrix} cos\theta \\ sin\theta \end{bmatrix} \tag{B.6}$$

$$C2 = \begin{bmatrix} -sin\theta \\ cos\theta \end{bmatrix} \tag{B.7}$$

$$C1^T \cdot C2 = -cos\theta sin\theta + cos\theta sin\theta = 0 \tag{B.8}$$

This means that the inverse of a rotation matrix is very easy the generate, just take the transpose of it.

$$R(\theta)^{-1} = R(\theta)^T \tag{B.9}$$

Another point is that an orthonormal matrix's determinate is equal to 1.

B.2 Selected List of Transformations

The primitive transformations discussed below are translations, scalings, and rotations. The rotation transformation is the one used most heavily in TSPI data reduction. There are other transformations that are not discussed here such as skew, reflection, ... etc.

B.2.1 Translations

A translation applies an offset to a vector by performing a vector addition. This can easily be extended to a three dimensional vector, and translation.

$$X' = X + Y = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{B.10}$$

$$X' = T(T_x, T_y)X = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \tag{B.11}$$

$$T(\Delta x, \Delta y) = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \tag{B.12}$$

B.2.2 Scaling

A scaling operation modifies the length of the vector. Each component is modified separately from the others. This Scaling matrix is symmetric.

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.13}$$

B.2.3 Rotations

A rotation is performed about an axis. The simplest case is a two-dimensional vector $[xy]$ rotated about an imaginary z–axis by an angle θ.

$$x' = a_{11}x - a_{12}y \tag{B.14}$$
$$y' = a_{21}x + a_{22}y \tag{B.15}$$

$$x' = cos\theta x - sin\theta y \tag{B.16}$$
$$y' = sin\theta x + cos\theta y \tag{B.17}$$

$$X' = \begin{bmatrix} x'_x \\ x'_y \end{bmatrix} = R(\theta) \cdot X = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x_x \\ x_y \end{bmatrix} \tag{B.18}$$

Note that the order of the matrix multiply determines the location of the negative $sin\theta$. The inverse transform of the rotation is R(-$sin\theta$). Pure rotation matrices are orthogonal, so the inverse of a rotation matrix is simply its transpose.

B.2.4 Rotations in Three Dimensions

When we have a three dimensional vector, each rotation has to specify the axis of rotation. The diagram below shows a right handed orthogonal coordinate system.

A rotation about the z–axis looks very similar to eq. (B.18), where the z component is not changed. The rotations matrices for rotations about the x, and y axes also keep one component the same. A rotation is positive when a vector is rotated about an axis in the direction of one of the arrows, and negative when rotated the reverse direction.

Here we use homogeneous coordinates for generality. If a rotation matrix is not going to be used in a composite matrix containing a translation then the rotation matrix can be simplified to a 3x3 matrix.

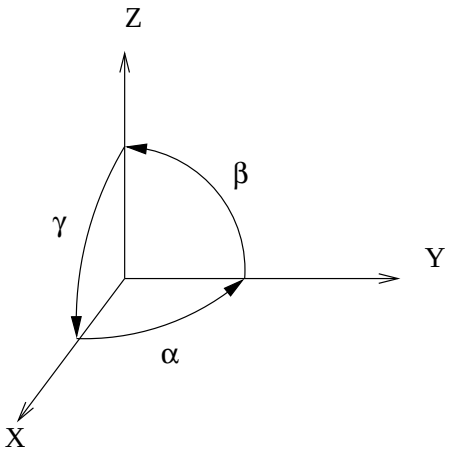

Figure B.7: Rotations in a Right Handed Coordinate System

The Primitive 3D Rotation Matrices are listed in order XYZ which forms a right handed coordinate system. Positive angles are shown in figure B.7.

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.19}$$

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & \cos\beta & 0 \\ -\sin\beta & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.20}$$

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma & 0 \\ 0 & \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.21}$$

B.3 More on Composite Matrices

Any sequence of transformations can be combined into a composite transformation matrix. This is created by multiplying each of the transformation matrices together. The order of multiplication depends upon the convention used.

Under the $X' = AX$ system (the conventional one) the last transformation is the leftmost. While in the $X' = XB$ system the last transform is the rightmost.

Example 1: To go from NED local coordinates to FRD (front/right/down) body coordinates, we use three rotations. These are (1) Heading, (2) Pitch, and finally (3) Roll.

Example 2: First rotate about the east–axis by $-\phi$ (latitude), then about the G–axis by $-\lambda$ (longitude) to convert from NED to EFG.

$$C(\phi, \lambda) = R_y(-\phi) R_z(-\lambda) \tag{B.22}$$

$$R_y(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & \cos\phi & 0 \\ -\sin\phi & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.23}$$

$$R_z(-\lambda) = \begin{bmatrix} \cos\lambda & \sin\lambda & 0 & 0 \\ -\sin\lambda & \cos\lambda & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.24}$$

Appendix C

The Geoid

The earth's surface can only be approximated by a spheroid, because the earth is "bumpy". The geoid is a gravitationally equi-potential surface which approximates the mean sea level. The geoid differs from the spheroid due to gravitational variances at different latitudes, and longitudes. Only with the advent of satellites orbiting the earth have we been able to accurately measure the position of the geoid. The geoid is described by a table of geoid separations at different positions over the surface of the earth.

The GEOID–84 geoidal separation is the difference between the local mean sea level and the WGS–84 spheroid. The geoid has to be empirically measured, it can be described by entries in a table (see below). The official geoid table contains entries for each $1/2^\circ$ of latitude, and longitude, this used to be classified. It is now available from the National Imagery and Mapping Agency (NIMA) at "http://www.nima.mil/". We supply a table in this appendix which has entries every 10 degrees. Here at China Lake the geoid is about -28.9 meters (the geoid is 94 feet below the spheroid). The continental United States has a negative geoid value, meaning that the geoid is below the spheroid defined by WGS–84.

The GEOID–84 was defined at the same time as the WGS–84, before the GPS/NAVSTAR satellite constellation was deployed. Using the more accurate results of GPS a new geoid model was developed called GEOID–90, which basically refines the GEOID–84 numbers with decimal points. The GEOID–90 data base is about 3 MB in binary, and is not included here. Rather the GEOID–84 in the form of a 10° by 10° grid is provided and a method of interpolation which should be accurate enough for our purpose.

The documentation that comes from DMA describes a spherical harmonic method of interpolation. This is very complicated. NIMA provides a much simpler (and faster) method (bi–linear interpolation). This is the method I use.

A bi–linear interpolation of the geoid tables may be performed to derive the geoid separation at any given latitude/longitude. Now we could perform this interpolation for every position of the aircraft, but this is not necessary. Since the geoid does not change radically within ± 20 miles we can determine the geoid for the local area, and use that for the entire area.

For example The airfield's cold–line at China Lake is located at

LAT ϕ	35 41 18	35.688333
LONG λ	-117 40 50	-117.680556
ELEV (ft)	2210	

The surrounding geoid values (in meters) are:

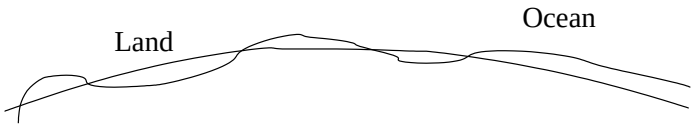

Figure C.1: The geoidal separation is the difference between the mean sea level and the spheroid.

			longitude	
			$\lambda 1$	$\lambda 2$
			-120	-110
latitude	$\phi 2$	40	-21	-16
[deg.]	$\phi 1$	30	-42	-29

Where the geoid values form an array:

$$\begin{bmatrix} N_4 & N_3 \\ N_1 & N_2 \end{bmatrix} = \begin{bmatrix} \text{-21} & \text{-16} \\ \text{-42} & \text{-29} \end{bmatrix}$$

Here we want $N(\phi, \lambda)$ and we are given $N_1(\phi_1, \lambda_1)$, $N_2(\phi_1, \lambda_2)$, $N_3(\phi_2, \lambda_2)$, and $N_4(\phi_2, \lambda_1)$. The bi-linear interpolation formula has the form:

$$N(\phi, \lambda) = a_0 + a_1 X + a_2 Y + a_3 XY \tag{C.1}$$

where:

$$\begin{aligned} a_0 &= N_1 && \text{(C.2)} \\ a_1 &= N_2 - N_1 && \text{(C.3)} \\ a_2 &= N_4 - N_1 && \text{(C.4)} \\ a_3 &= N_1 + N_3 - N_2 - N_4 && \text{(C.5)} \\ X &= (\lambda - \lambda_1) / (\lambda_2 - \lambda_1) && \text{(C.6)} \\ Y &= (\phi - \phi_1) / (\phi_2 - \phi_1) && \text{(C.7)} \\ & && \text{(C.8)} \end{aligned}$$

C.0.1 The Geoidal Separation at China Lake

For our sample location (China Lake):

$$\begin{aligned} a_0 &= -42 && \text{(C.9)} \\ a_1 &= -29 - (-42) = 13 && \text{(C.10)} \\ a_2 &= -21 - (-42) = 21 && \text{(C.11)} \\ a_3 &= -42 - 16 - (-29) - (-21) = -8 && \text{(C.12)} \\ X &= \frac{-117.680556 - (-120)}{-110 - (-120)} = 0.2319444 && \text{(C.13)} \\ Y &= \frac{35.68833 - 30}{40 - 30} = 0.568833 && \text{(C.14)} \end{aligned}$$

This gives the following geoid separation:

$$N(35.68033, -117.680556) = -28.09$$

The geoid value of -28.09 meters is very close to the actual measured value of -28.9 meters (-94.8 feet) used at the L20 surveyed point.

Here at China Lake the geoid (i.e., sea level) is 94 feet below the spheroid. Ground altitude (2200 MSL) is only 2106 feet above the spheroid.

C.1 GEOID–84 HEIGHT TABLES $(10° by 10°)$

Table C.1: Geoid–84 in Meters (Longitude 0 to 80)

	Longitude								
Latitude	0	10	20	30	40	50	60	70	80
90	13	13	13	13	13	13	13	13	13
80	33	34	28	23	17	13	9	4	4
70	51	43	29	20	12	5	−2	−10	−14
60	47	41	21	18	14	7	−3	−22	−29
50	47	48	42	28	12	−10	−19	−33	−43
40	52	48	35	40	33	−9	−28	−39	−48
30	36	28	29	17	12	−20	−15	−40	−33
20	31	26	15	6	1	−29	−44	−61	−67
10	22	23	2	−3	−7	−36	−59	−90	−95
0	18	12	−13	−9	−28	−49	−62	−89	−102
−10	12	13	−2	−14	−25	−32	−38	−60	−75
−20	17	23	21	8	−9	−10	−11	−20	−40
−30	22	27	34	29	14	15	15	7	−9
−40	18	26	31	33	39	41	30	24	13
−50	25	26	34	39	45	45	38	39	28
−60	16	19	25	30	35	35	33	30	27
−70	16	16	17	21	20	26	26	22	16
−80	−4	−1	1	4	4	6	5	4	2
−90	−30	−30	−30	−30	−30	−30	−30	−30	−30

Table C.2: Geoid84 in Meters (Longitude 90° to 170°)

	Longitude								
Latitude	90	100	110	120	130	140	150	160	170
90	13	13	13	13	13	13	13	13	13
80	1	−2	−2	0	2	3	2	1	1
70	−12	−10	−14	−12	−6	−2	3	6	4
60	−32	−32	−26	−15	−2	13	17	19	6
50	−42	−43	−29	−2	17	23	22	6	2
40	−59	−50	−28	3	23	37	18	−1	−11
30	−34	−34	−28	7	29	43	20	4	−6
20	−59	−36	−11	21	39	49	39	22	10
10	−63	−24	12	53	60	58	46	36	26
0	−63	−9	33	58	73	74	63	50	32
−10	−63	−26	0	35	52	68	76	64	52
−20	−47	−45	−25	5	23	45	58	57	63
−30	−25	−37	−39	−23	−14	15	33	34	45
−40	−2	−20	−32	−33	−27	−14	−2	5	20
−50	13	−1	−15	−22	−22	−18	−15	−14	−10
−60	10	−2	−14	−23	−30	−33	−29	−35	−43
−70	10	−1	−16	−29	−36	−46	−55	−54	−59
−80	−6	−15	−24	−33	−40	−48	−50	−53	−52
−90	−30	−30	−30	−30	−30	−30	−30	−30	−30

Table C.3: Geoid84 in Meters (Longitude 180 (-180) to 260 (-100))

	Longitude								
Latitude	180	190	200	210	220	230	240	250	260
	−180	−170	−160	−150	−140	−130	−120	−110	−100
90	13	13	13	13	13	13	13	13	13
80	3	1	−2	−3	−3	−3	−1	3	1
70	2	2	1	−1	−3	−7	−14	−24	−27
60	2	9	17	10	13	1	−14	−30	−39
50	−8	8	8	1	−11	−19	−16	−18	−22
40	−12	−10	−13	−20	−31	−34	−21	−16	−26
30	−7	−5	−8	−15	−28	−40	−42	−29	−22
20	5	10	7	−7	−23	−39	−47	−34	−9
10	13	12	11	2	−11	−28	−38	−29	−10
0	22	16	17	13	1	−12	−23	−20	−14
−10	36	22	11	6	−1	−8	−10	−8	−11
−20	51	27	10	0	−9	−11	−5	−2	−3
−30	46	22	5	−2	−8	−13	−10	−7	−4
−40	21	6	1	−7	−12	−12	−12	−10	−7
−50	−15	−18	−18	−16	−17	−15	−10	−10	−8
−60	−45	−43	−37	−32	−30	−26	−23	−22	−16
−70	−61	−60	−61	−55	−49	−44	−38	−31	−25
−80	−53	−54	−55	−52	−48	−42	−38	−38	−29
−90	−30	−30	−30	−30	−30	−30	−30	−30	−30

Table C.4: Geoid84 in Meters (Longitude 270 (-90) to 350 (-10))

	Longitude								
Latitude	270	280	290	300	310	320	330	340	350
	−90	−80	−70	−60	−50	−40	−30	−20	−10
90	13	13	13	13	13	13	13	13	13
80	5	9	11	19	27	31	34	33	34
70	−25	−19	3	24	37	47	60	61	58
60	−46	−42	−21	6	29	49	65	60	57
50	−35	−40	−26	−12	24	45	63	62	59
40	−34	−33	−35	−26	2	33	59	52	51
30	−26	−32	−51	−40	−17	17	31	34	44
20	−10	−20	−45	−48	−32	−9	17	25	31
10	3	1	−11	−41	−42	−16	3	17	33
0	−3	14	10	−15	−27	−18	3	12	20
−10	−9	1	32	4	−18	−13	−9	4	14
−20	−1	9	35	20	−5	−6	−5	0	13
−30	1	9	32	16	4	−8	4	12	15
−40	−1	8	23	15	−2	−6	6	21	24
−50	−2	6	14	13	3	3	10	20	27
−60	−10	−2	10	20	20	21	24	22	17
−70	−16	−6	1	4	5	4	2	6	12
−80	−26	−26	−24	−23	−21	−19	−16	−12	−8
−90	−30	−30	−30	−30	−30	−30	−30	−30	−30

Appendix D

SOURCE CODE

D.1 xfit.h

```
#ifndef XFIT_H
#define XFIT_H
/* ----------------------------------------------------------------
FILE      = xfit.h
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.2.4, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (DRTOOLS)
PURPOSE   = Parabolic smoothing in the least squares sense.
----------------------------------------------------------------
VERSION HISTORY:
3 (14-Jul-1999) PPE, Added xfit1().
2 (29-Mar-1999) PPE, Implemented normalize_matrix().
1 (26-Mar-1999) PPE, Original
---------------------------------------------------------------- */


#include <libdr/lists.h>



struct position_record {
   double time;
   double x, y, z;
};


struct state_vector_record {
   double time;
   double x, y, z;
   double vx, vy, vz;
   double ax, ay, az;
};
```

```
/* ========================================================= */


/* -------------------------------------------------------------
xfit - fit a parabolic curve to a position vector [pos_list],
to create a vector of state vectors (RETURN).  The elements of
the returned state vectors will be evaluated at the times
contained in the [time_list] with locality [twindow].  The
time window [twindow] is applied in both directions about
each element of [time_list] to choose the elements of
[pos_list] that will be used in smoothing and getting the
time derivatives (velocity, and acceleration).
Elements corresponding to times not within a region that can
be processed will have the state values set to zero.
If at least two positions are available (one on each side of
the requested time) then the position and velocity will be
estimated, but not the acceleration.
------------------------------------------------------------- */
LIST * xfit(LIST *pos_list, LIST *time_list, double twindow);

NODE * xfit1(LIST *pos_list, NODE *hint, double time,
 double twindow,
 struct state_vector_record *state);


#endif
```

D.2 xfit.c

```
/* ---------------------------------------------------------------
FILE      = xfit.c
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.2.4, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (DRTOOLS)
PURPOSE   = Parabolic smoothing in the least squares sense.
---------------------------------------------------------------
VERSION HISTORY:
3.10 (20-Jul-2001) PPE,
    [1] Ran the code (and comments) through a spell checker.
    [2] Added debug output for results of solve_stats().
    [3] Changed the use of a matrix element (mat[2][3])
        to the corresponding coefficient (c2) in solve_stats().
3.9 (28-Nov-2000) PPE,
    [1] Changed all #ifdef DEBUG to #if DEBUG, since DEBUG is
        now always defined, but normally set to zero in the main
Makefile.
3.8 (30-May-2000) PPE,
    [1] Replaced all the BUGOUTs with #ifdef DEBUG fprintf() #endif.
3.7 (16-Jul-1999) PPE,
```

```
    [1] Reduce roundoff errors, by re-normalizing the time elements
        to be centered about zero, rather than a huge number of
seconds from midnight of January 1.  Thus the time window
across which we are evaluating the data points is about the
same size as the values themselves.
3.6 (15-Jul-1999) PPE,
    [1] Now xfit() passes back the hint to xfit1() to
        give a massive speed improvement.
3.5 (14-Jul-1999) PPE,
    [1] Added xfit1(), which simplifies the logic since
        we are only evaluating data at one point.
    [2] Now xfit() calls xfit1(), but without the hint.
2.4 (05-Apr-1999) PPE, If a target time is specified with a zero
                       Julian date, use the Julian day from the
                       position list to evaluate the parabola.
2.3 (01-Apr-1999) PPE,
      [1] Allow times with zero Julian days to work
          regardless of which day is in the data.
2.2 (29-Mar-1999) PPE, Implemented normalize_matrix().
1.1 (26-Mar-1999) PPE, Original
------------------------------------------------------------------ */


#include <stdio.h>
#include <math.h>

#if 0
#define DEBUG 1
#endif

#if DEBUG
#endif

#include <libdr/drbase.h>
#include <libdr/xfit.h>
#include <libdr/lists.h>
#include <libdr/interpolate.h>
#include <libdr/irig.h>


#define MIN_ELEMENTS 5

#ifndef NULL
#   define  NULL  ((void *) 0)
#endif

#ifndef SECONDS_IN_DAY
#   define SECONDS_IN_DAY   86400
#endif

struct statistics {
```

```
   int n;
   double sx, sx2, sx3, sx4;
   double sy, sxy, sx2y;
};


/* ========================================================== */

/* ------------------------------------------------------------
print_stats -
------------------------------------------------------------ */
void print_stats(FILE *f, struct statistics *stats, char *msg) {
   fprintf(f, "%s\n", msg);
   fprintf(f, "  n = %d\n", stats->n);
   fprintf(f, "  sx  = %f\n"
              "  sx2 = %f\n"
      "  sx3 = %f\n"
      "  sx4 = %f\n",
                 stats->sx, stats->sx2, stats->sx3, stats->sx4);
   fprintf(f, "  sy = %f\n"
              "  sxy = %f\n"
      "  sx2y = %f\n",
                 stats->sy, stats->sxy, stats->sx2y);
   fprintf(f, "\n");
}



/* ------------------------------------------------------------
zero_stats - initialize a statistics record so that we
can accumulate a new set of data into it.
------------------------------------------------------------ */
void zero_stats(struct statistics *st) {
   st->n = 0;
   st->sx = st->sx2 = st->sx3 = st->sx4 = 0.0;
   st->sy = st->sxy = st->sx2y = 0.0;
}


/* ------------------------------------------------------------
accumulate_stats -
------------------------------------------------------------ */
void accumulate_stats(struct statistics *st, double x, double y) {
   double z;

#if DEBUG
   fprintf(stderr, "accumulate_stats: t = %f, y = %f\n", x, y);
#endif

   st->n++;                      st->sy   += y;
   z = x;     st->sx  += z;      st->sxy  += z * y;
```

```
   z *= x;    st->sx2 += z;      st->sx2y += z * y;
   z *= x;    st->sx3 += z;
   z *= x;    st->sx4 += z;

#if DEBUG
   print_stats(stderr, st, "partial accumulation of stats");
#endif
}


/* ------------------------------------------------------------
normalize_matrix - Diagonalize the augmented matrix to solve
the system of equations.
------------------------------------------------------------
Use Gaussian elimination.
------------------------------------------------------------ */
void normalize_matrix(double mat[3][4]) {
   int k, j, i;
   double x;

   for (k=0; k<3 ; k++) {
      /* --- normalize row k (get a 1 on the diagonal --- */
      x = mat[k][k];
      for (j=0 ; j<4 ; j++) {
         mat[k][j] /= x;
      };
      /* ---
      * Zeroize the k-th column (except for [k,k])
      * by subtracting x*row[k] from row[j].
      --- */
      for (j=0 ; j<3 ; j++) {
         if (k != j) {
            x = mat[j][k];
            for (i=0 ; i<4 ; i++) {
               mat[j][i] -= x * mat[k][i];
            };
         };
      };
   };
}


/* ------------------------------------------------------------
print_matrix -
------------------------------------------------------------ */
void print_matrix(FILE *f, double mat[3][4]) {
   int k, j;

   for (k=0 ; k<3 ; k++) {
      for (j=0 ; j<4 ; j++) {
         fprintf(f, " %10.3f", mat[k][j]);
```

```
      };
      fprintf(f, "\n");
   };
   fprintf(f, "\n");
}


/* ------------------------------------------------------------
solve_stats - evaluate the statistics [st] at the value
of [t] to give the return value.  The statistics model a
parabola to a maneuvering target.
The solution consists of position [x], velocity [v], and
acceleration [a].
------------------------------------------------------------
The statistics form a set of simultaneous equations.
We form an equation by diagonalizing a augmented matrix using
Gaussian elimination.  The matrix is built from the statistics,
then diagonalized.  The coefficients of the parabola are
are extracted from the diagonalized matrix.  The position
velocity, and acceleration are then evaluated.
   x = c0 + c1t + c2t^2
   v = c1 + 2c2t
   a = 2c2
This equation is then solved by using the supplied value [x].
------------------------------------------------------------ */
void solve_stats(struct statistics *st, double t,
                        double *x, double *v, double *a) {
   double mat[3][4], c0, c1, c2;

   mat[0][0] = st->n;
   mat[0][1] = st->sx;
   mat[0][2] = st->sx2;
   mat[0][3] = st->sy;

   mat[1][0] = st->sx;
   mat[1][1] = st->sx2;
   mat[1][2] = st->sx3;
   mat[1][3] = st->sxy;

   mat[2][0] = st->sx2;
   mat[2][1] = st->sx3;
   mat[2][2] = st->sx4;
   mat[2][3] = st->sx2y;

#if DEBUG
   fprintf(stderr, "matrix loaded with statistics\n");
   print_matrix(stderr, mat);
#endif
   normalize_matrix(mat);
#if DEBUG
   fprintf(stderr, "normalized matrix\n");
```

```
   print_matrix(stderr, mat);
#endif

   c0 = mat[0][3];
   c1 = mat[1][3];
   c2 = mat[2][3];
   *x = c0 + c1*t + c2*t*t;
   *v =      c1   + 2*c2*t;
   *a =             2*c2;
#if DEBUG
   fprintf(stderr, "x = %10.2f, v = %10.3f, a = %10.3f\n", *x, *v, *a);
#endif
}


/* ------------------------------------------------------------
 * xfit1 - fit a parabolic curve to a specific time point in
 * the position vector, resulting in a single state_vector_record.
 * The points from the position vector list [pos_list] used in
 * calculating the parabola are within [twindow] seconds from
 * [time].  The first time this routine is used on a position
 * list, use a [hint] of NULL.  On subsequent invocations on
 * a position list (and with monotonically incrementing [time])
 * use the [hint] provided by the previous invocation.
 * This routine creates the state_vector_record, and its node,
 * these are then the responsibility of the caller to deallocate.
 * ------------------------------------------------------------
 * Reduce round off errors by using an offset time rather than
 * absolute time.
 *
 * If at least MIN_ELEMENTS (5) elements are found within
 * that window then a least squares fit with a parabolic curve
 * is performed within that window.  The resulting curve is
 * evaluated at the specified time to give the state vector
 * which includes (position, velocity, and acceleration).
 * NOTE: the specified time is twindow seconds from offset_time.
 *
 * If not enough elements are found within the time
 * window, but two elements are found (one on each side of the
 * requested time) then a simple linear interpolation is used
 * to get the position, and the difference between the two is
 * used as the velocity.  The acceleration is set to zero.
 *
 * If none of the above can be found, all elements are set to
 * zero (except for time of course).
 * ------------------------------------------------------------ */
NODE *xfit1(LIST *pos_list, NODE *hint, double time,
               double twindow, struct state_vector_record *state) {
   struct statistics st_x, st_y, st_z;
   NODE   *pos_nd, *left_nd;
   struct position_record *pos_rec, *left_pos, *right_pos;
```

```
   int k;
   int julian_day;
   double pos_time, left_time, right_time;
   double delta_time, offset_time;
   int count=0;

#if DEBUG
   fprintf(stderr, "xfit1: time is %f\n", time);
   fprintf(stderr, "xfit1: zeroing state\n");
#endif

   state->time = time;
   state->x  = state->y  = state->z  = 0.0;
   state->vx = state->vy = state->vz = 0.0;
   state->ax = state->ay = state->az = 0.0;
   zero_stats(&st_x);
   zero_stats(&st_y);
   zero_stats(&st_z);

   k = 0;
   if (hint) {
      pos_nd = hint;
   } else {
      pos_nd = get_first_node_from_list(pos_list);
   };
   left_nd = NULL;
   left_pos = NULL;

   /* -------------------------------------------------
    * Traverse the position list, and accumulate the positions
    * of the points within the time window around the
    * requested time.
    * ------------------------------------------------- */
   while (pos_nd) {
      k++;
#if DEBUG
      fprintf(stderr, "xfit1: position record %d\n", k);
#endif
      pos_rec = (struct position_record *) pos_nd->data;

#if DEBUG
      fprintf(stderr, "xfit1: position record time %15.6f\n",
      pos_rec->time);
#endif
      if (time < SECONDS_IN_DAY) {
         julian_day = pos_rec->time / SECONDS_IN_DAY;
         pos_time = pos_rec->time - julian_day * SECONDS_IN_DAY;
      } else {
         pos_time = pos_rec->time;
      };
      delta_time = pos_time - time;
```

```
#if DEBUG
      fprintf(stderr, "xfit1: pos_time %15.6f\n", pos_time);
      fprintf(stderr, "xfit1: delta time %15.6f\n", delta_time);
#endif
#if 1
      if ( fabs(delta_time) < twindow ) {
#else
   int flag1, flag2, flag3, flag4;
      flag1 = (pos_rec->time - twindow) < time;
      flag2 = time < (pos_rec->time + twindow);
      flag3 = (pos_time_noday-twindow) < time_noday;
      flag4 = time_noday < (pos_time_noday+twindow);
      if ( (flag1 && flag2) || (flag3 && flag4) ) {
#endif
         count++;
         /* --- this element is within the time window --- */
#if DEBUG
         fprintf(stderr, "xfit1: found position node %d in time window\n",
                     k);
         fprintf(stderr, "xfit1:   x = %15.2f\n", pos_rec->x);
         fprintf(stderr, "xfit1:   y = %15.2f\n", pos_rec->y);
         fprintf(stderr, "xfit1:   z = %15.2f\n", pos_rec->z);
#endif
         if (count == 1) {
            left_nd = pos_nd;
            left_pos = pos_rec;
    left_time = pos_time;
 };
 if ((pos_time > time) && (!right_pos) ) {
            right_pos = pos_rec;
            right_time = pos_time;
         };
 offset_time = pos_time - time;
         accumulate_stats(&st_x, offset_time, pos_rec->x);
         accumulate_stats(&st_y, offset_time, pos_rec->y);
         accumulate_stats(&st_z, offset_time, pos_rec->z);
      };
      pos_nd = get_next_node(pos_nd);
      if (delta_time > twindow) break;
   };

   /* --- save the state vector's time --- */
   if (time < SECONDS_IN_DAY) {
       state->time = time + julian_day * SECONDS_IN_DAY;
   } else {
       state->time = time;
   };

   /* ------------------------------------------------
   * Decide the amount of data to extract given the
   * number of elements found within the time window
```

```
   * ------------------------------------------- */
#if DEBUG
   fprintf(stderr, "xfit1: found %d records in time window\n", st_x.n);
#endif
   if (st_x.n >= MIN_ELEMENTS) {
      /* ---------------------------------------------
       * Use least squares fit
      * --------------------------------------------- */
#if DEBUG
      fprintf(stderr, "xfit1: good, enough for full parabola\n");
#endif
      solve_stats(&st_x, 0.0,
                  &state->x, &state->vx, &state->ax);
      solve_stats(&st_y, 0.0,
                  &state->y, &state->vy, &state->ay);
      solve_stats(&st_z, 0.0,
                  &state->z, &state->vz, &state->az);
   } else if ((left_time < time) && (time < right_time)) {
      /* ---------------------------------------------
       * Use linear interpolation
      * --------------------------------------------- */
#if DEBUG
      fprintf(stderr, "xfit1: okay, linear interpolation\n");
#endif
      state->x = interpolate(0.0, left_time, right_time,
                             left_pos->x, right_pos->x);
      state->vx = (right_pos->x - left_pos->x)
           / (right_time - left_time);
      state->ax = 0.0;
      state->y = interpolate(0.0, left_time, right_time,
                             left_pos->y, right_pos->y);
      state->vy = (right_pos->y - left_pos->y)
           / (right_time - left_time);
      state->ay = 0.0;
      state->z = interpolate(0.0, left_time, right_time,
                             left_pos->z, right_pos->z);
      state->vz = (right_pos->z - left_pos->z)
           / (right_time - left_time);
      state->az = 0.0;
   } else {
#if DEBUG
      fprintf(stderr, "xfit1: not enough -> zero\n");
#endif
      /* -- leave the state vector nulled out. --- */
   };
   return left_nd;
}


/* ------------------------------------------------------------
xfit - fit a parabolic curve to a position vector [pos_list],
```

```
to create a vector of state vectors (RETURN).  The elements of
the returned state vectors will be evaluated at the times
contained in the [time_list] with locality [twindow].  The
time window [twindow] is applied in both directions about
each element of [time_list] to choose the elements of
[pos_list] that will be used in smoothing and getting the
time derivatives (velocity, and acceleration).
Elements corresponding to times not within a region that can
be processed will have the state values set to zero.
If at least two positions are available (one on each side of
the requested time) then the position and velocity will be
estimated, but not the acceleration.
-------------------------------------------------------------
Every element of [time_list] is evaluated to create a
state_vector_record.
A sliding window of regard is used with a window size of
[twindow] (nominally 1 second).

------------------------------------------------------------ */
LIST * xfit(LIST *pos_list, LIST *time_list, double twindow) {
   LIST   *state_list;
   NODE   *time_nd;
   NODE   *state_nd;
   NODE   *hint=NULL;
   double time;
   struct state_vector_record state;
   int j=0;
#if DEBUG
   char time_buffer[32]
#endif

#if DEBUG
   fprintf(stderr, "xfit: routine entered\n");
   fprintf(stderr, "xfit: creating new list\n");
#endif

   state_list = new_list();
   time_nd = get_first_node_from_list(time_list);
   while (time_nd) {
      j++;
#if DEBUG
      fprintf(stderr, "xfit: examining time element %d\n", j);
#endif
      time = *((double *) time_nd->data);
#if DEBUG
      fprintf(stderr, "xfit: time is %f (%s)\n", time,
      time2str(time_buffer, time));
#endif
#if 1
      hint = xfit1(pos_list, hint, time, twindow, &state);
#else
```

```
      hint = xfit1(pos_list, NULL, time, twindow, &state);
#endif
      state_nd = new_node();
      copy_data_to_node(state_nd, &state, sizeof(state));
      append_node_to_list(state_list, state_nd);
      time_nd = get_next_node(time_nd);
   };

#if DEBUG
   fprintf(stderr, "xfit: routine exiting\n");
#endif
   return state_list;
}
```

D.3 xform.h

```
#ifndef XFORM_H
#define XFORM_H
/* ----------------------------------------------------------------
FILE      = xform.h
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.0.35, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (HPM-TOOLS)
PURPOSE   = Coordinate transformations (WGS-84).
----------------------------------------------------------------
VERSION HISTORY:
4 (25-Feb-1999) PPE,
         [1] Corrected comment for ned2efg().  It is a simple
             rotation to the vector, the origins vector is not
             applied to the result.
         [2] Added cdu2ned().
3 (23-Sep-1998) Added DATUM_NAD27
CWT (23-Sep-1998)
2 (23-Sep-1998) PPE,
      [1] Allow different geodetic constants to be used.
      [2] Comment the routines, in particular the units.
1 (31-Aug-1998) PPE, Original
---------------------------------------------------------------- */


/* -------- geodetic datum which can be used --------- */
#define DATUM_WGS84   0
#define DATUM_HAYFORD 1
#define DATUM_NAD27   2


/* ----------------------------------------------------------------
set_datum - Set the geodetic constants to the specified [datum].
---------------------------------------------------------------- */
```

```
int set_datum(int datum);


/* ---------------------------------------------------------------
rae2ned - transform the coordinates [rng, azm, elev] (degrees)
to [north, east, down].
--------------------------------------------------------------- */
void rae2ned(double rng, double azm, double elev,
                double *north, double *east, double *down);


/* ---------------------------------------------------------------
ned2rae - transform the coordinates [north, east, down] to [rng,
azm, elev] (degrees).
--------------------------------------------------------------- */
void ned2rae( double north, double east, double down,
                double *rng, double *azm, double *elev);



/* ---------------------------------------------------------------
cdu2ned - Rotate a local vector in CDU into NED.  The [rot_angle] is
the angle from true north to the flight line on which the cross and
down range vectors are based.  The up component is simply converted
to a down component.

This same routine can be used as its own inverse, just negate the
rotation angle.
--------------------------------------------------------------- */
void cdu2ned(double rot_angle,
                double cross, double downrange, double up,
                double *north, double *east, double *down);



/* ---------------------------------------------------------------
llh2efg - transform the coordinate from [latitude, longitude, height]
(degrees, feet) (where the height is referenced to the spheroid) to
geocentric coordinates [e, f, g] (feet).
--------------------------------------------------------------- */
void llh2efg(double latitude, double longitude, double height,
                double *e, double *f, double *g);


/* ---------------------------------------------------------------
efg2llh - transform the geocentric coordinates [e, f, g] (feet) to
geodetic coordinates [latitude, longitude, height] (degrees, feet).
--------------------------------------------------------------- */
void efg2llh(double e, double f, double g,
               double *latitude, double *longitude, double *height);
```

```
/* ---------------------------------------------------------------
ned2efg - Rotate a vector in local coordinates [north, east, down]
(feet) which has an origin at [org_lat, org_long, org_ht] to a vector
aligned to geocentric axes [e, f, g] (feet).
--------------------------------------------------------------- */
void ned2efg(double north, double east, double down,
             double org_lat, double org_long, double org_ht,
             double *e, double *f, double *g);


/* ---------------------------------------------------------------
efg2ned - transform a geocentric coordinate [e, f, g] (feet) into
a local coordinate system [north, east, down] (feet) which has
an origin (org_lat, org_long, org_ht).
--------------------------------------------------------------- */
void efg2ned(double e, double f, double g,
             double org_lat, double org_long, double org_ht,
             double *north, double *east, double *down);

#endif
```

D.4 xform.c

```
/* ---------------------------------------------------------------
FILE      = xform.c
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.0.35, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (HPM-TOOLS)
PURPOSE   = Coordinate transformations (WGS-84).
---------------------------------------------------------------
VERSION HISTORY:
2.7 (01-Mar-1999) PPE,
      [1] Correct ned2efg(), change the sign of the middle matrix
          element.
2.6 (25-Feb-1999) PPE,
      [1] Correct comment about ned2efg(); it is a simple rotation,
          it does not apply the origins vector.
2.5 (23-Sep-1998) PPE,
      [1] Include xform.h, remove datum defines, use the ones from
          the header file.
      [2] Added NAD27 (Clarke-1886) datum.
CWT (23-Sep-1998)
2.4 (23-Sep-1998) PPE,
      [1] Allow different geodetic constants to be used.
      [2] Comment the routines, in particular the units.
1.3 (22-Sep-1998) PPE, We require some conversions r2d, d2r, ....
1.2 (15-Sep-1998) PPE, efg2llh() needed an atan() for the latitude.
1.1 (31-Aug-1998) PPE, Original
```

```
---------------------------------------------------------------- */

#include <math.h>
#include <stdio.h>

#include <libdr/conversions.h>
#include <libtspi/xform.h>


/* ------ geodetic constants for various spheroids ------- */
#define WGS84_MAJOR_AXIS_METERS    6378137
#define WGS84_MINOR_AXIS_METERS    6356752.3142
#define WGS84_ECCENT2              0.00669437999013

#if 0
#define HAYFORD_MAJOR_AXIS_METERS  6378388
#define HAYFORD_MINOR_AXIS_METERS  6356912
#define HAYFORD_ECCENT2            0.00672265318716
#endif

#define HAYFORD_MAJOR_AXIS_METERS  6378249.145
#define HAYFORD_MINOR_AXIS_METERS  6356514.870
#define HAYFORD_ECCENT2            0.0068035111

/* ------ North American Datum (Clarke 1886) ----- */
#define NAD27_MAJOR_AXIS_METERS  6378206.4
#define NAD27_MINOR_AXIS_METERS  6356583.8
#define NAD27_ECCENT2            0.0067686579978

/* -------- global variables --------- */
static double aa = WGS84_MAJOR_AXIS_METERS * METERS2FEET;
static double bb = WGS84_MINOR_AXIS_METERS * METERS2FEET;
static double ee = WGS84_ECCENT2;


/* ================================================================ */


/* ----------------------------------------------------------------
set_datum - Set the geodetic constants to the specified [datum].
Return: 1 when successful, 0 when failed.
---------------------------------------------------------------- */
int set_datum(int datum) {
   switch (datum) {
      case DATUM_WGS84:
         aa = WGS84_MAJOR_AXIS_METERS * METERS2FEET;
         bb = WGS84_MINOR_AXIS_METERS * METERS2FEET;
         ee = WGS84_ECCENT2;
         break;
      case DATUM_HAYFORD:
         aa = HAYFORD_MAJOR_AXIS_METERS * METERS2FEET;
```

```
         bb = HAYFORD_MINOR_AXIS_METERS * METERS2FEET;
         ee = HAYFORD_ECCENT2;
         break;
      case DATUM_NAD27:
         aa = NAD27_MAJOR_AXIS_METERS * METERS2FEET;
         bb = NAD27_MINOR_AXIS_METERS * METERS2FEET;
         ee = NAD27_ECCENT2;
         break;
      default:
         fprintf(stderr, "unknown datum\n");
         return 0;
   };
   return 1;
}


/* ----------------------------------------------------------------
rae2ned - transform the coordinates [rng, azm, elev] (degrees)
to [north, east, down].
---------------------------------------------------------------- */
void rae2ned(double rng, double azm, double elev,
               double *north, double *east, double *down) {
   double a = azm * DEGREES2RADIANS;
   double e = elev * DEGREES2RADIANS;
   double saz = sin(a);
   double caz = cos(a);
   double sel = sin(e);
   double cel = cos(e);

   *north = rng * cel * caz;
   *east  = rng * cel * saz;
   *down  = - rng * sel;
}


/* ----------------------------------------------------------------
ned2rae - transform the coordinates [north, east, down] to [rng,
azm, elev] (degrees).
---------------------------------------------------------------- */
void ned2rae( double north, double east, double down,
               double *rng, double *azm, double *elev) {
   double a, e;

   *rng  = sqrt(north*north + east*east + down*down);
   a  = atan2(east, north);
   *azm  = a * RADIANS2DEGREES;
   e = asin(-down / *rng);
   *elev = e * RADIANS2DEGREES;
}
```

```
/* ---------------------------------------------------------------
cdu2ned - Rotate a local vector in CDU into NED.  The [rot_angle] is
the angle from true north to the flight line on which the cross and
down range vectors are based.  The up component is simply converted
to a down component.

This same routine can be used as its own inverse, just negate the
rotation angle.
--------------------------------------------------------------- */
void cdu2ned(double rot_angle,
                double cross, double downrange, double up,
                double *north, double *east, double *down) {
   double sx, cx;

   if (rot_angle == 0.0) {
       *north = downrange;
       *east  = cross;
   } else {
       sx = sin(rot_angle * DEGREES2RADIANS);
       cx = cos(rot_angle * DEGREES2RADIANS);
       *north =  cx * downrange + sx * cross;
       *east  = -sx * downrange + cx * cross;
   };
       *down  = -up;
}


/* ---------------------------------------------------------------
PRIVATE radius_primary_vertical - calculate the radius of curvature
of the earth at the specified latitude (radians).
--------------------------------------------------------------- */
double radius_primary_vertical(double latitude) {
   double x, x2, denom;

   x = sin(latitude);
   x2 = 1.0 - ee * x * x;
   denom = sqrt(x2);
   return (aa / denom);
}


/* ---------------------------------------------------------------
llh2efg - transform the coordinate from [latitude, longitude, height]
(degrees, feet) (where the height is referenced to the spheroid) to
geocentric coordinates [e, f, g] (feet).
--------------------------------------------------------------- */
void llh2efg(double latitude, double longitude, double height,
                double *e, double *f, double *g) {

   double lat = latitude * DEGREES2RADIANS;
```

```
   double lon = longitude * DEGREES2RADIANS;
   double N = radius_primary_vertical(lat);
   double cphi = cos(lat);
   double sphi = sin(lat);
   double clamb = cos(lon);
   double slamb = sin(lon);

   *e = (N + height) * cphi * clamb;
   *f = (N + height) * cphi * slamb;
   *g = (N*(1.0-ee)+height) * sphi;
}



/* ----------------------------------------------------------------
efg2llh - transform the geocentric coordinates [e, f, g] (feet) to
geodetic coordinates [latitude, longitude, height] (degrees, feet).
---------------------------------------------------------------- */
void efg2llh(double e, double f, double g,
               double *latitude, double *longitude, double *height) {
   double x;
   double alpha;
   double sa;
   double ca;
   double lat, lon;
   double numer, denom, N;

   x = sqrt(e*e + f*f);
   alpha = atan(g*aa / (x*bb));
   sa = sin(alpha);
   ca = cos(alpha);

   lon = atan2(f, e);
   *longitude = lon * RADIANS2DEGREES;

   numer = g + bb*(ee/(1-ee))*sa*sa*sa;
   denom = x - aa*ee*ca*ca*ca;
   lat = atan(numer / denom);
   *latitude = lat * RADIANS2DEGREES;

   N = radius_primary_vertical(lat);
   sa = sin(lat);
   ca = cos(lat);
   *height = x*ca + g*sa - N*(1-ee*sa*sa);
}


/* ----------------------------------------------------------------
ned2efg - Rotate a vector in local coordinates [north, east, down]
(feet) which has an origin at [org_lat, org_long, org_ht] to a vector
aligned to geocentric axes [e, f, g] (feet).
```

```
------------------------------------------------------------------ */
void ned2efg(double north, double east, double down,
             double org_lat, double org_long, double org_ht,
             double *e, double *f, double *g) {
   double sphi = sin(org_lat * DEGREES2RADIANS);
   double cphi = cos(org_lat * DEGREES2RADIANS);
   double slamb = sin(org_long * DEGREES2RADIANS);
   double clamb = cos(org_long * DEGREES2RADIANS);

   *e = -sphi*clamb*north  -slamb*east  -cphi*clamb*down;
   *f = -sphi*slamb*north  +clamb*east  -cphi*slamb*down;
   *g =  cphi*north        +0           -sphi*down;

}



/* ----------------------------------------------------------------
efg2ned - transform a geocentric coordinate [e, f, g] (feet) into
a local coordinate system [north, east, down] (feet) which has
an origin (org_lat, org_long, org_ht).
------------------------------------------------------------------ */
void efg2ned(double e, double f, double g,
             double org_lat, double org_long, double org_ht,
             double *north, double *east, double *down) {
   double sphi = sin(org_lat * DEGREES2RADIANS);
   double cphi = cos(org_lat * DEGREES2RADIANS);
   double slamb = sin(org_long * DEGREES2RADIANS);
   double clamb = cos(org_long * DEGREES2RADIANS);

   *north = -sphi*clamb*e  -sphi*slamb*f  +cphi*g;
   *east  = -slamb*e       +clamb*f       +0;
   *down  = -cphi*clamb*e  -cphi*slamb*f  -sphi*g;
}



/* ============================================================
The following routines show some examples of using the above
primitive coordinate transforms.
============================================================ */



/* ----------------------------------------------------------------
ned2ned - Take a vector [n1, e1, d1] in one local coordinate system
[lat1, lon1, ht1] and rotate it to a different local coordinate
system [lat2, lon2, ht2] to give a vector [n2, e2, d2].
------------------------------------------------------------------ */
void ned2ned( double n1, double e1, double d1,
              double lat1, double lon1, double ht1,
```

```
              double lat2, double lon2, double ht2,
              double *n2, double *e2, double *d2) {
   double e, f, g;

   ned2efg(n1, e1, d1, lat1, lon1, ht1, &e, &f, &g);
   efg2ned(e, f, g, lat2, lon2, ht2, n2, e2, d2);
}


/* ---------------------------------------------------------------
orgllh_ned_llh - Given a NED vector from a specified origin determine
the latitude, longitude, and spheroidal height of the tip of the
vector.
---------------------------------------------------------------
NOTE: If the origin is constant you should get org_e, org_f, and
org_ht outside and use orgefg_ned_llh() for performance reasons.
--------------------------------------------------------------- */
void orgllh_ned_llh(double org_lat, double org_long, double org_ht,
                 double north, double east, double down,
                 double *lat, double *lon, double *ht) {
   double org_e, org_f, org_g;
   double e, f, g;

   llh2efg(org_lat, org_long, org_ht, &org_e, &org_f, &org_g);
   ned2efg(north, east, down, org_lat, org_long, org_ht, &e, &f, &g);
   efg2llh(org_e+e, org_f+f, org_g+g, lat, lon, ht);
}
```

D.5 geoid.h

```
#ifndef GEOID_H
#define GEOID_H
/* ---------------------------------------------------------------
FILE      = geoid.h
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.0.35, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (HPM-TOOLS)
PURPOSE   = .
---------------------------------------------------------------
VERSION HISTORY:
1.1 (31-Aug-1998) PPE, Original
--------------------------------------------------------------- */


/* ---------------------------------------------------------
--------------------------------------------------------- */
double get_geoid_value(double latitude, double longitude);
```

```
#endif
```

D.6 geoid.c

```
/* ----------------------------------------------------------------
FILE      = geoid.c
AUTHOR    = Peter P. Eiserloh
SYSTEM    = Linux 2.0.35, libc-6.0.7 (glibc-2.0.7)
COMPILER  = GCC-2.7.2
PROJECT   = AV-8B DATA REDUCTION TOOLS (HPM-TOOLS)
PURPOSE   = .
----------------------------------------------------------------
VERSION HISTORY:
1.1 (31-Aug-1998) PPE, Original
---------------------------------------------------------------- */

#include <math.h>
#include <stdio.h>

#include <libtspi/geoid.h>


#define METERS2FEET   3.28084

/* ------------------------------------------------
The geoid table gives the separation distance of the
WGS-84 spheroid from the geoid (MSL height) in units
of meters.
Each row corresponds to the longitude (0, 10, 20, ..., 170),
and each column to the latitude (90, 80, 70, ..., -90).
------------------------------------------------ */
double geoid_table[36][19] = {
   {13,33,51,47,47,52,36,31,22, 18, 12,17,22,18,25,16,16,-4,-30},
   {13,34,43,41,48,48,28,26,23, 12, 13,23,27,26,26,19,16,-1,-30},
   {13,28,29,21,42,35,29,15, 2,-13, -2,21,34,31,34,25,17, 1,-30},
   {13,23,20,18,28,40,17, 6,-3, -9,-14, 8,29,33,39,30,21, 4,-30},
   {13,17,12,14,12,33,12, 1,-7,-28,-25,-9,14,39,45,35,20, 4,-30},
   {13,13,  5,  7,-10, -9,-20,-29,-36, -49,-32,-10,15,41,45,35,26,6,-30},
   {13, 9, -2, -3,-19,-28,-15,-44,-59, -62,-38,-11,15,30,38,33,26,5,-30},
   {13, 4,-10,-22,-33,-39,-40,-61,-90, -89,-60,-20, 7,24,39,30,22,4,-30},
   {13, 4,-14,-29,-43,-48,-33,-67,-95,-102,-75,-40,-9,13,28,27,16,2,-30},

   /* -- 90 -- */
   {13, 1,-12,-32,-42,-59,-34,-59,-63,-63,-63,-47,-25,-2,13,10,10,-6,-30},
   {13,-2,-10,-32,-43,-50,-34,-36,-24,-9,-26,-45,-37,-20,-1,-2,-1,-15,-30},
   {13,-2,-14,-26,-29,-28,-28,-11,12,33,0,-25,-39,-32,-15,-14,-16,-24,-30},
   {13, 0,-12,-15, -2,  3, 7,21,53,58,35, 5,-23,-33,-22,-23,-29,-33,-30},
   {13, 2, -6, -2, 17, 23,29,39,60,73,52,23,-14,-27,-22,-30,-36,-40,-30},
   {13, 3, -2, 13, 23, 37,43,49,58,74,68,45, 15,-14,-18,-33,-46,-48,-30},
```

```
   {13, 2,  3, 17, 22, 18,20,39,46,63,76,58, 33,-2,-15,-29,-55,-50,-30},
   {13, 1,  6, 19,  6, -1, 4,22,36,50,64,57, 34, 5,-14,-35,-54,-53,-30},
   {13, 1,  4,  6,  2,-11,-6,10,26,32,52,63, 45,20,-10,-43,-59,-52,-30},

   /* -- 180 -- */
   {13,3,2,2,-8,-12,-7,5,13,22,36,51,46,21,-15,-45,-61,-53,-30},
   {13,1,2,9,8,-10,-5,10,12,16,22,27,22,6,-18,-43,-60,-54,-30},
   {13,-2,1,17,8,-13,-8,7,11,17,11,10,5,1,-18,-37,-61,-55,-30},
   {13,-3,-1,10,1,-20,-15,-7,2,13,6,0,-2,-7,-16,-32,-55,-52,-30},
   {13,-3,-3,13,-11,-31,-28,-23,-11,1,-1,-9,-8,-12,-17,-30,-49,-48,-30},
   {13,-3,-7,1,-19,-34,-40,-39,-28,-12,-8,-11,-13,-12,-15,-26,-44,-42,-30},
   {13,-1,-14,-14,-16,-21,-42,-47,-38,-23,-10,-5,-10,-12,-10,-23,-38,-38,-30},
   {13,3,-24,-30,-18,-16,-29,-34,-29,-20,-8,-2,-7,-10,-10,-22,-31,-38,-30},
   {13,1,-27,-39,-22,-26,-22,-9,-10,-14,-11,-3,-4,-7,-8,-16,-25,-29,-30},

   /* -- 270 -- */
   {13, 5,-25,-46,-35,-34,-26,-10,  3,-3,-9,-1, 1,-1,-2,-10,-16,-26,-30},
   {13, 9,-19,-42,-40,-33,-32,-20,  1,14, 1, 9, 9, 8, 6, -2, -6,-26,-30},
   {13,11,  3,-21,-26,-35,-51,-45,-11,10,32,35,32,23,14, 10,  1,-24,-30},
   {13,19, 24,  6,-12,-26,-40,-48,-41,-15,4,20,16,15,13, 20,  4,-23,-30},
   {13,27, 37, 29, 24,  2,-17,-32,-42,-27,-18,-5, 4,-2, 3,20, 5,-21,-30},
   {13,31, 47, 49, 45, 33, 17, -9,-16,-18,-13,-6,-8,-6, 3,21, 4,-19,-30},
   {13,34, 60, 65, 63, 59, 31, 17,  3,  3, -9,-5, 4, 6,10,24, 2,-16,-30},
   {13,33, 61, 60, 62, 52, 34, 25, 17, 12,  4, 0,12,21,20,22,6,-12,-30},
   {13,34, 58, 57, 59, 51, 44, 31, 33, 20, 14,13,15,24,27,17,12,-8,-30}
};



/* ------------------------------------------------------------
------------------------------------------------------------ */
double get_geoid_value(double latitude, double longitude) {
   double lat, lat1, lat2;
   double lon, lon1, lon2;
   int   row, col;
   double a0, a1, a2, a3;
   double n1, n2, n3, n4;
   double x, y, z;

   /* -- get lat/lon values of a box around the actual lat/long -- */
   lat = 90 - latitude;   /* values now increase when going south */
   lat1 = 10.0 * ceil(lat / 10.0);   /* bottom of box */
   lat2 = 10.0 * floor(lat / 10.0);  /* top of box */
   lon = longitude;
   if (lon < 0.0) {
      lon = 360 + lon;
   };
   lon1 = 10.0 * floor(lon / 10.0);   /* left edge of box */
   lon2 = 10.0 * ceil(lon / 10.0);    /* right edge of box */

#if 0
```

```
   printf("lat = %7.2f, lon = %7.2f\n", lat, lon);
   printf("lat1 = %7.2f, lon1 = %7.2f\n", lat1, lon1);
   printf("lat2 = %7.2f, lon2 = %7.2f\n", lat2, lon2);
#endif

   /* -- get geoid values in a box surrounding the actual lat/long -- */
   row = lon1 / 10;
   col = lat1 / 10;
   n1 = geoid_table[row][col];
#if 0
   printf("row = %d, col = %d, n1 = %5.1f\n", row, col, n1);
#endif

   row = lon2 / 10;
   col = lat1 / 10;
   n2 = geoid_table[row][col];
#if 0
   printf("row = %d, col = %d, n2 = %5.1f\n", row, col, n2);
#endif

   row = lon2 / 10;
   col = lat2 / 10;
   n3 = geoid_table[row][col];
#if 0
   printf("row = %d, col = %d, n3 = %5.1f\n", row, col, n3);
#endif

   row = lon1 / 10;
   col = lat2 / 10;
   n4 = geoid_table[row][col];
#if 0
   printf("row = %d, col = %d, n4 = %5.1f\n", row, col, n4);
#endif

   /* --- setup bi-linear interpolation coefficients --- */
   a0 = n1;
   a1 = n2 - n1;
   a2 = n4 - n1;
   a3 = n1 + n3 - n2 - n4;
   x = (lon - lon1) / (lon2 - lon1);
   y = (lat - lat1) / (lat2 - lat1);
#if 0
   printf("a0 = %f, a1 = %f, a2 = %f, a3 = %f\n", a0, a1, a2, a3);
   printf("x = %7.4f, y= %7.4f\n", x, y);
#endif

   /* --- evaluate bin-linear equation, and convert to feet --- */
   z = (a0 + a1*x + a2*y + a3*x*y);
#if 0
   printf("z = %f\n", z);
#endif
```

```
   return METERS2FEET * z;
}
```

GLOSSARY

Alrite A laser sensor operated by the range Dept., which can track the aircraft under test.

Altitude Height above the reference (mean sea level, or spheroidal)

ARDS Advanced Range Data System: A pod, looking like a sidewinder missile, containing a GPS receiver that mounts onto a wing station, and which collects GPS data to be used as TSPI data.

CDU Cross Range, Down Range, Up coordinate system. Similar to ENU, but rotated about a flight line.

China Lake The Naval Air Warfare Center - Weapons Division, China Lake. This is one of the US Navy's RD&TE laboratories for airborne weapons development.

CTDF Common Test Data Format (previously Common TSPI Data Format).

Datum A reference surface or line used in surveying (i.e., WGS-84, or Clarke 1880 spheroid).

DOP Dilution of Precision

Echo Range (See EWTES.)

ECEF Earth Centered Earth Fixed Coordinate System.

EFG The names (E, F, and G) of the axes of the ECEF coordinate system.

Elevation The angle up from the local horizontal plane. (contrast this with altitude).

Ellipsoid A geometric shape whose plane sections are all ellipses or circles. In the general case an ellipsoid has three axes (a, b, and c).

ENU East, North, Up local Cartesian coordinate system.

EWTES Electronic Warfare Threat Environment Simulation (was Echo Range). This is the EW test range in South portion of China Lake.

FLIR Forward Looking Infra-red Sensor

FRD Forward, Right, Down aircraft body coordinate system.

Geocentric An ECEF coordinate system defined by the axes E, F, and G.

Geodesy The science of the shape and size of the earth.

Geodetic A coordinate system defined by the surface of the earth using latitude, longitude, and height, based upon a chosen datum.

Geoid The gravitational equipotential surface at zero feet MSL, and which is perpendicular to the local gravity field.

GPS Global Positioning System.

ITDF Intermediate TSPI Data Format

IRIG Inter Range Instrumentation Group

INS Inertial Navigation Set, also “Inserted”.

Latitude The angle between the gravitational down vector and the equatorial plane. (i.e., North/South distance).

LLH Latitude, Longitude, Height (geodetic coordinates).

Longitude The angle from the prime meridian at Greenwich England towards the East (i.e., East/West distance).

LOQ Level of Quality

LSF Least Squares Fit.

MAGR Miniaturized Airborne GPS Receiver

NED North, East, Down local Cartesian coordinate system

NIKE The NIKE Radar is a ground based radar set from the Vietnam war era. Commonly used today as a source of TSPI data. It usually has a TV camera mounted on the same pedestal as the radar disk, to give an Electro-Optical track, and verification.

Oblate The shape of a spheroid when the axis of rotation is the semi-minor axis ‘b’. Thus an oblate spheroid is like a squashed ball.

Orthogonal Perpendicular. A system of independent equations.

Orthometric Height Height above mean sea level, in the direction normal to the local gravitational equipotential surface. (Contrast this with Spheroidal Height)

Prolate The shape of a spheroid where the axis of rotation is the semi-major axis ‘a’. Thus a prolate spheroid looks like a watermelon.

RAE Range, Azimuth, Elevation coordinate system

RCC Range Command Center, also Range Command Council

RIPS Range Instrumentation Processing System

Spheroid The shape generated by rotating an ellipse about one of its axes. A spheroid can be characterized as either oblate, or prolate.

Spheroidal Height Height above the spheroid, in the direction normal to the spheroid.

Transformation A mathematical process of changing a vector described in on coordinate system into a different coordinate system.

TAER Time Azimuth Elevation Range

TSPI Time Space Position Information

WGS World Geodetic System

Bibliography

[BOWRING-1976] "TRANSFORMATIONS FROM SPATIAL TO GEOGRAPHICAL COORDINATES", B. R Bowring, Survey Review XXIII, 181, July 1976, pg. 323-327

[Cullen-72] "Matrices and Linear Transformations" second edition, Charles G. Cullen, University of Pittsburgh. Addison Wesley Publishing Company Inc. Reading Massachusetts ISBN: 0-201-01209-X

[DMA-1987] "Distribution of the GPS UE Relevant WGS 84 DATA Base Package" dtd 12 JAN 87. HQ USAF Space Division, SD/CWNE, Michael J. Ellett, DMA Representative, Directorate of Systems Engineering, NAVSTAR Global Positioning System.

[HEARN-1986] "Computer Graphics", Donald Hearn, and M. Pauline Baker. Prentice Hall, Englewood Cliffs, New Jersey 07632. ISBN: 0-13-165382-2

[JAMES-1990] "Baseline Software Program Descriptions" dated January 24, 1990, revised Feb 19, 1990, by Robert James of GMD Systems, for Contract N60530-89-0083 for Naval Weapons Center, China Lake.

[LAMPREACH-1996] "Definition and Incorporation of the Common TSPI Data Format (CTDF)" MEMO dated 05/22/96. Jon A. Lampreach (MDA-NAWC) (760) 446-3507

[LANGLEY] "Basic Geodesy for GPS", Richard B. Langley, GPS World.

[MILBERT-XX] "GPS and Geoid90 - The New Leveling Rod", by Dennis Milbert (National Geodetic Survey), GPS World, February 1992

[RCC-1985] "Global Coordinate System", Range Commanders Council, Data Reduction and Computer Group. Document 151-85 (Revised June 1989)

[RIPS-1985] "RIPS DATA PRODUCTS TAPE SPECIFICATION" dtd 12-Mar-85. Floyd Hall: Code 6251A Range Dept, Test and Evaluation Directorate, Naval Weapons Center CA.

COLOPHON

This document was originally written on a Macintosh with Microsoft Word, then ported to Microsoft Windows, again with Microsoft Word. Both the equations and graphics had to be redone during this 'simple' re-hosting.

When my main OS transitioned from MS-Windows in its various incarnations to Linux, I could no longer use MS-Word. I had to either keep a copy of MS-Windows and MS-Word, freeze the document, or port the document over to some other documentation system. The latter was chosen.

There were a number of options available. Choose a word processor, SGML, HTML, TEX, LATEX, or other mark-up format.

One of many problems with using a word processor, was determining the changes between revisions. If this were source code (straight text) the program '*diff*' could be run against the document, and its previous version to generate a complete list of changes.

Another big problem with word processors, especially the proprietary ones such as Word Perfect, was long term viability. I wanted a system that could build the document twenty or more years from now. This was most easily satisfied by using the open source software readily available on the Internet.

The SGML document looked to be the best match. It not only could be processed to generate Postscript (tm), but also LATEX, text, or even HTML.

An initial port was done, but it was disappointing due to problems with the style sheets. Orphan control was very poor. The headings of paragraphs would in many cases end on a page, with the body starting the next one. Also the method of equation handling was difficult. The equations were written in TEX and converted to Encapsulated Postscript (EPS) files, one for each equation. The figures were similarly handled (they were built with *xfig*), and saved as EPS files.

TEX, and LATEXwere mature, open source, and came with all Linux distributions. It was available under most operating systems including other versions of Unix, Mac-OS, MS-Windows, MS-DOS, IBM's OS/390, Amiga-DOS, and many others. It provided a consistent output regardless of the particular OS on which it was currently hosted.

LATEX was a macro language on top of TEX, supplying a structuring for the document. The document files were text, and could be edited with a simple text editor, *diff* works on these, as does document configuration software such as *CVS*.

Version 0.6.4 of the document was re-hosted using LATEX 2_ε files. At first, the computer modern fonts were used, these were the default fonts with both TEX, and LATEX. These looked okay when the output was only Postscript, but when converted to a Portable Document File (PDF) format it looked horrible, so I changed the font to the standard Postscript font 'Times'. This was accomplished by simply using the package 'times'.

The equations are written in the main text. The figures are created with *xfig* and exported as EPS files. These EPS files are then referenced at the appropriate places by the LATEX files. Each chapter, appendix, or other entity has its own LATEX file. The source code in Appendix D, was imported by simply nesting the source code within a *verbatim* environment.

The entire system is managed via a Makefile. Executing *make* against a target within the Makefile provides the method of interacting with the system. The commands used to build the document to the output formats are stored within the Makefile.

Index