Freescale Semiconductor
Application Note

Document Number:AN4684
Rev. 1.0, 4/2013

Magnetic Calibration of Hard and Soft Iron Interference

by: Mark Pedley

1 Introduction

This Application Note documents the magnetic calibration functions in Freescale's Xtrinsic eCompass and Magnetic Calibration software provided under license at www.freescale.com/ecompass. This Application Note is part of the technical documentation for that software and its use and distribution are controlled by the license agreement.

Contents

1.1 Mathematical glossary

The following are used in this application note:

$\boldsymbol{A}$	Soft iron ellipsoid fit where $\boldsymbol{A} = \{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1}$
B	Geomagnetic field strength
$\boldsymbol{B}_c$	Calibrated magnetometer reading after correction for magnetic hard and soft iron effects
$\boldsymbol{B}_p$	Raw magnetometer reading before correction for magnetic hard and soft iron effects
$\boldsymbol{B}_r$	Geomagnetic field vector in the earth frame
M	Number of measurements used in fit
N	Number of unknowns
P	Performance function which is in to be minimized
$\boldsymbol{r}$	Vector of fit errors (residuals)
$\boldsymbol{R}$	Rotation matrix defining the smartphone orientation
$\boldsymbol{V}$	Hard iron offset vector
$\boldsymbol{W}$	Soft iron matrix
$\boldsymbol{X}$	Measurement matrix
$\boldsymbol{Y}$	Vector of dependent variables
$\boldsymbol{\beta}$	Solution vector
δ	Geomagnetic inclination angle
ε	Normalized fit error
λ	Eigenvalue

2 Magnetometer Measurements Buffer

2.1 Structure

The magnetometer measurement buffer is a 6x6x6=216 array which selects, stores and prioritizes the magnetometer measurements for use by the calibration algorithms. Its purpose is two-fold:

- to ensure that the stored magnetometer measurements are sufficiently different from each other that a stable magnetic calibration is always obtained.
- to ensure that calibration uses the most recent magnetometer measurements in preference to older measurements.

The magnetometer measurement buffer structure is defined as:

```
#define MAGBUFFSIZE 6              // magnetic buffer size: 6 implies 6^3 = 216 entries
struct MagneticBuffer
{
    int16 iBx[MAGBUFFSIZE][MAGBUFFSIZE][MAGBUFFSIZE];   // array of x magnetic fields
    int16 iBy[MAGBUFFSIZE][MAGBUFFSIZE][MAGBUFFSIZE];   // array of y magnetic fields
    int16 iBz[MAGBUFFSIZE][MAGBUFFSIZE][MAGBUFFSIZE];   // array of z magnetic fields
    int32 index[MAGBUFFSIZE][MAGBUFFSIZE][MAGBUFFSIZE]; // array of time indices
    int32 iMagBufferCount;                              // number of magnetometer readings
};
```

Each of the 6x6x6=125 measurement entries comprises raw the *x*, *y*, and *z* magnetometer 16 bit readings as read from the magnetometer sensor (`iBx`, `iBy`, `iBz`) and a time index used for prioritizing the most recent measurements over older measurements (`index`).

The time index is a signed 32-bit integer which increments by 1 each iteration of the eCompass. A value of –1 denotes that the measurement is invalid as a result of never being entered or through having aged so that it should no longer be used. Values from 0 to 2147483647 inclusive are therefore valid with smaller index values denoting earlier, and therefore older, measurements. At a sampling rate of 50 Hz, the index will wrap around to –2147483648 after 1.36 years of continual operation. The software makes no test for wraparound on the assumption that the eCompass will never be powered up for this length of time.

2.2 Array indexing

The three indices of the magnetometer buffer are derived from *pseudo-angles* calculated from the arctangents of the accelerometer readings. Each magnetometer buffer bin corresponds approximately to an angle range of 180º / `MAGBUFFSIZE` which, for `MAGBUFFSIZE=6`, equates to 30º. This simple technique ensures that the buffer contains magnetometer measurements taken at significantly different orientation angles relative to the geomagnetic field and therefore a high quality calibration estimate.

```
// map -90 to 90 degrees onto range 0 to MAGBUFFSIZE - 1
ftmp = (float) MAGBUFFSIZE * FRECIP180;
j = (int32) ((atan2(pthisAccel->fGpx, fabs(pthisAccel->fGpy)) * FRADTODEG + 90.0F) * ftmp);
k = (int32) ((atan2(pthisAccel->fGpy, fabs(pthisAccel->fGpz)) * FRADTODEG + 90.0F) * ftmp);
l = (int32) ((atan2(pthisAccel->fGpz, fabs(pthisAccel->fGpx)) * FRADTODEG + 90.0F) * ftmp);
```

Figure 1 shows an example of the locus of magnetometer measurements under arbitrary rotation lying on the surface of the soft iron ellipsoid offset from the origin by the hard iron offset. Each of the 216 bins in the three-dimensional magnetometer buffer contains one measurement taken from a specific region of the ellipsoid surface and, taken together, the set of 216 bins cover the ellipsoid surface. This indexing by accelerometer reading has proven to be a very effective, robust and low overhead method of selecting magnetometer measurements for use by the calibration algorithms.

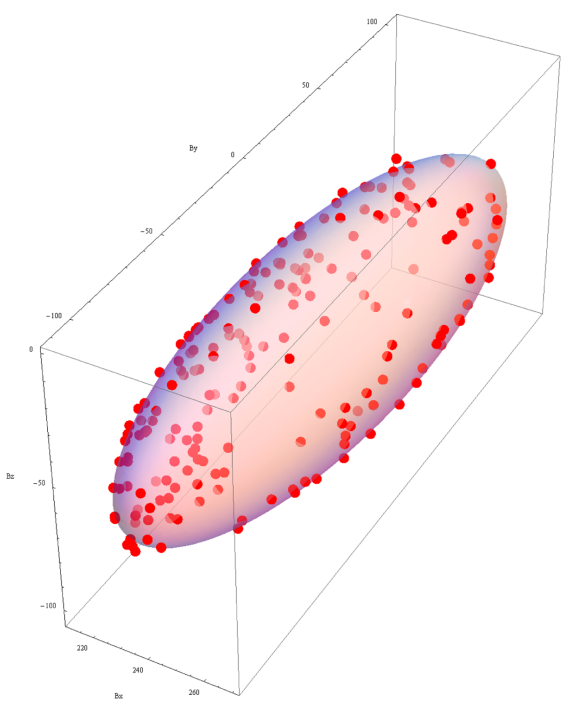

Figure 1. Locus of Magnetometer Measurements

If the eCompass PCB is left stationary for a long period, on a desk for example, only the one measurement bin corresponding to that orientation will be continually overwritten. The magnetic calibration will not, therefore, become unstable as a result of being filled with essentially identical magnetic data.

This code to update the magnetometer buffer has low overhead and should, ideally, execute regularly even if the eCompass is not being used in a mapping application. Normal orientation changes of the smartphone during calls or body motion will ensure that the magnetometer buffer is always filled with recent magnetometer data typical of the current environment and operating temperature so that no *figure of 8* movement is required when the eCompass application is eventually executed.

2.3 Data selection and aging

The default settings for the minimum and maximum number of measurements taken from the magnetometer buffer for use in the calibration algorithms are:

```
#define MINEQUATIONS 24        // minimum number of measurements used for calibration
#define MEDEQUATIONS 80        // new solution always accepted up to this point
#define MAXEQUATIONS 120       // maximum number of measurements used for calibration
```

The calibration algorithms first execute when there are `MINEQUATIONS` measurements in the magnetic buffer.

From `MINEQUATIONS` to `MEDEQUATIONS`, the calibration algorithms execute every `INITIALCALINTERVAL` or `FINALCALINTERVAL` iterations. The resulting solution is always accepted to replace the existing calibration even if the Fit Error increases on the assumption that the calibration becomes more accurate as more measurements become available. Between `MEDEQUATIONS` and `MAXEQUATIONS`, the new calibration is only accepted if the Fit Error is reduced."

In the absence of any sensor noise, the four element calibration algorithm could produce a robust solution using just four measurements and the seven element calibration algorithm using just seven measurements. In practice, using 24 measurements taken from the magnetometer buffer has been found to be the minimum required for an initial calibration in the presence of sensor noise. After power-up, with a completely empty magnetometer buffer, these 24 measurements will be rapidly captured with a very brief 'figure of 8' motion of the handset. In practice, if the approach described in the previous paragraphs is used where the magnetometer buffer is continually updated even when the eCompass is not required, then no 'figure of 8' motion will be required when the eCompass is required to execute.

In general, the more measurements used for calibration, the better but with the caveat that it is sensible to allow some space between the value of `MAXEQUATIONS` (120 measurements) and the buffer size (6x6x6=216 measurements) to allow old measurements to age out of use.

The code to update and control the number of active measurements in the buffer is in function `fUpdateMagnetometerBuffer` and listed in Function fUpdateMagnetometerBuffer. The algorithm is:

- If the current bin contains an active measurement (one of the most recent `MAXEQUATIONS` measurements) then no further action is required after the new magnetometer and time stamp data is written into the buffer
- If the current bin was empty (`previndex == -1`) but the number of measurements is still below `MAXEQUATIONS` then all that needs to be done is to increment the number of active measurements `iMagBufferCount`
- If the current bin was empty (`previndex == -1`) but there are already `MAXEQUATIONS` active entries in the buffer, then the oldest measurement is found and deactivated by setting its index to the empty value of –1.

2.4 Function fUpdateMagnetometerBuffer

```
// update the magnetic measurement buffer with most recent data
void fUpdateMagnetometerBuffer(struct MagneticBuffer *pthisMagneticBuffer,
  struct MagSensor *pthisMag, struct AccelSensor *pthisAccel, int32 loopcounter)
{
  // local variables
  int32 j, k, l;                  // magnetic buffer indices
  int32 previndex;                // previous time index in the bin being over-written
```

```
  int32 oldestj, oldestk, oldestl; // indices of bin with oldest data
  int32 oldestindex;               // time index of bin with oldest data
  float ftmp;                      // scratch variable

  // map -90 to 90 degrees onto range 0 to MAGBUFFSIZE - 1
  ftmp = (float) MAGBUFFSIZE * FRECIP180;
  j = (int32) ((atan2(pthisAccel->fGpx, fabs(pthisAccel->fGpy)) * FRADTODEG + 90.0F) * ftmp);
  k = (int32) ((atan2(pthisAccel->fGpy, fabs(pthisAccel->fGpz)) * FRADTODEG + 90.0F) * ftmp);
  l = (int32) ((atan2(pthisAccel->fGpz, fabs(pthisAccel->fGpx)) * FRADTODEG + 90.0F) * ftmp);

  // bounds safety check in case exactly on boundary
  if (j >= MAGBUFFSIZE) j = MAGBUFFSIZE - 1;
  if (k >= MAGBUFFSIZE) k = MAGBUFFSIZE - 1;
  if (l >= MAGBUFFSIZE) l = MAGBUFFSIZE - 1;

  // save the previous time index in this bin for later use
  previndex = pthisMagneticBuffer->index[j][k][l];

  // store the raw short integer magnetometer reading
  pthisMagneticBuffer->iBx[j][k][l] = pthisMag->iBpx;
  pthisMagneticBuffer->iBy[j][k][l] = pthisMag->iBpy;
  pthisMagneticBuffer->iBz[j][k][l] = pthisMag->iBpz;

  // set the time index to the loop counter (valid value 0 at start of first pass)
  pthisMagneticBuffer->index[j][k][l] = loopcounter;

  // no additional action needed if current bin was active
  // but action is needed if bin was inactive (value -1)
  if (previndex == -1)
  {
    if (pthisMagneticBuffer->iMagBufferCount < MAXEQUATIONS)
    {
      // simply increase the count of active measurements for calibration
      (pthisMagneticBuffer->iMagBufferCount)++;
    }
    else
    {
      // we need to find and de-activate the oldest reading
      oldestindex = loopcounter;
      oldestj = oldestk = oldestl = 0; // to avoid compiler error
      for (j = 0; j < MAGBUFFSIZE; j++)
      {
        for (k = 0; k < MAGBUFFSIZE; k++)
        {
          for (l = 0; l < MAGBUFFSIZE; l++)
          {
            if ((pthisMagneticBuffer->index[j][k][l] != -1) &&
                (pthisMagneticBuffer->index[j][k][l] < oldestindex))
            {
              oldestj = j;
              oldestk = k;
              oldestl = l;
              oldestindex = pthisMagneticBuffer->index[oldestj][oldestk][oldestl];
            }
          }
        }
      }
      // deactivate the oldest reading without bothering to zero the measurement data
      pthisMagneticBuffer->index[oldestj][oldestk][oldestl] = -1;
    }
  }

  return;
}
```

3 Hard and Soft Iron Model

3.1 General linear model

The calibrated magnetometer reading $\boldsymbol{B}_c$ (where *calibrated* means that hard and soft iron distortions are not present or have been removed) measured by the smartphone is simply the geomagnetic field $\boldsymbol{B}_r$ rotated by the orientation matrix $\boldsymbol{R}$:

$$\boldsymbol{B}_c = \boldsymbol{R}\boldsymbol{B}_r \tag{1}$$

The general linear model for the actual magnetometer reading $\boldsymbol{B}_p$, including the effects of hard and soft iron distortion, is:

$$\boldsymbol{B}_p = \boldsymbol{W}\boldsymbol{B}_c + \boldsymbol{V} = \boldsymbol{W}\boldsymbol{R}\boldsymbol{B}_r + \boldsymbol{V} \tag{2}$$

The 3x1 vector $\boldsymbol{V}$ is termed the hard iron offset and the 3x3 matrix $\boldsymbol{W}$ is termed the soft iron matrix. Physically, and simplifying a complicated subject, the hard iron offset models the effect of permanently magnetized components on the smartphone PCB and the soft iron matrix models the directional effect of induced magnetic fields and differing sensitivities in the three axes of the magnetometer sensor.

The calibration algorithms described later in this document estimate the hard iron offset $\boldsymbol{V}$ and the soft iron matrix $\boldsymbol{W}$ from magnetometer measurements stored in the magnetometer buffer and then invert equation (2) to remove their effect from the magnetometer readings to allow an accurate eCompass heading to be determined.

3.2 Measurement locus

Under arbitrary rotation of the phone, the locus of the calibrated magnetometer reading $\boldsymbol{B}_c = \boldsymbol{R}\boldsymbol{B}_r$ lies on the surface of a sphere with radius B.

$$(\boldsymbol{B}_c)^T\boldsymbol{B}_c = (\boldsymbol{R}\boldsymbol{B}_r)^T\boldsymbol{R}\boldsymbol{B}_r = \boldsymbol{B}_r{}^T\boldsymbol{R}^T\boldsymbol{R}\boldsymbol{B}_r = B^2 \tag{3}$$

In the presence of hard and soft iron effects, the locus of the uncalibrated magnetometer reading $\boldsymbol{B}_p$ lies on the surface defined using equation (2) to be:

$$\left\{\boldsymbol{W}^{-1}(\boldsymbol{B}_p - \boldsymbol{V})\right\}^T\left\{\boldsymbol{W}^{-1}(\boldsymbol{B}_p - \boldsymbol{V})\right\} = (\boldsymbol{R}\boldsymbol{B}_r)^T\boldsymbol{R}\boldsymbol{B}_r = \boldsymbol{B}_r{}^T\boldsymbol{R}^T\boldsymbol{R}\boldsymbol{B}_r = B^2 \tag{4}$$

$$\Rightarrow (\boldsymbol{B}_p - \boldsymbol{V})^T(\boldsymbol{W}^{-1})^T\boldsymbol{W}^{-1}(\boldsymbol{B}_p - \boldsymbol{V}) = B^2 \tag{5}$$

The general expression for the locus of a vector $\boldsymbol{R}$ lying on the surface of an ellipsoid with center at $\boldsymbol{R}_0$ is:

$$(\boldsymbol{R} - \boldsymbol{R}_0)^T\boldsymbol{A}(\boldsymbol{R} - \boldsymbol{R}_0) = const \tag{6}$$

where $\boldsymbol{A}$ is a symmetric matrix defining the shape of the ellipsoid.

Equations (5) and (6) are of the same form since it can be easily proved that the matrix $\boldsymbol{A} = \{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1}$ is symmetric:

$$\boldsymbol{A}^T = \{\{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1}\}^T = \{\boldsymbol{W}^{-1}\}^T\{\{\boldsymbol{W}^{-1}\}^T\}^T = \{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1} = \boldsymbol{A} \quad (7)$$

The hard and soft iron distortion therefore force the geomagnetic vector, as measured in the smartphone reference frame, to lie on the surface on an ellipsoid centered at the hard iron offset $\boldsymbol{V}$ with shape determined by the transposed product of the inverse soft iron matrix with itself $\{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1}$. The precise point on the ellipsoid where the measurement lies is determined by the orientation of the smartphone.

3.3 Inverting the model

Equation (2) can be inverted to give the calibrated magnetometer reading from the uncalibrated magnetometer reading once the calibration algorithms have estimated the hard iron offset and the soft iron matrix:

$$\boldsymbol{B}_c = \boldsymbol{W}^{-1}\left(\boldsymbol{B}_p - \boldsymbol{V}\right) \quad (8)$$

Figure 2 shows the calibration process visually using real measured magnetometer data from a strong hard and soft iron environment. The red data points are the raw measured magnetometer data $\boldsymbol{B}_p$ which, as equation (5) predicts, lies on the surface of an ellipsoid. The calibration process fits the optimum ellipsoid to the raw magnetometer data which is shown superimposed on the red data points.

Removal of the hard iron offset only will transform the measured data points to the lightly shaded ellipsoid centered at the origin. Removal of both hard and soft iron effect transforms the measured data points $\boldsymbol{B}_p$ to the blue calibrated points $\boldsymbol{B}_c$ which, as equation (3) predicts, lie on the surface of a sphere centered at the origin with radius equal to the geomagnetic field strength of approximately 50 µT.

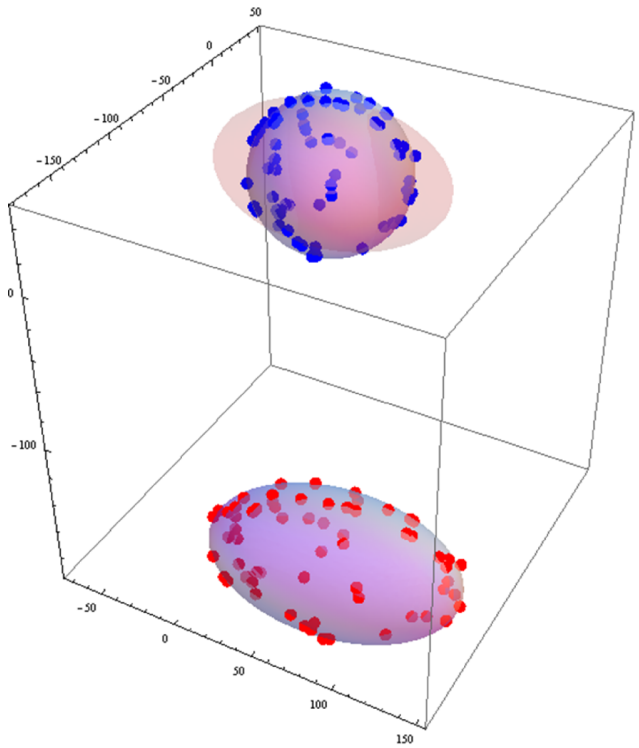

Figure 2. Measurement Loci of Raw (red) and Calibrated (blue) Magnetometer Data

3.4 Function fInvertMagCal

The code to perform the mapping of equation (8) is implemented in function `fInvertMagCal` and listed below.

```
// map the uncalibrated magnetometer data Bp (uT) onto calibrated data Bc (uT)
void fInvertMagCal(struct MagSensor *pthisMag, struct MagCalibration *pthisMagCal)
{
    // local variables
    float ftmpx, ftmpy, ftmpz;

    // remove the computed hard iron offset
    pthisMag->fBcx = pthisMag->fBpx - pthisMagCal->fVx;
    pthisMag->fBcy = pthisMag->fBpy - pthisMagCal->fVy;
    pthisMag->fBcz = pthisMag->fBpz - pthisMagCal->fVz;

    // remove the computed soft iron offset
    ftmpx = pthisMagCal->finvW[0][0] * pthisMag->fBcx +
            pthisMagCal->finvW[0][1] * pthisMag->fBcy +
            pthisMagCal->finvW[0][2] * pthisMag->fBcz;
    ftmpy = pthisMagCal->finvW[1][0] * pthisMag->fBcx +
            pthisMagCal->finvW[1][1] * pthisMag->fBcy +
            pthisMagCal->finvW[1][2] * pthisMag->fBcz;
    ftmpz = pthisMagCal->finvW[2][0] * pthisMag->fBcx +
            pthisMagCal->finvW[2][1] * pthisMag->fBcy +
            pthisMagCal->finvW[2][2] * pthisMag->fBcz;
    pthisMag->fBcx = ftmpx;
    pthisMag->fBcy = ftmpy;
    pthisMag->fBcz = ftmpz;

    return;
}
```

4 Linear Least Squares

4.1 Measurement model

This section provides the mathematical foundation for the 7-parameter and 4-parameter calibration algorithms discussed later in this document.

The general linear model relating independent variables $X_j[i]$ to dependent variable $Y[i]$ at measurement i via the N fitted model parameters β_j is:

$$Y[i] = \beta_0 X_0[i] + \beta_1 X_1[i] + \cdots + \beta_{N-1} X_{N-1}[i] \tag{9}$$

The fit to the model will, in general, not be perfectly accurate and will result in an error term $r[i]$:

$$r[i] = Y[i] - \beta_0 X_0[i] - \beta_1 X_1[i] - \cdots - \beta_{N-1} X_{N-1}[i] \tag{10}$$

For a series of M measurements, equation (10) can be written in the form:

$$\begin{pmatrix} r[0] \\ r[1] \\ \cdots \\ r[M-1] \end{pmatrix} = \begin{pmatrix} Y[0] \\ Y[1] \\ \cdots \\ Y[M-1] \end{pmatrix} - \begin{pmatrix} X_0[0] & X_1[0] & \cdots & X_{N-1}[0] \\ X_0[1] & X_1[1] & \cdots & X_{N-1}[1] \\ \cdots & \cdots & \cdots & \cdots \\ X_0[M-1] & X_1[M-1] & \cdots & X_{N-1}[M-1] \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \cdots \\ \beta_{N-1} \end{pmatrix} \tag{11}$$

Equation (11) can then be written in the more compact form below:

$$\boldsymbol{r} = \boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\beta} \tag{12}$$

$\boldsymbol{r}$ is the column vector of errors or residuals to the fit of the solution vector $\boldsymbol{\beta}$:

$$\boldsymbol{r} = \begin{pmatrix} r[0] \\ r[1] \\ \cdots \\ r[M-1] \end{pmatrix} \tag{13}$$

$\boldsymbol{Y}$ is the M by 1 column vector of M measurements on the dependent variable Y:

$$\boldsymbol{Y} = \begin{pmatrix} Y[0] \\ Y[1] \\ \cdots \\ Y[M-1] \end{pmatrix} \tag{14}$$

X is the M by N matrix of M measurements of the independent variable X:

$$\boldsymbol{X} = \begin{pmatrix} X_0[0] & X_1[0] & \dots & X_{N-1}[0] \\ X_0[1] & X_1[1] & \dots & X_{N-1}[1] \\ \dots & \dots & \dots & \dots \\ X_0[M-1] & X_1[M-1] & \dots & X_{N-1}[M-1] \end{pmatrix} \tag{15}$$

$\boldsymbol{\beta}$ is the N by 1 column vector of unknown model coefficients β_0 to β_{N-1} to be determined:

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_{N-1} \end{pmatrix} \tag{16}$$

If there are more measurements M than there are unknowns N, then the equations are typically solved in a least squares sense by minimizing the performance function P defined as the modulus squared of the error vector $\boldsymbol{r}$:

$$P = \boldsymbol{r}^T\boldsymbol{r} = \|\boldsymbol{r}\|^2 = (\boldsymbol{Y} - \boldsymbol{X\beta})^T(\boldsymbol{Y} - \boldsymbol{X\beta}) = \|\boldsymbol{Y} - \boldsymbol{X\beta}\|^2 \tag{17}$$

4.2 Derivation of Normal Equations for non-homogeneous case

If the vector of dependent measurements $\boldsymbol{Y}$ is not zero, then the equations are termed non-homogeneous. The performance function will be a minimum when it is stationary with respect to any perturbation $\delta\boldsymbol{\beta}$ about the optimal least squares solution $\boldsymbol{\beta}$:

$$\nabla P = \frac{\partial P}{\partial \boldsymbol{\beta}} = 0 \Rightarrow P(\boldsymbol{\beta} + \delta\boldsymbol{\beta}) = P(\boldsymbol{\beta}) \tag{18}$$

Substituting equation (12) into (18) and ignoring second order terms gives:

$$\big(\boldsymbol{Y} - \boldsymbol{X}(\boldsymbol{\beta} + \delta\boldsymbol{\beta})\big)^T\big(\boldsymbol{Y} - \boldsymbol{X}(\boldsymbol{\beta} + \delta\boldsymbol{\beta})\big) - (\boldsymbol{Y} - \boldsymbol{X\beta})^T(\boldsymbol{Y} - \boldsymbol{X\beta}) = 0 \tag{19}$$

$$\Rightarrow -\boldsymbol{Y}^T\boldsymbol{X}\delta\boldsymbol{\beta} + (\boldsymbol{X\beta})^T\boldsymbol{X}\delta\boldsymbol{\beta} - (\boldsymbol{X}\delta\boldsymbol{\beta})^T\,(\boldsymbol{Y} - \boldsymbol{X\beta}) = 0 \tag{20}$$

$$\Rightarrow -\boldsymbol{Y}^T\boldsymbol{X}\delta\boldsymbol{\beta} + (\boldsymbol{X\beta})^T\boldsymbol{X}\delta\boldsymbol{\beta} - \delta\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{Y} + \delta\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} = 0 \tag{21}$$

Since $\delta\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{Y}$ and $\delta\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta}$ are scalars, their values are unchanged by the transpose operation and equation (21) can be re-written as:

$$(-\boldsymbol{Y}^T\boldsymbol{X} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X})\delta\boldsymbol{\beta} = 0 \tag{22}$$

$$\Rightarrow \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{Y}^T\boldsymbol{X} \Rightarrow \boldsymbol{X}^T\boldsymbol{X\beta} = \boldsymbol{X}^T\boldsymbol{Y} \tag{23}$$

$$\Rightarrow \boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \tag{24}$$

Equation (24) is termed the *Normal Equations* solution for the vector $\boldsymbol{\beta}$ for the non-homogeneous equations $\boldsymbol{Y} = \boldsymbol{X\beta}$.

The performance function P at the optimal least squares solution can be found by substituting equation (24) into equation (17) to give:

$$P = \boldsymbol{r}^T\boldsymbol{r} = \{\boldsymbol{Y} - \boldsymbol{X\beta}\}^T\{\boldsymbol{Y} - \boldsymbol{X\beta}\} = (\boldsymbol{Y}^T - \boldsymbol{\beta}^T\boldsymbol{X}^T)(\boldsymbol{Y} - \boldsymbol{X\beta}) = \boldsymbol{Y}^T\boldsymbol{Y} - \boldsymbol{Y}^T\boldsymbol{X\beta} - \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{Y} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} \quad (25)$$

$$= \boldsymbol{Y}^T\boldsymbol{Y} - \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{Y} - (\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{Y})^T + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} \quad (26)$$

Since each term of equation (26) is a scalar and equal to its transpose, the performance function can be written as:

$$P = \boldsymbol{Y}^T\boldsymbol{Y} - 2\boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{Y}) + \boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{\beta} \quad (27)$$

In the special case where the number of measurements M equals the number of model parameters N to be fitted, the matrix $\boldsymbol{X}$ is square and the standard result for invertible square matrices holds that:

$$(\boldsymbol{X}^T\boldsymbol{X})^{-1} = \boldsymbol{X}^{-1}(\boldsymbol{X}^T)^{-1} \Rightarrow \boldsymbol{\beta} = \boldsymbol{X}^{-1}(\boldsymbol{X}^T)^{-1}\boldsymbol{X}^T\boldsymbol{Y} = \boldsymbol{X}^{-1}\boldsymbol{Y} \quad (28)$$

Substituting equation (28) into (27) gives the expected result that the performance function is zero and the fit is perfect when the number of model parameters equals the number of measurements:

$$P = \boldsymbol{Y}^T\boldsymbol{Y} - 2(\boldsymbol{X}^{-1}\boldsymbol{Y})^T\boldsymbol{X}^T\boldsymbol{Y} + (\boldsymbol{X}^{-1}\boldsymbol{Y})^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{X}^{-1}\boldsymbol{Y} = \boldsymbol{Y}^T\boldsymbol{Y} - (\boldsymbol{X}^{-1}\boldsymbol{Y})^T\boldsymbol{X}^T\boldsymbol{Y} = \boldsymbol{Y}^T\boldsymbol{Y} - \boldsymbol{Y}^T(\boldsymbol{X}^{-1})^T\boldsymbol{X}^T\boldsymbol{Y} = 0 \quad (29)$$

4.3 Derivation of solution for homogeneous case

If the dependent measurement vector $\boldsymbol{Y}$ is zero, then the equations are termed homogeneous. The model being fitted is now:

$$\boldsymbol{X\beta} = 0 \quad (30)$$

The performance function P to be minimized simplifies to:

$$P = \|\boldsymbol{r}\|^2 = \|\boldsymbol{X\beta}\|^2 = (\boldsymbol{X\beta})^T\boldsymbol{X\beta} = \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} \quad (31)$$

Using the Normal Equations of equation (24) for the non-homogeneous case simply gives the null vector $\boldsymbol{\beta}$ as solution:

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} = 0 \quad (32)$$

A solution method is required which minimizes the performance function P in equation (31) subject to the constraint that $\boldsymbol{\beta}$ has non-zero magnitude. Since equation (30) is linear, we can arbitrarily scale the solution vector $\boldsymbol{\beta}$ to have unit magnitude:

$$1 - \boldsymbol{\beta}^T\boldsymbol{\beta} = 0 \quad (33)$$

The method of Lagrange Multipliers can be used to find constrained minima in this case. Using this approach, the performance function to be minimized is:

$$P(\boldsymbol{\beta}) = (\boldsymbol{X\beta})^{\boldsymbol{T}}\boldsymbol{X\beta} + \lambda(1 - \boldsymbol{\beta}^{\boldsymbol{T}}\boldsymbol{\beta}) = \boldsymbol{\beta}^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X\beta} + \lambda(1 - \boldsymbol{\beta}^{\boldsymbol{T}}\boldsymbol{\beta}) \quad (34)$$

Applying the stationary constraint that $P(\boldsymbol{\beta} + \delta\boldsymbol{\beta}) = P(\boldsymbol{\beta})$ gives:

$$(\boldsymbol{\beta} + \delta\boldsymbol{\beta})^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}(\boldsymbol{\beta} + \delta\boldsymbol{\beta}) + \lambda\left(1 - (\boldsymbol{\beta} + \delta\boldsymbol{\beta})^{\boldsymbol{T}}(\boldsymbol{\beta} + \delta\boldsymbol{\beta})\right) = \boldsymbol{\beta}^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X\beta} + \lambda(1 - \boldsymbol{\beta}^{\boldsymbol{T}}\boldsymbol{\beta}) \quad (35)$$

Ignoring second order terms gives:

$$\delta\boldsymbol{\beta}^{T}\boldsymbol{X}^{T}\boldsymbol{X\beta} + \boldsymbol{\beta}^{T}\boldsymbol{X}^{T}\boldsymbol{X}\delta\boldsymbol{\beta} - \lambda(\boldsymbol{\beta}^{T}\delta\boldsymbol{\beta} + \delta\boldsymbol{\beta}^{T}\boldsymbol{\beta}) = 0 \quad (36)$$

Since each term in equation (36) is a scalar and equal to its transpose, the solution for the optimum $\boldsymbol{\beta}$ which constrains the performance function is:

$$\delta\boldsymbol{\beta}^{T}(\boldsymbol{X}^{T}\boldsymbol{X\beta} - \lambda\boldsymbol{\beta}) = 0 \quad (37)$$

$$\Rightarrow \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X\beta} = \lambda\boldsymbol{\beta} \quad (38)$$

Equation (38) states that the required solution vector $\boldsymbol{\beta}$ is an eigenvector of the product matrix $\boldsymbol{X}^{T}\boldsymbol{X}$ associated with eigenvalue λ.

Substituting equation (38) into equation (31) for the performance function P_i with eigenvalue λ_i and eigenvector $\boldsymbol{\beta}_i$ gives:

$$P_i = \boldsymbol{\beta}_i{}^{T}\boldsymbol{X}^{T}\boldsymbol{X}\boldsymbol{\beta}_i = \lambda_i\boldsymbol{\beta}_i{}^{T}\boldsymbol{\beta}_i = \lambda_i \quad (39)$$

The performance function P_i for the *i-th* eigenvector solution therefore equals the eigenvalue λ_i of $\boldsymbol{X}^{T}\boldsymbol{X}$ and the minimum performance function is equal to the minimum eigenvalue. The optimal solution $\boldsymbol{\beta}$ is then the eigenvector corresponding to the smallest eigenvalue λ_{min}.

5 Four Element Magnetic Calibration

This section documents the algorithm and the C code implementation of the 4-element calibration algorithm which solves for the three components of the hard iron vector $\boldsymbol{V}$ and the geomagnetic field strength B.

5.1 Matrix inverse least squares solution

Equation (5) for the case of a PCB subject to hard iron interference only simplifies to:

$$\left(\boldsymbol{B}_p - \boldsymbol{V}\right)^{T}\left(\boldsymbol{B}_p - \boldsymbol{V}\right) = B^2 \quad (40)$$

$$\Rightarrow \boldsymbol{B_p}^T\boldsymbol{B_p} - 2\boldsymbol{B_p}^T\boldsymbol{V} + \boldsymbol{V}^T\boldsymbol{V} - B^2 = 0 \quad \textbf{(41)}$$

The residual error $r[i]$ for the *i-th* observation is then:

$$r[i] = B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2 - 2B_{px}[i]V_x - 2B_{py}[i]V_y - 2B_{pz}[i]V_z + {V_x}^2 + {V_y}^2 + {V_z}^2 - B^2 \quad \textbf{(42)}$$

Simplifying and returning to matrix format gives:

$$r[i] = \left(B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2\right) - \begin{pmatrix} B_{px}[i] \\ B_{py}[i] \\ B_{pz}[i] \\ 1 \end{pmatrix}^T \begin{pmatrix} 2V_x \\ 2V_y \\ 2V_z \\ B^2 - {V_x}^2 - {V_y}^2 - {V_z}^2 \end{pmatrix} \quad \textbf{(43)}$$

The dependent measurement variable $y[i]$ can be defined to be:

$$y[i] = B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2 \quad \textbf{(44)}$$

and the solution vector $\boldsymbol{\beta}$ as:

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} 2V_x \\ 2V_y \\ 2V_z \\ B^2 - {V_x}^2 - {V_y}^2 - {V_z}^2 \end{pmatrix} \quad \textbf{(45)}$$

The error residual is now given by:

$$r[i] = B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2 - \begin{pmatrix} B_{px}[i] \\ B_{py}[i] \\ B_{pz}[i] \\ 1 \end{pmatrix}^T \begin{pmatrix} 2V_x \\ 2V_y \\ 2V_z \\ B^2 - {V_x}^2 - {V_y}^2 - {V_z}^2 \end{pmatrix} \quad \textbf{(46)}$$

$$= B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2 - \begin{pmatrix} B_{px}[i] \\ B_{py}[i] \\ B_{pz}[i] \\ 1 \end{pmatrix}^T \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} \quad \textbf{(47)}$$

Equation **(47)** can be expanded to represent *M* measurements as:

$$\begin{pmatrix} r[0] \\ r[1] \\ \dots \\ r[M-1] \end{pmatrix} = \begin{pmatrix} B_{px}[0]^2 + B_{py}[0]^2 + B_{pz}[0]^2 \\ B_{px}[1]^2 + B_{py}[1]^2 + B_{pz}[1]^2 \\ \dots \\ B_{px}[M-1]^2 + B_{py}[M-1]^2 + B_{pz}[M-1]^2 \end{pmatrix} - \begin{pmatrix} B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1 \\ B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1 \\ \dots & \dots & \dots & 1 \\ B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1 \end{pmatrix} \begin{pmatrix} 2V_x \\ 2V_y \\ 2V_z \\ B^2 - {V_x}^2 - {V_y}^2 - {V_z}^2 \end{pmatrix} \quad \textbf{(48)}$$

The equivalent equation in matrix form is:

$$\boldsymbol{r} = \boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\beta} \tag{49}$$

where $\boldsymbol{r}$ is the error residual vector.

$$\boldsymbol{r} = \begin{pmatrix} r[0] \\ r[1] \\ \dots \\ r[M-1] \end{pmatrix} \tag{50}$$

$\boldsymbol{Y}$ is the vector of dependent variables:

$$\boldsymbol{Y} = \begin{pmatrix} B_{px}[0]^2 + B_{py}[0]^2 + B_{pz}[0]^2 \\ B_{px}[1]^2 + B_{py}[1]^2 + B_{pz}[1]^2 \\ \dots \\ B_{px}[M-1]^2 + B_{py}[M-1]^2 + B_{pz}[M-1]^2 \end{pmatrix} \tag{51}$$

$\boldsymbol{X}$ is the M by 4 measurement matrix:

$$\boldsymbol{X} = \begin{pmatrix} B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1 \\ B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1 \\ \dots & \dots & \dots & 1 \\ B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1 \end{pmatrix} \tag{52}$$

The model being fitted is the non-homogeneous model $\boldsymbol{r} = \boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\beta}$ which is most simply solved for $\boldsymbol{\beta}$ using the Normal Equations of equation (24) :

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \tag{53}$$

The performance function P is given by equation (27) :

$$P = \boldsymbol{Y}^T\boldsymbol{Y} - 2\boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{Y}) + \boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{\beta} \tag{54}$$

The matrix products can be written in the form:

$$\boldsymbol{X}^T\boldsymbol{X} = \begin{pmatrix} B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1 \\ B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1 \\ \dots & \dots & \dots & 1 \\ B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1 \end{pmatrix}^T \begin{pmatrix} B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1 \\ B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1 \\ \dots & \dots & \dots & 1 \\ B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1 \end{pmatrix} \tag{55}$$

Table continues on the next page...

$$= \sum_{i=0}^{M-1} \begin{pmatrix} B_{px}[i]^2 & B_{px}[i]B_{py}[i] & B_{px}[i]B_{pz}[i] & B_{px}[i] \\ B_{px}[i]B_{py}[i] & B_{py}[i]^2 & B_{py}[i]B_{pz}[i] & B_{py}[i] \\ B_{px}[i]B_{pz}[i] & B_{py}[i]B_{pz}[i] & B_{pz}[i]^2 & B_{pz}[i] \\ B_{px}[i] & B_{py}[i] & B_{pz}[i] & 1 \end{pmatrix} \tag{56}$$

$$\boldsymbol{X}^T\boldsymbol{Y} = \begin{pmatrix} B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1 \\ B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1 \\ \dots & \dots & \dots & 1 \\ B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1 \end{pmatrix}^T \begin{pmatrix} B_{px}[0]^2 + B_{py}[0]^2 + B_{pz}[0]^2 \\ B_{px}[1]^2 + B_{py}[1]^2 + B_{pz}[1]^2 \\ \dots \\ B_{px}[M-1]^2 + B_{py}[M-1]^2 + B_{pz}[M-1]^2 \end{pmatrix} \tag{57}$$

$$= \sum_{i=0}^{M-1} \begin{pmatrix} B_{px}[i]\left(B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2\right) \\ B_{py}[i]\left(B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2\right) \\ B_{pz}[i]\left(B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2\right) \\ B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2 \end{pmatrix} \tag{58}$$

$$\boldsymbol{Y}^T\boldsymbol{Y} = \begin{pmatrix} B_{px}[0]^2 + B_{py}[0]^2 + B_{pz}[0]^2 \\ B_{px}[1]^2 + B_{py}[1]^2 + B_{pz}[1]^2 \\ \dots \\ B_{px}[M-1]^2 + B_{py}[M-1]^2 + B_{pz}[M-1]^2 \end{pmatrix}^T \begin{pmatrix} B_{px}[0]^2 + B_{py}[0]^2 + B_{pz}[0]^2 \\ B_{px}[1]^2 + B_{py}[1]^2 + B_{pz}[1]^2 \\ \dots \\ B_{px}[M-1]^2 + B_{py}[M-1]^2 + B_{pz}[M-1]^2 \end{pmatrix} \tag{59}$$

$$= \sum_{i=0}^{M-1} \left(B_{px}[i]^2 + B_{py}[i]^2 + B_{pz}[i]^2\right)^2 \tag{60}$$

5.2 Hard iron calibration

The soft iron model is given directly by equation (45) as:

$$\begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} = \left(\frac{1}{2}\right)\begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} \tag{61}$$

The soft iron matrix is the 3x3 identity matrix in the 4-element, hard iron-only, algorithm:

$$\boldsymbol{V} = \boldsymbol{I} \tag{62}$$

5.3 Fit error

Since $P = \boldsymbol{r}^T\boldsymbol{r}$ and $\boldsymbol{r}$ has M elements with each element having dimensions B^2, a suitable normalized and dimensionless fit error ε can be defined as:

$$\varepsilon = \frac{1}{2B^2}\sqrt{\frac{P}{M}} = \frac{1}{2B^2}\sqrt{\frac{\boldsymbol{Y}^T\boldsymbol{Y} - 2\boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{Y}) + \boldsymbol{\beta}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{\beta}}{M}} \tag{63}$$

5.4 Geomagnetic field strength

The geomagnetic field strength can be computed from the fourth row of matrix equation (45) as.

$$B^2 = \beta_3 + V_x{}^2 + V_y{}^2 + V_z{}^2 \Rightarrow B = \sqrt{\beta_3 + V_x{}^2 + V_y{}^2 + V_z{}^2} \quad (64)$$

5.5 Function fUpdateCalibration4INV

The function `fUpdateCalibration4INV` implements the 4-element calibration algorithm documented in this section.

```
// 4 element calibration using 4x4 matrix inverse
void fUpdateCalibration4INV(struct MagCalibration *pthisMagCal,
 struct MagneticBuffer *pthisMagneticBuffer,
 float **ftmpA4x4, float **ftmpB4x4, float **ftmpA4x1,
 float **ftmpB4x1, int32 **icolind, int32 **irowind, int32 **ipivot)
{
 int32 i, j, k, l, m, n;                // loop counters
 int32 ilocalMagBufferCount;            // local count of measurements for this process
 float fOffsetx, fOffsety, fOffsetz;    // offset to remove large DC hard iron bias in matrix
 float ftmpBpx, ftmpBpy, ftmpBpz;       // x, y, z magnetometer readings (uT)
 float ftmpBpxsq, ftmpBpysq, ftmpBpzsq; // squares of x, y, z magnetometer readings (uT)
 float fy;                              // dependent variable
 float fYTY;                            // scalar equal to Y^T.Y
 float fscaling;                        // set to FUTPERCOUNT * FMATRIXSCALING
 float fP;                              // performance function = r^T.r

 // compute fscaling to reduce multiplications later
 fscaling = FUTPERCOUNT * FMATRIXSCALING;

 // set trial inverse soft iron matrix invW to the identity matrix for 4 element calibration
 pthisMagCal->ftrinvW[0][0] = pthisMagCal->ftrinvW[1][1] = pthisMagCal->ftrinvW[2][2] = 1.0F;
 pthisMagCal->ftrinvW[0][1] = pthisMagCal->ftrinvW[0][2] = pthisMagCal->ftrinvW[1][0] = 0.0F;
 pthisMagCal->ftrinvW[1][2] = pthisMagCal->ftrinvW[2][0] = pthisMagCal->ftrinvW[2][1] = 0.0F;

 // zero fYTY=Y^T.Y, ftmpA4x1=X^T.Y and on and above diagonal elements of ftmpA4x4=X^T*X
 fYTY = 0.0F;
 for (m = 0; m < 4; m++)
 {
     ftmpA4x1[m][0] = 0.0F;
     for (n = m; n < 4; n++)
     {
         ftmpA4x4[m][n] = 0.0F;
     }
 }

 // the offsets are guaranteed to be set from the first element but to avoid compiler error
 fOffsetx = fOffsety = fOffsetz = 0.0F;

 // use from MINEQUATIONS up to MAXEQUATIONS entries from magnetic buffer to compute matrices
 i = 0;
 for (j = 0; j < MAGBUFFSIZE; j++)
 {
     for (k = 0; k < MAGBUFFSIZE; k++)
     {
         for (l = 0; l < MAGBUFFSIZE; l++)
         {
             if (pthisMagneticBuffer->index[j][k][l] != -1)
             {
```

```
                // use first valid magnetic buffer entry as estimate (in counts) for offset
                if (i == 0)
                {
                    fOffsetx = (float)pthisMagneticBuffer->iBx[j][k][l];
                    fOffsety = (float)pthisMagneticBuffer->iBy[j][k][l];
                    fOffsetz = (float)pthisMagneticBuffer->iBz[j][k][l];
                }

                // calculate offset and scaled magnetic buffer vector data Bx, By, Bz
(scaled uT)
                ftmpBpx = ((float)pthisMagneticBuffer->iBx[j][k][l] - fOffsetx) * fscaling;
                ftmpBpy = ((float)pthisMagneticBuffer->iBy[j][k][l] - fOffsety) * fscaling;
                ftmpBpz = ((float)pthisMagneticBuffer->iBz[j][k][l] - fOffsetz) * fscaling;

                // calculate y = Bx^2 + By^2 + Bz^2 (scaled uT^2) and accumulate Y^T.Y
                ftmpBpxsq = ftmpBpx * ftmpBpx;
                ftmpBpysq = ftmpBpy * ftmpBpy;
                ftmpBpzsq = ftmpBpz * ftmpBpz;
                fy = ftmpBpxsq + ftmpBpysq + ftmpBpzsq;
                fYTY += fy * fy;

                // accumulate ftmpA4x1 = X^T.Y
                ftmpA4x1[0][0] += ftmpBpx * fy;
                ftmpA4x1[1][0] += ftmpBpy * fy;
                ftmpA4x1[2][0] += ftmpBpz * fy;
                ftmpA4x1[3][0] += fy;

                // accumulate on and above-diagonal terms of ftmpA4x4 = X^T.X
                ftmpA4x4[0][0] += ftmpBpxsq;
                ftmpA4x4[0][1] += ftmpBpx * ftmpBpy;
                ftmpA4x4[0][2] += ftmpBpx * ftmpBpz;
                ftmpA4x4[0][3] += ftmpBpx;
                ftmpA4x4[1][1] += ftmpBpysq;
                ftmpA4x4[1][2] += ftmpBpy * ftmpBpz;
                ftmpA4x4[1][3] += ftmpBpy;
                ftmpA4x4[2][2] += ftmpBpzsq;
                ftmpA4x4[2][3] += ftmpBpz;

                // increment the counter for next iteration
                i++;
            }
        }
    }
 }

 // store the number of measurements accumulated
 ilocalMagBufferCount = i;

 // set the last element of the measurement matrix to the number of buffer elements used
 ftmpA4x4[3][3] = (float) i;

 // use above diagonal elements of symmetric matrix ftmpA4x4 to create ftmpB4x4 = ftmpA4x4 =
X^T.X
 for (m = 0; m < 4; m++)
 {
     for (n = m; n < 4; n++)
     {
     ftmpB4x4[m][n] = ftmpB4x4[n][m] = ftmpA4x4[n][m] = ftmpA4x4[m][n];
     }
 }

 // calculate in situ inverse of ftmpB4x4 = inv(X^T.X) (4x4)
 fmatrixAeqInvA(ftmpB4x4, 4, icolind, irowind, ipivot);

 // calculate ftmpB4x1 = solution vector beta (4x1) = inv(X^T.X).X^T.Y = ftmpB4x4 * ftmpA4x1
 fmatrixAeqBxC(ftmpB4x1, ftmpB4x4, ftmpA4x1, 4, 4, 1);

 // calculate P = r^T.r = Y^T.Y - 2 * beta^T.(X^T.Y) + beta^T.(X^T.X).beta
 // = fYTY - 2 * ftmpB4x1^T.ftmpA4x1 + ftmpB4x1^T.ftmpA4x4.ftmpB4x1
 // first set P = Y^T.Y - 2 * beta^T.(X^T.Y) = fYTY - 2 * ftmpB4x1^T.ftmpA4x1
```

```
 fP = fYTY - 2.0F * (ftmpA4x1[0][0] * ftmpB4x1[0][0] + ftmpA4x1[1][0] * ftmpB4x1[1][0] +
     ftmpA4x1[2][0] * ftmpB4x1[2][0] + ftmpA4x1[3][0] * ftmpB4x1[3][0]);
 // set ftmpA4x1 = (X^T.X).beta = ftmpA4x4.ftmpB4x1
 fmatrixAeqBxC(ftmpA4x1, ftmpA4x4, ftmpB4x1, 4, 4, 1);
 // add beta^T.(X^T.X).beta = ftmpB4x1^T * ftmpA4x1 to P
 fP += ftmpA4x1[0][0] * ftmpB4x1[0][0] + ftmpA4x1[1][0] * ftmpB4x1[1][0] +
     ftmpA4x1[2][0] * ftmpB4x1[2][0] + ftmpA4x1[3][0] * ftmpB4x1[3][0];

 // compute the hard iron vector (in uT but offset and scaled by FMATRIXSCALING)
 pthisMagCal->ftrVx = 0.5F * ftmpB4x1[0][0];
 pthisMagCal->ftrVy = 0.5F * ftmpB4x1[1][0];
 pthisMagCal->ftrVz = 0.5F * ftmpB4x1[2][0];

 // compute the scaled geomagnetic field strength B (in uT but scaled by FMATRIXSCALING)
 pthisMagCal->ftrB = (float)sqrt(ftmpB4x1[3][0] + pthisMagCal->ftrVx * pthisMagCal->ftrVx +
     pthisMagCal->ftrVy * pthisMagCal->ftrVy + pthisMagCal->ftrVz * pthisMagCal->ftrVz);

 // calculate the trial fit error (percent) normalized to number of measurements and scaled
geomagnetic field strength
 pthisMagCal->ftrFitErrorpc = (float)sqrt(fP / (float) ilocalMagBufferCount) * 100.0F /
     (2.0F * pthisMagCal->ftrB * pthisMagCal->ftrB);
 printf("\n\nTrial new calibration fit error=%9.4f%% versus previous %9.4f%%", pthisMagCal-
>ftrFitErrorpc, pthisMagCal->fFitErrorpc);

 // correct the hard iron estimate for FMATRIXSCALING and the offsets applied (result in uT)
 pthisMagCal->ftrVx = pthisMagCal->ftrVx * FINVMATRIXSCALING + fOffsetx * FUTPERCOUNT;
 pthisMagCal->ftrVy = pthisMagCal->ftrVy * FINVMATRIXSCALING + fOffsety * FUTPERCOUNT;
 pthisMagCal->ftrVz = pthisMagCal->ftrVz * FINVMATRIXSCALING + fOffsetz * FUTPERCOUNT;
 printf("\n\nTrial new calibration hard iron (uT) Vx=%9.3f Vy=%9.3f Vz=%9.3f", pthisMagCal-
>ftrVx, pthisMagCal->ftrVy, pthisMagCal->ftrVz);

 // correct the geomagnetic field strength B to correct scaling (result in uT)
 pthisMagCal->ftrB *= FINVMATRIXSCALING;
 printf("\n\nTrial new calibration geomagnetic field (uT) B=%9.3f", pthisMagCal->ftrB);

 // set the valid calibration flag to true
 pthisMagCal->iValidMagCal = 1;

 return;
}
```

6 Eigen Analysis by Jacobi Algorithm

This section documents the algorithm underlying function `eigencompute` which computes the eigenvalues and eigenvectors of a symmetric matrix and is used by both the 7- and 10-element calibration algorithms.

The matrix of real eigenvectors $\boldsymbol{X}$ and diagonal matrix of real eigenvalues $\boldsymbol{\Lambda}$ of the real symmetric matrix $\boldsymbol{A}$ satisfy:

$$\boldsymbol{AX} = \boldsymbol{X\Lambda} \Rightarrow \boldsymbol{A} = \boldsymbol{X\Lambda X}^{-1} \Rightarrow \boldsymbol{\Lambda} = \boldsymbol{X}^{-1}\boldsymbol{AX} \tag{65}$$

Pre-and post multiplying equation (65) by a sequence of matrices ${\boldsymbol{R}_i}^{-1}$ and $\boldsymbol{R}_i$ selected to zero the off-diagonal elements of $\boldsymbol{A}$ gives:

$${\boldsymbol{R}_N}^{-1} \ldots {\boldsymbol{R}_2}^{-1}{\boldsymbol{R}_1}^{-1}\boldsymbol{AR}_1\boldsymbol{R}_2 \ldots \boldsymbol{R}_N = {\boldsymbol{R}_N}^{-1} \ldots {\boldsymbol{R}_2}^{-1}{\boldsymbol{R}_1}^{-1}\boldsymbol{X\Lambda X}^{-1}\boldsymbol{R}_1\boldsymbol{R}_2 \ldots \boldsymbol{R}_N \tag{66}$$

By comparison with equation (65), equation (66) can be written as:

$$\boldsymbol{\Lambda} = \boldsymbol{R}_N{}^{-1} \dots \boldsymbol{R}_2{}^{-1}\boldsymbol{R}_1{}^{-1}\boldsymbol{X}\boldsymbol{\Lambda}\boldsymbol{X}^{-1}\boldsymbol{R}_1\boldsymbol{R}_2 \dots \boldsymbol{R}_N \quad (67)$$

$$\Rightarrow \boldsymbol{X} = \boldsymbol{R}_1\boldsymbol{R}_2 \dots \boldsymbol{R}_N \quad (68)$$

The eigenvalues of the matrix $\boldsymbol{A}$ are then the elements of the diagonal matrix $\boldsymbol{\Lambda}$ derived by zeroing off-diagonal elements in $\boldsymbol{A}$ and the matrix of eigenvectors $\boldsymbol{X}$ is the product of the sequence of matrices used to perform the diagonalization.

6.1 Jacobi rotation matrix

Jacobi rotation matrices $\boldsymbol{R}_{pq}$ are used to successively zero off-diagonal matrix elements. The Jacobi matrices have form:

$$\boldsymbol{R}_{pq} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & c & \dots & s & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (69)$$

All the diagonal elements are zero except for the two elements at positions p,p and q,q. All off-diagonal elements are zero except for the two elements at positions p,q and q,p.

The elements c and $s$are the sine and cosine of the Jacobi rotation angle ϕ:

$$\begin{aligned} c &= cos(\phi) \\ s &= sin(\phi) \\ \Longrightarrow c^2 + s^2 &= 1 \end{aligned} \quad (70)$$

A general matrix $\boldsymbol{A}$ is transformed by the Jacobi rotation matrix by:

$$\boldsymbol{A}' = \boldsymbol{R}_{pq}{}^{T}\boldsymbol{A}\boldsymbol{R}_{pq} \quad (71)$$

The changed elements of $\boldsymbol{A}$ after this transformation are those shown below:

$$\boldsymbol{A}' = \begin{pmatrix} \dots & \dots & a'_{0,p} & \dots & a'_{0,q} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a'_{p,0} & \dots & a'_{p,p} & \dots & a'_{p,q} & \dots & a'_{p,n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a'_{q,0} & \dots & a'_{q,p} & \dots & a'_{q,q} & \dots & a'_{q,n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & a'_{n-1,p} & \dots & a'_{n-1,q} & \dots & \dots \end{pmatrix} \quad (72)$$

The changed elements of equation (72) are:

$$a'_{r,p} = ca_{r,p} - sa_{r,q} \; (r \neq p, r \neq q) \quad (73)$$

Table continues on the next page...

$$a'_{r,q} = ca_{r,q} + sa_{r,p}\ (r \neq p, r \neq q) \quad (74)$$

$$a'_{p,p} = c^2 a_{p,p} + s^2 a_{q,q} - 2sca_{p,q} \quad (75)$$

$$a'_{q,q} = s^2 a_{p,p} + c^2 a_{q,q} + 2sca_{p,q} \quad (76)$$

$$a'_{p,q} = (c^2 - s^2)a_{p,q} + sc(a_{p,p} - a_{q,q}) \quad (77)$$

6.2 Jacobi algorithm

Setting $a'_{p,q}$ to zero gives:

$$(c^2 - s^2)a_{p,q} + sc(a_{p,p} - a_{q,q}) = 0 \Rightarrow \frac{(c^2 - s^2)}{sc} = \frac{(a_{q,q} - a_{p,p})}{a_{p,q}} \quad (78)$$

Equation (79) is a simple trigonometric identity:

$$cot(2\phi) = \frac{1}{tan(2\phi)} = \frac{cos(2\phi)}{sin(2\phi)} = \frac{cos^2(\phi) - sin^2(\phi)}{2sin(\phi)cos(\phi)} \quad (79)$$

Combining equations (78) and (79) defines the required Jacobi rotation angle as:

$$\theta = cot(2\phi) = \frac{c^2 - s^2}{2sc} = \frac{a_{q,q} - a_{p,p}}{2a_{p,q}} \quad (80)$$

Setting $t = tan\ (\phi)$ gives:

$$t = tan(\phi) = \frac{sin(\phi)}{cos(\phi)} = \frac{s}{c} \quad (81)$$

$$\theta = \frac{c^2 - s^2}{2sc} = \frac{1 - \left(\frac{s}{c}\right)^2}{2\left(\frac{s}{c}\right)} = \frac{1 - t^2}{2t} \Rightarrow 1 - t^2 = 2\theta t \Rightarrow t^2 + 2\theta t - 1 = 0 \quad (82)$$

$$t = \frac{-2\theta \pm \sqrt{4\theta^2 + 4}}{2} = -\theta \pm \sqrt{\theta^2 + 1} \quad (83)$$

$$t = -\theta + \sqrt{\theta^2 + 1} = \frac{(-\theta + \sqrt{\theta^2 + 1})(-\theta - \sqrt{\theta^2 + 1})}{(-\theta - \sqrt{\theta^2 + 1})} = \frac{-1}{(-\theta - \sqrt{\theta^2 + 1})} \quad (84)$$

$$t = -\theta - \sqrt{\theta^2 + 1} = \frac{(-\theta - \sqrt{\theta^2 + 1})(-\theta + \sqrt{\theta^2 + 1})}{(-\theta + \sqrt{\theta^2 + 1})} = \frac{-1}{(-\theta + \sqrt{\theta^2 + 1})} \quad (85)$$

For θ negative, the smaller magnitude solution is:

$$t = \frac{-1}{\left(-\theta + \sqrt{\theta^2 + 1}\right)} = \frac{sgn(\theta)}{\left(|\theta| + \sqrt{\theta^2 + 1}\right)} \quad (86)$$

For θ positive, the smaller magnitude solution is:

$$t = \frac{-1}{\left(-\theta - \sqrt{\theta^2 + 1}\right)} = \frac{1}{\left(\theta + \sqrt{\theta^2 + 1}\right)} = \frac{sgn(\theta)}{\left(|\theta| + \sqrt{\theta^2 + 1}\right)} \quad (87)$$

In both cases:

$$t = \frac{sgn(\theta)}{\left(|\theta| + \sqrt{\theta^2 + 1}\right)} \quad (88)$$

If θ is so large that θ squared would overflow, the alternative is:

$$t = \frac{sgn(\theta)}{2|\theta|} = \frac{-1}{2\theta} \quad (89)$$

To avoid roundoff error, an iterative update is used to replace equations (73) to (77) :

$$a'_{p,q} = (c^2 - s^2)a_{p,q} + sc\left(a_{p,p} - a_{q,q}\right) = 0 \quad (90)$$

$$\Rightarrow (c^2 - s^2)a_{p,q} + sca_{p,p} = sca_{q,q} \Rightarrow a_{q,q} = \left\{\frac{(c^2 - s^2)a_{p,q} + sca_{p,p}}{sc}\right\} \quad (91)$$

$$a'_{p,p} = c^2 a_{p,p} + s^2 a_{q,q} - 2sca_{p,q} = c^2 a_{p,p} + s\left\{\frac{(c^2 - s^2)a_{p,q} + sca_{p,p}}{c}\right\} - 2sca_{p,q} \quad (92)$$

$$= c^2 a_{p,p} + \left(\frac{s(c^2 - s^2)a_{p,q}}{c}\right) + s^2 a_{p,p} - 2sca_{p,q} = a_{p,p} + \left(\frac{s(c^2 - s^2)a_{p,q} - 2sc^2 a_{p,q}}{c}\right) \quad (93)$$

$$= a_{p,p} + \left(\frac{-s^3 a_{p,q} - sc^2 a_{p,q}}{c}\right) = a_{p,p} - a_{p,q}\left(\frac{s^3 + sc^2}{c}\right) = a_{p,p} - a_{p,q}\left(\frac{s}{c}\right) = a_{p,p} - ta_{p,q} \quad (94)$$

$$a'_{p,p} = a_{p,p} - ta_{p,q} \quad (95)$$

$$a'_{q,q} = s^2 a_{p,p} + c^2 a_{q,q} + 2sca_{p,q} \quad (96)$$

$$(c^2 - s^2)a_{p,q} + sc\left(a_{p,p} - a_{q,q}\right) = 0 \quad (97)$$

$$a_{p,p} = \frac{sca_{q,q} - (c^2 - s^2)a_{p,q}}{sc} = a_{q,q} - \left\{\frac{(c^2 - s^2)a_{p,q}}{sc}\right\} \quad (98)$$

$$a'_{q,q} = s^2\left\{a_{q,q} - \left\{\frac{(c^2 - s^2)a_{p,q}}{sc}\right\}\right\} + c^2 a_{q,q} + 2sca_{p,q} \quad (99)$$

$$= a_{q,q} - \left\{\frac{s(c^2 - s^2) - 2sc^2}{c}\right\}a_{p,q} = a_{q,q} + \left\{\frac{s^3 + sc^2}{c}\right\}a_{p,q} = a_{q,q} + \left(\frac{s}{c}\right)a_{p,q} = a_{q,q} + ta_{p,q} \quad (100)$$

$$a'_{q,q} = a_{q,q} + ta_{p,q} \quad (101)$$

Table continues on the next page...

$$a'_{r,p} = ca_{r,p} - sa_{r,q} = a_{r,p} - (1-c)a_{r,p} - sa_{r,q} = a_{r,p} - s\left\{a_{r,q} + \frac{(1-c)(1+c)}{s(1+c)}a_{r,p}\right\} \quad (102)$$

$$a'_{r,p} = a_{r,p} - s\left\{a_{r,q} + \frac{s}{(1+c)}a_{r,p}\right\} = a_{r,p} - s(a_{r,q} + \tau a_{r,p}) \quad (103)$$

$$\tau = \frac{s}{(1+c)} = \frac{sin(\phi)}{(1+cos(\phi))} = \frac{2sin\left(\frac{\phi}{2}\right)cos\left(\frac{\phi}{2}\right)}{2cos^2\left(\frac{\phi}{2}\right)} = tan\left(\frac{\phi}{2}\right) \quad (104)$$

$$a'_{r,q} = ca_{r,q} + sa_{r,p} = a_{r,q} - (1-c)a_{r,q} + sa_{r,p} = a_{r,q} + s\left\{a_{r,p} - \frac{(1-c)(1+c)a_{r,q}}{s(1+c)}\right\} \quad (105)$$

$$a'_{r,q} = a_{r,q} + s\left\{a_{r,p} - \frac{sa_{r,q}}{(1+c)}\right\} = a_{r,q} + s(a_{r,p} - \tau a_{r,q}) \quad (106)$$

6.3 Function eigencompute

Function `eigencompute` implement the Jacobi rotation solution described in this section.

```
// function computes all eigenvalues and eigenvectors of a real symmetric matrix m[0..n-1]
[0..n-1]
// m[][] is unchanged on output.
// eigval[0..n-1] returns the eigenvalues of m[][].
// eigvec[0..n-1][0..n-1] returns the normalized eigenvectors of m[][]
// the eigenvectors are not sorted by value.
void eigencompute(float **m, int32 n, float **eigval, float **eigvec)
{
  // maximum number of iterations to achieve convergence: in practice 6 is typical
#define NITERATIONS 15

  // matrix row and column indices
  int32 ir, ic;
  // general loop counter
  int32 j;
  // various trig functions of the jacobi rotation angle phi
  float cot2phi, tanhalfphi, tanphi, sinphi, cosphi;
  // scratch variable to prevent over-writing during rotations
  float ftmp;
  // residue from remaining non-zero above diagonal terms
  float residue;
  // timeout ctr for number of passes of the algorithm
  int32 ctr;

  // initialise eigenvectors matrix and eigenvalues array
  for (ir = 0; ir < n; ir++)
  {
    // loop over all columns
    for (ic = 0; ic < n; ic++)
    {
      // set on diagonal and off-diagonal elements to zero
      eigvec[ir][ic] = 0.0F;
    }

    // correct the diagonal elements to 1.0
    eigvec[ir][ir] = 1.0F;

    // initialise the array of eigenvalues to the diagonal elements of m
    eigval[ir][0] = m[ir][ir];
  }

  // initialise the counter and loop until converged or NITERATIONS reached
```

```
ctr = 0;
do
{
  // compute the absolute value of the above diagonal elements as exit criterion
  residue = 0.0F;
  // loop over rows excluding last row
  for (ir = 0; ir < n - 1; ir++)
  {
    // loop over above diagonal columns
    for (ic = ir + 1; ic < n; ic++)
    {
      // accumulate the residual off diagonal terms which are being driven to zero
      residue += fabs(m[ir][ic]);
    }
  }

  // check if we still have work to do
  if (residue > 0.0F)
  {
    // loop over all rows with the exception of the last row
    // (since only rotating above diagonal elements)
    for (ir = 0; ir < n - 1; ir++)
    {
      // loop over columns ic (where ic is always greater than ir since above diagonal)
      for (ic = ir + 1; ic < n; ic++)
      {
        // only continue with this element if the element is non-zero
        if (fabs(m[ir][ic]) > 0.0F)
        {
          // calculate cot(2*phi) where phi is the Jacobi rotation angle
          cot2phi = 0.5F * (eigval[ic][0] - eigval[ir][0]) / (m[ir][ic]);

          // calculate tan(phi) correcting sign to ensure the smaller solution is used
          tanphi = 1.0F / (float) (fabs(cot2phi) + sqrt(1.0F + cot2phi * cot2phi));
          if (cot2phi < 0.0F)
          {
            tanphi = -tanphi;
          }

          // calculate the sine and cosine of the Jacobi rotation angle phi
          cosphi = 1.0F / (float)sqrt(1.0F + tanphi * tanphi);
          sinphi = tanphi * cosphi;

          // calculate tan(phi/2)
          tanhalfphi = sinphi / (1.0F + cosphi);

          // set tmp = tan(phi) * current matrix element
          // used in update of leading diagonal elements
          ftmp = tanphi * m[ir][ic];

          // apply the jacobi rotation to diagonal elements [ir][ir] and [ic][ic]
          // stored in the eigenvalue array
          // eigval[ir] = eigval[ir] - tan(phi) * m[ir][ic]
          eigval[ir][0] -= ftmp;
          // eigval[ic] = eigval[ic] + tan(phi) * m[ir][ic]
          eigval[ic][0] += ftmp;

          // by definition, applying the jacobi rotation on element ir, ic results in 0.0
          m[ir][ic] = 0.0F;

          // apply the jacobi rotation to all elements of the eigenvector matrix
          for (j = 0; j < n; j++)
          {
            // store eigvec[j][ir]
            ftmp = eigvec[j][ir];
            // eigvec[j][ir] = eigvec[j][ir] - sin(phi) *
            // (eigvec[j][ic] + tan(phi/2) * eigvec[j][ir])
            eigvec[j][ir] = ftmp - sinphi * (eigvec[j][ic] + tanhalfphi * ftmp);
            // eigvec[j][ic] = eigvec[j][ic] + sin(phi) *
            // (eigvec[j][ir] - tan(phi/2) * eigvec[j][ic])
```

```
            eigvec[j][ic] = eigvec[j][ic] + sinphi * (ftmp - tanhalfphi * eigvec[j][ic]);
          }

          // apply the jacobi rotation only to those elements of matrix m that can change
          for (j = 0; j <= ir - 1; j++)
          {
            // store m[j][ir]
            ftmp = m[j][ir];
            // m[j][ir] = m[j][ir] - sin(phi) * (m[j][ic] + tan(phi/2) * m[j][ir])
            m[j][ir] = ftmp - sinphi * (m[j][ic] + tanhalfphi * ftmp);
            // m[j][ic] = m[j][ic] + sin(phi) * (m[j][ir] - tan(phi/2) * m[j][ic])
            m[j][ic] = m[j][ic] + sinphi * (ftmp - tanhalfphi * m[j][ic]);
          }
          for (j = ir + 1; j <= ic - 1; j++)
          {
            // store m[ir][j]
            ftmp = m[ir][j];
            // m[ir][j] = m[ir][j] - sin(phi) * (m[j][ic] + tan(phi/2) * m[ir][j])
            m[ir][j] = ftmp - sinphi * (m[j][ic] + tanhalfphi * ftmp);
            // m[j][ic] = m[j][ic] + sin(phi) * (m[ir][j] - tan(phi/2) * m[j][ic])
            m[j][ic] = m[j][ic] + sinphi * (ftmp - tanhalfphi * m[j][ic]);
          }
          for (j = ic + 1; j < n; j++)
          {
            // store m[ir][j]
            ftmp = m[ir][j];
            // m[ir][j] = m[ir][j] - sin(phi) * (m[ic][j] + tan(phi/2) * m[ir][j])
            m[ir][j] = ftmp - sinphi * (m[ic][j] + tanhalfphi * ftmp);
            // m[ic][j] = m[ic][j] + sin(phi) * (m[ir][j] - tan(phi/2) * m[ic][j])
            m[ic][j] = m[ic][j] + sinphi * (ftmp - tanhalfphi * m[ic][j]);
          }
        }   // end of test for matrix element already zero
      }   // end of loop over columns
    }   // end of loop over rows
  }  // end of test for non-zero residue
} while ((residue > 0.0F) && (ctr++ < NITERATIONS)); // end of main loop

  // recover the above diagonal matrix elements from unaltered below diagonal elements
  // this code can be deleted if the input matrix m[][] is not required to be unaltered on
exit
  for (ir = 0; ir < n - 1; ir++)
  {
    // loop over above diagonal columns
    for (ic = ir + 1; ic < n; ic++)
    {
      // copy below diagonal element to above diagonal element
      m[ir][ic] = m[ic][ir];
    }
  }
  return;
}
```

7 Seven-Element Magnetic Calibration

This section documents the C code implementation of the 7-element calibration algorithm but does not provide the mathematical derivation of the algorithm. The 7-element calibration algorithm expands on the 4-element calibration algorithm by solving for the three diagonal elements of the soft iron matrix. The off-diagonal soft iron matrix values remain zero.

7.1 Eigenvector least squares solution

The 7-element magnetic calibration can be written as a function of the vector $\boldsymbol{\beta}$ which is a eigenvector solution with eigenvalue λ of the form of equation (38) to a set of homogeneous linear equations. The derivation of equation (107) and subsequent equations is proprietary to Freescale and will not be released.

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta} = \lambda\boldsymbol{\beta} \tag{107}$$

where:

$$\boldsymbol{\beta} = \begin{pmatrix}\beta_0\\ \beta_1\\ \beta_2\\ \beta_3\\ \beta_4\\ \beta_5\\ \beta_6\end{pmatrix} = \begin{pmatrix} A_{xx}\\ A_{yy}\\ A_{zz}\\ -2A_{xx}V_x\\ -2A_{yy}V_y\\ -2A_{zz}V_z\\ A_{xx}{V_x}^2 + A_{yy}{V_y}^2 + A_{zz}{V_z}^2 - B^2\end{pmatrix} \tag{108}$$

$\boldsymbol{X}$ is the M by 7 measurement matrix constructed from M magnetometer vector measurements:

$$\boldsymbol{X} = \begin{pmatrix} B_{px}[0]^2 & B_{py}[0]^2 & B_{pz}[0]^2 & B_{px}[0] & B_{py}[0] & B_{pz}[0] & 1\\ B_{px}[1]^2 & B_{py}[1]^2 & B_{pz}[1]^2 & B_{px}[1] & B_{py}[1] & B_{pz}[1] & 1\\ \dots & \dots & \dots & \dots & \dots & \dots & 1\\ B_{px}[M-1]^2 & B_{py}[M-1]^2 & B_{pz}[M-1]^2 & B_{px}[M-1] & B_{py}[M-1] & B_{pz}[M-1] & 1\end{pmatrix} \tag{109}$$

$\boldsymbol{A}$ is the magnetic calibration ellipsoid fit matrix defined by equation (7) as the transposed product of the inverse soft iron matrix with itself $\boldsymbol{A} = \{\boldsymbol{W}^{-1}\}^T\boldsymbol{W}^{-1}$:

The transposed product $\boldsymbol{X}^T\boldsymbol{X}$ evaluates to:

$$\boldsymbol{X}^T\boldsymbol{X} = \sum_{i=0}^{M-1}\begin{pmatrix} B_{px}[i]^4 & B_{px}[i]^2B_{py}[i]^2 & B_{px}[i]^2B_{pz}[i]^2 & B_{px}[i]^3 & B_{px}[i]^2B_{py}[i] & B_{px}[i]^2B_{pz}[i] & B_{px}[i]^2\\ B_{py}[i]^2B_{px}[i]^2 & B_{py}[i]^4 & B_{py}[i]^2B_{pz}[i]^2 & B_{py}[i]^2B_{px}[i] & B_{py}[i]^3 & B_{py}[i]^2B_{pz}[i] & B_{py}[i]^2\\ B_{pz}[i]^2B_{px}[i]^2 & B_{pz}[i]^2B_{py}[i]^2 & B_{pz}[i]^4 & B_{pz}[i]^2B_{px}[i] & B_{pz}[i]^2B_{py}[i] & B_{pz}[i]^3 & B_{pz}[i]^2\\ B_{px}[i]^3 & B_{px}[i]B_{py}[i]^2 & B_{px}[i]B_{pz}[i]^2 & B_{px}[i]^2 & B_{px}[i]B_{py}[i] & B_{px}[i]B_{pz}[i] & B_{px}[i]\\ B_{py}[i]B_{px}[i]^2 & B_{py}[i]^3 & B_{py}[i]B_{pz}[i]^2 & B_{py}[i]B_{px}[i] & B_{py}[i]^2 & B_{py}[i]B_{pz}[i] & B_{py}[i]\\ B_{pz}[i]B_{px}[i]^2 & B_{pz}[i]B_{py}[i]^2 & B_{pz}[i]^3 & B_{pz}[i]B_{px}[i] & B_{pz}[i]B_{py}[i] & B_{pz}[i]^2 & B_{pz}[i]\\ B_{px}[i]^2 & B_{py}[i]^2 & B_{pz}[i]^2 & B_{px}[i] & B_{py}[i] & B_{pz}[i] & 1\end{pmatrix} \tag{110}$$

Equation (39) gives the performance function P_i resulting from the solution vector with eigenvalue λ_i and eigenvector $\boldsymbol{\beta}_i$:

$$P_i = \lambda_i \tag{111}$$

The M by M matrix $\boldsymbol{X}^T\boldsymbol{X}$ has M eigenvalues and eigenvectors. The required solution which minimizes the fit error is, according to equation (111), the eigenvector associated with the smallest eigenvalue.

7.2 Hard iron and soft iron calibration

The soft iron model is given directly by equation (108) as:

$$\boldsymbol{A} = \begin{pmatrix} A_{xx} & 0 & 0 \\ 0 & A_{yy} & 0 \\ 0 & 0 & A_{zz} \end{pmatrix} = \begin{pmatrix} \beta_0 & 0 & 0 \\ 0 & \beta_1 & 0 \\ 0 & 0 & \beta_2 \end{pmatrix} \tag{112}$$

The inverse soft iron matrix by easily be found from the square root of the soft iron components as:

$$\boldsymbol{W}^{-1} = \sqrt{\boldsymbol{A}} = \begin{pmatrix} A_{xx} & 0 & 0 \\ 0 & A_{yy} & 0 \\ 0 & 0 & A_{zz} \end{pmatrix} = \begin{pmatrix} \sqrt{\beta_0} & 0 & 0 \\ 0 & \sqrt{\beta_1} & 0 \\ 0 & 0 & \sqrt{\beta_2} \end{pmatrix} \tag{113}$$

The hard iron model is also given by equation (108) as:

$$\begin{pmatrix} -2A_{xx}V_x \\ -2A_{yy}V_y \\ -2A_{zz}V_z \end{pmatrix} = \begin{pmatrix} \beta_3 \\ \beta_4 \\ \beta_5 \end{pmatrix} \Rightarrow \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} = \begin{pmatrix} \left(\frac{-\beta_3}{2A_{xx}}\right) \\ \left(\frac{-\beta_4}{2A_{yy}}\right) \\ \left(\frac{-\beta_5}{2A_{zz}}\right) \end{pmatrix} = \begin{pmatrix} \left(\frac{-\beta_3}{2\beta_0}\right) \\ \left(\frac{-\beta_4}{2\beta_1}\right) \\ \left(\frac{-\beta_5}{2\beta_2}\right) \end{pmatrix} \tag{114}$$

7.3 Fit error

Since $P = \boldsymbol{r}^T\boldsymbol{r}$ and $\boldsymbol{r}$ has M elements with each element having dimensions B^2, a suitable normalized and dimensionless fit error ε can be defined as:

$$\varepsilon = \left(\frac{1}{B}\right)\sqrt{\frac{P}{4MB^2}} = \frac{1}{2B^2}\sqrt{\frac{\lambda_{min}}{M}} \tag{115}$$

7.4 Geomagnetic field strength

The geomagnetic field strength B is given by the last component of equation (108) :

$$\beta_6 = A_{xx}{V_x}^2 + A_{yy}{V_y}^2 + A_{zz}{V_z}^2 - B^2 \tag{116}$$

$$\Rightarrow B^2 = A_{xx}{V_x}^2 + A_{yy}{V_y}^2 + A_{zz}{V_z}^2 - \beta_6 \tag{117}$$

Since the solution vector and ellipsoid matrix elements may be negated in software to force a positive determinant, it is necessary to compute the geomagnetic field from the absolute value of the right hand side of equation (117).

$$B = \sqrt{\left|A_{xx}{V_x}^2 + A_{yy}{V_y}^2 + A_{zz}{V_z}^2 - \beta_6\right|} \quad (118)$$

The fitted geomagnetic field strength B and the elements of the soft iron matrix both determine the overall magnitude of the measured magnetometer signal and cannot be separated from each other. It is conventional therefore to normalize the inverse soft iron matrix to unit determinant and compute the geomagnetic field on the assumption that the gain (determinant) of the soft iron matrix is unity.

7.5 Function fUpdateCalibration7EIG

The function `fUpdateCalibration7EIG` implements the 7-element calibration algorithm.

```
    // 7 element calibration using direct eigen-decomposition
    void fUpdateCalibration7EIG(struct MagCalibration *pthisMagCal,
      struct MagneticBuffer *pthisMagneticBuffer,
      float **ftmpA7x7, float **ftmpB7x7, float **ftmpA7x1)
    {
      // local variables
      int32 i, j, k, l, m, n;              // loop counters
      int32 ilocalMagBufferCount;          // local count of measurements for this process
      float fOffsetx, fOffsety, fOffsetz;  // offset to remove large DC hard iron bias
      float det;                           // matrix determinant
      float fscaling;                      // set to FUTPERCOUNT * FMATRIXSCALING

      // compute fscaling to reduce multiplications later
      fscaling = FUTPERCOUNT * FMATRIXSCALING;

      // the offsets are guaranteed to be set from the first element but to avoid compiler
error
      fOffsetx = fOffsety = fOffsetz = 0.0F;

      // zero the on and above diagonal elements of the 7x7 symmetric measurement matrix
ftmpA7x7
      for (m = 0; m < 7; m++)
      {
          for (n = m; n < 7; n++)
          {
              ftmpA7x7[m][n] = 0.0F;
          }
      }

      // last entry of vector ftmpA7x1 is always 1.0F so move assignment outside the loop
      ftmpA7x1[6][0] = 1.0F;

      // place from MINEQUATIONS to MAXEQUATIONS entries into product matrix ftmpA7x7
      i = 0;
      for (j = 0; j < MAGBUFFSIZE; j++)
      {
          for (k = 0; k < MAGBUFFSIZE; k++)
          {
              for (l = 0; l < MAGBUFFSIZE; l++)
              {
                  if (pthisMagneticBuffer->index[j][k][l] != -1)
                  {
                      // use first valid magnetic buffer entry as offset estimate (bit
counts)
                      if (i == 0)
                      {
                          fOffsetx = (float)pthisMagneticBuffer->iBx[j][k][l];
                          fOffsety = (float)pthisMagneticBuffer->iBy[j][k][l];
                          fOffsetz = (float)pthisMagneticBuffer->iBz[j][k][l];
```

```
                    }

                    // apply the offset, scaling and conversion to uT
                    ftmpA7x1[3][0] = ((float)pthisMagneticBuffer->iBx[j][k][l] - fOffsetx)
* fscaling;
                    ftmpA7x1[4][0] = ((float)pthisMagneticBuffer->iBy[j][k][l] - fOffsety)
* fscaling;
                    ftmpA7x1[5][0] = ((float)pthisMagneticBuffer->iBz[j][k][l] - fOffsetz)
* fscaling;

                    // compute the remaining members of the measurement vector ftmpA7x1
                    ftmpA7x1[0][0] = ftmpA7x1[3][0] * ftmpA7x1[3][0];
                    ftmpA7x1[1][0] = ftmpA7x1[4][0] * ftmpA7x1[4][0];
                    ftmpA7x1[2][0] = ftmpA7x1[5][0] * ftmpA7x1[5][0];

                    // accumulate the on-and above-diagonal terms of ftmpA7x7
                    // = X^T*X = Sigma{ftmpA7x1^T * ftmpA7x1}
                    for (m = 0; m < 7; m++)
                    {
                        for (n = m; n < 7; n++)
                        {
                            ftmpA7x7[m][n] += ftmpA7x1[m][0] * ftmpA7x1[n][0];
                        }
                    }

                    // increment the counter for the next iteration
                    i++;
                }
            }
        }
    }

    // store the number of measurements accumulated
    ilocalMagBufferCount = i;

    // copy the above diagonal elements of ftmpA7x7 to below the diagonal
    for (m = 1; m < 7; m++)
    {
        for (n = 0; n < m; n++)
        {
            ftmpA7x7[m][n] = ftmpA7x7[n][m];
        }
    }

    // set tmpA7x1 to the unsorted eigenvalues and ftmpB7x7 to the unsorted eigenvectors
of ftmpA7x7
    eigencompute(ftmpA7x7, 7, ftmpA7x1, ftmpB7x7);

    // set ellipsoid matrix A to the solution vector column j with smallest eigenvalue
    j = 0;
    for (i = 1; i < 7; i++)
    {
        if (ftmpA7x1[i][0] < ftmpA7x1[j][0])
        {
            j = i;
        }
    }
    pthisMagCal->fA[0][0] = ftmpB7x7[0][j];
    pthisMagCal->fA[1][1] = ftmpB7x7[1][j];
    pthisMagCal->fA[2][2] = ftmpB7x7[2][j];
    pthisMagCal->fA[0][1] = pthisMagCal->fA[0][2] = pthisMagCal->fA[1][0] = 0.0F;
    pthisMagCal->fA[1][2] = pthisMagCal->fA[2][0] = pthisMagCal->fA[2][1] = 0.0F;

    // compute the trial hard iron vector in offset and scaled uT
    pthisMagCal->ftrVx = -0.5F * ftmpB7x7[3][j] / pthisMagCal->fA[0][0];
    pthisMagCal->ftrVy = -0.5F * ftmpB7x7[4][j] / pthisMagCal->fA[1][1];
    pthisMagCal->ftrVz = -0.5F * ftmpB7x7[5][j] / pthisMagCal->fA[2][2];

    // negate A if it has negative determinant
    det = pthisMagCal->fA[0][0] * pthisMagCal->fA[1][1] * pthisMagCal->fA[2][2];
```

```
      if (det < 0.0F)
      {
          fmatrixAeqMinusA(pthisMagCal->fA, 3, 3);
          ftmpB7x7[6][j] = -ftmpB7x7[6][j];
          det = -det;
      }

      // compute the trial geomagnetic field strength B in bit counts times FMATRIXSCALING
      pthisMagCal->ftrB = (float)sqrt(fabs(pthisMagCal->fA[0][0] * pthisMagCal->ftrVx *
pthisMagCal->ftrVx +
          pthisMagCal->fA[1][1] * pthisMagCal->ftrVy * pthisMagCal->ftrVy +
          pthisMagCal->fA[2][2] * pthisMagCal->ftrVz * pthisMagCal->ftrVz - ftmpB7x7[6][j]));

      // calculate the trial normalized fit error as a percentage
      pthisMagCal->ftrFitErrorpc = 100.0F * (float) sqrt(fabs(ftmpA7x1[j][0]) /
          (double) ilocalMagBufferCount) / (2.0F * pthisMagCal->ftrB * pthisMagCal->ftrB);
      printf("\n\nTrial new calibration fit error=%9.4f%% versus previous %9.4f%%",
          pthisMagCal->ftrFitErrorpc, pthisMagCal->fFitErrorpc);

      // correct for the measurement matrix offset and scaling and get the computed trial
hard iron offset in uT
      pthisMagCal->ftrVx = pthisMagCal->ftrVx * FINVMATRIXSCALING + fOffsetx * FUTPERCOUNT;
      pthisMagCal->ftrVy = pthisMagCal->ftrVy * FINVMATRIXSCALING + fOffsety * FUTPERCOUNT;
      pthisMagCal->ftrVz = pthisMagCal->ftrVz * FINVMATRIXSCALING + fOffsetz * FUTPERCOUNT;
      printf("\n\nTrial new calibration hard iron (uT) Vx=%9.3f Vy=%9.3f Vz=%9.3f",
          pthisMagCal->ftrVx, pthisMagCal->ftrVy, pthisMagCal->ftrVz);

      // convert the geomagnetic field strength B into uT for current soft iron matrix A
      pthisMagCal->ftrB *= FINVMATRIXSCALING;

      // normalize the ellipsoid matrix A to unit determinant and correct B by root of this
multiplicative factor
      fmatrixAeqAxScalar(pthisMagCal->fA, (float)pow((double)det, (double) (-1.0F / 3.0F)),
3, 3);
      pthisMagCal->ftrB *= (float)pow((double)det, (double) (-1.0F / 6.0F));
      printf("\n\nTrial new calibration geomagnetic field (uT) B=%9.3f", pthisMagCal->ftrB);
      printf("\n\nTrial new calibration ellipsoid matrix A (normalized)");
      fmatrixPrintA(pthisMagCal->fA, 0, 2, 0, 2);

      // compute trial invW from the square root of A also with normalized determinant
      pthisMagCal->ftrinvW[0][0] = (float)sqrt(fabs(pthisMagCal->fA[0][0]));
      pthisMagCal->ftrinvW[1][1] = (float)sqrt(fabs(pthisMagCal->fA[1][1]));
      pthisMagCal->ftrinvW[2][2] = (float)sqrt(fabs(pthisMagCal->fA[2][2]));
      pthisMagCal->ftrinvW[0][1] = pthisMagCal->ftrinvW[0][2] = pthisMagCal->ftrinvW[1][0] =
0.0F;
      pthisMagCal->ftrinvW[1][2] = pthisMagCal->ftrinvW[2][0] = pthisMagCal->ftrinvW[2][1] =
0.0F;
      printf("\n\nTrial new calibration inverse soft iron matrix invW (normalized)");
      fmatrixPrintA(pthisMagCal->ftrinvW, 0, 2, 0, 2);

      // set the valid calibration flag to true
      pthisMagCal->iValidMagCal = 1;
      return;
   }
```

8 Ten-Element Magnetic Calibration

The highest performance Xtrinsic magnetic calibration algorithm fits a 10-parameter model to the magnetometer buffer by extending the soft iron matrix to be a symmetric matrix with off-diagonal terms. Neither the mathematical derivation of this algorithm nor the source code are publicly released by Freescale.

8.1 Object code

The file calibration10_c.obj in the distribution file contains object code compiled for the Thumb2 instruction set for ARM processors. The object code is available to customers by sending a request to **sfusion@freescale.com**.

8.2 Function prototype

The function prototype for the 10-element calibration is identical to that for the 7-element calibration with the exception of the temporary arrays required.

```
// 10 element calibration using direct eigen-decomposition
void fUpdateCalibration10EIG(struct MagCalibration *pthisMagCal,
  struct MagneticBuffer *pthisMagneticBuffer,
  float **ftmpA10x10, float **ftmpB10x10,
  float **ftmpA10x1, float **ftmpA3x3, float **ftmpA3x1)
```

The sizes of the temporary arrays are indicated in their names. The matrices are passed as arrays of pointers to one dimensional arrays in the same manner as for the 7-element calibration. The initialization of these arrays follows the approach used elsewhere in the software.

```
float xftmpA10x10[10][10], *ftmpA10x10[10]; // scratch 10x10 matrix
float xftmpB10x10[10][10], *ftmpB10x10[10]; // scratch 10x10 matrix
float xftmpA10x1[10][1], *ftmpA10x1[10];    // scratch 10x1 matrix
float xftmpA3x3[3][3], *ftmpA3x3[3];        // scratch 3x3 matrix
float xftmpA3x1[3][1], *ftmpA3x1[3];        // scratch 3x1 vector

// 10 row arrays
for (i = 0; i < 10; i++)
{
  ftmpA10x10[i] = xftmpA10x10[i];
  ftmpB10x10[i] = xftmpB10x10[i];
  ftmpA10x1[i] = xftmpA10x1[i];
}

// 3 row arrays
for (i = 0; i < 3; i++)
{
  ftmpA3x3[i] = xftmpA3x3[i];
  ftmpA3x1[i] = xftmpA3x1[i];
}
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4684
Rev. 1.0, 4/2013