DT0059
Design tip

Ellipsoid or sphere fitting for sensor calibration

By Andrea Vitali

Main components	
LSM303AGR	Ultra compact high-performance e-compass: ultra-low-power 3D accelerometer and 3D magnetometer
LSM6DS3	iNEMO inertial module: 3D accelerometer and 3D gyroscope

Purpose and benefits

This design tip explains how to compute offsets, gains, and cross-axis gains for a 3-axis sensor by performing a sphere (ellipsoid) fitting. The technique is typically used to calibrate and compensate magnetometers, but it can also be used with other sensors, such as accelerometers.

Benefits:

- Added functionality with respect to calibration provided by the MotionFX library which only provides offsets for the Magnetometer.
- Short and essential implementation, which enables easy customization and enhancement by the end-user (osxMotionFX is available only in binary format, not as source code)
- Easy to use on every microcontroller (osxMotionFX can only be run on the STM32 and only when the proper license has been issued by Open.MEMS license server).

Algorithm description

Measurements are taken on a number of positions (N) and combined to find the unknowns (offsets, gains and cross-axis gains).

For 6-tumble calibration, positioning the sensor accurately is required. However, for the ellipsoid fitting described here, there is no need to know the true stimulus of the sensor, as the only requirement is that the modulus of the true stimulus be constant (square root of sum of squares of X, Y, and Z).

- For the case of the magnetometer: in order to measure only the earth magnetic field, any other spurious (and often time varying) magnetic anomalies must be absent; the modulus of the true stimulus is then the modulus of the earth magnetic field

- For the case of the accelerometer: in order to measure only the gravity, the sensor must not be subject to any other acceleration; the modulus of the true stimulus is then the modulus of the gravity

In the most general case, the following equation has 9 unknowns, $\mathbf{v} = [\, a, b, c, d, e, f, g, h, i \,]^T$ with the data points being on a rotated ellipsoid. If the ellipsoid is not rotated, the axis will be aligned with X, Y and Z, where the corresponding equation has only 6 unknowns $\mathbf{v} = [\, a, b, c, g, h, i \,]^T$. If the axes are all the same length, then it is a sphere, and the corresponding equation has only 4 unknowns $\mathbf{v} = [\, a+b+c, g, h, i \,]^T$. The general equation is the following:

$$a X^2 + b Y^2 + c Z^2 + d\, 2XY + e\, 2XZ + f\, 2YZ + g\, 2X + h\, 2Y + i\, 2Z = 1$$

The set of N data points is used to build a data matrix D where the data points must not be co-planar:

- Rotated ellipsoid: line of $\mathbf{D} = [X^2, Y^2, Z^2, 2XY, 2XZ, 2YZ, 2X, 2Y, 2Z]$, where D is [Nx9]. At least 9 data points are needed to compute offsets, gains and cross-axis gains
- Non-rotated ellipsoid: line of $\mathbf{D} = [X^2, Y^2, Z^2, 2X, 2Y, 2Z]$, **where** D is [Nx6], At least 6 data points are needed to compute offsets and gains
- Sphere: line of $\mathbf{D} = [X^2+Y^2+Z^2, 2X, 2Y, 2Z]$, where D is [Nx4]. At least 4 data points are needed to compute offsets

Rotated ellipsoid fitting

Now, the least-square error approximation can be computed for the unknowns in **v** by using the pseudo-inverse of the non-square matrix. First, both sides are multiplied by the transpose $\mathbf{D}^T$. Second, both sides are multiplied by the inverse of the square matrix $\mathbf{D}\,\mathbf{D}^T$. There can be 9, 6 or 4 unknowns, depending on the aforementioned constraints. For the most general case:

$$D[Nx9]\; v[9x1] = 1\;[Nx1] \rightarrow D^T[9xN]\; D[Nx9]\; v[9x1] = D^T[9xN]\; 1[Nx1] \rightarrow$$

$$(D^T\, D)[9x9]\; v[9x1] = (D^T\, 1)[9x1] \rightarrow v[9x1] = inv(D^T\, D)[9x9]\; (D^T\, 1)[9x1]$$

Next, the auxiliary matrix $\mathbf{A_4[4x4]}$ and $\mathbf{A_3[3x3]}$, and the auxiliary vector $\mathbf{v_{ghi}[3x1]}$ are built using the unknowns **v[9x1]:**

$$v = [\, a, b, c, d, e, f, g, h, i \,]^T, \qquad v_{ghi} = [g\; h\; i]^T$$

$$A_4 = [\, a\; d\; e\; g;\;\; d\; b\; f\; h;\;\; e\; f\; c\; i;\;\; g\; h\; i\; {-1} \,], \qquad A_3 = [\, a\; d\; e;\; d\; b\; f;\; e\; f\; c]$$

Offsets $\mathbf{o} = (ox, oy, oz)$ can be computed as follows:

$$A_3[3x3]\; o[3x1] = -v_{ghi}[3x1] \rightarrow o[3x1] = -inv(A_3)[3x3]\; v_{ghi}[3x1]$$

Once the offsets are known, another auxiliary matrix $\mathbf{B_4[4x4]}$ is computed, which represents the ellipsoid translated into the origin:

$$T = [\,1\ 0\ 0\ 0;\ \ 0\ 1\ 0\ 0;\ 0\ 0\ 1\ 0;\ ox\ oy\ oz\ 1\,] \rightarrow B_4[4x4] = T[4x4]\ A[4x4]\ T^T[4x4],$$

$$B_4[4x4] = [\ b_{11}\ b_{12}\ b_{13}\ b_{14};\ b_{21}\ b_{22}\ b_{23}\ b_{24};\ b_{31}\ b_{32}\ b_{33}\ b_{34};\ b_{41}\ b_{42}\ b_{43}\ b_{44}\ ],$$

$$B_3[3x3] = [\ b_{11}\ b_{12}\ b_{13};\ b_{21}\ b_{22}\ b_{23};\ b_{31}\ b_{32}\ b_{33}\ ] / -b_{44}$$

Gains and cross-axis gains can be computed from eigenvalues and eigenvectors respectively of **B_3[3x3].**

- Ellipsoid radii are the square root of the inverse of the 3 eigenvalues; these are the axis gains **g = [gx, gy, gz]^T**
- Ellipsoid rotation matrix **R[3x3]** is obtained by juxtaposition of the 3 eigenvectors; gains and cross-axis gains are obtained by multiplying the 3x3 matrix where the diagonal contains the gains

Compensation of offsets, gains and cross-axis gains to map the data point **p = [x, y, z]^T** on the unit sphere can then be done in 3 steps:

1. Subtraction of the offsets, **p' = p - o = [x-ox, y-oy, z-oz]^T = [x', y', z']^T**
2. Multiplication by the inverse of the rotation matrix, **p" = p' inv(R) = [x", y", z"]^T**
3. Division by the gains, **p'"= [x"/gx, y"/gy, z"/gz]^T = [x'", y'", z'"]^T**

Non-rotated ellipsoid fitting

In this case, the data matrix **D[Nx6]** has only 6 columns and there are only 6 unknowns to be computed **v = [a, b, c, g, h, i]:**

$$D[Nx6]\ v[6x1] = 1\ [Nx1] \rightarrow D^T[6xN]\ D[Nx6]\ v[6x1] = D^T[6xN]\ 1[Nx1] \rightarrow$$

$$(D^T\ D)[6x6]\ v[6x1] = (D^T\ 1)[6x1] \rightarrow v[6x1] = inv(D^T\ D)[6x6]\ (D^T\ 1)[6x1]$$

Offsets **o = (oX, oY, oZ)** can be computed as follows:

$$o = [\ g/a,\ h/b,\ i/c\ ]^T$$

Gains **g = [gx, gy, gz]^T** can be computed as follows:

$$G = 1 + g^2/a + h^2/b + i^2/c \rightarrow g = [\ sqrt(a/G)\ sqrt(b/G)\ sqrt(c/G)\ ]^T$$

Sphere fitting

In this case, the data matrix **D[Nx4]** has only 4 columns and there are only 4 unknowns to be computed **v = [a+b+c, g, h, i]^T = [a", g, h, i]^T:**

$$D[Nx4]\ v[4x1] = 1\ [Nx1] \rightarrow D^T[4xN]\ D[Nx4]\ v[4x1] = D^T[4xN]\ 1[Nx1] \rightarrow$$

$$(D^T\ D)[4x4]\ v[4x1] = (D^T\ 1)[4x1] \rightarrow v[4x1] = inv(D^T\ D)[4x4]\ (D^T\ 1)[4x1]$$

Offsets **o = (oX, oY, oZ)** can be computed as follows:

$$\mathbf{o = [\ g/a'',\ h/a'',\ i/a''\ ]^T}$$

Gains **g = [gx, gy, gz]**^{T} can be computed as follows:

$$\mathbf{G = 1 + g^2/a'' + h^2/a'' + i^2/a'' \rightarrow g = [\ sqrt(a''/G)\ sqrt(a''/G)\ sqrt(a''/G)\ ]^T}$$

Notes

Hints for a compact real-time implementation on a microcontroller:

- Only the product $D^T[MxN]\ D[NxM]$ needs to be maintained in memory, this is a MxM matrix, M=9, 6 or 4; worst case is that 9x9=81 elements are to be maintained in memory
- Only the product $D^T[MxN]\ 1[Nx1]$ needs to be maintained in memory, this is a Mx1 vector, M=9, 6, or 4; worst case is that 9 elements are to be maintained in memory
- Gaussian elimination can be implemented to compute the inverse of the aforementioned MxM matrix when enough data points (at least M) have been collected
- For the case of the rotated ellipsoid fitting, eigenvalues and eigenvectors of a 3x3 matrix can be computed by using closed formulas
- For the case of a rotated ellipsoid when there is no or little rotation, the system does not easily converge to the correct solution; this is especially true if data points are affected by noise. If little or no rotation is expected (matrix R has small values out of the diagonal) and/or if data points are affected by a significant noise, the following alternate equation system is suggested:

$$\mathbf{D[Nx9] = [\ X^2+Y^2-2Z^2,\ X^2-2Y^2+Z^2,\ 4XY, 2XZ,\ 2YZ,\ 2X,\ 2Y,\ 2Z,\ 1\ ]}$$

$$\mathbf{E[Nx1] = [\ X^2+Y^2+Z^2\ ]}$$

$$\mathbf{D[Nx9]\ u[9x1] = E[Nx1] \rightarrow D^T[9xN]\ D[Nx9]\ u[9x1] = D^T[9xN]\ E[Nx1] \rightarrow}$$

$$\mathbf{(D^T\ D)[9x9]\ u[9x1] = (D^T\ 1)[9x1] \rightarrow u[9x1] = inv(D^T\ D)[9x9]\ (D^T\ E)[9x1]}$$

$$\mathbf{S'[3x3] = [\ 3, 1, 1;\ 3, 1, -2;\ 3, -2, 1\ ]}$$

$$\mathbf{S[10x10] = [\ S'[3x3], 0[3x7];\ 0[7x3]\ eye[7x7]\ ]} \text{ then } \mathbf{set\ s_{44} = 2}$$

$$\mathbf{v' = S[10x10]\ [\ -1/3;\ u[9x1]\ ] = [\ a', b', c', d', e', f', g', h', i', j'\ ]^T}$$

$$\mathbf{v = -\ [\ a', b', c', d', e', f', g', h', i'\ ]^T / j' = [\ a, b, c, d, e, f, g, h, i\ ]^T}$$

Then, the computation proceeds as before for the case of a rotated ellipsoid.

Rotation matrix refinement for optimal data compensation

As already mentioned, there are 3 steps to compensate for offsets, gains and cross-axis gains and map the data point on the unit sphere:

- Translate the center of the ellipsoid to the origin (0,0,0) by subtracting the offsets
- De-rotate the ellipsoid by multiplying by the inverse of the rotation matrix (the inverse is simply the transpose)
- Scale the de-rotated ellipsoid by multiplying by the inverse of the gains.

After the de-rotation, it can be seen that one could flip any axis and the ellipsoid would still map to the unit sphere: X can be exchanged with -X, and the same can be done with Y and Z. This means that the ellipsoid can be de-rotated so that a given axis is aligned with any of the reference axes in the positive or negative direction: X, -X, Y, -Y, Z or -Z.

One may want to impose the constraint that the de-rotation is the least possible: a given axis should be aligned with the nearest reference axis and no flipping should happen. This means that the rotation matrix has the largest coefficients along the diagonal and those coefficients are positive. This can be obtained by swapping and changing the sign of selected columns (or rows) of the rotation matrix (or its inverse which is the transpose). See below the reference code for this refinement step.

MatLab code for ellipsoid/sphere fitting

Reference implementation.

```
function [ofs,gain,rotM]=ellipsoid_fit(XYZ,varargin)
% Fit an (non)rotated ellipsoid or sphere to a set of xyz data points
% XYZ: N(rows) x 3(cols), matrix of N data points (x,y,z)
% optional flag f, default to 0 (fitting of rotated ellipsoid)
x=XYZ(:,1); y=XYZ(:,2); z=XYZ(:,3); if nargin>1, f=varargin{1}; else f=0; end;
if     f==0, D=[x.*x,  y.*y,  z.*z,  2*x.*y,2*x.*z,2*y.*z, 2*x,2*y,2*z]; % any axes (rotated ellipsoid)
elseif f==1, D=[x.*x,  y.*y,  z.*z,                        2*x,2*y,2*z]; % XYZ axes (non-rotated ellipsoid)
elseif f==2, D=[x.*x+y.*y,    z.*z,                        2*x,2*y,2*z]; %   and radius x=y
elseif f==3, D=[x.*x+z.*z,    y.*y,                        2*x,2*y,2*z]; %   and radius x=z
elseif f==4, D=[y.*y+z.*z,    x.*x,                        2*x,2*y,2*z]; %   and radius y=z
elseif f==5, D=[x.*x+y.*y+z.*z,                            2*x,2*y,2*z]; %   and radius x=y=z (sphere)
end;
v = (D'*D)\(D'*ones(length(x),1)); % least square fitting
if f==0, % rotated ellipsoid
  A = [ v(1) v(4) v(5) v(7); v(4) v(2) v(6) v(8); v(5) v(6) v(3) v(9); v(7) v(8) v(9) -1 ];
  ofs=-A(1:3,1:3)\[v(7);v(8);v(9)]; % offset is center of ellipsoid
  Tmtx=eye(4); Tmtx(4,1:3)=ofs'; AT=Tmtx*A*Tmtx'; % ellipsoid translated to (0,0,0)
  [rotM ev]=eig(AT(1:3,1:3)/-AT(4,4)); % eigenvectors (rotation) and eigenvalues (gain)
  gain=sqrt(1./diag(ev)); % gain is radius of the ellipsoid
else % non-rotated ellipsoid
  if     f==1, v = [ v(1) v(2) v(3) 0 0 0 v(4) v(5) v(6) ];
  elseif f==2, v = [ v(1) v(1) v(2) 0 0 0 v(3) v(4) v(5) ];
  elseif f==3, v = [ v(1) v(2) v(1) 0 0 0 v(3) v(4) v(5) ];
  elseif f==4, v = [ v(2) v(1) v(1) 0 0 0 v(3) v(4) v(5) ];
  elseif f==5, v = [ v(1) v(1) v(1) 0 0 0 v(2) v(3) v(4) ]; % sphere
  end;
  ofs=-(v(1:3).\v(7:9))'; % offset is center of ellipsoid
  rotM=eye(3); % eigenvectors (rotation), identity = no rotation
  g=1+(v(7)^2/v(1)+v(8)^2/v(2)+v(9)^2/v(3));
  gain=(sqrt(g./v(1:3)))'; % find radii of the ellipsoid (scale)
end;
```

Alternative implementation for near spherical data with little or no rotation

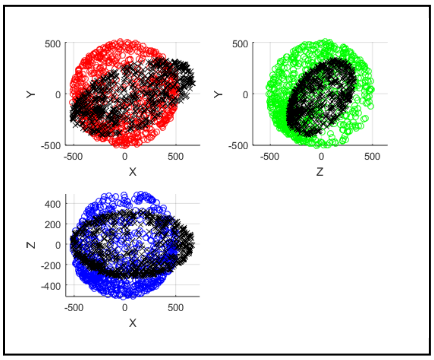
```
function [ofs,gain,rotM]=ellipsoid_fit(XYZ)
% Fit a rotated ellipsoid to a set of xyz data points
% XYZ: N(rows) x 3(cols), matrix of N data points (x,y,z)
x=XYZ(:,1); y=XYZ(:,2); z=XYZ(:,3);
x2=x.*x; y2=y.*y; z2=z.*z;
D = [x2+y2-2*z2, x2-2*y2+z2, 4*x.*y, 2*x.*z, 2*y.*z, 2*x, 2*y, 2*z, ones(length(x),1)];
R = x2+y2+z2;
b = (D'*D)\(D'*R); % least square solution
mtxref = [ 3  1  1   0  0  0   0  0  0  0; 3  1 -2   0  0  0   0  0  0  0; 3 -2  1   0  0  0   0  0  0  0; ...
           0  0  0   2  0  0   0  0  0  0; 0  0  0   0  1  0   0  0  0  0; 0  0  0   0  0  1   0  0  0  0; ...
           0  0  0   0  0  0   1  0  0  0; 0  0  0   0  0  0   0  1  0  0; 0  0  0   0  0  0   0  0  1  0; ...
           0  0  0   0  0  0   0  0  0  1];
v = mtxref*[-1/3; b]; nn=v(10); v = -v(1:9);
A = [ v(1) v(4) v(5) v(7); v(4) v(2) v(6) v(8); v(5) v(6) v(3) v(9); v(7) v(8) v(9) -nn ];
ofs=-A(1:3,1:3)\[v(7);v(8);v(9)]; % offset is center of ellipsoid
Tmtx=eye(4); Tmtx(4,1:3)=ofs'; AT=Tmtx*A*Tmtx'; % ellipsoid translated to (0,0,0)
[rotM ev]=eig(AT(1:3,1:3)/-AT(4,4)); % eigenvectors (rotation) and eigenvalues (gain)
gain=sqrt(1./diag(ev)); % gain is radius of the ellipsoid
```

Rotation matrix refinement

```
function [gain,rotM]=refine_3D_fit(gain,rotM)
  % largest element should be on diagonal
  m=0; rm=0; cm=0;
  for r=1:3, for c=1:3,
    if abs(rotM(r,c))>m, m=abs(rotM(r,c)); rm=r; cm=c; end; % record max
  end; end;
  if rm~=cm, % swap cols if not on diagonal
    t=rotM(:,cm); rotM(:,cm)=rotM(:,rm); rotM(:,rm)=t;
    t=gain(cm); gain(cm)=gain(rm); gain(rm)=t;
  end; % largest now in the diagonal, in row rm

  % do the same on remaining 2x2 matrix
  switch rm, case 1, i=[2 3]; case 2, i=[1 3]; case 3, i=[1 2]; end;
  m=0; rm=0; cm=0;
  for r=1:2, for c=1:2,
    if abs(rotM(i(r),i(c)))>m, m=abs(rotM(i(r),i(c))); rm=i(r); cm=i(c); end;
  end; end;
  if rm~=cm, % swap cols if not on diagonal
    t=rotM(:,cm); rotM(:,cm)=rotM(:,rm); rotM(:,rm)=t;
    t=gain(cm); gain(cm)=gain(rm); gain(rm)=t;
  end;

  % neg cols to make it positive along diagonal
  if rotM(1,1)<0, rotM(:,1)=-rotM(:,1); end;
  if rotM(2,2)<0, rotM(:,2)=-rotM(:,2); end;
  if rotM(3,3)<0, rotM(:,3)=-rotM(:,3); end;
end
```

Test code and sample output

```
[ofs,gain,rotM]=ellipsoid_fit([X Y Z]);
[gain,rotM]=refine_3D_fit(gain,rotM); % optional refinement

XC=X-ofs(1); YC=Y-ofs(2); ZC=Z-ofs(3); % translate to (0,0,0)
XYZC=[XC,YC,ZC]*rotM; % rotate to XYZ axes
refr = 500; % reference radius
XC=XYZC(:,1)/gain(1)*refr;
YC=XYZC(:,2)/gain(2)*refr;
ZC=XYZC(:,3)/gain(3)*refr; % scale to sphere

figure;
subplot(2,2,1); hold on; plot(XC,YC,'ro'); plot(X,Y,'kx');
xlabel('X'); ylabel('Y'); axis equal; grid on;
subplot(2,2,2); hold on; plot(ZC,YC,'go'); plot(Z,Y,'kx');
xlabel('Z'); ylabel('Y'); axis equal; grid on;
subplot(2,2,3); hold on; plot(XC,ZC,'bo'); plot(X,Z,'kx');
xlabel('X'); ylabel('Z'); axis equal; grid on;
```



Support material

Related design support material
BlueMicrosystem1, Bluetooth low energy and sensors software expansion for STM32Cube
Open.MEMS, MotionFX, Real-time motion-sensor data fusion software expansion for STM32Cube
Documentation
Application note, AN4508, Parameters and calibration of a low-g 3-axis accelerometer
Application note, AN4615, Fusion and compass calibration APIs for the STM32 Nucleo with the X-NUCLEO-IKS01A1 sensors expansion board
Desing tip, DT0053, 6-point tumble sensor calibration

Revision history

Date	Version	Changes
09-Feb-2016	1	Initial release
26-Aug-2018	2	Updated equations, added Matlab code
29-Oct-2018	3	Added paragraph on rotation matrix refinement for optimal data compensation