Freescale Semiconductor
Application Note

Document Number:AN4697
Rev. 1.0, 4/2013

Low Pass Filtering of Orientation Estimates

by: Mark Pedley

Contents

1 Introduction

This Application Note documents the low pass filtering of orientation estimates 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.

1.1 Summary

Application Note AN4676 discusses the representation of orientation in terms of Euler angles, Rotation Matrix, Rotation Vector and Quaternion formats and Application Note AN4685 describes the estimation of the instantaneous orientation in these four formats using readings from the accelerometer and magnetometer sensors.

This Application Note documents the low pass filtering of the instantaneous orientation estimates to reduce sample to sample jitter noise. The problem is more complex than it first appears for the following reasons:

- The Euler angle representation suffers from the unavoidable mathematical limitation of discontinuities and instability at *gimbal lock* orientations
- The Rotation Vector and Quaternion representations are discontinuous at rotation angles of 180° about any axis

since a rotation of 181° about any axis is equivalent to a rotation of 179° about the negated axis.

Low pass filtering any of these three orientation types will result in unexpected behavior such as an orientation estimates in the vicinity of pointing southwards averaging to pointing northwards.

In contrast, the rotation matrix estimate of orientation never suffers from discontinuities and can be low pass filtered provided that care is taken to ensure that the low pass filtered remains a valid orthonormal rotation matrix. Once the low pass filtered orientation matrix is available, the low pass filtered quaternion, rotation vector and Euler angles can be computed using the conversion routines documented in AN4676.

Low Pass Butterworth Filter documents the second order low pass Butterworth filter with adjustable cutoff frequency which is used in the Xtrinsic eCompass software. The source code for filtering the orientation matrix is listed in Source Code Listings.

2 Low Pass Butterworth Filter

2.1 General second order pole-zero filter

The general second order pole-zero filter has difference equation:

$$y[k] = b_0x[k] + b_1x[k-1] + b_2x[k-2] + a_1y[k-1] + a_2y[k-2] \tag{1}$$

The z-transform $H(z)$ and transfer function $H(f)$ at frequency f Hz for sampling frequency f_s Hz have form:

$$H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{1 - a_1z^{-1} - a_2z^{-2}} \tag{2}$$

$$H(f) = \frac{b_0 + b_1e^{-\left(\frac{2\pi if}{f_s}\right)} + b_2e^{-\left(\frac{4\pi if}{f_s}\right)}}{1 - a_1e^{-\left(\frac{2\pi if}{f_s}\right)} - a_2e^{-\left(\frac{4\pi if}{f_s}\right)}} \tag{3}$$

The power gain $|H(f)|^2$ is:

$$|H(f)|^2 = H(f)H(f)^* = \left(\frac{b_0 + b_1e^{-\left(\frac{2\pi if}{f_s}\right)} + b_2e^{-\left(\frac{4\pi if}{f_s}\right)}}{1 - a_1e^{-\left(\frac{2\pi if}{f_s}\right)} - a_2e^{-\left(\frac{4\pi if}{f_s}\right)}}\right)\left(\frac{b_0 + b_1e^{\left(\frac{2\pi if}{f_s}\right)} + b_2e^{\left(\frac{4\pi if}{f_s}\right)}}{1 - a_1e^{\left(\frac{2\pi if}{f_s}\right)} - a_2e^{\left(\frac{4\pi if}{f_s}\right)}}\right) \tag{4}$$

$$= \frac{\left(b_0{}^2 + b_1{}^2 + b_2{}^2\right) + 2b_1(b_0 + b_2)cos\left(\frac{2\pi f}{f_s}\right) + 2b_0b_2cos\left(\frac{4\pi f}{f_s}\right)}{(1 + a_1{}^2 + a_2{}^2) + 2a_1(a_2 - 1)cos\left(\frac{2\pi f}{f_s}\right) - 2a_2cos\left(\frac{4\pi f}{f_s}\right)} \tag{5}$$

$$\Rightarrow |H(f)| = \sqrt{\frac{\left(b_0{}^2 + b_1{}^2 + b_2{}^2\right) + 2b_1(b_0 + b_2)cos\left(\frac{2\pi f}{f_s}\right) + 2b_0b_2cos\left(\frac{4\pi f}{f_s}\right)}{(1 + a_1{}^2 + a_2{}^2) + 2a_1(a_2 - 1)cos\left(\frac{2\pi f}{f_s}\right) - 2a_2cos\left(\frac{4\pi f}{f_s}\right)}} \tag{6}$$

2.2 Second order Butterworth low pass filter

The specific form of the coefficients for the general second order filter for the low pass Butterworth filter with a cutoff at f_c Hz is given by.

$$b_0 = \frac{tan^2\left(\frac{\pi f_c}{f_s}\right)}{1+\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)} \tag{7}$$

$$b_1 = \frac{2tan^2\left(\frac{\pi f_c}{f_s}\right)}{1+\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)} = 2b_0 \tag{8}$$

$$b_2 = \frac{tan^2\left(\frac{\pi f_c}{f_s}\right)}{1+\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)} = b_0 \tag{9}$$

$$a_1 = \frac{2\left\{1-tan^2\left(\frac{\pi f_c}{f_s}\right)\right\}}{1+\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)} \tag{10}$$

$$a_2 = \frac{-\left\{1-\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)\right\}}{1+\sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right)+tan^2\left(\frac{\pi f_c}{f_s}\right)} \tag{11}$$

Useful results are that the power gain is 1 at DC, 0 at Nyquist and 0.5 at the cutoff frequency f_c:

$$|H(f=0)|^2 = \frac{\left({b_0}^2+{b_1}^2+{b_2}^2\right)+2b_1(b_0+b_2)+2b_0b_2}{\left(1+{a_1}^2+{a_2}^2\right)+2a_1(a_2-1)-2a_2} = 1\ for\ all\ f_c \tag{12}$$

$$\left|H\left(f=\frac{f_s}{2}\right)\right|^2 = \frac{\left({b_0}^2+{b_1}^2+{b_2}^2\right)-2b_1(b_0+b_2)+2b_0b_2}{\left(1+{a_1}^2+{a_2}^2\right)-2a_1(a_2-1)-2a_2} = 0\ for\ all\ f_c \tag{13}$$

$$|H(f_c)|^2 = \frac{\left({b_0}^2+{b_1}^2+{b_2}^2\right)+2b_1(b_0+b_2)cos\left(\frac{2\pi f_c}{f_s}\right)+2b_0b_2cos\left(\frac{4\pi f_c}{f_s}\right)}{\left(1+{a_1}^2+{a_2}^2\right)+2a_1(a_2-1)cos\left(\frac{2\pi f_c}{f_s}\right)-2a_2cos\left(\frac{4\pi f_c}{f_s}\right)} = 0.5\ for\ all\ f_c \tag{14}$$

2.3 Filter coefficients

Table 1 lists the filter coefficients for cutoff frequencies $f_c = f_s/2N$ for N = 2, 4, 8, 16, 32, 64, 128. N = 2 therefore corresponds to cutoff at half Nyquist, N = 4 to cutoff at quarter Nyquist, and so on.

Table 1. Filter coefficients as a function of cutoff frequency

N	2	4	8	16	32	64	128
b0	0.2928932	0.0976311	0.0299546	0.0084427	0.0022516	0.0005821	0.0001480
b1	0.5857864	0.1952621	0.0599092	0.0168854	0.0045032	0.0011642	0.0002960
b2	0.2928932	0.0976311	0.0299546	0.0084427	0.0022516	0.0005821	0.0001480
a1	0.0000000	0.9428090	1.4542436	1.7237762	1.8613611	1.9306064	1.9652934
a2	-0.1715729	-0.3333333	-0.5740619	-0.7575469	-0.8703675	-0.9329347	-0.9658855

2.4 Transfer function plots

Figure 1 is a linear plot of the amplitude gain $|H(f)| = \sqrt{|H(f)|^2}$ and Figure 2 is a logarithmic plot of the power gain $|H(f)|^2$ for $f_c = f_s/2N$ for N = 2, 4, 8, 16, 32, 64, 128 as a function of normalized frequency f/f_s ranging up to Nyquist at $f/f_s = 0.5$.

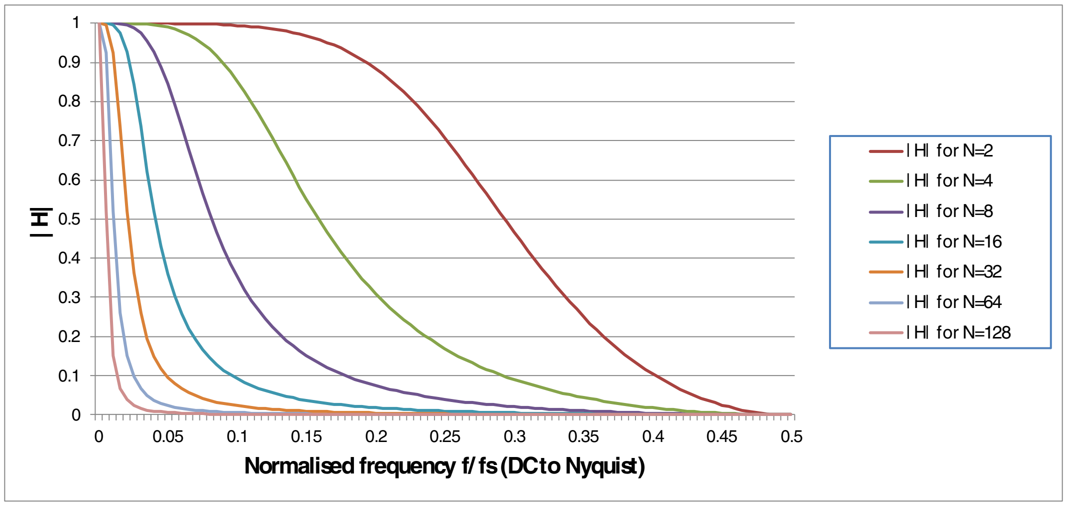

Figure 1. Amplitude gain transfer function of low pass Butterworth filter

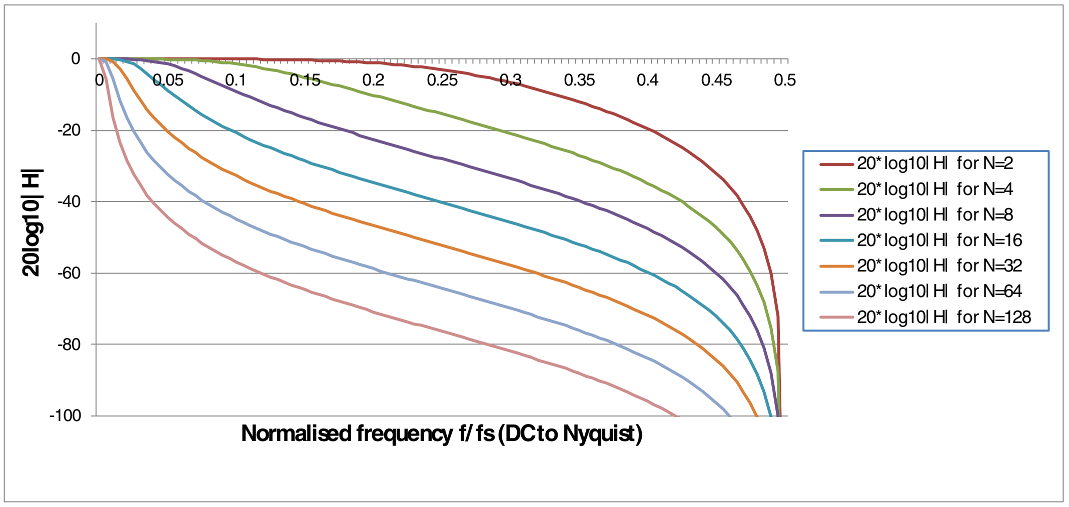

Figure 2. Logarithmic power gain transfer function of low pass Butterworth filter

2.5 Pole-zero diagrams

Zeroes in the transfer function $H(z)$ occur at solutions to:

$$b_0 z^2 + b_1 z + b_2 = 0 \tag{15}$$

Substituting for b_1 and b_2 and then factorizing demonstrates the two zeroes occur at $z = -1$ irrespective of the cutoff frequency f_c:

$$b_0 (z_{Zero} + 1)^2 = 0 \Rightarrow z_{Zero} = -1 \tag{16}$$

Poles in the transfer function occur at solutions to:

$$z_{Pole}{}^2 - a_1 z_{Pole} - a_2 = 0 \tag{17}$$

$$\Rightarrow z_{Pole} = \frac{a_1 \pm \sqrt{a_1{}^2 + 4a_2}}{2} \tag{18}$$

$$= \frac{1 - tan^2\left(\frac{\pi f_c}{f_s}\right) \pm i\sqrt{2} tan\left(\frac{\pi f_c}{f_s}\right)}{1 + \sqrt{2} tan\left(\frac{\pi f_c}{f_s}\right) + tan^2\left(\frac{\pi f_c}{f_s}\right)} = \frac{1 - tan^2\left(\frac{\pi}{2N}\right) \pm i\sqrt{2} tan\left(\frac{\pi}{2N}\right)}{1 + \sqrt{2} tan\left(\frac{\pi}{2N}\right) + tan^2\left(\frac{\pi}{2N}\right)} \tag{19}$$

The pole radius is given by:

$$|z_{Pole}|^2 = \{Re(z_{Pole})\}^2 + \{Im(z_{Pole})\}^2 = \frac{a_1{}^2}{4} + \frac{-a_1{}^2 - 4a_2}{4} = -a_2 \tag{20}$$

Table continues on the next page...

$$\Rightarrow |z_{Pole}| = \sqrt{-a_2} = \sqrt{\frac{1 - \sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right) + tan^2\left(\frac{\pi f_c}{f_s}\right)}{1 + \sqrt{2}tan\left(\frac{\pi f_c}{f_s}\right) + tan^2\left(\frac{\pi f_c}{f_s}\right)}} \quad (21)$$

Figure 3 plots the locations of the zeroes (in blue) and the poles (in red) for the discrete values of N = 2, 4, 8, 16, 32, 64, and 128. There is no reason that N cannot be fractional and Figure 3 also therefore plots the continuous red trajectory of the two poles for fractional N varying from 1 to infinity. The situation N = 1 is a degenerate all-pass filter with pole-zero cancellation at $z = -1$.

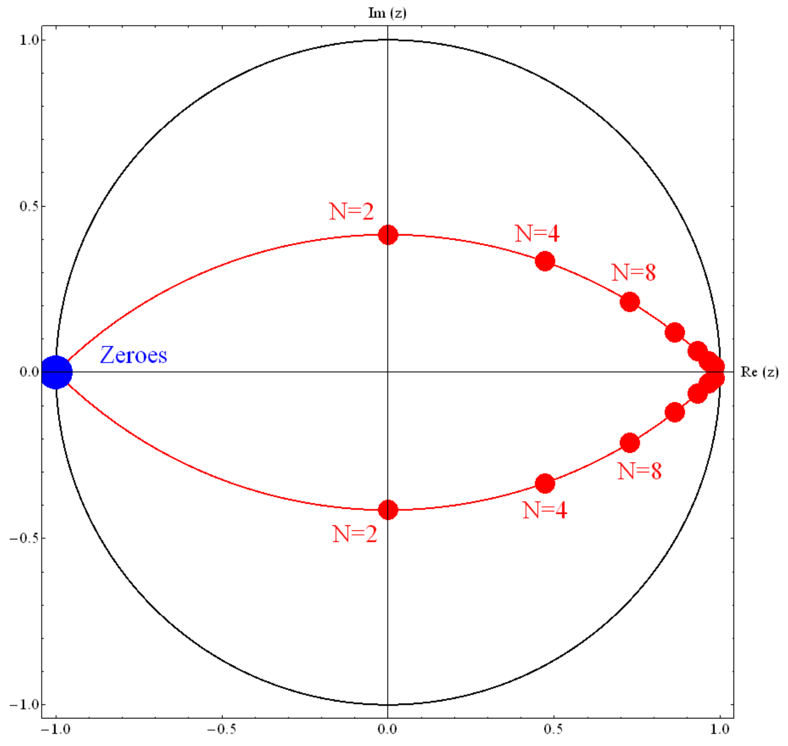

Figure 3. Pole-zero plot for low pass second order Butterworth filter

2.6 Application to the orientation matrix

The Butterworth difference equation (1) is applied to estimate the low pass filtered orientation matrix $\boldsymbol{R}_{LP}[k]$ at time sample k in terms of previous low pass filtered orientation matrices $\boldsymbol{R}_{LP}[k-1]$ and $\boldsymbol{R}_{LP}[k-2]$ and unfiltered orientation matrices $\boldsymbol{R}[k]$, $\boldsymbol{R}[k-1]$ and $\boldsymbol{R}[k-2]$.

$$\boldsymbol{R}_{LP}[k] = b_0\boldsymbol{R}[k] + b_1\boldsymbol{R}[k-1] + b_2\boldsymbol{R}[k-2] + a_1\boldsymbol{R}_{LP}[k-1] + a_2\boldsymbol{R}_{LP}[k-2] \quad (22)$$

$$= b_0(\boldsymbol{R}[k] + 2\boldsymbol{R}[k-1] + \boldsymbol{R}[k-2]) + a_1\boldsymbol{R}_{LP}[k-1] + a_2\boldsymbol{R}_{LP}[k-2] \quad (23)$$

Equation (23) should be interpreted as applying separately to each of the nine elements of the orientation matrix. For the specific element R_{00}, for example, the low pass filter has form:

$$R_{LP,00}[k] = b_0(R_{00}[k] + 2R_{00}[k-1] + R_{00}[k-2]) + a_1 R_{LP,00}[k-1] + a_2 R_{LP,00}[k-2] \tag{24}$$

A consequence of the low pass filtering operation is that the resulting low pass filtered orientation matrix will not, in general, have the orthonormal properties of a valid rotation matrix. Specifically, the row and column norms of a valid rotation matrix have unit magnitude, the columns and rows must be orthogonal and the determinant must equal 1.

The low pass filtered rotation matrix can easily be corrected to be orthonormal with the algorithm below. The discussion of the algorithm refers to the un-normalized rotation matrix in terms of its column vectors as:

$$\boldsymbol{R} = (\boldsymbol{c}_x \quad \boldsymbol{c}_y \quad \boldsymbol{c}_z) \tag{25}$$

Step 1: The z column vector is normalized to unit magnitude:

$$\boldsymbol{c}_z{}' = \frac{\boldsymbol{c}_z}{|\boldsymbol{c}_z|} \tag{26}$$

Step 2: The x column vector is set to the vector product of the y and z column vectors and the result normalized. This results in the x column vector being normal to the y and z column vectors.

$$\boldsymbol{c}_x{}' = \frac{\boldsymbol{c}_y \times \boldsymbol{c}_z{}'}{|\boldsymbol{c}_y \times \boldsymbol{c}_z{}'|} \tag{27}$$

Step 3: The y column vector is set to the vector product of the z and x column vectors and the result normalized. This results in the y column vector being normal to the z and x column vectors and all three column vectors being mutually orthornormal.

$$\boldsymbol{c}_y{}' = \frac{\boldsymbol{c}_z{}' \times \boldsymbol{c}_x{}'}{|\boldsymbol{c}_z{}' \times \boldsymbol{c}_x{}'|} \tag{28}$$

This algorithm is implemented by function `fmatrixAeqRenormRotA` documented in Matrix Renormalization Function.

3 Source Code Listings

3.1 Structures

```
// quaternion structure definition
struct fquaternion
{
    float q0;    // scalar component
    float q1;    // x vector component
    float q2;    // y vector component
    float q3;    // z vector component
};

// 6DOF orientation structure definition
```

```
struct SV6DOF
{
    // Euler angles
    float fPhi6DOF;                  // 6DOF roll (deg)
    float fThe6DOF;                  // 6DOF pitch (deg)
    float fPsi6DOF;                  // 6DOF yaw (deg)
    float fRho6DOF;                  // 6DOF compass (deg)
    float fLPPhi6DOF;                // 6DOF low pass roll (deg)
    float fLPThe6DOF;                // 6DOF low pass pitch (deg)
    float fLPPsi6DOF;                // 6DOF low pass yaw (deg)
    float fLPRho6DOF;                // 6DOF low pass compass (deg)
    float fDelta6DOFn;               // 6DOF inclination angle (deg) sample n
    float fDelta6DOFnm1;             // 6DOF inclination angle (deg) sample n-1
    float fDelta6DOFnm2;             // 6DOF inclination angle (deg) sample n-2
    float fLPDelta6DOFn;             // 6DOF low pass inclination angle (deg) sample n
    float fLPDelta6DOFnm1;           // 6DOF low pass inclination angle (deg) sample n-1
    float fLPDelta6DOFnm2;           // 6DOF low pass inclination angle (deg) sample n-2
    // orientation matrices and quaternions
    float xfR6DOFn[3][3];            // 6DOF rotation matrix for current sample n
    float *fR6DOFn[3];
    float xfR6DOFnm1[3][3];          // 6DOF rotation matrix for sample n-1
    float *fR6DOFnm1[3];
    float xfR6DOFnm2[3][3];          // 6DOF rotation matrix for sample n-2
    float *fR6DOFnm2[3];
    float xfLPR6DOFn[3][3];          // low pass 6DOF rotation matrix for sample n
    float *fLPR6DOFn[3];
    float xfLPR6DOFnm1[3][3];        // low pass 6DOF rotation matrix for sample n-1
    float *fLPR6DOFnm1[3];
    float xfLPR6DOFnm2[3][3];        // low pass 6DOF rotation matrix for sample n-1
    float *fLPR6DOFnm2[3];
    struct fquaternion fq6DOFn;      // current 6DOF orientation quaternion
    struct fquaternion fLPq6DOFn;    // low pass 6DOF orientation quaternion
};
```

3.2 Compile Time Constants

The compile time constant `ANGLE_LPF_FPU` in file include.h is equivalent to the parameter *N* in Filter coefficients which defines the cutoff frequency as a function of the Nyquist frequency. The value defaults to 1.0, or all pass, in the supplied software and should be changed to a higher value (perhaps 8 or 16) for a sampling frequency of 50 Hz.

```
// orientation LPF cutoff = Nyquist / ANGLE_LPF_FPU
// so about ANGLE_LPF_FPU samples response
// ANGLE_LPF_FPU equal to 1.0F is valid and corresponds to all pass.
#define ANGLE_LPF_FPU 1.0F
```

3.3 Initialization using fInitLPFOrientationMatrix

Function `fInitLPFOrientationMatrix` initializes the Butterworth filter coefficients using equations **(7)** through **(11)** and the compile-time constant `ANGLE_LPF_FPU`. The all pass case is handled specially in the function `fLPFOrientationMatrix` and the initialization function returns without filter coefficients if `ANGLE_LPF_FPU` is 1.0.

```
// function initializes the low pass filter
void fInitLPFOrientationMatrix(float *pfb0, float *pfa1, float *pfa2)
{
    // function computes the Butterworth low pass filter values from ANGLE_LPF_FPU

    // local variables
    float fwc;                    // bilinear transformed frequency
    float ftmp;                   // scratch variable

    // return nothing for the all pass case which is tested for explicitly in the filter
    // ANGLE_LPF_FPU == 1.0F results in pole zero cancellation
    if (ANGLE_LPF_FPU == 1.0F)
        return;

    // calculate the filter coefficients for the general low pass case
    fwc = (float)tan(0.5F * PI / ANGLE_LPF_FPU);
    ftmp = 1.0F + (float)sqrt(2.0F) * fwc + fwc * fwc;
    *pfb0 = fwc * fwc / ftmp;
    *pfa1 = 2.0F * (1.0F - fwc * fwc) / ftmp;
    *pfa2 = -(1.0F - (float) sqrt(2.0F) * fwc + fwc * fwc) / ftmp;

    return;
}
```

3.4 Low pass filter fLPFOrientationMatrix

Function `fLPFOrientationMatrix` implements the low pass Butterworth filter of Low Pass Butterworth Filter and then computes the low pass filtered Euler angles and quaternion from the resulting low pass filtered rotation matrix.

```
// function low pass filters the orientation matrix
void fLPFOrientationMatrix(struct SV6DOF *pthisSV6DOF, int32 iCoordSystem, int32 loopcounter,
    float fb0, float fa1, float fa2)
{
    // function is called with the instantaneous orientation matrix fR6DOFn
    // and inclination angle fdelta6DOFn and computes the low pass filtered
    // rotation matrix, Euler angles, quaternion and inclination angle

    // local variables
    int32 i, j;                        // loop counters

    // initialize delay lines on first pass
    if (loopcounter == 0)
    {
        // set R[LP,n-2]=R[LP,n-1]=R[n-2]=R[n-1]=R[n]
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                pthisSV6DOF->fLPR6DOFnm2[i][j] = pthisSV6DOF->fLPR6DOFnm1[i][j] =
                pthisSV6DOF->fR6DOFnm2[i][j] = pthisSV6DOF->fR6DOFnm1[i][j] =
                pthisSV6DOF->fR6DOFn[i][j];

        // set delta[LP,n-2]=delta[LP,n-1]=delta[n-2]=delta[n-1]=delta[n]
        pthisSV6DOF->fLPDelta6DOFnm2 = pthisSV6DOF->fLPDelta6DOFnm1 =
            pthisSV6DOF->fDelta6DOFnm2 = pthisSV6DOF->fDelta6DOFnm1 =
            pthisSV6DOF->fDelta6DOFn;
    }

    // low pass filter the nine elements of the rotation matrix
    // the resulting matrix will no longer be perfectly orthonormal
    if (ANGLE_LPF_FPU == 1.0F)
    {
        // all pass case for orientation matrix
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                pthisSV6DOF->fLPR6DOFn[i][j] = pthisSV6DOF->fR6DOFn[i][j];
        // all pass case for the inclination angle delta
        pthisSV6DOF->fLPDelta6DOFn = pthisSV6DOF->fDelta6DOFn;
    }
    else
    {
        // apply the low pass Butterworth filter to the orientation matrix
        for (i = 0; i < 3; i++)
           for (j = 0; j < 3; j++)
             pthisSV6DOF->fLPR6DOFn[i][j] = fb0 * (pthisSV6DOF->fR6DOFn[i][j] +
                2.0F * pthisSV6DOF->fR6DOFnm1[i][j] + pthisSV6DOF->fR6DOFnm2[i][j]) +
                fa1 * pthisSV6DOF->fLPR6DOFnm1[i][j] + fa2 * pthisSV6DOF->fLPR6DOFnm2[i][j];
        // apply the low pass Butterworth filter to the inclination angle delta
        pthisSV6DOF->fLPDelta6DOFn = fb0 * (pthisSV6DOF->fDelta6DOFn +
            2.0F * pthisSV6DOF->fDelta6DOFnm1 + pthisSV6DOF->fDelta6DOFnm2) +
            fa1 * pthisSV6DOF->fLPDelta6DOFnm1 + fa2 * pthisSV6DOF->fLPDelta6DOFnm2;
    }

    // renormalize the orientation matrix
    fmatrixAeqRenormRotA(pthisSV6DOF->fLPR6DOFn);

    // calculate Euler angles from the low pass orientation matrix
    if (iCoordSystem == NED) // NED
    {
      fNEDAnglesDegFromRotationMatrix(pthisSV6DOF->fLPR6DOFn, &(pthisSV6DOF->fLPPhi6DOF),
      &(pthisSV6DOF->fLPThe6DOF), &(pthisSV6DOF->fLPPsi6DOF),
      &(pthisSV6DOF->fLPRho6DOF));
    }
    else if (iCoordSystem == ANDROID) // Android
    {
      fAndroidAnglesDegFromRotationMatrix(pthisSV6DOF->fLPR6DOFn, &(pthisSV6DOF->fLPPhi6DOF),
      &(pthisSV6DOF->fLPThe6DOF), &(pthisSV6DOF->fLPPsi6DOF),
      &(pthisSV6DOF->fLPRho6DOF));
    }
    else  // Windows 8
    {
      fWin8AnglesDegFromRotationMatrix(pthisSV6DOF->fLPR6DOFn, &(pthisSV6DOF->fLPPhi6DOF),
      &(pthisSV6DOF->fLPThe6DOF), &(pthisSV6DOF->fLPPsi6DOF),
      &(pthisSV6DOF->fLPRho6DOF));
    }

    // compute a normalized low pass quaternion from the low pass normalized matrix
    fQuaternionFromRotationMatrix(pthisSV6DOF->fLPR6DOFn, &(pthisSV6DOF->fLPq6DOFn));

    return;
}
```

3.5 Delay line shuffle

The low pass filter delay lines are updated in the main control loop in the file main.c as follows.

```
// shuffle the rotation matrix low pass filter delay lines
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        {
        thisSV6DOF.fLPR6DOFnm2[i][j] = thisSV6DOF.fLPR6DOFnm1[i][j];
        thisSV6DOF.fLPR6DOFnm1[i][j] = thisSV6DOF.fLPR6DOFn[i][j];
        thisSV6DOF.fR6DOFnm2[i][j] = thisSV6DOF.fR6DOFnm1[i][j];
        thisSV6DOF.fR6DOFnm1[i][j] = thisSV6DOF.fR6DOFn[i][j];
        }

// update the inclination angle low pass filter delay lines
thisSV6DOF.fLPDelta6DOFnm2 = thisSV6DOF.fLPDelta6DOFnm1;
thisSV6DOF.fLPDelta6DOFnm1 = thisSV6DOF.fLPDelta6DOFn;
thisSV6DOF.fDelta6DOFnm2 = thisSV6DOF.fDelta6DOFnm1;
thisSV6DOF.fDelta6DOFnm1 = thisSV6DOF.fDelta6DOFn;
```

3.6 Matrix Renormalization Function

The function `fmatrixAeqRenormRotA` forces the input rotation matrix to be a valid orthornormal matrix as defined in equations **(25)** to **(28)**.

```
// function re-orthonormalizes a 3x3 rotation matrix
void fmatrixAeqRenormRotA(float **A)
{
    float ftmp;                     // scratch variable
#define CORRUPTMATRIX 0.01F         // threshold for deciding rotation matrix is corrupt

    // normalize the z vector [column 2]
    ftmp = (float) sqrt(A[0][2] * A[0][2] + A[1][2] * A[1][2] + A[2][2] * A[2][2]);
    if (ftmp > CORRUPTMATRIX)
    {
        // general case
        A[0][2] /= ftmp;
        A[1][2] /= ftmp;
        A[2][2] /= ftmp;
    }
    else
    {
        // return with identity matrix since the matrix is corrupted
        A[0][0] = A[1][1] = A[2][2] = 1.0F;
        A[0][1] = A[0][2] = A[1][0] = A[1][2] = A[2][0] = A[2][1] = 0.0F;
        return;
    }

    // set the x vector [column 0] to y [column 1] cross z [column 2]
    A[0][0] = A[1][1] * A[2][2] - A[2][1] * A[1][2];
    A[1][0] = A[2][1] * A[0][2] - A[0][1] * A[2][2];
    A[2][0] = A[0][1] * A[1][2] - A[1][1] * A[0][2];

    // normalize the resulting x vector [column 0]
    ftmp = (float) sqrt(A[0][0] * A[0][0] + A[1][0] * A[1][0] + A[2][0] * A[2][0]);
    if (ftmp > CORRUPTMATRIX)
    {
        // general case
        A[0][0] /= ftmp;
        A[1][0] /= ftmp;
        A[2][0] /= ftmp;
    }
    else
    {
        // return with identity matrix since the matrix is corrupted
        A[0][0] = A[1][1] = A[2][2] = 1.0F;
        A[0][1] = A[0][2] = A[1][0] = A[1][2] = A[2][0] = A[2][1] = 0.0F;
        return;
    }

    // set the y vector [column 1] to z [column 2] cross x [column 0]
    // this will be normalized since z and x already are
    A[0][1] = A[1][2] * A[2][0] - A[2][2] * A[1][0];
    A[1][1] = A[2][2] * A[0][0] - A[0][2] * A[2][0];
    A[2][1] = A[0][2] * A[1][0] - A[1][2] * A[0][0];

    return;
}
```

Document Number: AN4697
Rev. 1.0, 4/2013