//一阶互补

// a=tau / (tau + loop time)

// newAngle = angle measured with atan2 using the accelerometer

//加速度传感器输出值

// newRate = angle measured using the gyro

// looptime = loop time in millis()

float tau = 0.075;

float a = 0.0;

float Complementary(float newAngle, float newRate, int looptime)

{

float dtC = float(looptime) / 1000.0;

a = tau / (tau + dtC);

x\_angleC = a \* (x\_angleC + newRate \* dtC) + (1 - a) \* (newAngle);

return x\_angleC;

}

//二阶互补

// newAngle = angle measured with atan2 using the accelerometer

// newRate = angle measured using the gyro

// looptime = loop time in millis()

float Complementary2(float newAngle, float newRate, int looptime)

{

float k = 10;

float dtc2 = float(looptime) / 1000.0;

x1 = (newAngle - x\_angle2C) \* k \* k;

y1 = dtc2 \* x1 + y1;

x2 = y1 + (newAngle - x\_angle2C) \* 2 \* k + newRate;

x\_angle2C = dtc2 \* x2 + x\_angle2C;

return x\_angle2C;

}

//Here too we just have to set the k and magically we get the angle. 卡尔曼滤波

// KasBot V1 - Kalman filter module

float Q\_angle = 0.01; //0.001

float Q\_gyro = 0.0003; //0.003

float R\_angle = 0.01; //0.03

float x\_bias = 0;

float P\_00 = 0, P\_01 = 0, P\_10 = 0, P\_11 = 0;

float y, S;

float K\_0, K\_1;

// newAngle = angle measured with atan2 using the accelerometer

// newRate = angle measured using the gyro

// looptime = loop time in millis()

float kalmanCalculate(float newAngle, float newRate, int looptime)

{

float dt = float(looptime) / 1000;

x\_angle += dt \* (newRate - x\_bias);

P\_00 += - dt \* (P\_10 + P\_01) + Q\_angle \* dt;

P\_01 += - dt \* P\_11;

P\_10 += - dt \* P\_11;

P\_11 += + Q\_gyro \* dt;

y = newAngle - x\_angle;

S = P\_00 + R\_angle;

K\_0 = P\_00 / S;

K\_1 = P\_10 / S;

x\_angle += K\_0 \* y;

x\_bias += K\_1 \* y;

P\_00 -= K\_0 \* P\_00;

P\_01 -= K\_0 \* P\_01;

P\_10 -= K\_1 \* P\_00;

P\_11 -= K\_1 \* P\_01;

return x\_angle;

}

//To get the answer, you have to set 3 parameters: Q\_angle, R\_angle, R\_gyro.

//详细卡尔曼滤波

/\* -\*- indent-tabs-mode:T; c-basic-offset:8; tab-width:8; -\*- vi: set ts=8:

\* $Id: tilt.c,v 1.1 2003/07/09 18:23:29 john Exp $

\*

\* 1 dimensional tilt sensor using a dual axis accelerometer

\* and single axis angular rate gyro. The two sensors are fused

\* via a two state Kalman filter, with one state being the angle

\* and the other state being the gyro bias. \*

\* Gyro bias is automatically tracked by the filter. This seems

\* like magic.

\*

\* Please note that there are lots of comments in the functions and

\* in blocks before the functions. Kalman filtering is an already complex

\* subject, made even more so by extensive hand optimizations to the C code

\* that implements the filter. I've tried to make an effort of explaining

\* the optimizations, but feel free to send mail to the mailing list,

\* autopilot-devel@lists.sf.net, with questions about this code.

\*

\*

\* (c) 2003 Trammell Hudson <hudson@rotomotion.com>

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* This file is part of the autopilot onboard code package.

\*

\* Autopilot is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* Autopilot is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with Autopilot; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*

\*/

#include <math.h>

/\*

\* Our update rate. This is how often our state is updated with

\* gyro rate measurements. For now, we do it every time an

\* 8 bit counter running at CLK/1024 expires. You will have to

\* change this value if you update at a different rate.

\*/

static const float dt = ( 1024.0 \* 256.0 ) / 8000000.0;

/\*

\* Our covariance matrix. This is updated at every time step to

\* determine how well the sensors are tracking the actual state.

\*/

static float P[2][2] =

{

{ 1, 0 },

{ 0, 1 },

};

/\*

\* Our two states, the angle and the gyro bias. As a byproduct of computing

\* the angle, we also have an unbiased angular rate available. These are

\* read-only to the user of the module.

\*/

float angle;

float q\_bias;

float rate;

/\*

\* R represents the measurement covariance noise. In this case,

\* it is a 1x1 matrix that says that we expect 0.3 rad jitter

\* from the accelerometer.

\*/

static const float R\_angle = 0.3;

/\*

\* Q is a 2x2 matrix that represents the process covariance noise.

\* In this case, it indicates how much we trust the acceleromter

\* relative to the gyros.

\*/

static const float Q\_angle = 0.001;

static const float Q\_gyro = 0.003;

/\*

\* state\_update is called every dt with a biased gyro measurement

\* by the user of the module. It updates the current angle and

\* rate estimate.

\*

\* The pitch gyro measurement should be scaled into real units, but \* does not need any bias removal. The filter will track the bias.

\*

\* Our state vector is:

\*

\* X = [ angle, gyro\_bias ]

\*

\* It runs the state estimation forward via the state functions:

\*

\* Xdot = [ angle\_dot, gyro\_bias\_dot ]

\*

\* angle\_dot = gyro - gyro\_bias

\* gyro\_bias\_dot = 0

\*

\* And updates the covariance matrix via the function:

\*

\* Pdot = A\*P + P\*A' + Q

\*

\* A is the Jacobian of Xdot with respect to the states:

\*

\* A = [ d(angle\_dot)/d(angle) d(angle\_dot)/d(gyro\_bias) ]

\* [ d(gyro\_bias\_dot)/d(angle) d(gyro\_bias\_dot)/d(gyro\_bias) ]

\*

\* = [ 0 -1 ]

\* [ 0 0 ]

\*

\* Due to the small CPU available on the microcontroller, we've

\* hand optimized the C code to only compute the terms that are

\* explicitly non-zero, as well as expanded out the matrix math

\* to be done in as few steps as possible. This does make it harder

\* to read, debug and extend, but also allows us to do this with

\* very little CPU time.

\*/

void state\_update( const float q\_m /\* Pitch gyro measurement \*/)

{

/\* Unbias our gyro \*/

const float q = q\_m - q\_bias;

/\*

\* Compute the derivative of the covariance matrix

\*

\* Pdot = A\*P + P\*A' + Q

\*

\* We've hand computed the expansion of A = [ 0 -1, 0 0 ] multiplied \* by P and P multiplied by A' = [ 0 0, -1, 0 ]. This is then added

\* to the diagonal elements of Q, which are Q\_angle and Q\_gyro.

\*/

const float Pdot[2 \* 2] =

{

Q\_angle - P[0][1] - P[1][0], /\* 0,0 \*/

- P[1][1], /\* 0,1 \*/

- P[1][1], /\* 1,0 \*/

Q\_gyro /\* 1,1 \*/

};

/\* Store our unbias gyro estimate \*/

rate = q;

/\*

\* Update our angle estimate

\* angle += angle\_dot \* dt

\* += (gyro - gyro\_bias) \* dt

\* += q \* dt

\*/

angle += q \* dt;

/\* Update the covariance matrix \*/

P[0][0] += Pdot[0] \* dt;

P[0][1] += Pdot[1] \* dt;

P[1][0] += Pdot[2] \* dt;

P[1][1] += Pdot[3] \* dt;

}

/\*

\* kalman\_update is called by a user of the module when a new

\* accelerometer measurement is available. ax\_m and az\_m do not

\* need to be scaled into actual units, but must be zeroed and have

\* the same scale.

\*

\* This does not need to be called every time step, but can be if

\* the accelerometer data are available at the same rate as the

\* rate gyro measurement.

\*

\* For a two-axis accelerometer mounted perpendicular to the rotation

\* axis, we can compute the angle for the full 360 degree rotation

\* with no linearization errors by using the arctangent of the two

\* readings.

\* \* As commented in state\_update, the math here is simplified to

\* make it possible to execute on a small microcontroller with no

\* floating point unit. It will be hard to read the actual code and

\* see what is happening, which is why there is this extensive

\* comment block.

\*

\* The C matrix is a 1x2 (measurements x states) matrix that

\* is the Jacobian matrix of the measurement value with respect

\* to the states. In this case, C is:

\*

\* C = [ d(angle\_m)/d(angle) d(angle\_m)/d(gyro\_bias) ]

\* = [ 1 0 ]

\*

\* because the angle measurement directly corresponds to the angle

\* estimate and the angle measurement has no relation to the gyro

\* bias.

\*/

void kalman\_update(

const float ax\_m, /\* X acceleration \*/

const float az\_m /\* Z acceleration \*/

)

{

/\* Compute our measured angle and the error in our estimate \*/

const float angle\_m = atan2( -az\_m, ax\_m );

const float angle\_err = angle\_m - angle;

/\*

\* C\_0 shows how the state measurement directly relates to

\* the state estimate.

\*

\* The C\_1 shows that the state measurement does not relate

\* to the gyro bias estimate. We don't actually use this, so

\* we comment it out.

\*/

const float C\_0 = 1;

/\* const float C\_1 = 0; \*/

/\*

\* PCt<2,1> = P<2,2> \* C'<2,1>, which we use twice. This makes

\* it worthwhile to precompute and store the two values.

\* Note that C[0,1] = C\_1 is zero, so we do not compute that

\* term. \*/

const float PCt\_0 = C\_0 \* P[0][0]; /\* + C\_1 \* P[0][1] = 0 \*/

const float PCt\_1 = C\_0 \* P[1][0]; /\* + C\_1 \* P[1][1] = 0 \*/

/\*

\* Compute the error estimate. From the Kalman filter paper:

\*

\* E = C P C' + R

\*

\* Dimensionally,

\*

\* E<1,1> = C<1,2> P<2,2> C'<2,1> + R<1,1>

\*

\* Again, note that C\_1 is zero, so we do not compute the term.

\*/

const float E =

R\_angle

+ C\_0 \* PCt\_0

/\* + C\_1 \* PCt\_1 = 0 \*/

;

/\*

\* Compute the Kalman filter gains. From the Kalman paper:

\*

\* K = P C' inv(E)

\*

\* Dimensionally:

\*

\* K<2,1> = P<2,2> C'<2,1> inv(E)<1,1>

\*

\* Luckilly, E is <1,1>, so the inverse of E is just 1/E.

\*/

const float K\_0 = PCt\_0 / E;

const float K\_1 = PCt\_1 / E;

/\*

\* Update covariance matrix. Again, from the Kalman filter paper:

\*

\* P = P - K C P

\*

\* Dimensionally:

\*

\* P<2,2> -= K<2,1> C<1,2> P<2,2>

\* \* We first compute t<1,2> = C P. Note that:

\*

\* t[0,0] = C[0,0] \* P[0,0] + C[0,1] \* P[1,0]

\*

\* But, since C\_1 is zero, we have:

\*

\* t[0,0] = C[0,0] \* P[0,0] = PCt[0,0]

\*

\* This saves us a floating point multiply.

\*/

const float t\_0 = PCt\_0; /\* C\_0 \* P[0][0] + C\_1 \* P[1][0] \*/

const float t\_1 = C\_0 \* P[0][1]; /\* + C\_1 \* P[1][1] = 0 \*/

P[0][0] -= K\_0 \* t\_0;

P[0][1] -= K\_0 \* t\_1;

P[1][0] -= K\_1 \* t\_0;

P[1][1] -= K\_1 \* t\_1;

/\*

\* Update our state estimate. Again, from the Kalman paper:

\*

\* X += K \* err

\*

\* And, dimensionally,

\*

\* X<2> = X<2> + K<2,1> \* err<1,1>

\*

\* err is a measurement of the difference in the measured state

\* and the estimate state. In our case, it is just the difference

\* between the two accelerometer measured angle and our estimated

\* angle.

\*/

angle += K\_0 \* angle\_err;

q\_bias += K\_1 \* angle\_err;

}