/\*

http://stackoverflow.com/questions/6937433/ios-high-pass-filter-equation-for-accelerometer

A moving average is just a crude low pass filter. In this case it's what is known as ARMA (auto-regressive moving average)

you can think of it as a smoothing function" - it's "smoothing out" all the high frequency energy and leaving you with a slowly

varying estimate of the mean value of the signal. If you then subtract this smoothed signal from the instantaneous signal then

the difference will be the content that you have filtered out, i.e. the high frequency stuff, hence you get a high pass filter.

In other words: high\_pass\_filtered\_signal = signal - smoothed\_signal

With kFilteringFactor 0.1 you are taking 10% of the current value and adding 90% of the previous value. Therefore the value retains a 90% similarity to the previous value, which increases its resistance to sudden changes. This decreases noise but it also makes it less responsive to changes in the signal.

\*/

#define kFilteringFactor 0.1

//These "rolling" values are just low pass filtered versions of the input values (aka "moving averages")

UIAccelerationValue rollingX, rollingY, rollingZ;

- (void)accelerometer:(UIAccelerometer \*)accelerometer didAccelerate:(UIAcceleration \*)acceleration {

// Subtract the low-pass value from the current value to get a simplified high-pass filter

//low pass values subtracted from the instantaneous values to give you a high pass filtered output, i.e. you are getting the current deviation from the moving average.

rollingX = (acceleration.x \* kFilteringFactor) + (rollingX \* (1.0 - kFilteringFactor));

rollingY = (acceleration.y \* kFilteringFactor) + (rollingY \* (1.0 - kFilteringFactor));

rollingZ = (acceleration.z \* kFilteringFactor) + (rollingZ \* (1.0 - kFilteringFactor));

float accelX = acceleration.x - rollingX;

float accelY = acceleration.y - rollingY;

float accelZ = acceleration.z - rollingZ;

// Use the acceleration data.

}

----------------------------

Most IMU's have 6 DOF (Degrees Of Freedom). This means that there are 3 accelerometers, and 3 gyrosocopes inside the unit.

We will use both the accelerometer and gyroscope data for the same purpose: obtaining the angular position of the object. The gyroscope can do this by integrating the angular velocity over time. To obtain the angular position with the accelerometer, we are going to determine the position of the gravity vector (g-force) which is always visible on the accelerometer. This can easily be done by using an [atan2](http://en.wikipedia.org/wiki/Atan2) function. In both these cases, there is a big problem, which makes the data very hard to use without filter.

**The problem with accelerometers**

As an accelerometer measures all forces that are working on the object, it will also see a lot more than just the gravity vector. Every small force working on the object will disturb our measurement completely. The accelerometer data is reliable only on the long term, so a "low pass" filter has to be used.

**The problem with gyroscopes**

[In one of the previous articles](http://www.pieter-jan.com/node/7) I explained how to obtain the angular position by use of a gyroscope. We saw that it was very easy to obtain an accurate measurement that was not susceptible to external forces. The less good news was that, because of the integration over time, the measurement has the tendency to drift, not returning to zero when the system went back to its original position.

**The complementary filter**

The complementary filter gives us a "best of both worlds" kind of deal. On the short term, we use the data from the gyroscope, because it is very precise and not susceptible to external forces. On the long term, we use the data from the accelerometer, as it does not drift. In it's most simple form, the filter looks as follows:

http://www.pieter-jan.com/images/equations/CompFilter_Eq.gif

The function "ComplementaryFilter" has to be used in a infinite loop. Every iteration the pitch and roll angle values are updated with the new gyroscope values by means of integration over time. The filter then checks if the magnitude of the force seen by the accelerometer has a reasonable value that could be the real g-force vector. Afterwards, it will update the pitch and roll angles with the accelerometer data by taking 98% of the current value, and adding 2% of the angle calculated by the accelerometer. This will ensure that the measurement won't drift, but that it will be very accurate on the short term.

#define ACCELEROMETER\_SENSITIVITY 8192.0

#define GYROSCOPE\_SENSITIVITY 65.536

#define M\_PI 3.14159265359

#define dt 0.01 // 10 ms sample rate!

void ComplementaryFilter(short accData[3], short gyrData[3], float \*pitch, float \*roll) {

float pitchAcc, rollAcc;

*// Integrate the gyroscope data -> int(angularSpeed) = angle*

\*pitch += ((float)gyrData[0] / GYROSCOPE\_SENSITIVITY) \* dt;

*// Angle around the X-axis*

\*roll -= ((float)gyrData[1] / GYROSCOPE\_SENSITIVITY) \* dt;

*// Angle around the Y-axis*

*// Compensate for drift with accelerometer data if !bullshit*

*// Sensitivity = -2 to 2 G at 16Bit -> 2G = 32768 && 0.5G = 8192* int forceMagnitudeApprox = [abs](http://www.opengroup.org/onlinepubs/009695399/functions/abs.html)(accData[0]) + [abs](http://www.opengroup.org/onlinepubs/009695399/functions/abs.html)(accData[1]) + [abs](http://www.opengroup.org/onlinepubs/009695399/functions/abs.html)(accData[2]);

if (forceMagnitudeApprox > 8192 && forceMagnitudeApprox < 32768) {

*// Turning around the X axis results in a vector on the Y-axis* pitchAcc = atan2f((float)accData[1], (float)accData[2]) \* 180 / M\_PI; \*pitch = \*pitch \* 0.98 + pitchAcc \* 0.02;

*// Turning around the Y axis results in a vector on the X-axis* rollAcc = atan2f((float)accData[0], (float)accData[2]) \* 180 / M\_PI; \*roll = \*roll \* 0.98 + rollAcc \* 0.02;

} }