First Order Lag Filter

The formula for a first order lag filter is fairly simple:

```
new_filtered_value = k * raw_sensor_value + (1 - k) * old_filtered_value
```

The "magic" is determining an approprate value for '**k**'. The hard approach is to characterize the noise which will be present in the system and do some frequency-domain calculations to diminish the noise to acceptable levels. One thing that makes this approach hard (or even impossible) is that it requires knowing what noise sources will be present and what their frequency and magnitude characteristics will be. The easier approach is to cast a wide net and just filter the hell out of the signal.

The one catch with the easy approach is known as "lag". That is, when the input changes, the filtered value may not reflect enough of the change fast enough to meet system response requirements. For example, if you are using an IR LED and IR sensor to detect when you are within a short distance of an obstacle, too much filtering may result in the sensor not indicating there is an obstacle until *after* you run into the obstacle (or after there is insufficient time to stop).

The good thing about working from the perspective of lag is that determining how much lag is acceptable is generally easier than characterizing noise. Consider the previous example about running into an obstacle. If you know that you read the sensor 50 times a second, that you need the filtered value to reach 90% of the actual sensor value before your robot will think there is an obstacle, and that (based on your robot's speed and the sensor range) you need the filtered value to reach that level within 0.3 seconds, then you have all of the data you need to calculate 'k'.

The following sections will:

- Explain the input data you need to provide.
- Show how to calculate a suitable value for 'k' based on your input.

Input #1: Maximum Response Time (Tr)

This is the (maximum) amount of time (in seconds) between the instant the raw sensor value changes and the instant the filtered sensor value reflects that change.

For example, say your robot has an IR sensor that detects obstacles within 1 meter and your robot requires 0.2 meters to come to a complete stop. That means your robot can continue traveling full speed *at most* 1 - 0.2 = 0.8 meters after the sensor detects an obstacle.

Furthermore, say your robot's full speed is 2 meters/second. That means your robot needs to start stopping *at most* 0.8 / 2 = 0.4 seconds after the sensor detects an obstacle.

Finally, you decide you don't want to set things up so your robot just barely manages to stop in time (with its nose right against the obstacle). Instead, you would like to leave some margin for error. So rather than using 0.4 seconds, you go with a maximum response time (**Tr**) of 0.3 seconds (leaving you a 0.1 second margin of error).

Input #2: Sampling Period (Ts)

This is the amount of time (in seconds) between one reading of the raw sensor value and the next.

For example, if the robot samples the sensor 50 times a second, then the sampling period (**Ts**) is 1 / 50 = 0.02 seconds. (This is assuming that the sensor is read at roughly regular intervals.)

Note: Your sampling period should generally be *at least* 4 times (and preferably 10 times or more) smaller than your maximum response time.

Input #3: Desired Fraction of Change (Frac)

This is the (minimum) accuracy with which a change in the raw sensor value (at time **t**) will be reflected in the filtered sensor value (at or before time **t + Tr**).

For example, suppose the output from your IR sensor changes from 0 volts to 2 volts when an obstacle comes into range. Furthermore, your robot considers any value > 1.8 volts to indicate there is an obstacle present. That means that your robot needs to see 1.8 / 2 = 0.9 (90%) of the change before the change will be acted upon, so **Frac** is 0.9 in this case.

Calculate k

To calculate **k**, enter **Tr**, **Ts** and **Frac** from above into the following equation:

$$\mathbf{k = 1 - e^{ln(1 - Frac) * Ts / Tr}}$$

where:

- "**e**" is approximately 2.71828182846, and
- "**ln**" is the natural logorithm function.

For the example values given in the previous sections (**Tr** = 0.3, **Ts** = 0.02, **Frac** = 0.9), **k** comes out to be about 0.142304101409.

Results

So, does it actually work as advertised? To demonstrate it in action, the following C code is used:

```
void test() {
   const double Ts = 0.02;
   const double k = 0.142304101409;
   const double old_raw_sensor_value = 0;
   const double raw_sensor_value = 2;
   double t = 0;
   double filtered_value = old_raw_sensor_value; // assume we have settled on the old value
   do {
      filtered_value = k * raw_sensor_value + (1 - k) * filtered_value;
      t += Ts;
      printf("t = %.02f: filtered value = %.8f\n", t, filtered_value);
   } while (t < 0.4);
}
```

And the output produced is:

```
t = 0.02: filtered value = 0.28460820
t = 0.04: filtered value = 0.52871549
t = 0.06: filtered value = 0.73808531
t = 0.08: filtered value = 0.91766095
t = 0.10: filtered value = 1.07168223
t = 0.12: filtered value = 1.20378566
t = 0.14: filtered value = 1.31709023
t = 0.16: filtered value = 1.41427109
t = 0.18: filtered value = 1.49762271
t = 0.20: filtered value = 1.56911306
t = 0.22: filtered value = 1.63043004
t = 0.24: filtered value = 1.68302136
t = 0.26: filtered value = 1.72812872
t = 0.28: filtered value = 1.76681712
t = 0.30: filtered value = 1.80000000
t = 0.32: filtered value = 1.82846082
t = 0.34: filtered value = 1.85287155
t = 0.36: filtered value = 1.87380853
t = 0.38: filtered value = 1.89176609
t = 0.40: filtered value = 1.90716822
```

As you can see, the desired filter value (1.8) was achieved in the desired response time (0.3 seconds).

Beyond k

Handling Startup

What do you do at the start of time? The lag filter formula is undefined because you don't have an "old_filtered_value" yet. Some approaches for computing the very first filtered value are:

- Use the raw sensor data instead of the lag filter formula.
- Use some "safe" value.
- Use the raw sensor data adjusted towards some "safe" value. (Adjust it enough so that the adjustment is likely to be larger than any noise.)

Another option which can be combined with the above is to simply ignore the filtered sensor data entirely until several samples have been run through the filter. (For example, ignore the sensor for '**Tr**' time after startup.)

Scaled Integer Math

k is always some value between 0 and 1, and rounding to 0 or 1 is no good, so a straight-forward implementation involves the use of floating point or fixed point. If you are operating in an environment where neither floating point nor fixed point are available/desirable, and you have adequately large integers at your disposal, scaled integer math can be used instead.

Scaled integer math is basically the same as fixed point, except the location of the binary point is implicit in your code. The following is some example code that only uses integer operations (add, subtract, multiply, shift) to implement a lag filter:

```
// shift_amount is the number of bits past the binary point.
// k_scaled is k times 2 to the power of shift_amount, rounded to the next largest integer.
// The expression "((1 << shift_amount) - k_scaled)" can, of course, be precomputed.

scaled_filtered_value =   k_scaled * raw_sensor_value
                        + ((((1 << shift_amount) - k_scaled) * scaled_filtered_value) >> shift_amount)
filtered_value = scaled_filtered_value >> shift_amount
```

Note that the second product will require the largest number of integer bits. You can reduce the amount of bits required by feeding "filtered_value" instead of "scaled_filtered_value" into the equation (and then removing the ">> shift_amount" in that same equation), but that negatively impacts the accuracy and can result in the 'filtered_value' getting stuck before reaching its target value.

Last Modified: November 11th, 2006
home: http://my.execpc.com/~steidl/
e-mail: steidl@execpc.com