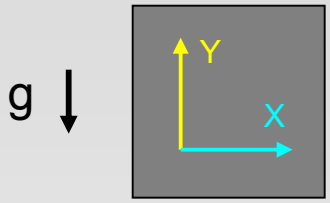
The Balance Filter

A Simple Solution for Integrating Accelerometer and Gyroscope Measurements for a Balancing Platform

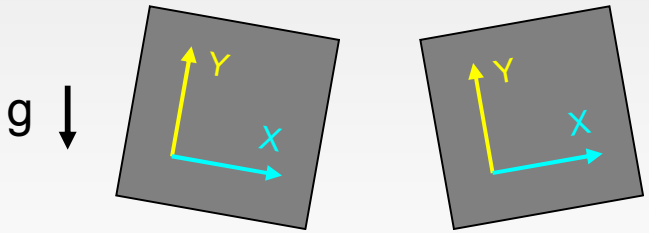
Shane Colton <scolton@mit.edu>
Mentor, FRC 97

Rev.1: Submitted as a Chief Delphi white paper - June 25, 2007.

Sensors



2-Axis Accelerometer:
• Measures “acceleration,” but really force per unit mass. (*F* = *ma,* so *a* = *F*/*m*)
• Can be used to measure the force of gravity. Above, X-axis reads 0g, Y-axis reads -1g.
• Can be used to measure tilt:

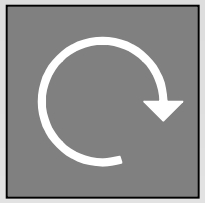


X now sees some gravity.

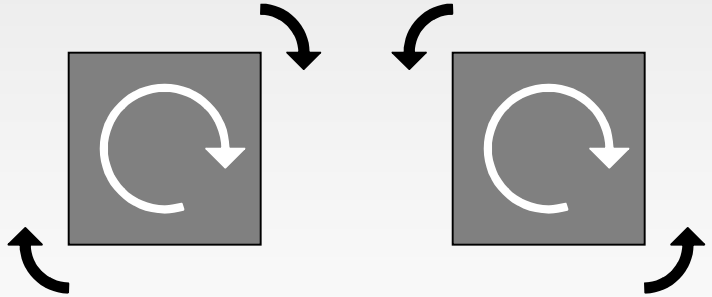
X reads slightly positive.

X reads slightly negative

Y sees slightly less gravity.
Is Y useful information? Probably not:
a) It is far less sensitive to small changes in angle than X.
b) It does not depend on direction of tilt.



Gyroscope:
• Measures angular rate (speed of rotation).
• Reads “zero” when stationary.
• Reads positive or negative when rotating:



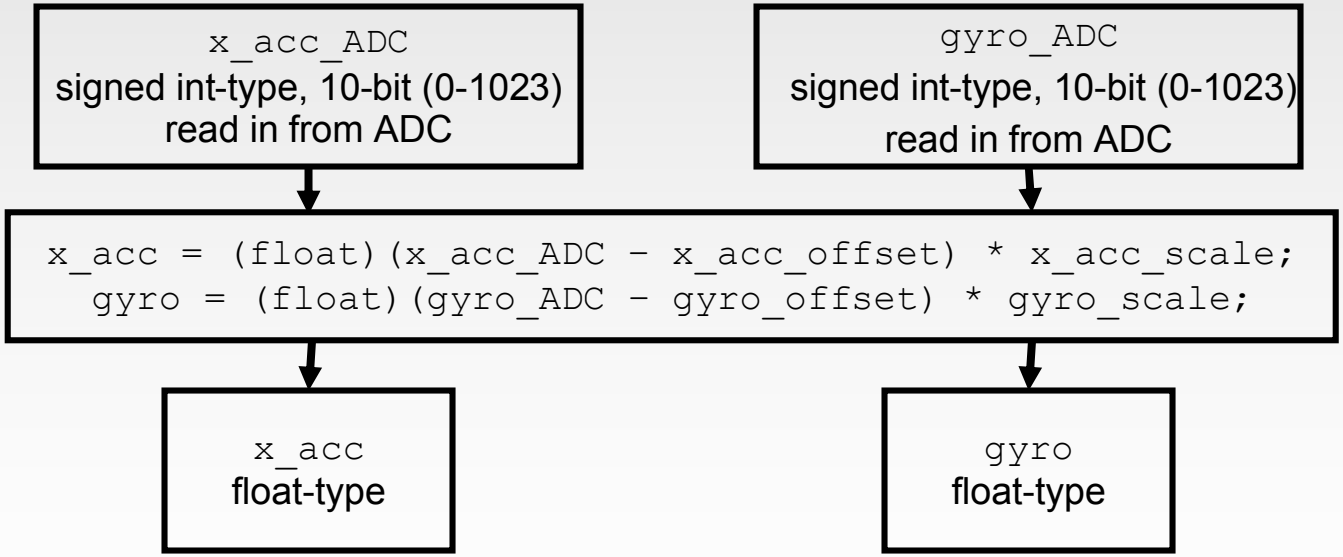
Gyro reads positive.

Gyro reads negative.

Reading Values from the Sensors

The first step is to read in analog inputs (through the analog-to-digital converter, ADC) for each sensor and get them into useful units. This requires adjustment for **offset** and **scale**:

• The **offset** is easy to find: see what integer value the sensor reads when it is horizontal and/or stationary. If it flickers around, choose an average value. The offset should be a signed* int-type variable (or constant).
• The **scale** depends on the sensor. It is the factor by which to multiply to get to the desired units†. This can be found in the sensor datasheet or by experiment. It is sometimes called the sensor constant, gain, or sensitivity. The scale should be a float-type variable (or constant).

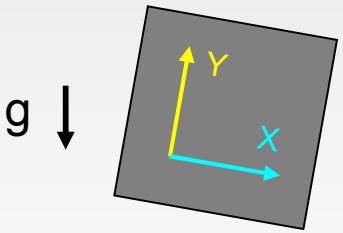


*Even though neither the ADC result nor the offset can be negative, they will be subtracted, so it couldn't hurt to make them signed variables now.
†Units could be degrees or radians [per second for the gyro]. They just have to be consistent.

A bit more about the accelerometer…

If it was necessary to have an estimate of angle for 360º of rotation, having the Y-axis measurement would be useful, but not necessary. With it, we could use trigonometry to find the inverse tangent of the two axis readings and calculate the angle. Without it, we can still use sine or cosine and the X-axis alone to figure out angle, since we know the magnitude of gravity. But trig kills processor time and is non-linear, so if it can be avoided, it should.

For a balancing platform, the most important angles to measure are near vertical. If the platform tilts more than 30º in either direction, there's probably not much the controller can do other than drive full speed to try to catch it. Within this window, we can use **small angle approximation** and the X-axis to save processor time and coding complexity:



Platform is tilted forward by and angle θ, but stationary (not accelerating horizontally).

X-axis reads: (1g) × sin(θ)

small angle approximation: sin(θ) ≈ θ, **in radians**
This works well (within 5%) up to $\theta = \pm\pi/6 = \pm 30º$.

So in the following bit of code,

```
x_acc = (float)(x_acc_ADC - x_acc_offset) * x_acc_scale;
```

`x_acc` will be the angle in **radians** if `x_acc_scale` is set to scale the output to 1[g] when the X-axis is pointed straight downward.

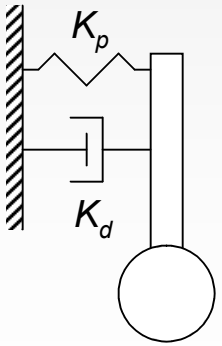
To get the angle in **degrees**, `x_acc_scale` should be multiplied by 180/π.

Desired Measurements

In order to control the platform, it would be nice to know both the **angle** and the **angular velocity** of the base platform. This could be the basis for an angle PD (proportional/derivative) control algorithm, which has been proven to work well for this type of system. Something like this:

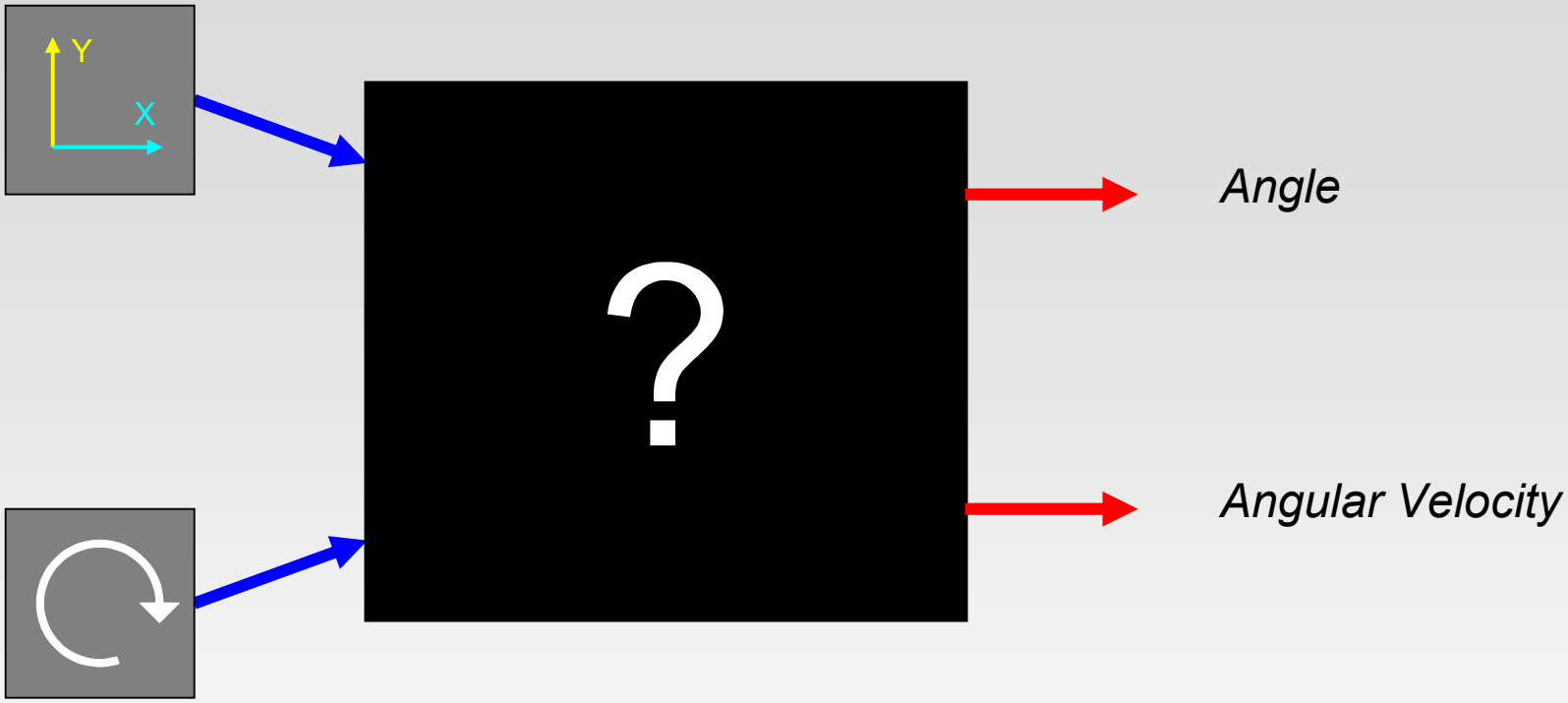
$$\textit{Motor Output} = K_p \times (\textit{Angle}) + K_d \times (\textit{Angular Velocity})$$

What exactly *Motor Output* does is another story. But the general idea is that this control setup can be tuned with K_p and K_d to give stability and smooth performance. It is less likely to overshoot the horizontal point than a proportional-only controller. (If *angle* is positive but *angular velocity* is negative, i.e. it is heading back toward being horizontal, the motors are slowed in advance.)



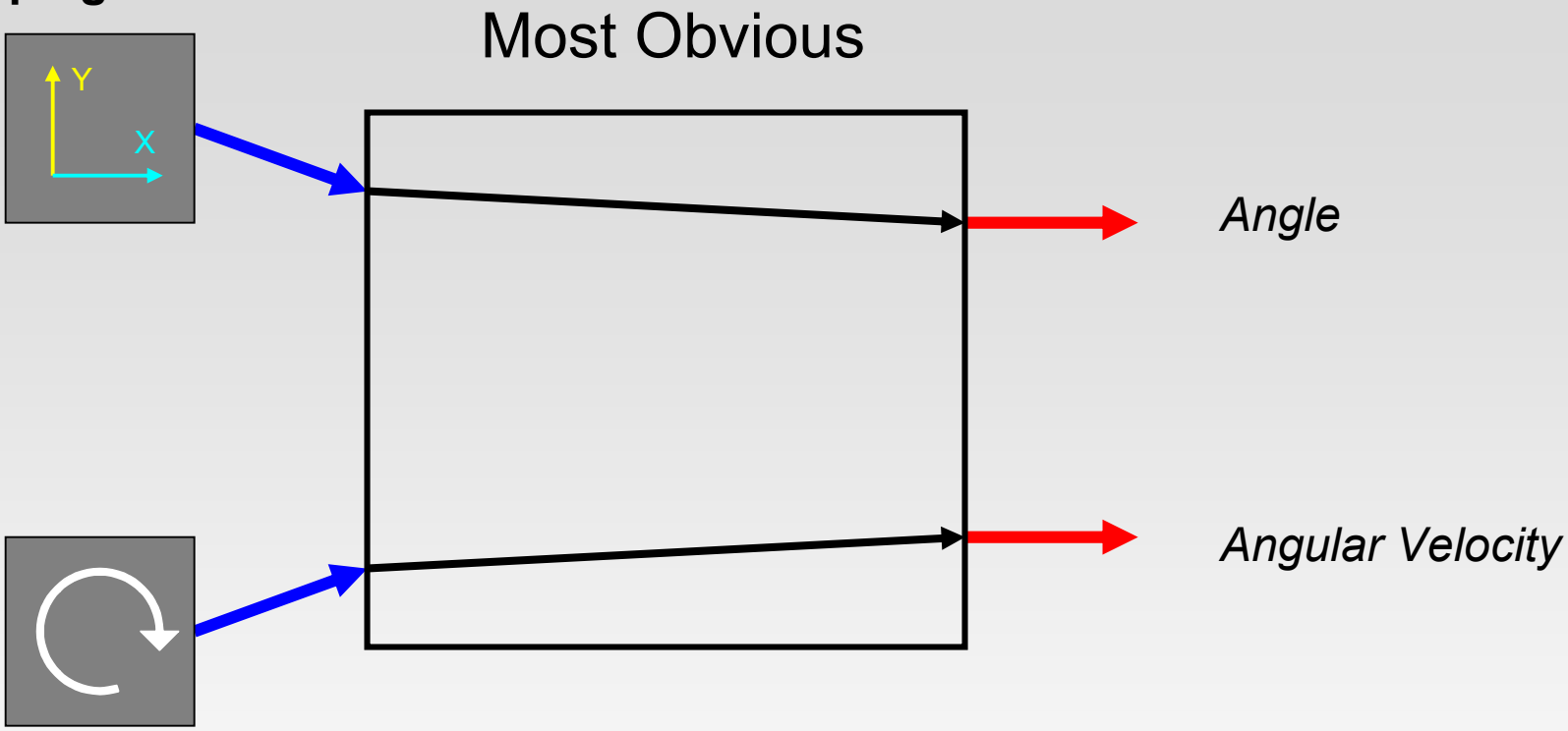
In effect, the PD control scheme is like adding an adjustable spring and damper to the Segway.

Mapping Sensors



Best approach?

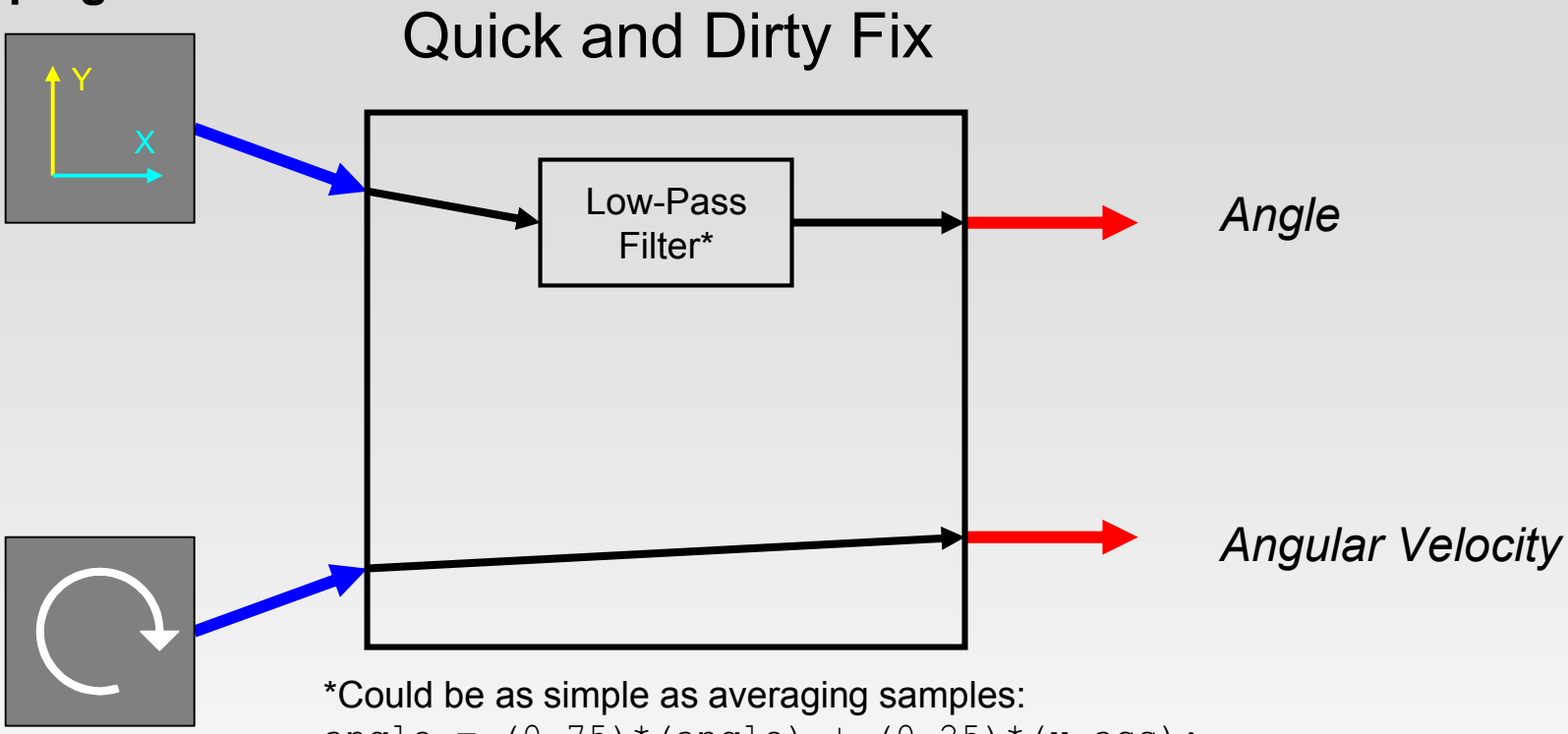
Mapping Sensors



Pros:
- Intuitive.
- Easy to code.
- Gyro gives fast and accurate angular velocity measurement.

Cons:
- Noisy.
- X-axis will read any horizontal acceleration as a change in angle. (Imagine the platform is horizontal, but the motors are causing it to accelerate forward. The accelerometer cannot distinguish this from gravity.)

Mapping Sensors



*Could be as simple as averaging samples:

```
angle = (0.75)*(angle) + (0.25)*(x_acc);
```

0.75 and 0.25 are example values. These could be tuned to change the time constant of the filter as desired.

Pros:

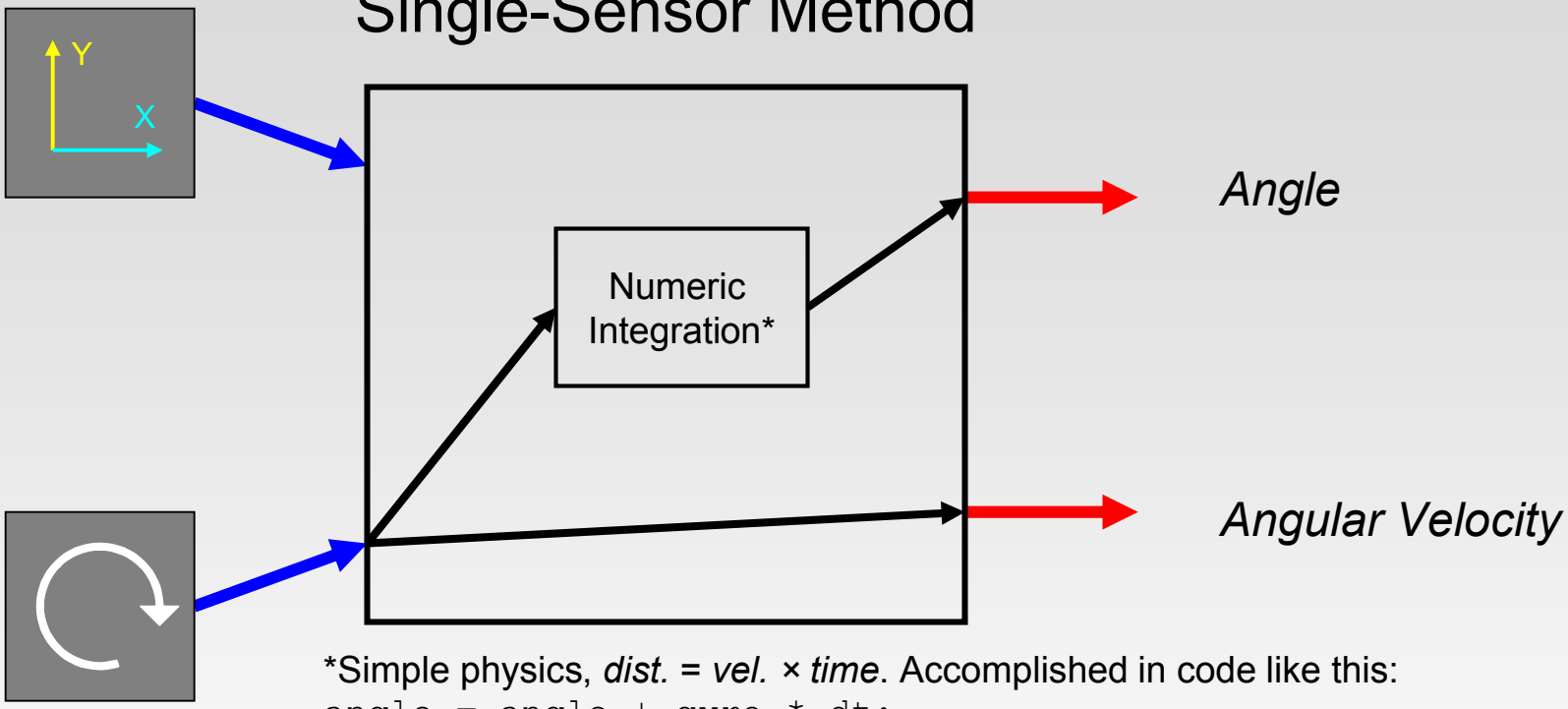
- Still Intuitive.
- Still easy to code.
- Filters out short-duration horizontal accelerations. Only long-term acceleration (gravity) passes through.

Cons:

- Angle measurement will lag due to the averaging. The more you filter, the more it will lag. Lag is generally bad for stability.

Mapping Sensors

Single-Sensor Method



*Simple physics, *dist.* = *vel.* × *time*. Accomplished in code like this:

```
angle = angle + gyro * dt;
```

Requires that you know the time interval between updates, *dt*.

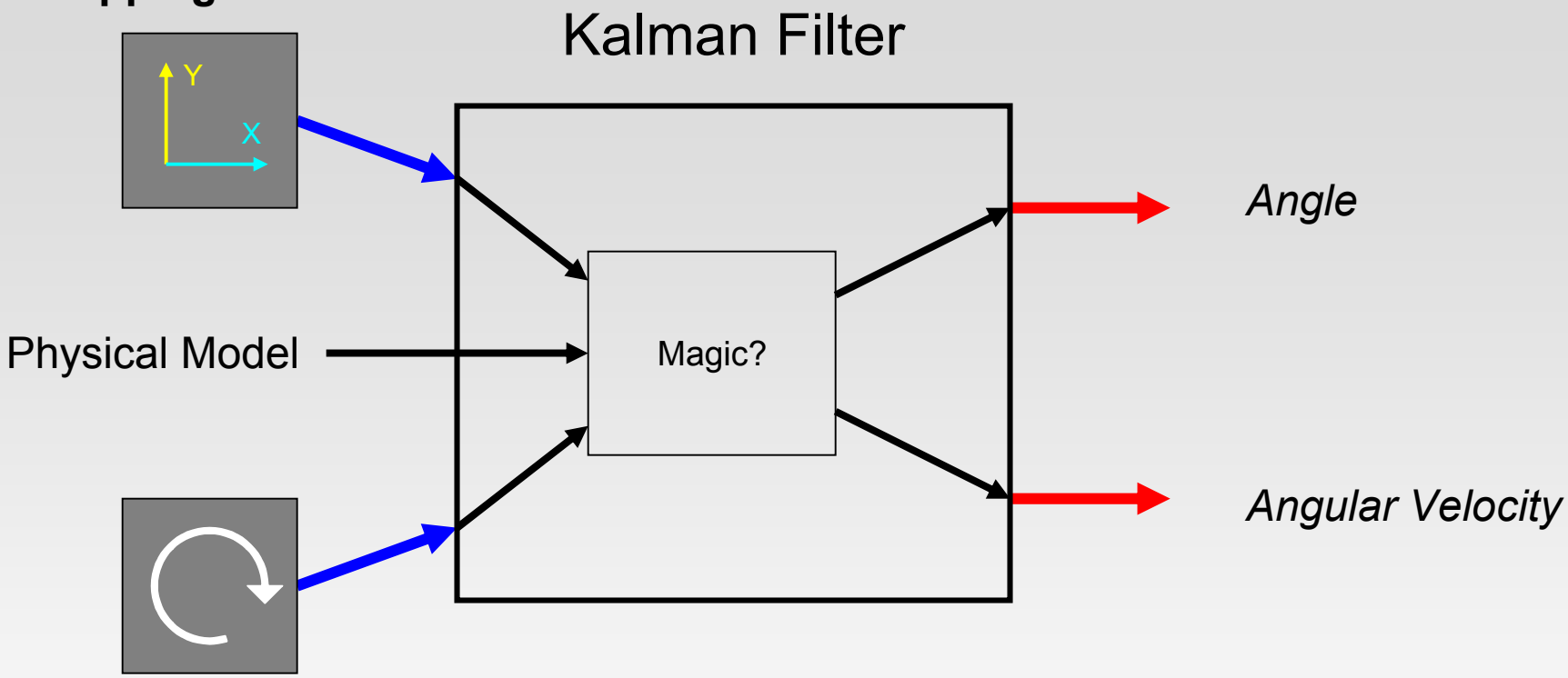
Pros:

- Only one sensor to read.
- Fast, lag is not a problem.
- Not subject to horizontal accelerations.
- Still easy to code.

Cons:

- The dreaded gyroscopic drift. If the gyro does not read perfectly zero when stationary (and it won't), the small rate will keep adding to the angle until it is far away from the actual angle.

Mapping Sensors



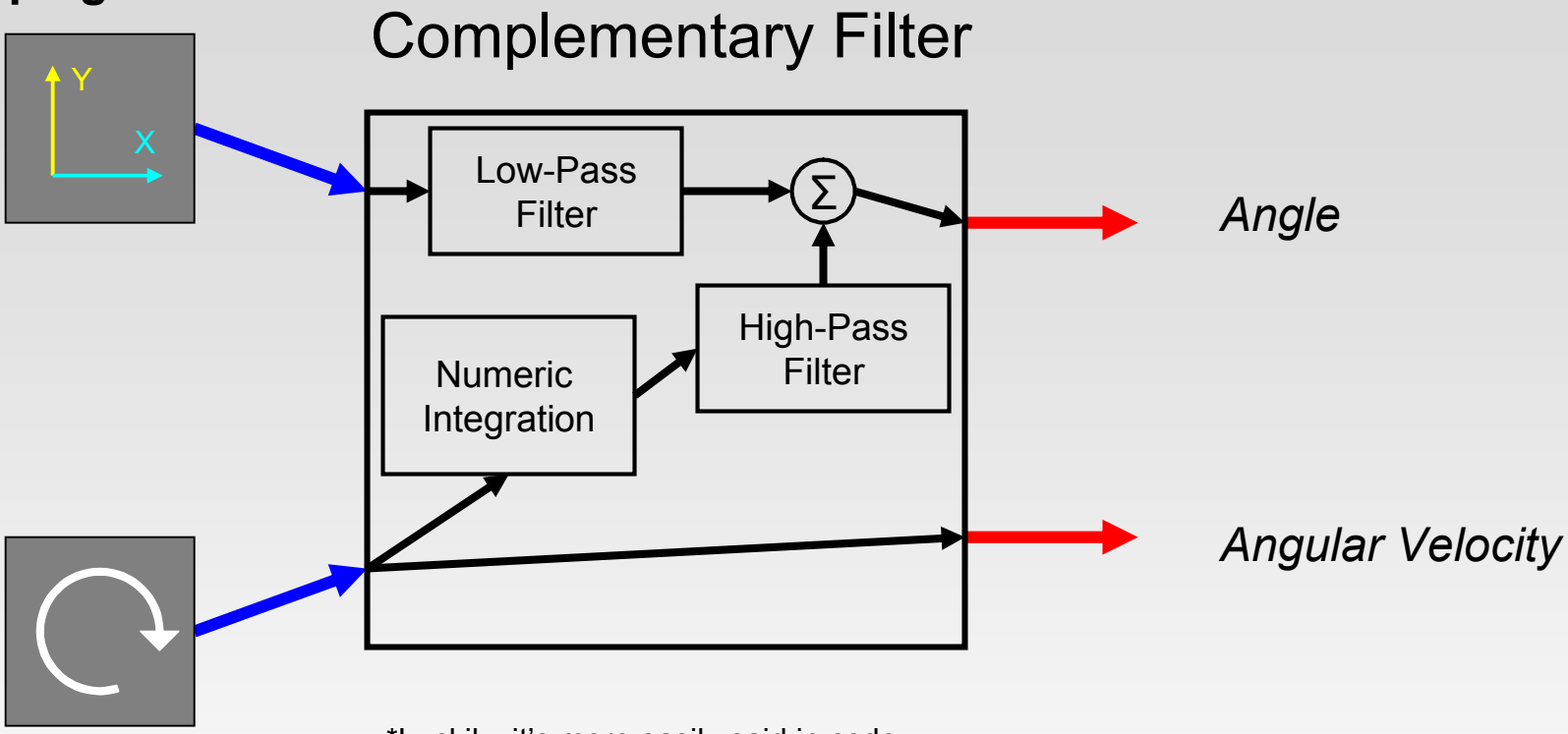
Pros:

- Supposedly the theoretically-ideal filter for combining noisy sensors to get clean, accurate estimates.
- Takes into account known physical properties of the system (mass, inertia, etc.).

Cons:

- I have no idea how it works. It's mathematically complex, requiring some knowledge of linear algebra. There are different forms for different situations, too.
- Probably difficult to code.
- Would kill processor time.

Mapping Sensors



*Luckily, it's more easily-said in code:

```
angle = (0.98)*(angle + gyro * dt) + (0.02)*(x_acc);
```

More explanation to come…

Pros:

- Can help fix noise, drift, and horizontal acceleration dependency.
- Fast estimates of angle, much less lag than low-pass filter alone.
- Not very processor-intensive.

Cons:

- A bit more theory to understand than the simple filters, but nothing like the Kalman filter.

More on Digital Filters

There is a lot of theory behind digital filters, most of which I don't understand, but the basic concepts are fairly easy to grasp without the theoretical notation (*z-domain transfer functions*, if you care to go into it). Here are some definitions:

Integration: This is easy. Think of a car traveling with a known speed and your program is a clock that ticks once every few milliseconds. To get the new position at each tick, you take the old position and add the change in position. The change in position is just the speed of the car multiplied by the time since the last tick, which you can get from the timers on the microcontroller or some other known timer. In code:
`position += speed*dt;`, or for a balancing platform, `angle += gyro*dt;`.

Low-Pass Filter: The goal of the low-pass filter is to only let through long-term changes, filtering out short-term fluctuations. One way to do this is to force the changes to build up little by little in subsequent times through the program loop. In code:

```
angle = (0.98)*angle + (0.02)*x_acc;
```

If, for example, the angle starts at zero and the accelerometer reading suddenly jumps to 10º, the angle estimate changes like this in subsequent iterations:

Iter.	1	2	3	4	5	6	7	8	9	10
θ	0.20º	0.40º	0.59º	0.78º	0.96º	1.14º	1.32º	1.49º	1.66º	1.83º

If the sensor stays at 10º, the angle estimate will rise until it levels out at that value. The time it takes to reach the full value depends on both the filter constants (0.98 and 0.02 in the example) and the sample rate of the loop (*dt*).

More on Digital Filters

High-Pass Filter: The theory on this is a bit harder to explain than the low-pass filter, but conceptually it does the exact opposite: It allows short-duration signals to pass through while filtering out signals that are steady over time. This can be used to cancel out drift.

Sample Period: The amount of time that passes between each program loop. If the sample rate is 100 Hz, the sample period is 0.01 sec.

Time Constant: The time constant of a filter is the relative duration of signal it will act on. For a low-pass filter, signals much longer than the time constant pass through unaltered while signals shorter than the time constant are filtered out. The opposite is true for a high-pass filter. The time constant, τ, of a digital low-pass filter,

```
y = (a)*(y) + (1-a)*(x);,
```

running in a loop with sample period, *dt*, can be found like this*:

$$\tau = \frac{a \cdot dt}{1-a} \leftrightarrow a = \frac{\tau}{\tau + dt}$$

So if you know the desired time constant and the sample rate, you can pick the filter coefficient *a*.

Complementary: This just means the two parts of the filter always add to one, so that the output is an accurate, linear estimate in units that make sense. After reading a bit more, I think the filter presented here is not exactly complementary, but is a very good approximation when the time constant is much longer than the sample rate (a necessary condition of digital control anyway).

*http://en.wikipedia.org/wiki/Low-pass_filter#Passive_digital_realization

A Closer Look at the Angle Complementary Filter

```
angle = (0.98)*(angle + gyro*dt) + (0.02)*(x_acc);
```

Integration.

Low-pass portion acting on the accelerometer.

Something resembling a high-pass filter on the integrated gyro angle estimate. It will have approximately the same time constant as the low-pass filter.

If this filter were running in a loop that executes 100 times per second, the time constant for both the low-pass and the high-pass filter would be:

$$\tau = \frac{a \cdot dt}{1 - a} = \frac{0.98 \cdot 0.01\,\text{sec}}{0.02} = 0.49\,\text{sec}$$

This defines where the boundary between trusting the gyroscope and trusting the accelerometer is. For time periods shorter than half a second, the gyroscope integration takes precedence and the noisy horizontal accelerations are filtered out. For time periods longer than half a second, the accelerometer average is given more weighting than the gyroscope, which may have drifted by this point.

A Closer Look at the Angle Complementary Filter

For the most part, designing the filter usually goes the other way. First, you pick a time constant and then use that to calculate filter coefficients. Picking the time constant is the place where you can tweak the response. If your gyroscope drifts on average 2º per second (probably a worst-case estimate), you probably want a time constant less than one second so that you can be guaranteed never to have drifted more than a couple degrees in either direction. But the lower the time constant, the more horizontal acceleration noise will be allowed to pass through. Like many other control situations, there is a tradeoff and the only way to really tweak it is to experiment.

Remember that the sample rate is *very* important to choosing the right coefficients. If you change your program, adding a lot more floating point calculations, and your sample rate goes down by a factor of two, your time constant will go up by a factor of two unless you recalculate your filter terms.

As an example, consider using the 26.2 msec radio update as your control loop (generally a slow idea, but it does work). If you want a time constant of 0.75 sec, the filter term would be:

$$a = \frac{\tau}{\tau + dt} = \frac{0.75\,\text{sec}}{0.75\,\text{sec} + 0.0262\,\text{sec}} = 0.966$$

So, `angle = (0.966)*(angle + gyro*0.0262) + (0.034)*(x_acc);`.
The second filter coefficient, 0.034, is just (1 - 0.966).

A Closer Look at the Angle Complementary Filter

It's also worthwhile to think about what happens to the gyroscope bias in this filter. It definitely doesn't cause the drifting problem, but it can still effect the angle calculation. Say, for example, we mistakenly chose the wrong offset and our gyroscope reports a rate of 5 º/sec rotation when it is stationary. It can be proven mathematically (I won't here) that the effect of this on the angle estimate is just the offset rate multiplied by the time constant. So if we have a 0.75 sec time constant, this will give a constant angle offset of 3.75º.

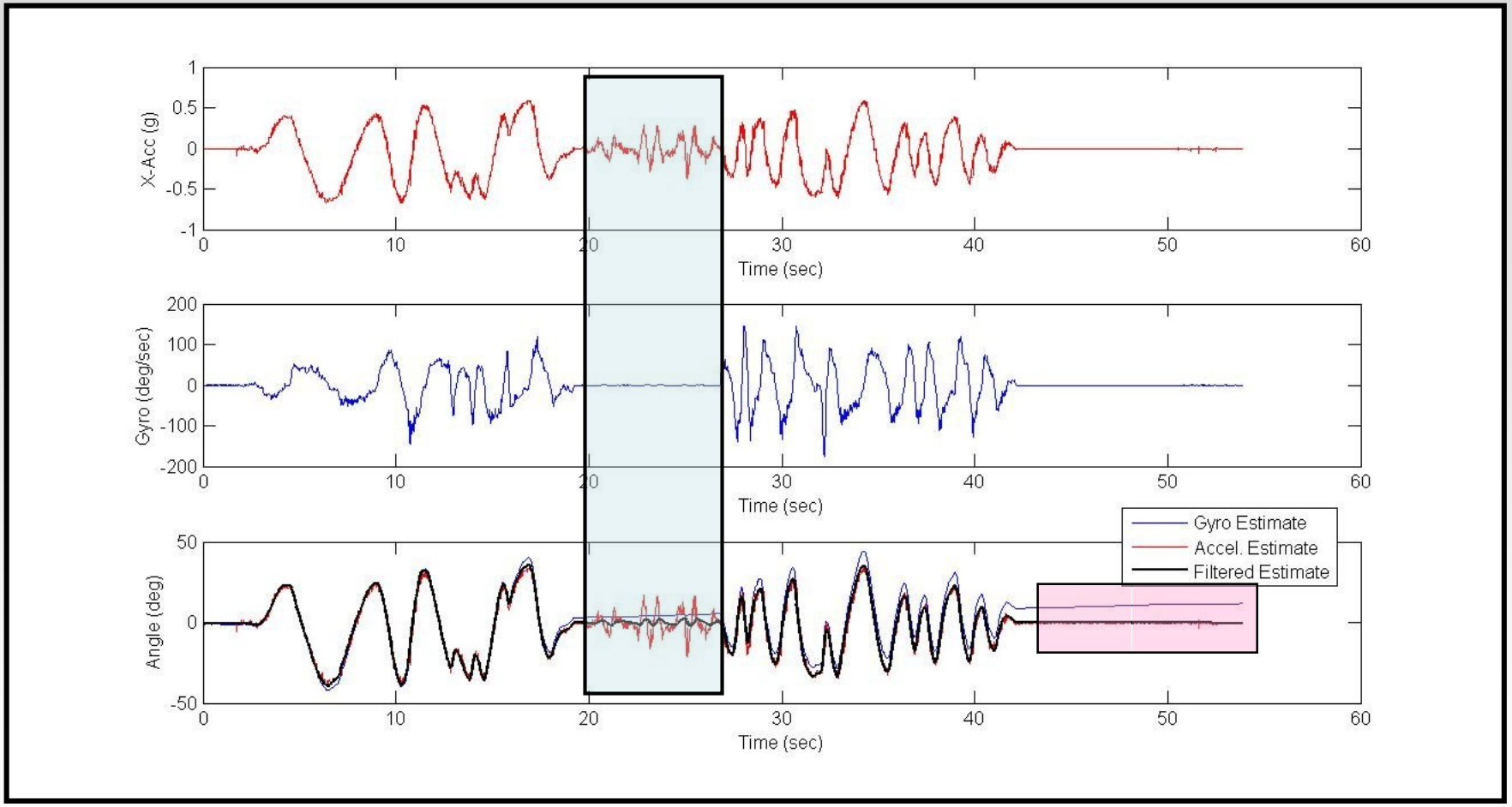
Besides the fact that this is probably a worst-case scenario (the gyro should never be that far offset), a constant angle offset is much easier to deal with than a drifting angle offset. You could, for example, just rotate the accelerometer 3.75º in the opposite direction to accommodate for it.

Enough theory. Time for some experimental results.

Control Platform:	Custom PIC-based wireless controller, 10-bit ADCs. Based on the Machine Science XBoard*.
Data Acquisition:	Over a serial USB radio, done in Visual Basic.
Gyroscope:	ADXRS401, Analog Devices iMEMS 75 º/sec angular rate sensor
Accelerometer:	ADXL203, Analog Devices iMEMS 2-axis accelerometer

*http://www.machinescience.org

Enough theory. Time for some experimental results.

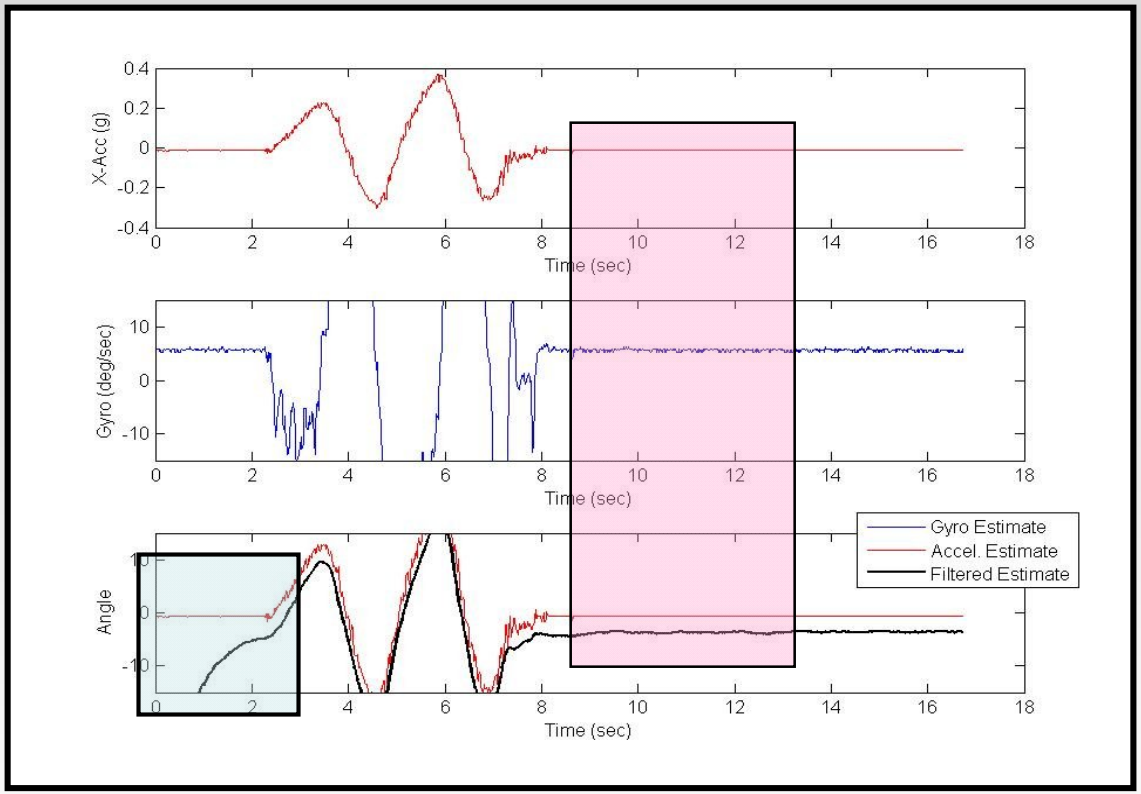


Sample Rate: 79 Hz
Filter Coefficients: 0.98 and 0.02

} Time Constant: 0.62 sec

Notice how the filter handles both problems: horizontal acceleration disturbances while not rotating (highlighted blue) and gyroscope drift (highlighted red).

Enough theory. Time for some experimental results.



Sample Rate: 84 Hz
Filter Coefficients: 0.98 and 0.02

} Time Constant: 0.58 sec

Two things to notice here: First, the unanticipated startup problem (blue highlight). This is what can happen if you don't initialize your variables properly. The long time constant means the first few seconds can be uncertain. This is easily fixed by making sure all important variables are initialized to zero, or whatever a "safe" value would be. Second, notice the severe gyro offset (red highlight), about 6 º/sec, and how it creates a constant angle offset in the angle estimate. (The angle offset is about equal to the gyro offset multiplied by the time constant.) This is a good worst-case scenario example.

Conclusions

I think this filter is well-suited to D.I.Y. balancing solutions for the following reasons:

1. It seems to work. The angle estimate is responsive and accurate, not sensitive to horizontal accelerations or to gyroscope drift.
2. It is microprocessor-friendly. It requires a small number of floating-point operations, but chances are you are using these in your control code anyway. It can easily be run in control loops at or above 100 Hz.
3. It is intuitive and much easier to explain the theory than alternatives like the Kalman filter. This might not have anything to do with how well it works, but in educational programs like FIRST, it can be an advantage.

Before I say with 100% certainty that this is a perfect solution for balancing platforms, I'd like to see it tested on some hardware…perhaps a D.I.Y. Segway?

Also, I'm not sure how much of this applies to horizontal positioning. I suspect not much: without gravity, there is little an accelerometer can do to give an absolute reference. Sure, you can integrate it twice to estimate position, but this will drift a lot. The filtering technique, though, could be implemented with a different set of sensors – maybe an accelerometer and an encoder set – but the scenario is not exactly analogous. (Encoders are not absolute positioning devices…they can drift too if wheels lose traction. A better analogy for horizontal positioning would be using GPS to do the long-term estimate and inertial sensors for the short-term integrations.)