

CESE4030

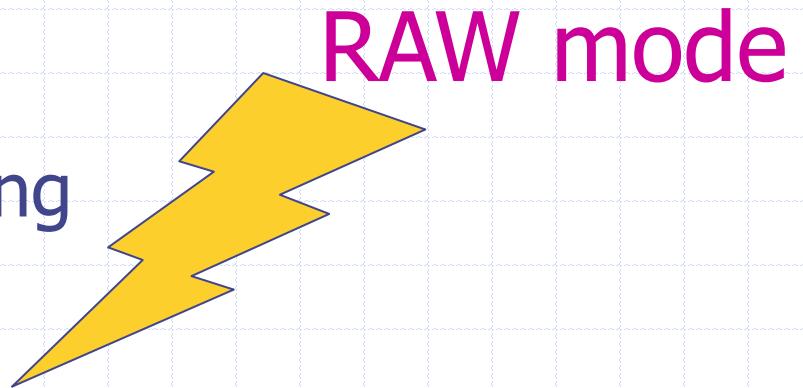
Embedded Systems Laboratory

System Integration

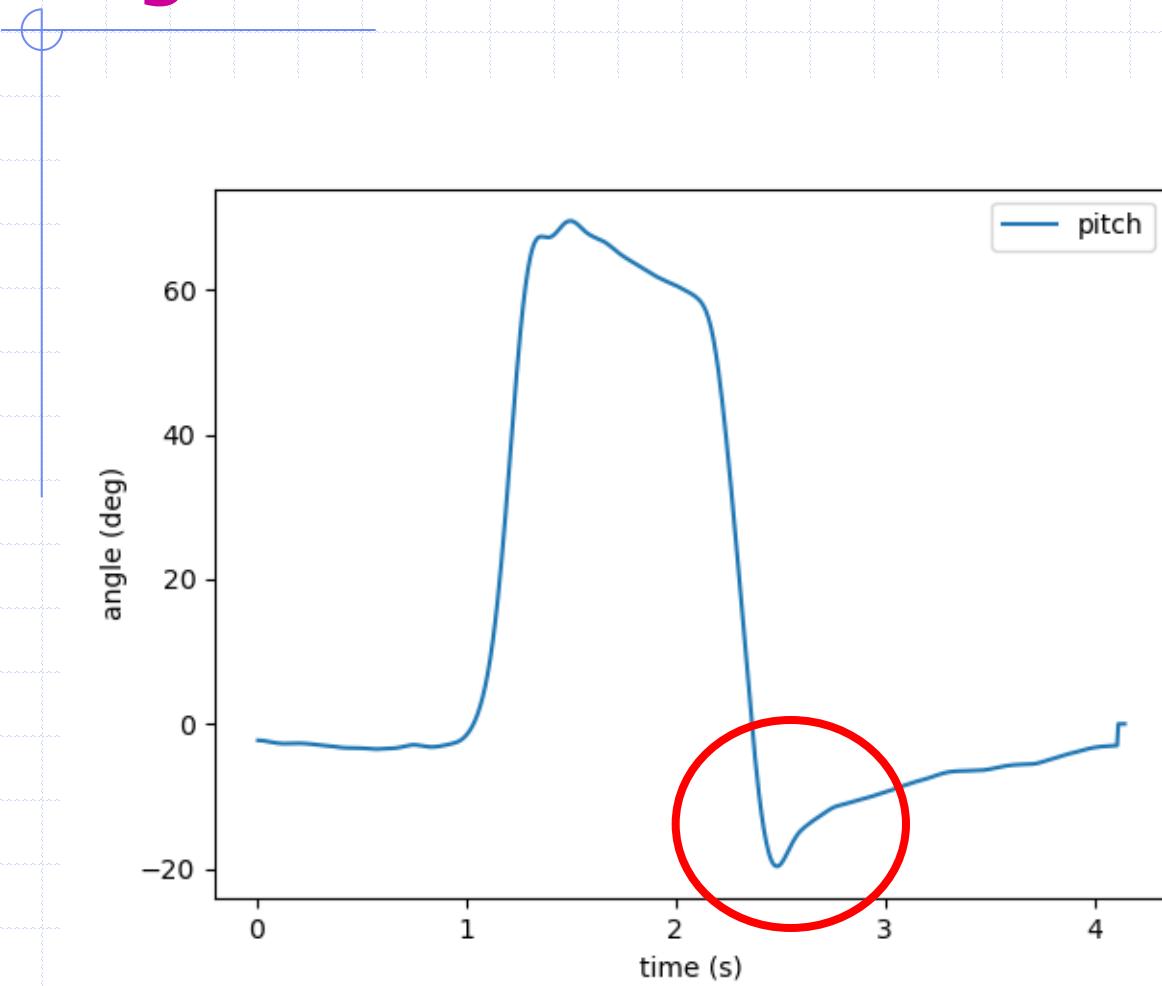
takes time

Putting it all together

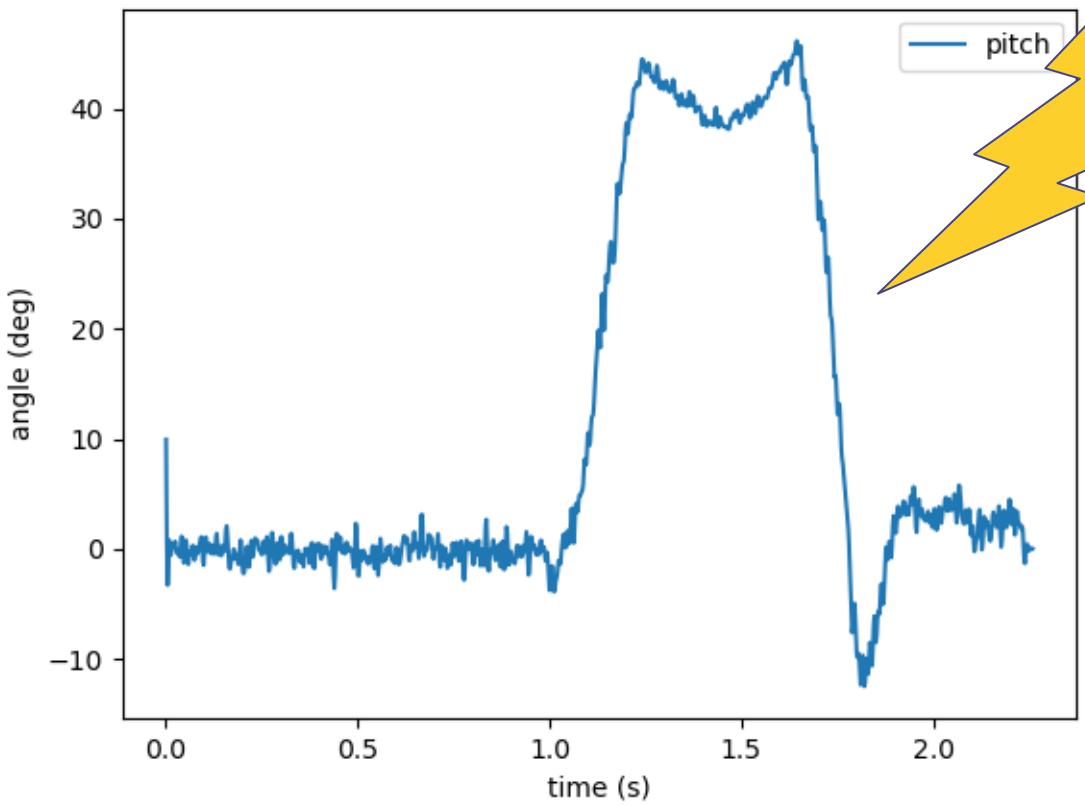
1. System identification
2. Embedded programming
3. Control theory
4. Signal processing



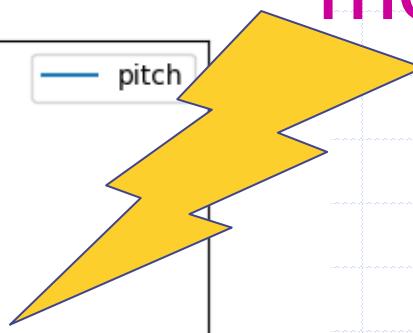
Signals – DMP



Signals – accelerometer

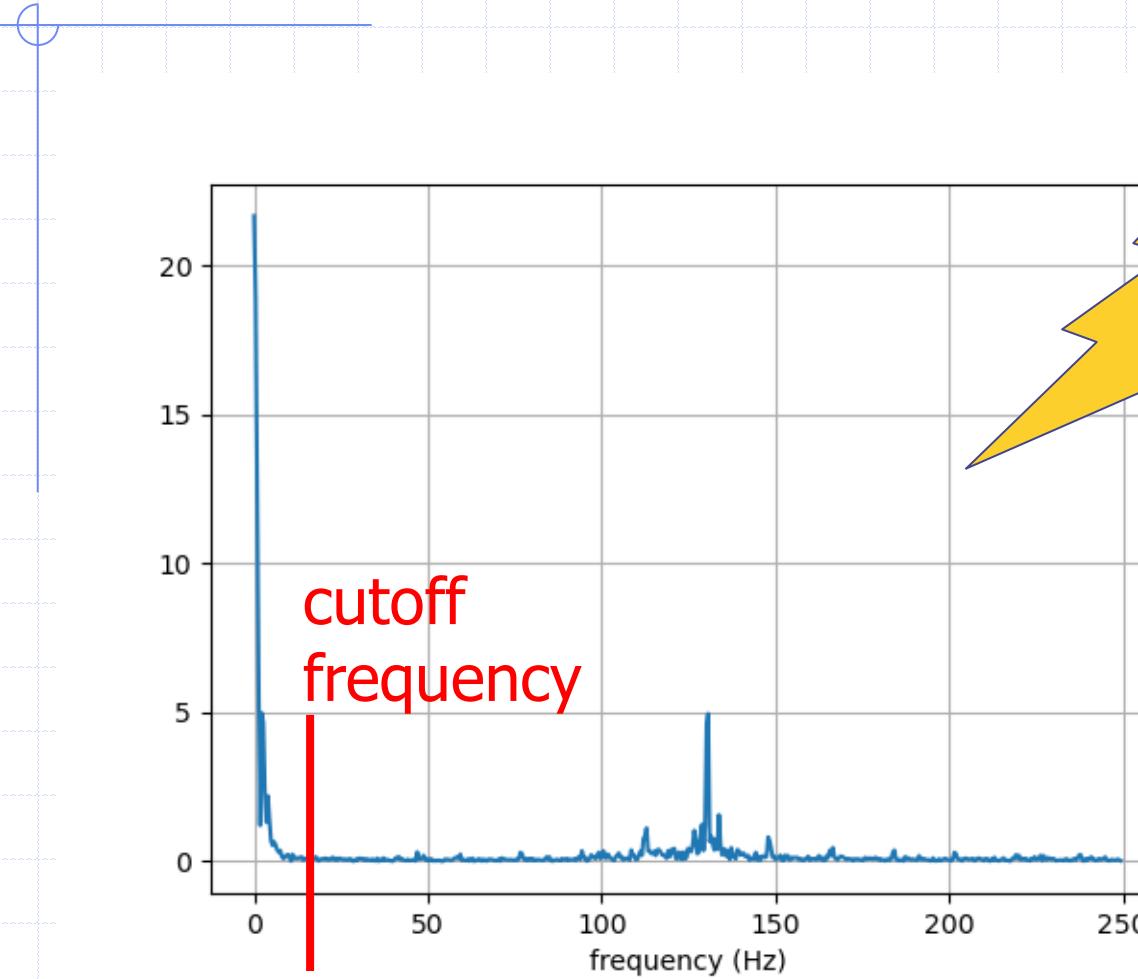


motors on

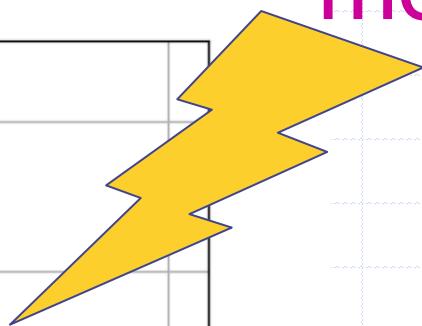


$$\theta = \arctan\left(\frac{sax}{saz}\right)$$

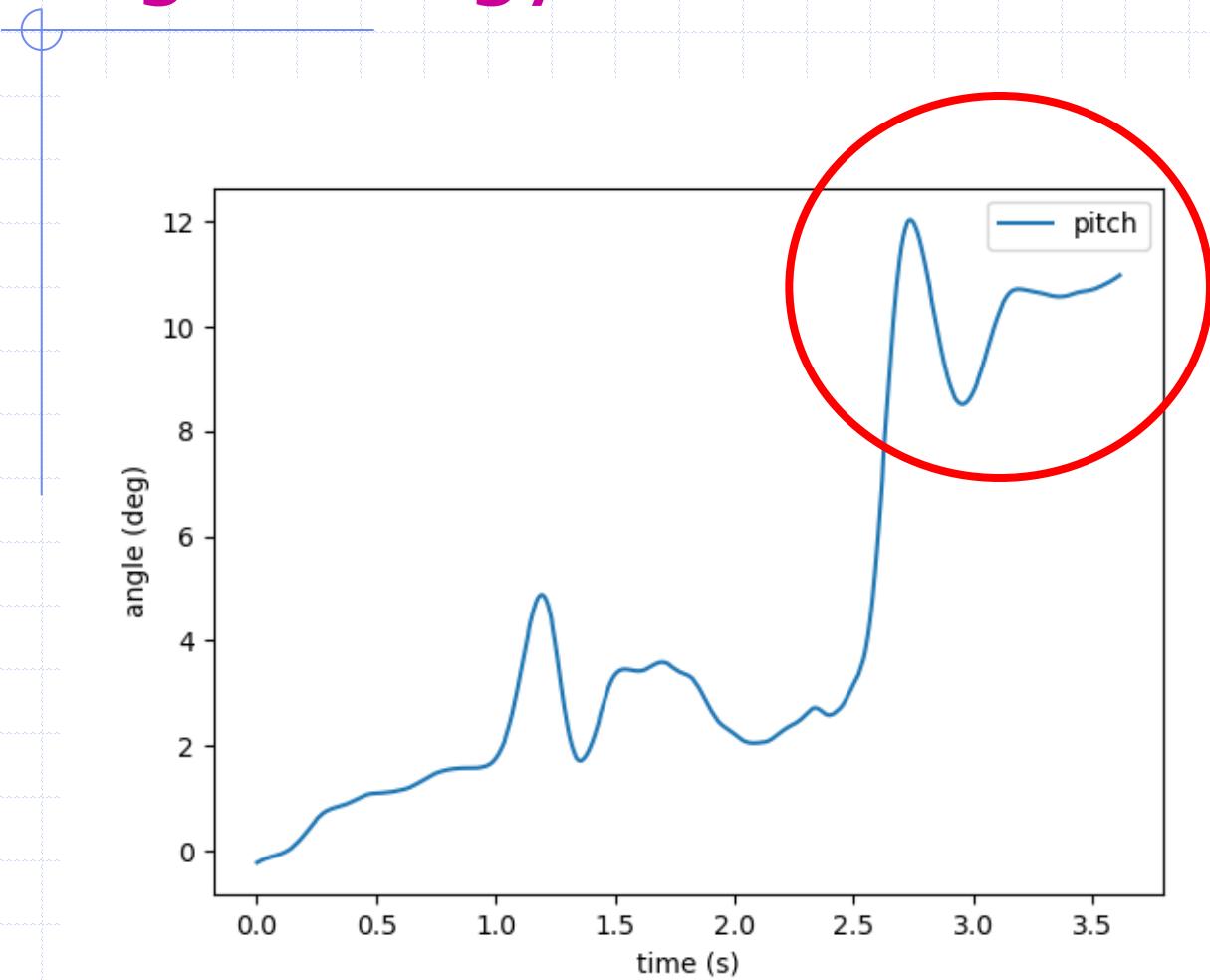
FFT – accelerometer



motors on

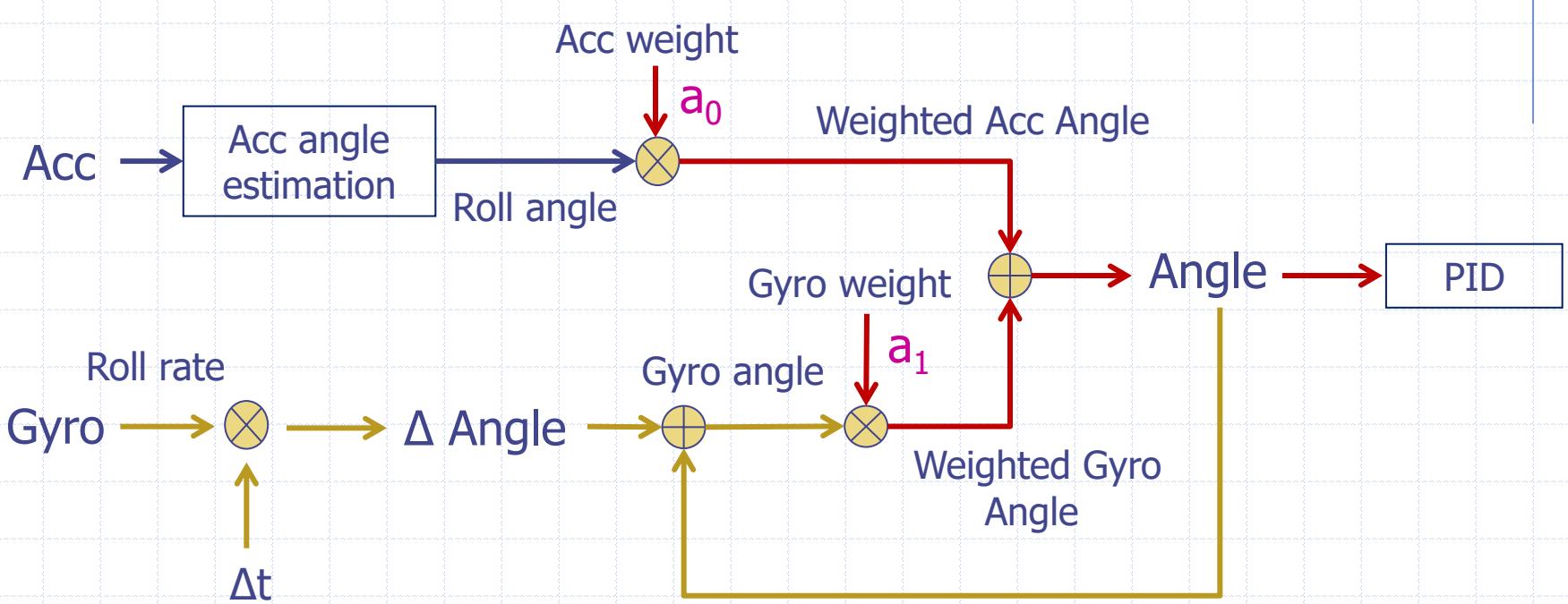


Signals – gyro



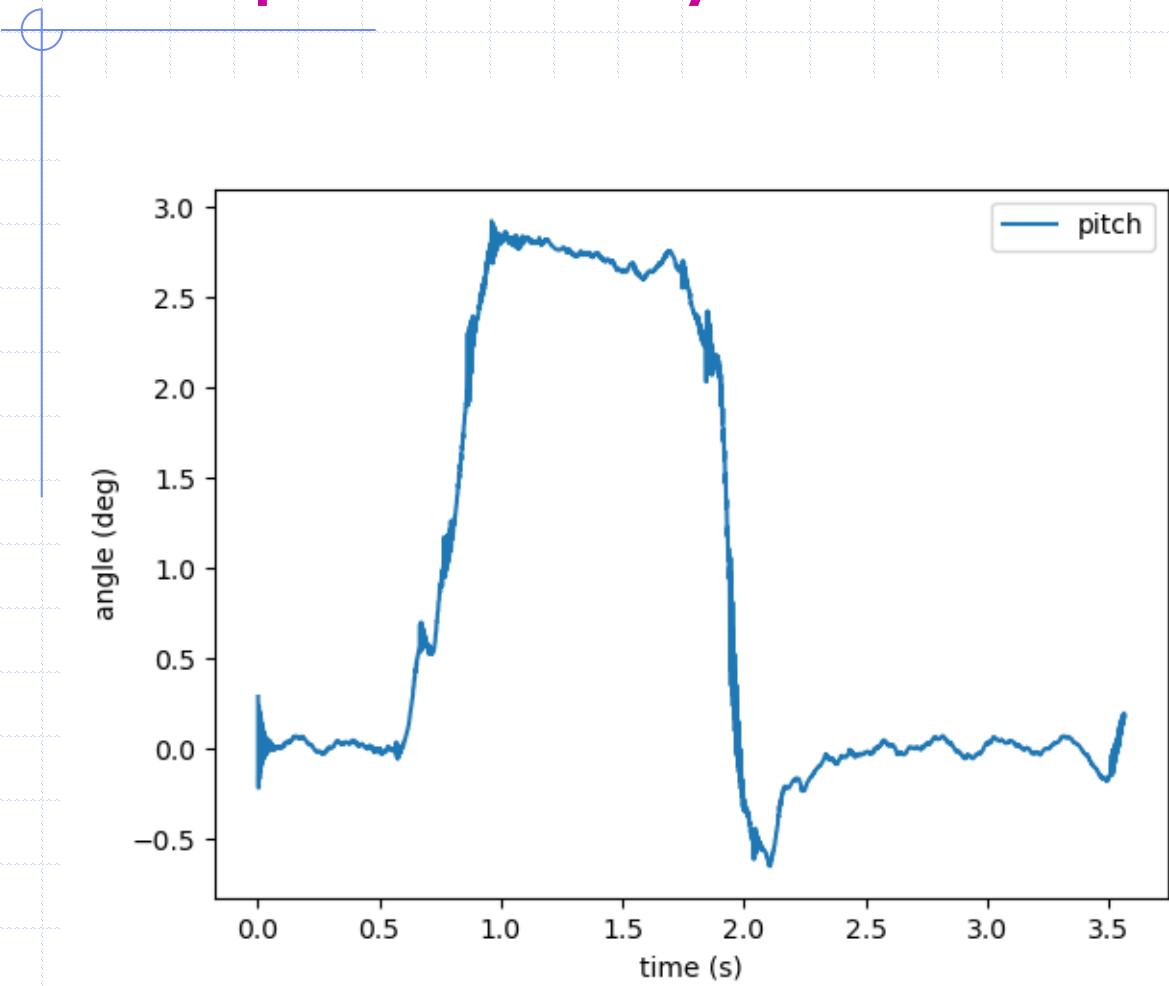
$$\theta = \int q$$

Complementary Filter

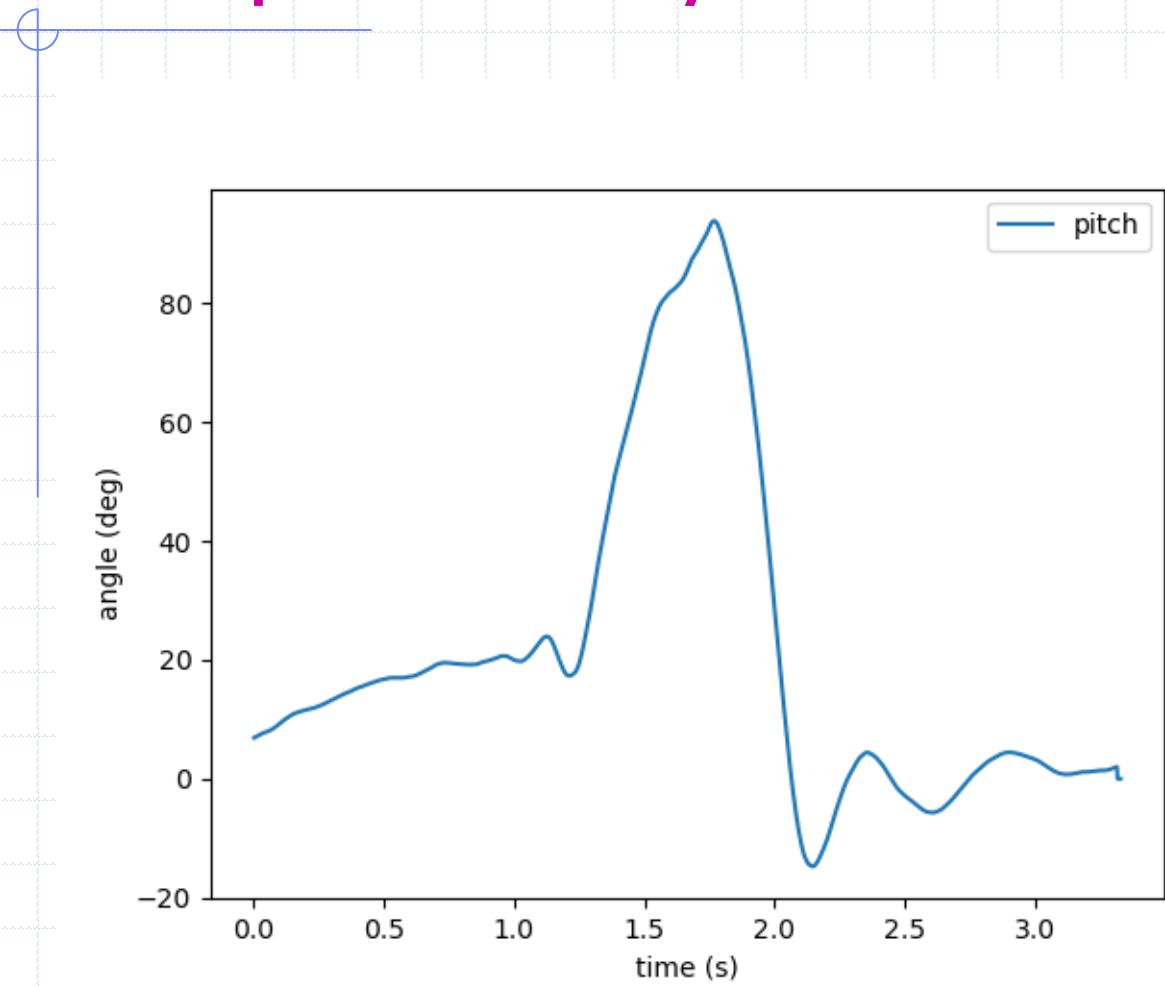


$$\theta[n] = a_0 * s\alpha[n] + (1-a_0) * (\theta[n-1] + s\alpha[n] * \Delta t)$$

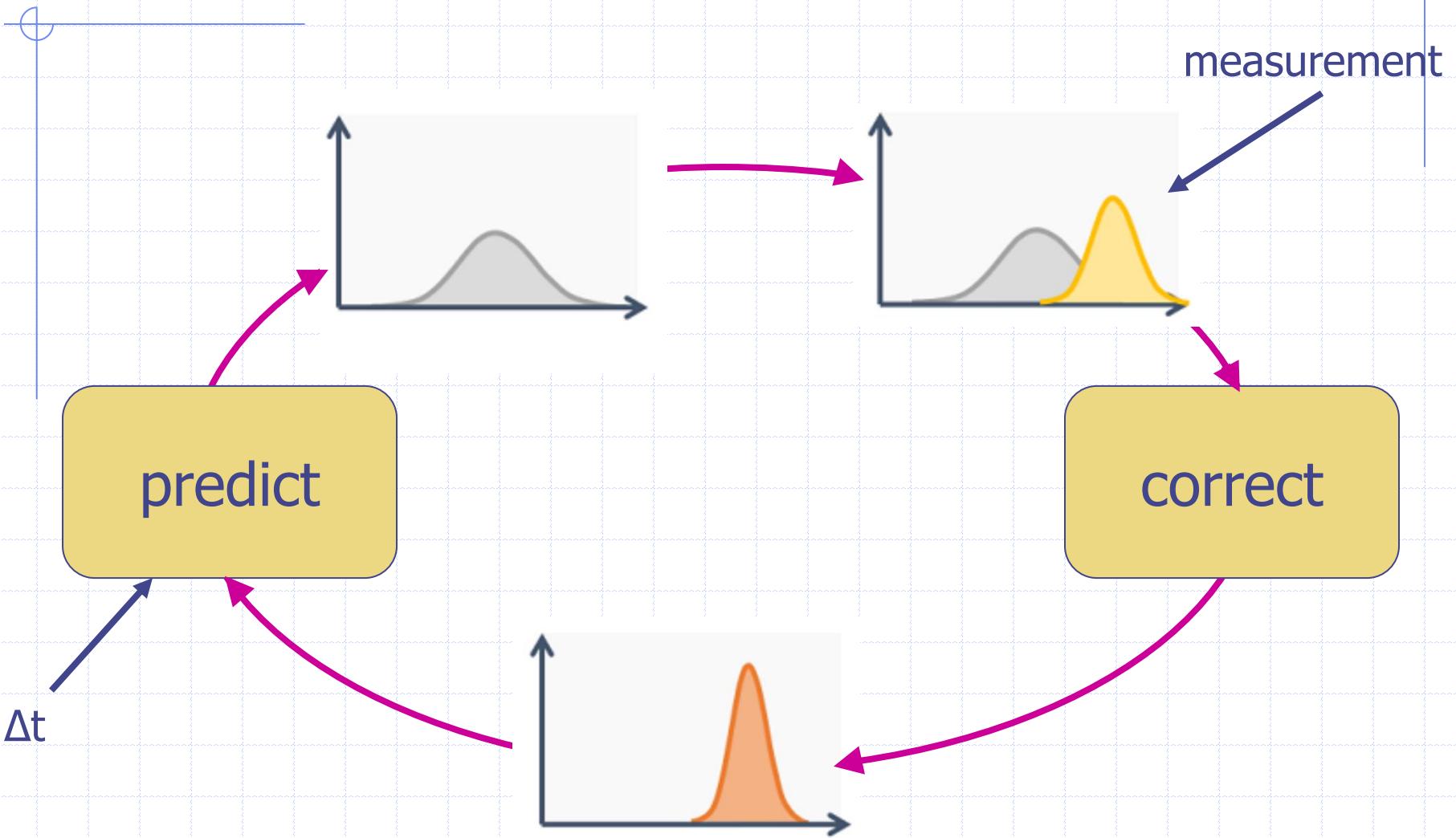
Complementary Filter – 90%



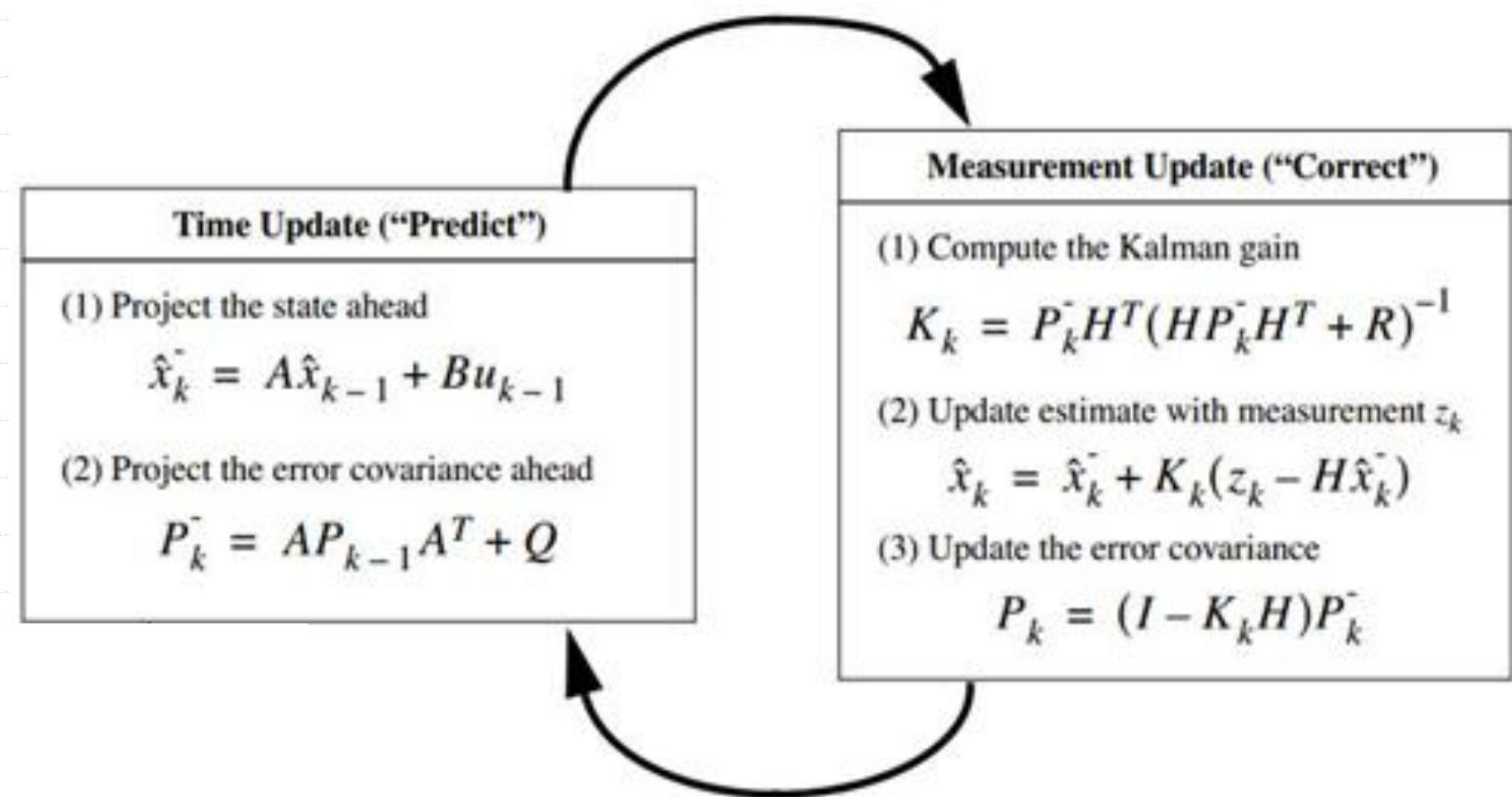
Complementary Filter – 99%



Kalman Filter – sensor fusion



Kalman Filter – sensor fusion



Kalman Filter (simplified)

- ◆ Sensor Fusing: gyro and accel share same information

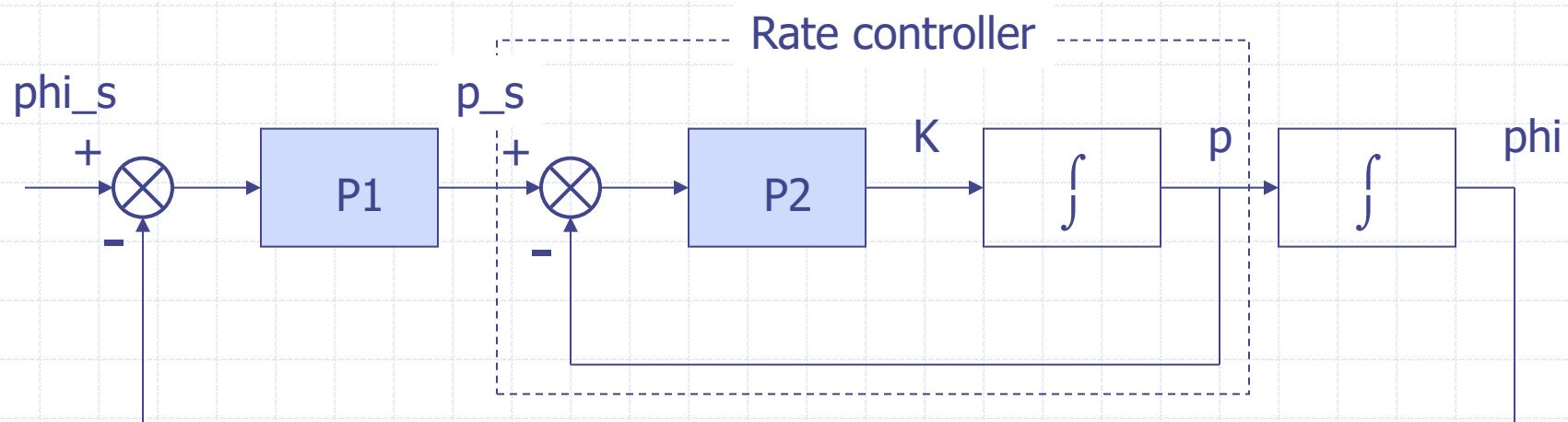


- ◆ Integrate sp to phi
- ◆ Adjust integration for sp (drift) bias b by comparing phi to sphi, averaged over *long* period ($\phi \sim \text{constant}$)
- ◆ Return phi, and p (= sp – bias)

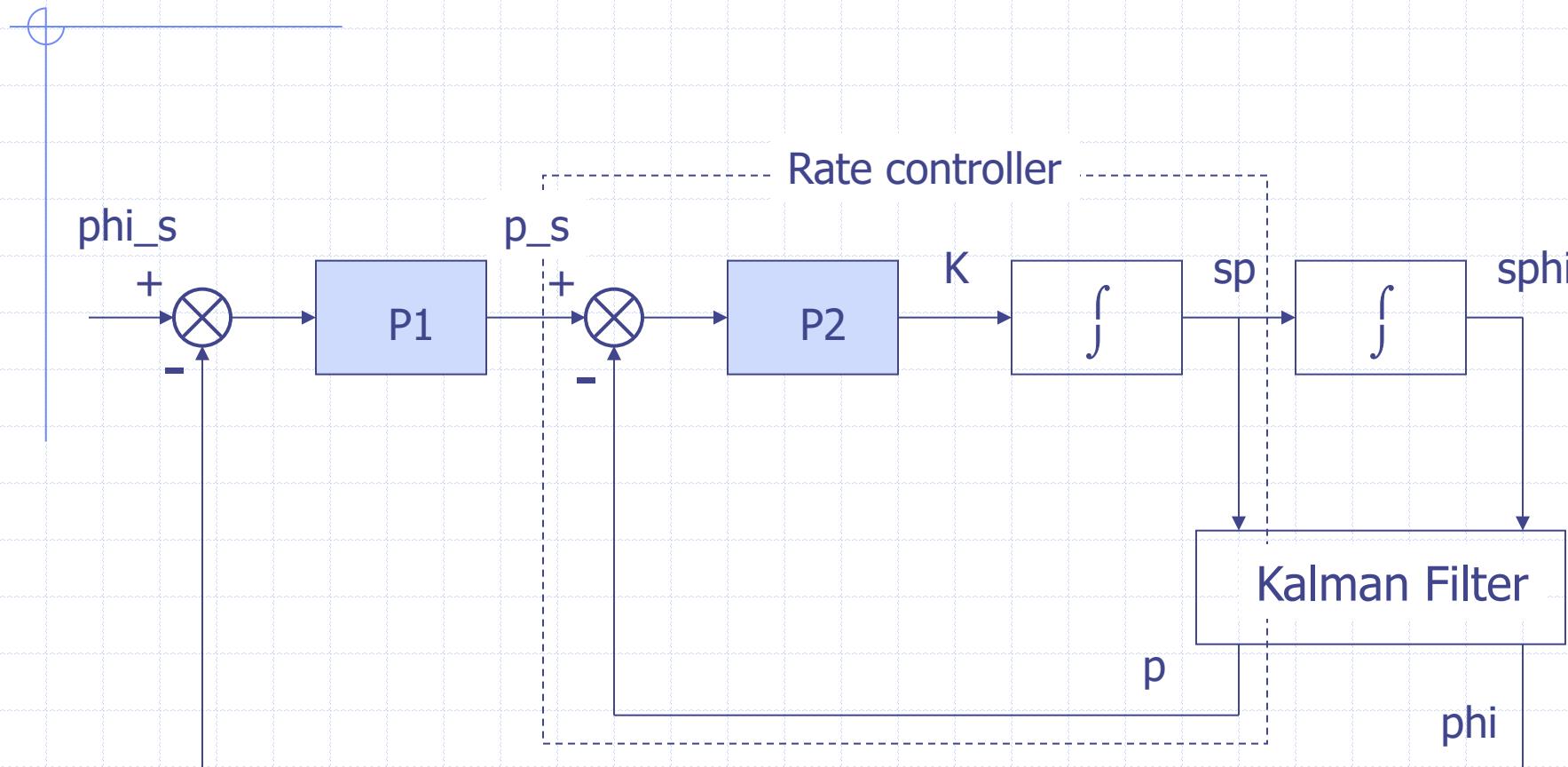
Algorithm

- ◆ $p = sp - b$ // estimate real p
 - ◆ $\phi_i = \phi_i + p * P2PHI$ // predict ϕ
 - ◆ $e = \phi_i - s\phi_i$ // compare to measured ϕ
 - ◆ $\phi_i = \phi_i - e / C1$ // correct ϕ to *some* extent
 - ◆ $b = b + (e/P2PHI) / C2$ // adjust bias term
-
- ◆ P2PHI: depends on loop freq -> compute/measure
 - ◆ C1 small: believe $s\phi_i$; C1 large: believe sp
 - ◆ C2 large (typically > 1,000 C1): slow drift

Cascaded P control



Cascaded P control + Kalman filter



Fixed-point Arithmetic

- Simple/cheap microcontrollers have no floating-point unit
- Software floating-point often (too) slow
- Need to implement filters in fixed-point arithmetic

2's-complement bit representation (e.g., 32 bits, 14 bits fraction):

Are we done?			
3.75:	0000000000000000000011	1100000000000000	
0.02:	0000000000000000000000	00000101001001	
-1.5:	00000000000000000001	1000000000000000	$\wedge -1 + 1 \Rightarrow$
	1111111111111110	1000000000000000	

Integer part

• Fractional part

Fixed-point Arithmetic

- Addition, subtraction as usual
- Multiplication: result must be post-processed:
 - make sure intermediate fits in variable! (e.g., 32 bits)
 - shift right by $|fraction|$

Example multiplication (32 bits, 14 bits fraction):

3.75: 00000000000000001111000000000000 times:

-1.5: 11111111111111110100000000000000 equals:

10100110000000000000000000000000

(value just fits in 32 bits!)

(now shift right by 14 bits and sign-extend):

11111111111111010011000000000000 which is:

-5.625 1111111111111010 0110000000000000

Filter Example

- Second-order Butterworth LP Filter $f_c = 10\text{Hz}$, $f_s = 1250\text{Hz}$

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] - b_1 y[n-1] - b_2 y[n-2]$$

- Coefficients:

$$a_0 = 0.0006098548$$

$$b_0 = 1$$

$$a_1 = 2 a_0$$

$$b_1 = -1.9289423$$

$$a_2 = a_0$$

$$b_2 = 0.9313817$$

Bit representation (e.g., 32 bits, 14 bits fraction):

a[0] 00000000000000000000000000000000 000000000001010 ($a_0 << 14$)

a[1] 00000000000000000000000000000000 000000000010100

a[2] 00000000000000000000000000000000 000000000001010

b[1] 00000000000000000000000000000001 $11101101110100 \wedge -1 + 1$

b[2] 00000000000000000000000000000000 11101110011100

pkg load signal

```
% 2nd-order Butterworth - fc 10 Hz at fs 1250 Hz  
nmax = 400;  
n = (1:nmax);  
fs = 1250;  
f = 10;  
t = n / fs;  
x = sin(2*pi*f*n/1250);  
  
% real coefficients  
[a,b] = butter(2, f/(fs/2));  
a0 = a(1); a1 = a(2); a2 = a(3); b1 = b(2); b2 = b(3);  
x1 = 0; x2 = 0; y1 = 0; y2 = 0;  
for i = 1:nmax  
    x0 = x(i);  
    y0 = a0 * x0 + a1 * x1 + a2 * x2 - b1 * y1 - b2 * y2;  
    y(i) = y0;  
    x2 = x1; x1 = x0; y2 = y1; y1 = y0;  
end;
```

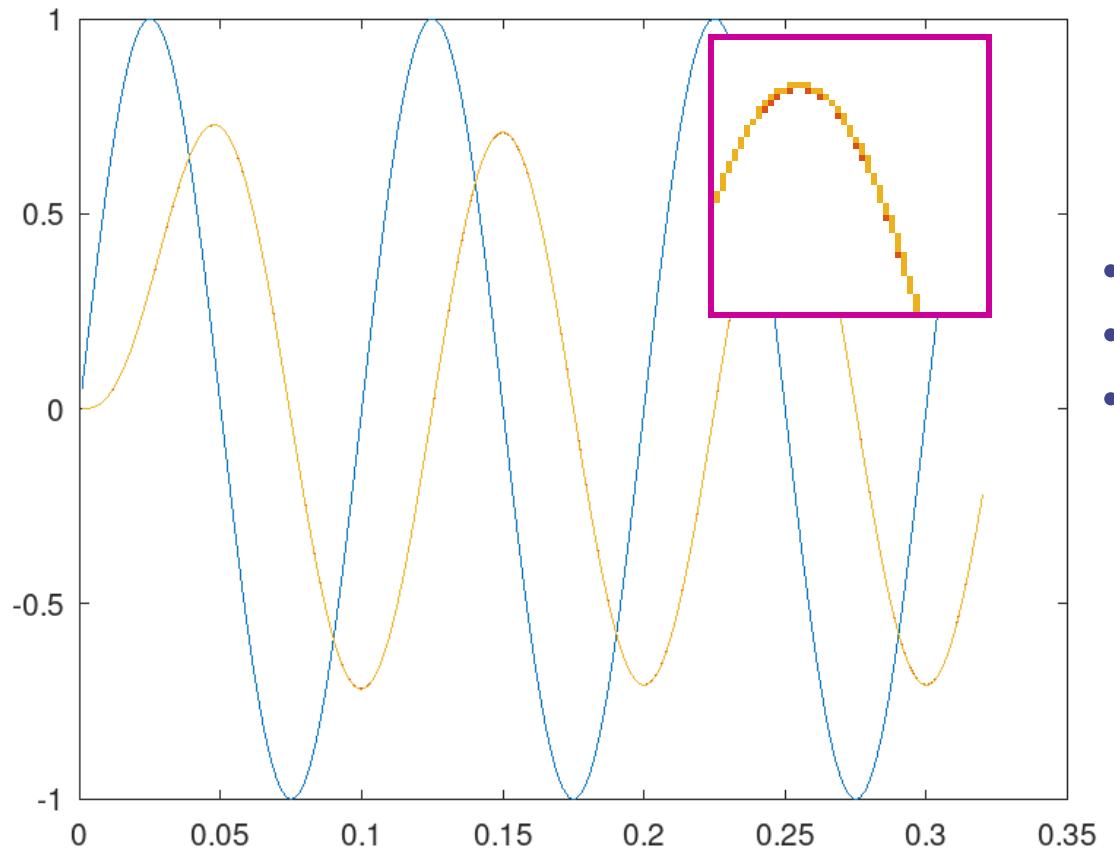


Filter Example

```
% 14 bits fraction, approximated in terms of powers of 2
a0 = 8 + 2; % 9.9919
a1 = 2 * a0;
a2 = a0;
b0 = 16384; % 0x1<<14
b1 = -(16384 + 8192 + 4096 + 2048 + 512 + 256 + 64 + 32 + 16 + 4);
b2 = 8192 + 4096 + 2048 + 512 + 256 + 128 + 16 + 8 + 4;

x1 = 0; x2 = 0; y1 = 0; y2 = 0;
for i = 1:nmax
    x0 = x(i);
    y0 = (a0 * x0 + a1 * x1 + a2 * x2 - b1 * y1 - b2 * y2) / 16384;
    z(i) = y0;
    x2 = x1; x1 = x0; y2 = y1; y1 = y0;
end;
```

Results



- blue: input
- red: filtered (real)
- yellow: filtered (FP)

Implementation (high-cost)

```
int mul(int c, int d) {  
    int result = c * d;  
    return (result >> 14);  
}  
  
void filter() {  
    y0 = mul(a0,x0) + mul(a1,x1) + mul(a2,x2) -  
        mul(b1,y1) - mul(b2,y2);  
    x2 = x1; x1 = x0; y2 = y1; y1 = y0;  
}
```

Scaling: tips and tricks

- One size fits all? NO!
 - number of bits depends on needed precision (sensor vs. joystick)
- special case for proportional controller: $P * \epsilon$
- $fp_n * fp_n = fp_{2n}$ (overflow! requires an additional shift)
- scalar * $fp_n = fp_n$ (overflow? no shift needed)
- $fp_m * fp_n = fp_{m+n}$ (when P can't be represented as a scalar)
- document precision for every data type (part of softw arch)
- fp_n to scalar
 - be patient, shift at last instant (when feeding the engines)

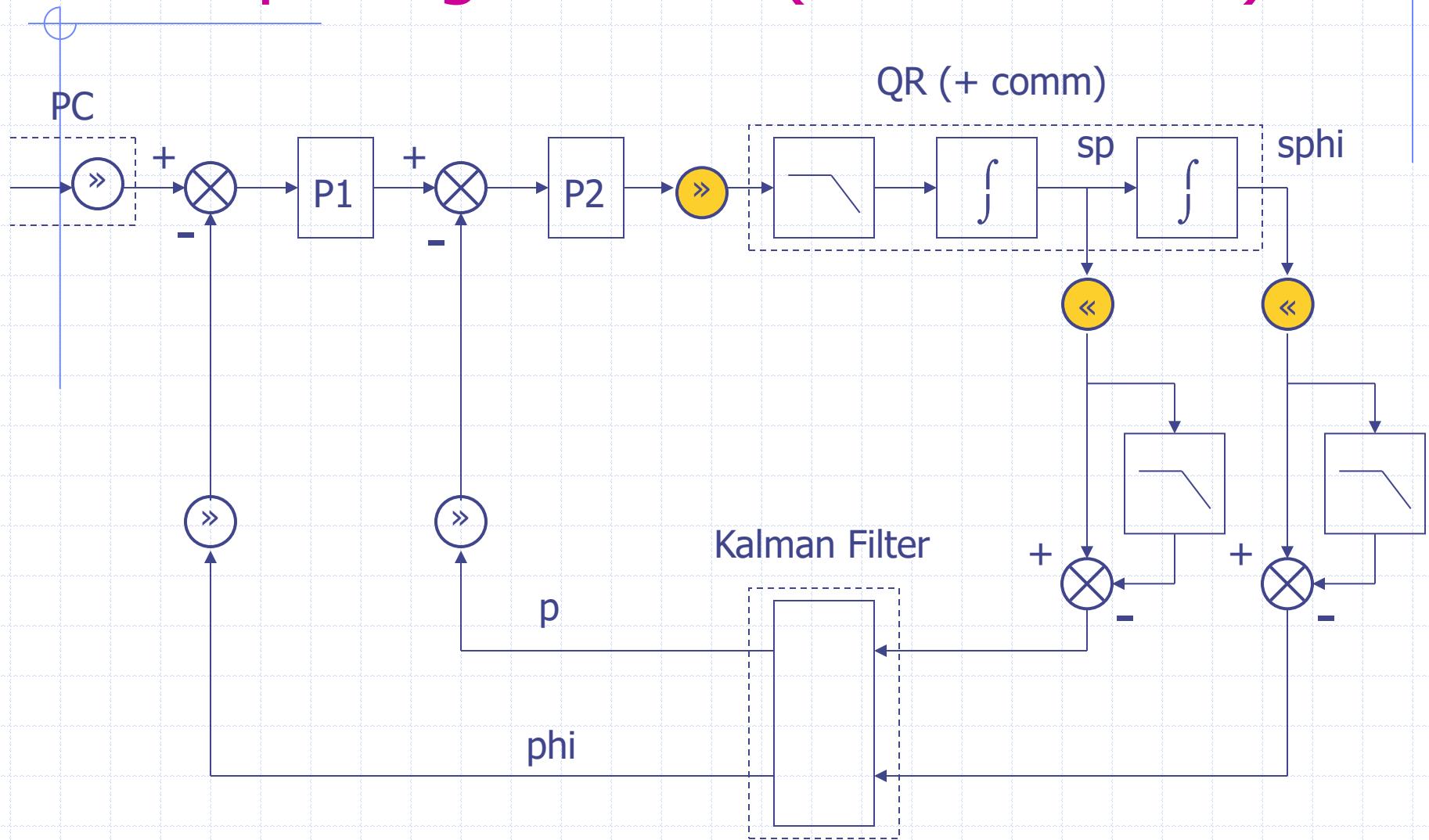
Fixed – the Rust way

The screenshot shows a web browser window displaying the Rust documentation for the `fixed` crate. The URL in the address bar is `docs.rs/fixed/latest/fixed/`. The page title is "fixed". On the left, there's a sidebar with navigation links: All Items, Modules, Macros, Structs, Enums, and Crates. The "Crates" section is expanded, and "fixed" is selected. The main content area has a search bar with placeholder text "Click or press 'S' to search, '?' for more options...". Below it, there's a "source" link with a minus sign icon. The main content starts with a heading "Fixed-point numbers". It says: "The `fixed` crate provides fixed-point numbers." followed by a bulleted list:

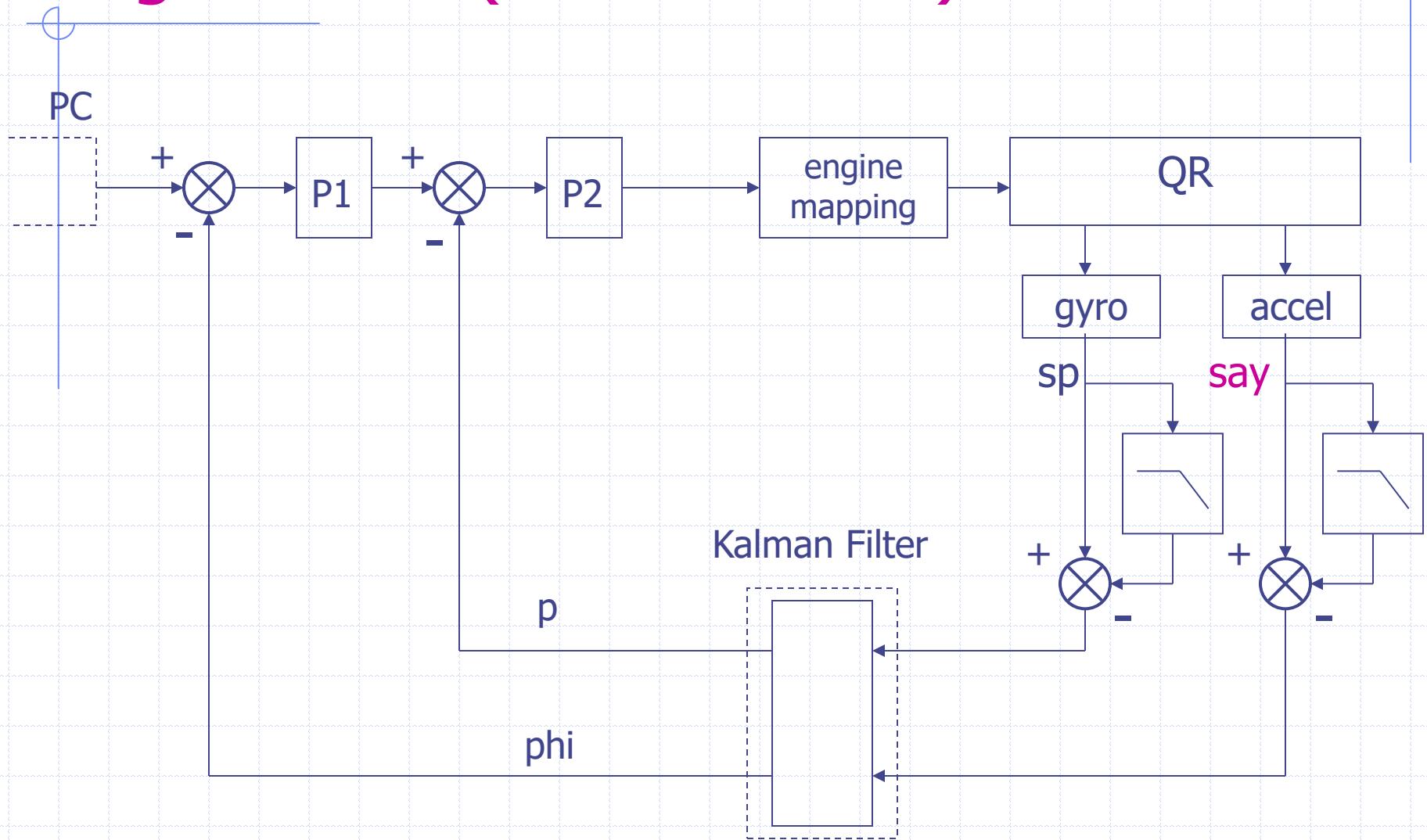
- `FixedI8` and `FixedU8` are eight-bit fixed-point numbers.
- `FixedI16` and `FixedU16` are 16-bit fixed-point numbers.
- `FixedI32` and `FixedU32` are 32-bit fixed-point numbers.
- `FixedI64` and `FixedU64` are 64-bit fixed-point numbers.
- `FixedI128` and `FixedU128` are 128-bit fixed-point numbers.

Below this, there's a paragraph about the properties of fixed-point numbers: "An n -bit fixed-point number has $f = \text{Frac}$ fractional bits where $0 \leq f \leq n$, and $n - f$ integer bits. For example, `FixedI32<U24>` is a 32-bit signed fixed-point number with $n = 32$ total bits, $f = 24$ fractional bits, and $n - f = 8$ integer bits. `FixedI32<U0>` behaves like `i32`, and `FixedU32<U0>` behaves like `u32`".

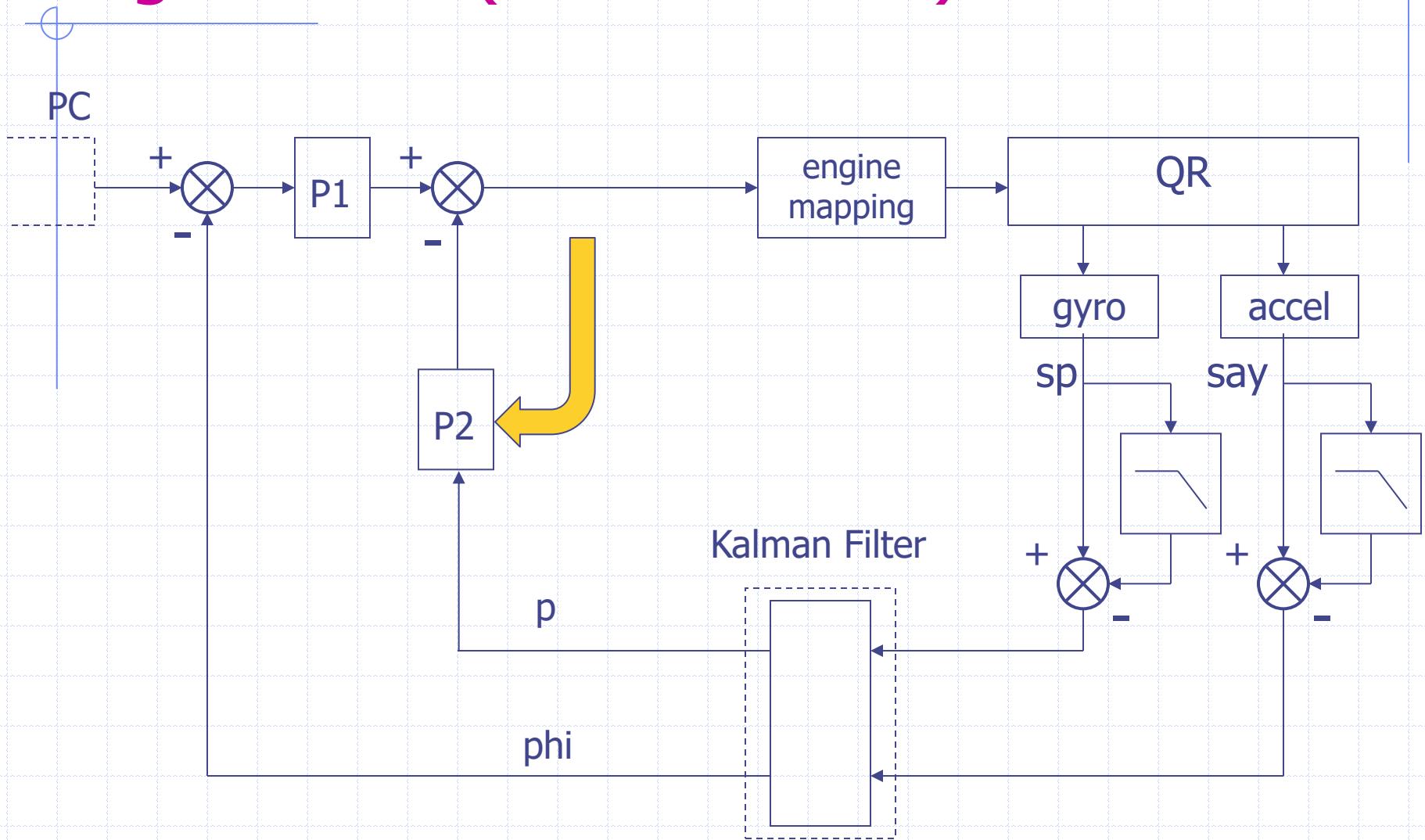
Example Signal Flow (Kalman Filter)



Signal Flow (Kalman Filter)



Signal Flow (Kalman Filter)



PID tuning

D: address oscillations

P: address errors

[I: address steady state]

Guard
ratio

Integral
windup

Max the
error

What's next?

- ◆ Week 3.6 – midterm: yaw control
- ◆ Week 3.9 – final demo: full cntrl, raw, ...
- ◆ Week 3.9 – report
 - what you did and **why**
 - results: plots, perf numbers
 - analysis/reflection