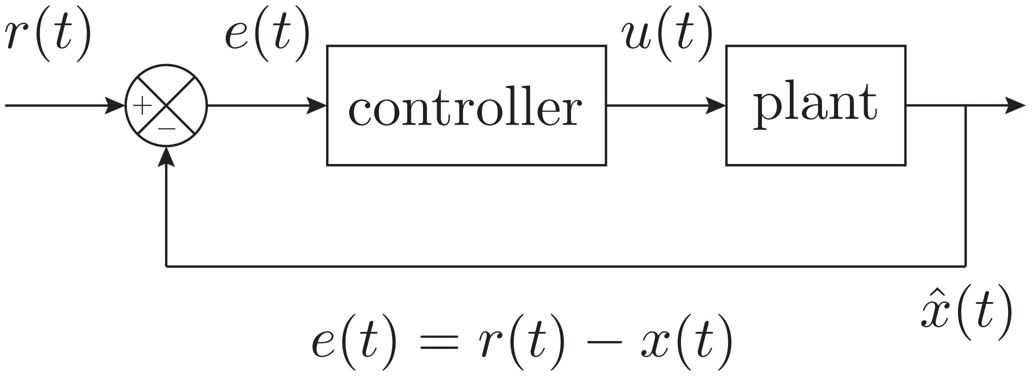
MBot
Motor Speed Control

Lecture 3

M | ROBOTICS

Feedback Controllers

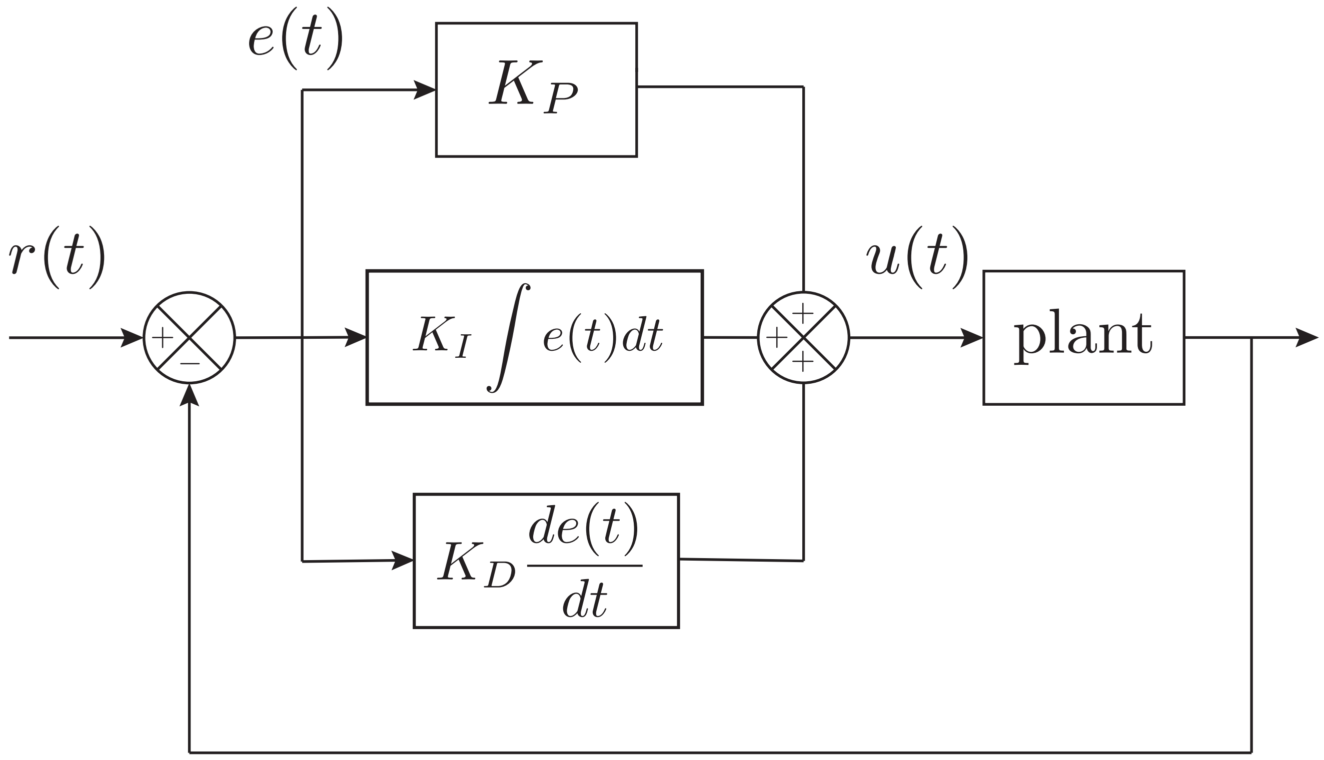


- We want a controller such that after applying feedback, our plant behavior to be well defined
- Our controller takes in the error between a reference and measured value and computes a control signal
- With feedback we can drive the error to zero
- We are interested in the error dynamics of the controlled system

PID Control

- Proportional Integral Derivative control
- Simple to implement, simple to understand
- Proportional term corrects for current error
- Integral term corrects for past error
- Derivative term corrects for future error

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt}$$



Discrete Time PID

- Discrete PID can be implemented easily and efficiently
- Need to store previous I-term and previous feedback reading only
- More advanced features can be added to this simple implementation

$$u(k) = u_P(k) + u_I(k) + u_D(k)$$

$$u_P(k) = K_P(x_r(k) - \hat{x}(k))$$

$$u_I(k) = u_I(k-1) + K_I(x_r(k) - \hat{x}(k))$$

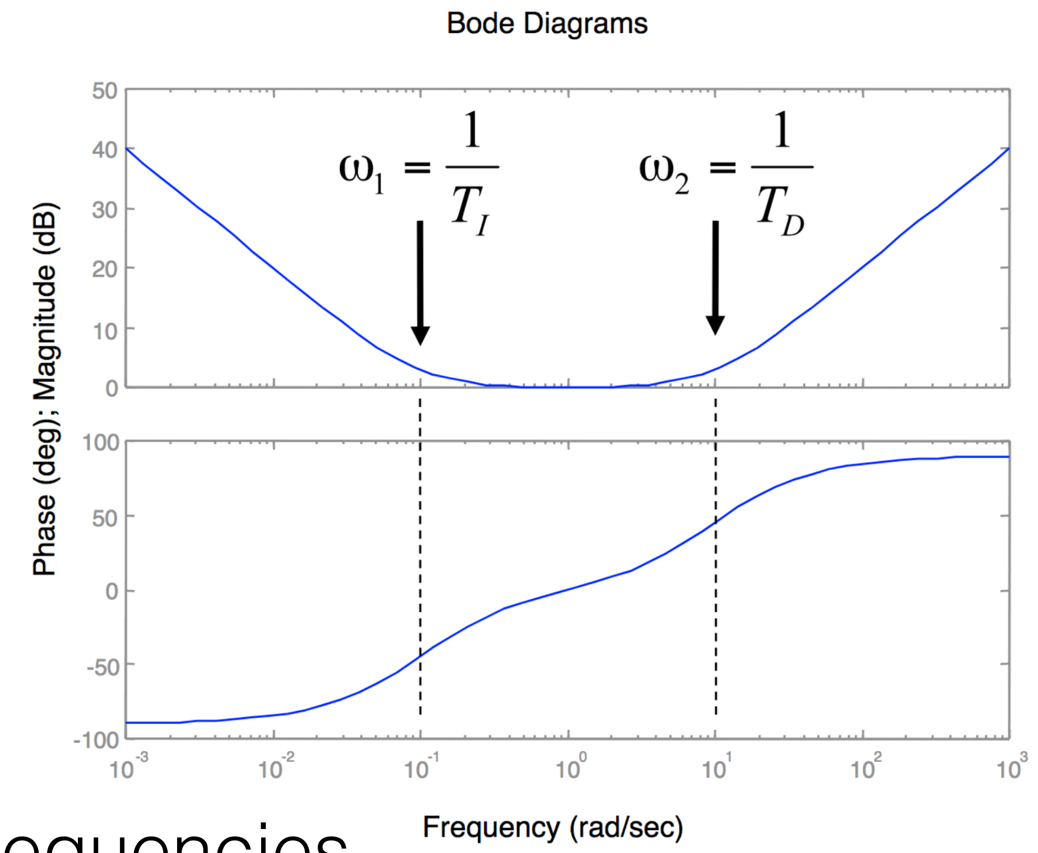
$$u_D(k) = K_D(\dot{x}_r(k) - \dot{\hat{x}}(k))$$

$$\dot{\hat{x}}(k) = \frac{\hat{x}(k) - \hat{x}(k-1)}{\Delta t}$$

PID in Laplace Domain

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt}$$

$$\begin{aligned} U(s) &= K_p + \frac{K_I}{s} + K_D s \\ &= K(1 + \frac{1}{T_I} + T_D s) \\ &= K \frac{T_D}{T_I} \frac{(s + 1/T_I)(s + 1/T_D)}{s} \end{aligned}$$



- Integral Increases gain at low frequencies
- Derivative increases gain at high frequencies

Problems with Integral Control

- Windup: control or actuator saturates (i.e. max PWM)
- Integral term builds up due to error
- Can overshoot and take a long time to recover

Solutions

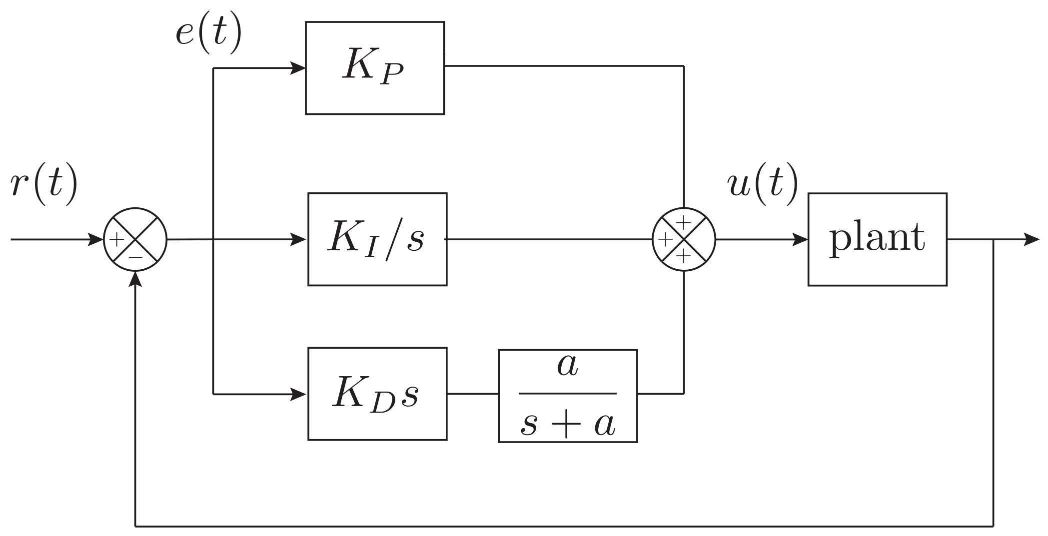
- Saturate integral term
- Reset integral term to zero when setpoint is reached

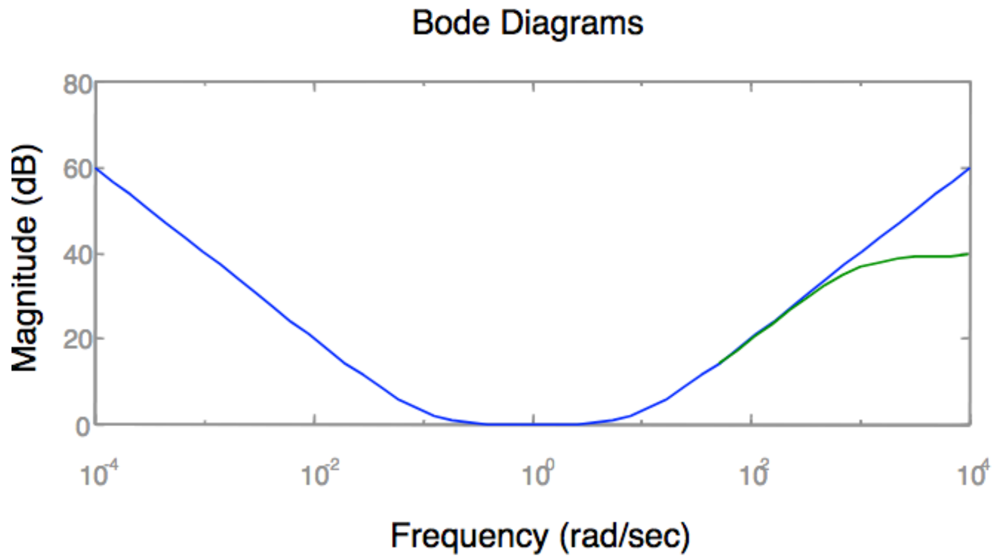
Problems with Derivative Control

- Derivative control amplifies high frequency noise in the error signal
- Step inputs can produce large derivative response

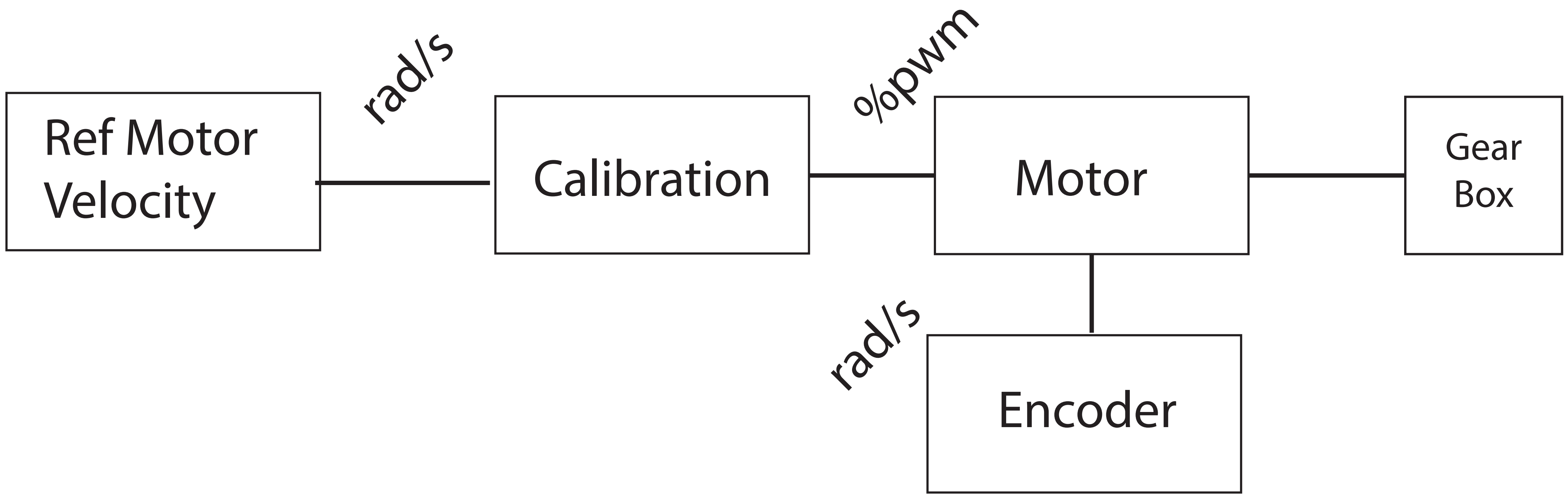
Solutions

- Use high frequency roll off filter (LPF) in derivative term
  - 1st order filter will give constant gain at high frequencies
  - Higher order filters will reduce gain
- Avoid step inputs to your controller – LPF on reference signal

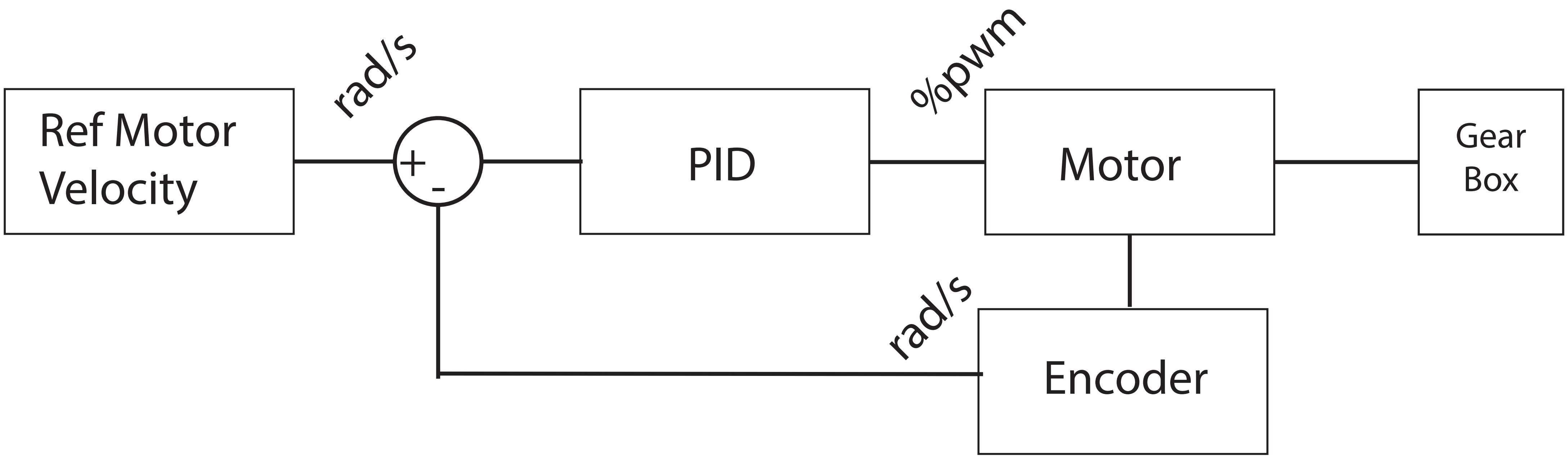




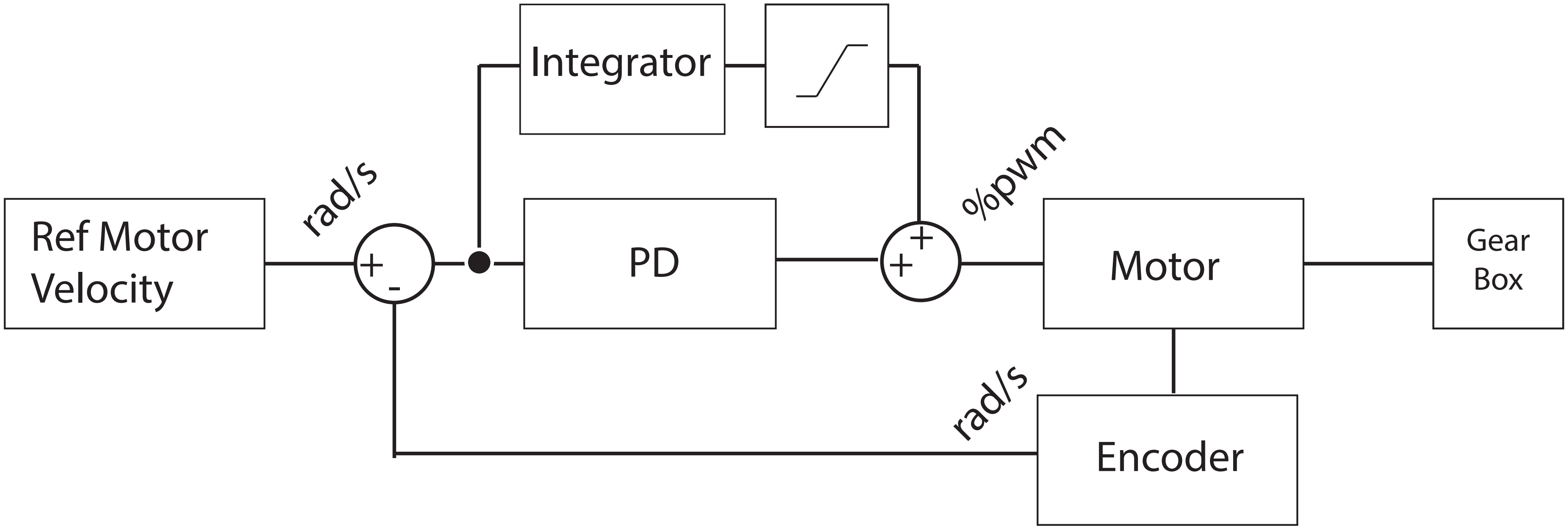
Feed Forward



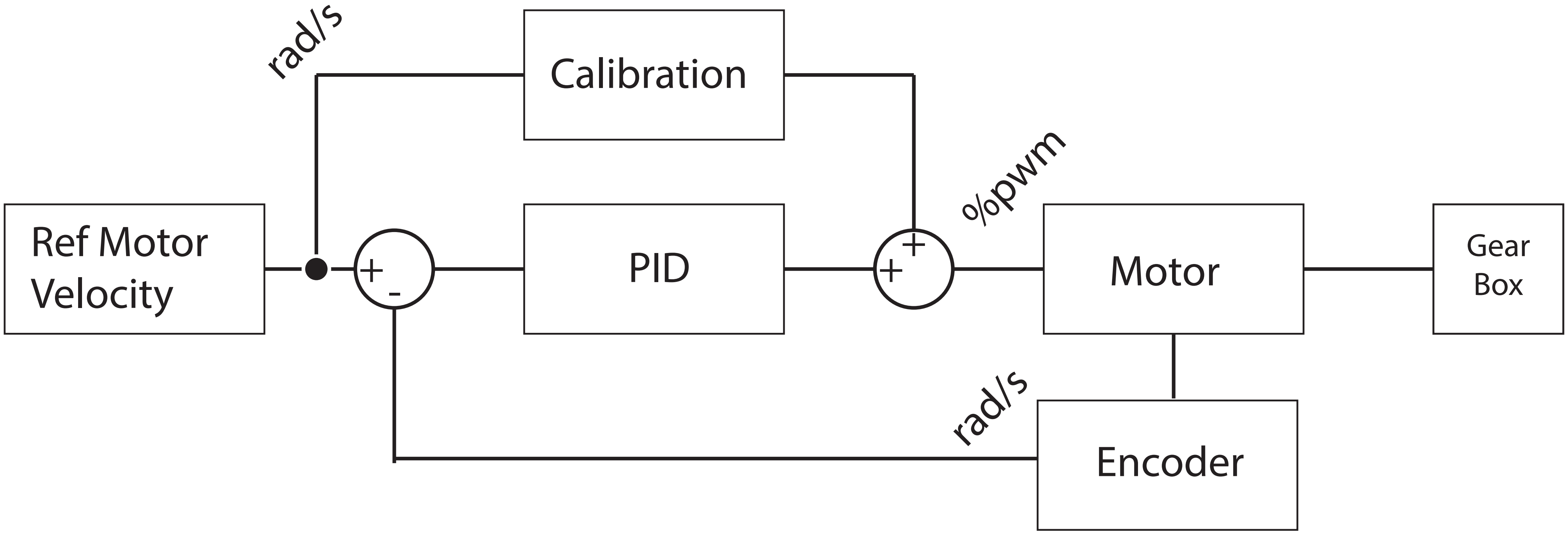
Basic Wheel Speed PID



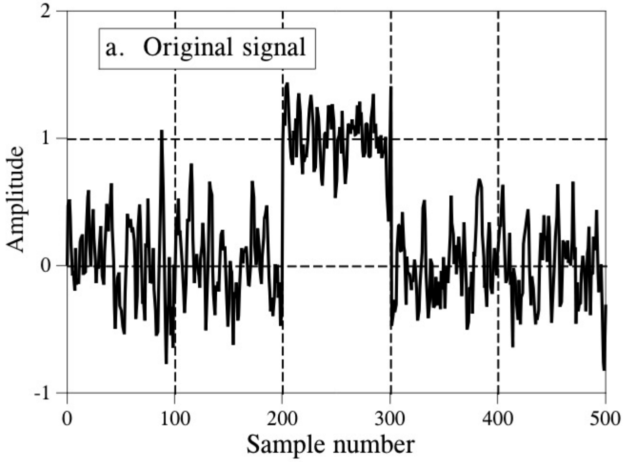
Advanced Wheel Speed PID

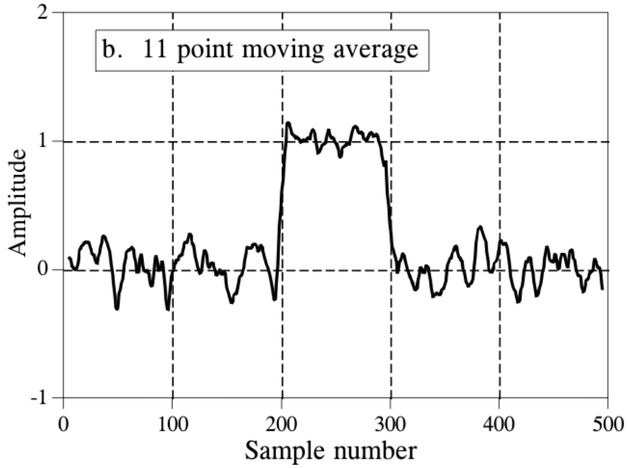


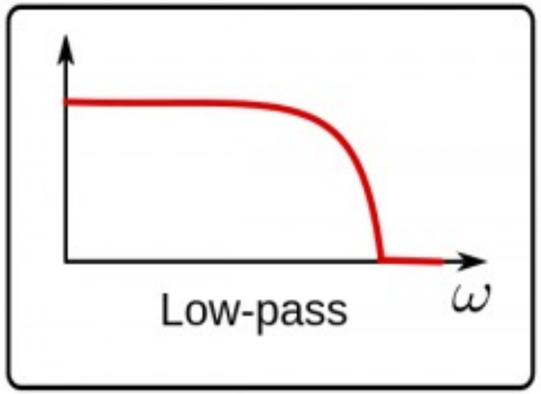
Feed Forward w/ PID

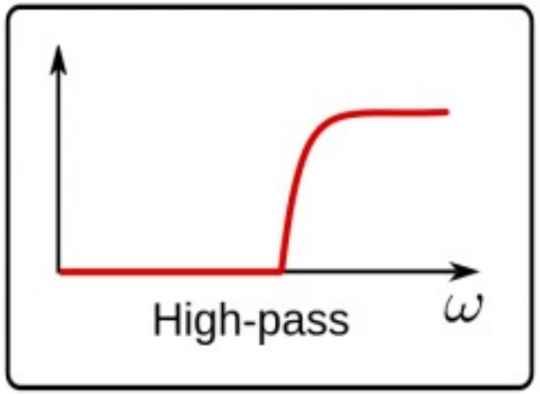


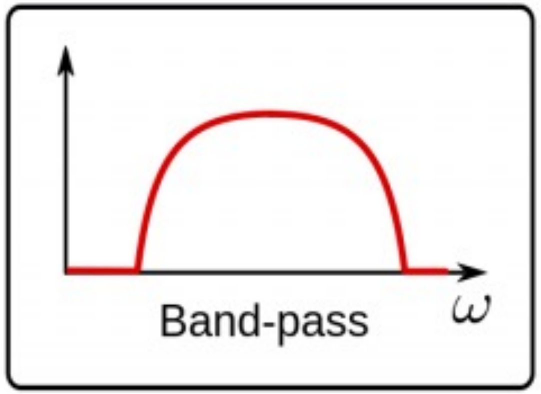
Digital Filters

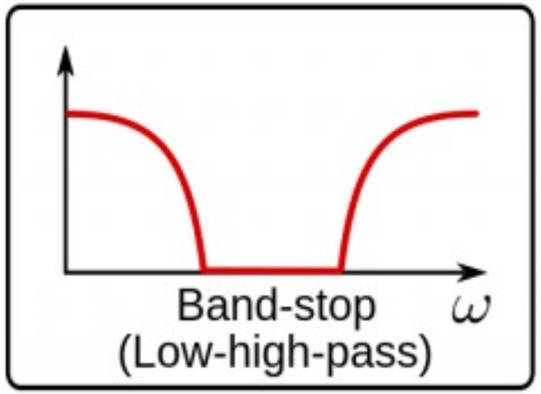












Implementing Digital Filters

- Moving Average 2 samples: $y[n] = \frac{1}{2}(x[n] + x[n-1])$
- Moving Average: $y[n] = \frac{1}{N}\sum_{k=0}^{N} x[n-k]$
- These weight each sample equally
- The output is a function of a finite number of input samples
- Referred to as Finite Impulse Response filters

Infinite Impulse Response Filters

- Instead of keeping track of a finite number of previous samples we can use recursive definition:

$$y[n] = \alpha x[n] + (1 - \alpha) y[n-1]$$

- Since this keeps a history of all previous samples, it is an infinite impulse response filter

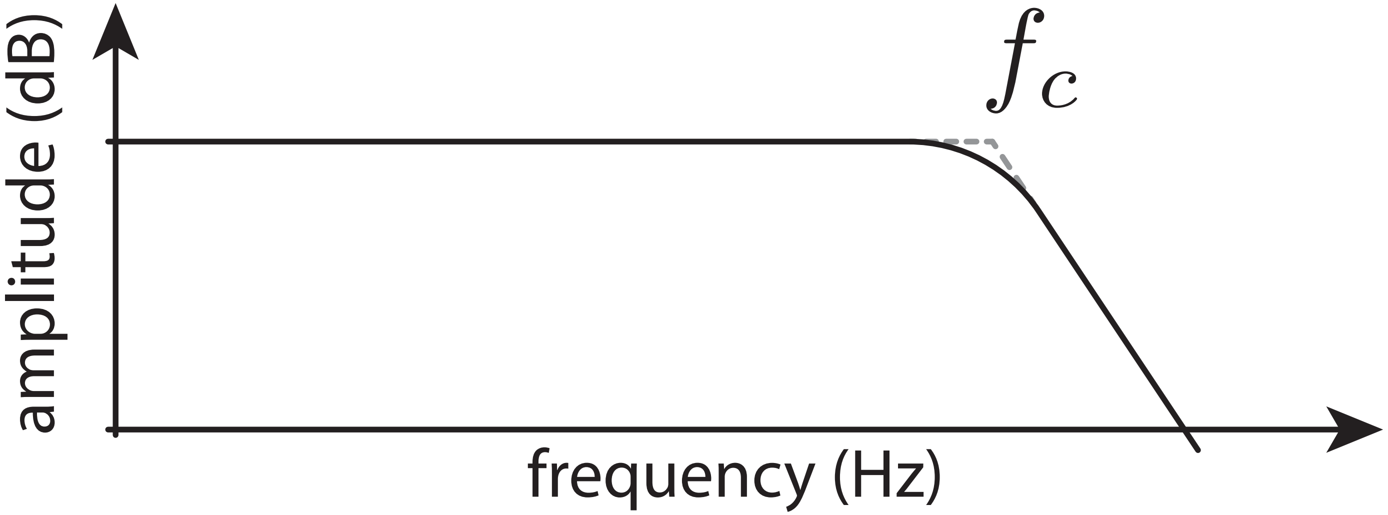
- If $\alpha = \dfrac{\Delta t}{\tau + \Delta t}$ we get a first order LPF with time constant τ

IIR First Order Low Pass Filter

$$y[n] = \underbrace{\alpha x[n]}_{\text{Contribution from new input}} + \underbrace{(1-\alpha)y[n-1]}_{\text{Exponentially decaying contribution from previous inputs}}$$

$$\alpha = \frac{\Delta t}{\tau + \Delta t}$$

$$\tau = \frac{1}{2\pi f_c}$$



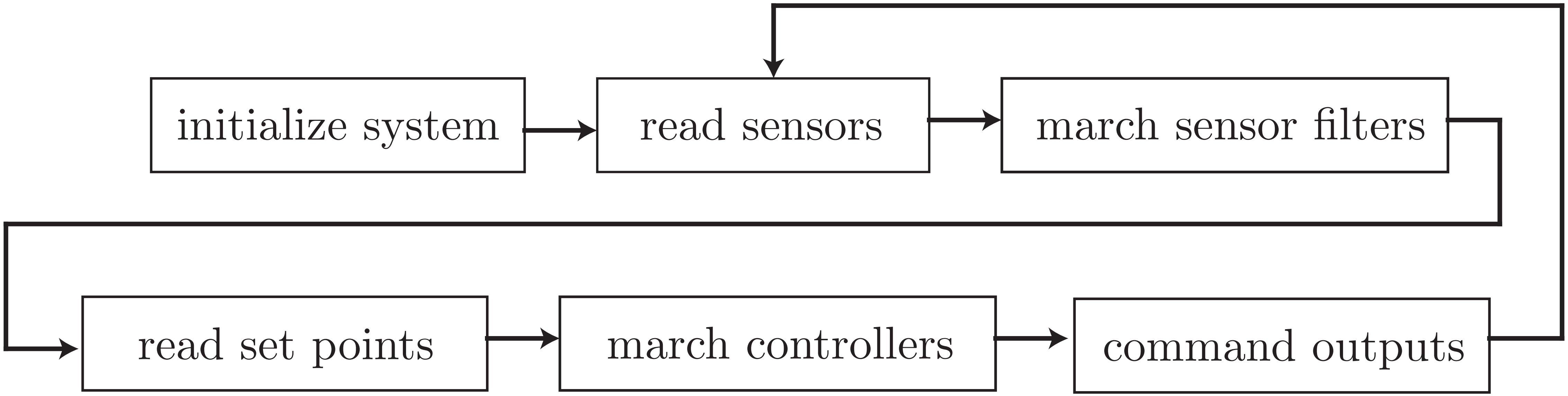
Using RCLib Filters

```
#include <rc/math.h>
#include <pico/time.h>
#define SAMPLE_RATE 100
#define TIME_CONSTANT 0.1

int main(){
    rc_filter_t low_pass = rc_filter_empty();
    const double dt = 1.0/SAMPLE_RATE;
    rc_filter_first_order_lowpass(&low_pass, dt, TIME_CONSTANT);

    while(1){
        input = get_input();
        output = rc_filter_march(&low_pass, input);
        sleep_us(1000000/SAMPLE_RATE);
    }
}
```

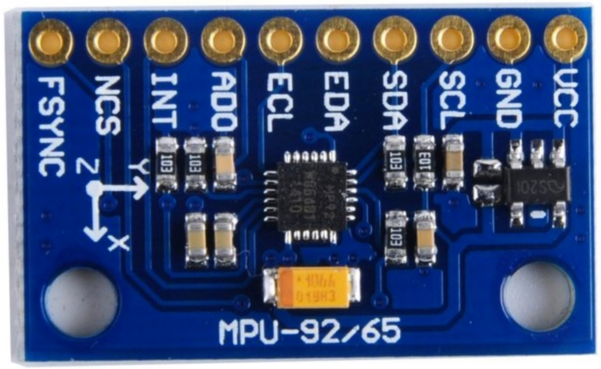
Available RCLib Filters

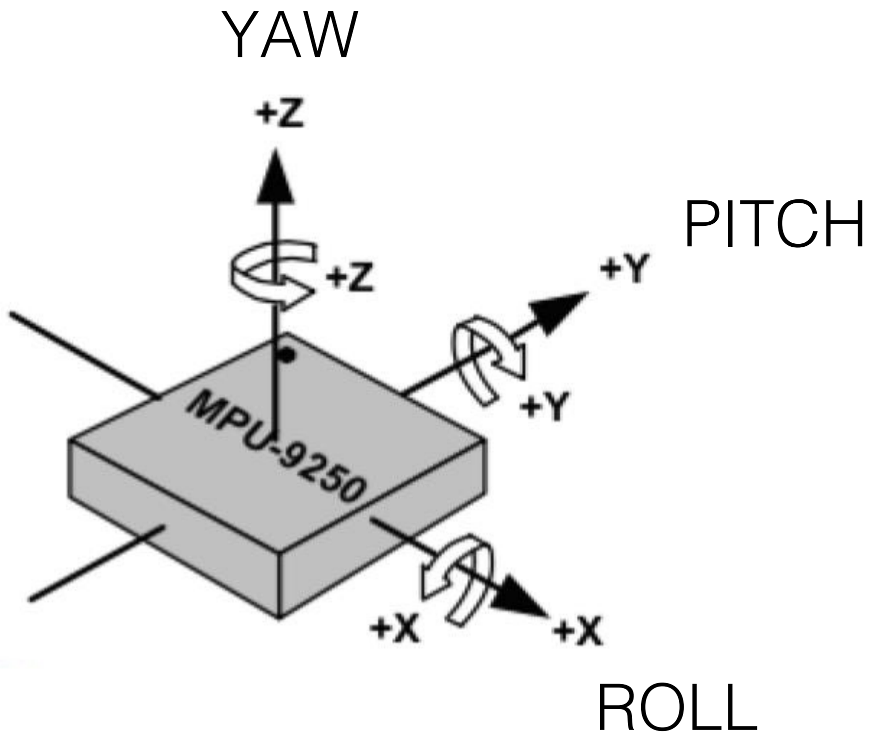
```
rc_filter_first_order_lowpass (rc_filter_t *f, double dt, double tc)

rc_filter_first_order_highpass (rc_filter_t *f, double dt, double tc)

rc_filter_butterworth_lowpass (rc_filter_t *f, int order, double dt, double wc)

rc_filter_butterworth_highpass (rc_filter_t *f, int order, double dt, double wc)

rc_filter_integrator (rc_filter_t *f, double dt)

rc_filter_pid (rc_filter_t *f, double kp, double ki, double kd, double Tf, double dt)

rc_filter_third_order_complement (rc_filter_t *lp, rc_filter_t *hp,
                                  double freq, double damp, double dt)
```

Basic Control System Programming



IMU – MPU9250





- Accelerometer, Gyroscope, Magnetometer
- Contains data-fusion processor – runs a fusion algorithm (Kalman filter) to determine orientation
- DMP interrupts at fixed rate (200Hz, 100Hz, 50Hz etc.)

Real Time Software Model

- System produces an output in response to stimulus within a specified time
- Stimuli can be:
  - Event based aperiodic (i.e. a bump switch)
  - External periodic (i.e. an external sensor which interrupts the system periodically)
  - System defined periodic (i.e. an external sensor whose current state is read)
- Timing demands of different stimuli are unique, so a simple sequential loop is not typically adequate

Handling Events

- External events typically handled by *Interrupts*
- A hardware interrupt immediately switches processor execution to an interrupt service routine (ISR)
- A soft interrupt schedules the interrupt service routine to be run when convenient
- Periodic events are typically handled with a timer which sends an interrupt when it reaches the desired time

Timed Reading

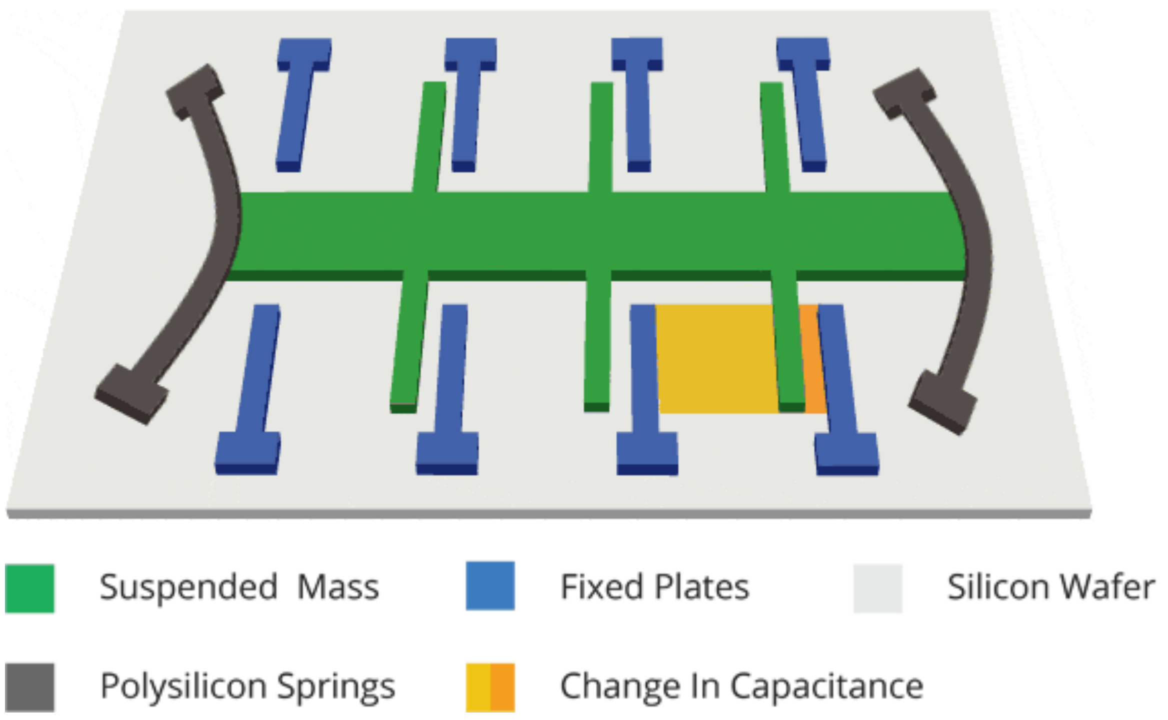
```
#include <rc/mpu.h>
#include <rc/time.h>
#define READ_HZ 100
...
rc_mpu_data_t data; //struct to hold new data
...
while (running) {
    rc_mpu_read_accel(&data)
    rc_mpu_read_gyro(&data)
    printf("%6.2f %6.2f %6.2f", data.accel[0],
data.accel[1], data.accel[2]);
    printf("%6.1f %6.1f %6.1f \n",
data.gyro[0], data.gyro[1], data.gyro[2]);
    fflush(stdout);
    rc_usleep(1E6/READ_HZ);
}
```

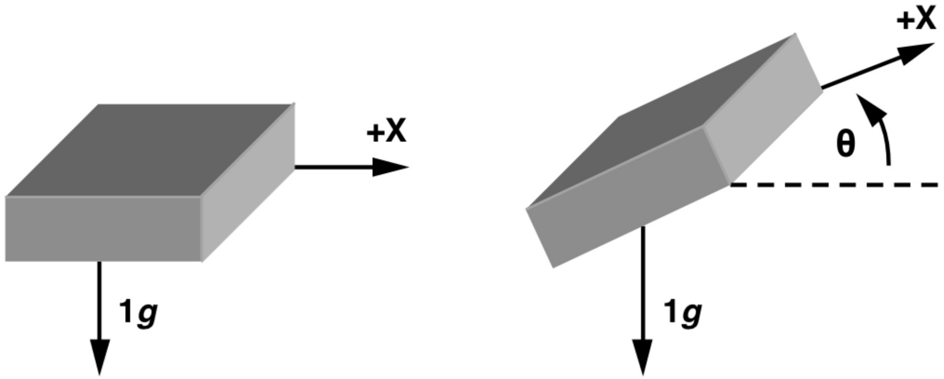
Interrupt Callback

```
#include <rc/mpu.h>
#include <rc/time.h>
...
rc_mpu_data_t data;

//Callback function
void print_data(void){
    printf("%6.2f %6.2f %6.2f", data.accel[0], data.accel[1], data.accel[2]);
    printf("%6.1f %6.1f %6.1f \n", data.gyro[0], data.gyro[1], data.gyro[2]);
    printf("%6.1f %6.1f %6.1f |", data.dmp_TaitBryan[TB_PITCH_X],data.dmp_TaitBryan[TB_ROLL_Y],
                                  data.dmp_TaitBryan[TB_YAW_Z]);
}

int main(int argc, char *argv[])
{
    ...
    rc_mpu_config_t conf = rc_mpu_default_config();
    rc_mpu_initialize_dmp(&data, conf)
    rc_mpu_set_dmp_callback(&print_data); //Set callback function
    while(running){
        rc_usleep(100000);  //Do nothing but sleep
    }
    ...
}
```

Accelerometers

- MEMS accelerometers measure the small change in capacitance between a suspended mass and fixed electrodes
- Susceptible to electronic and mechanical noise

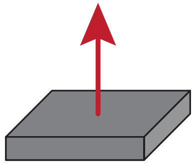




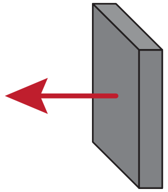
$$A_{X,OUT}\,[g] = 1\,g \times \sin(\theta)$$

What will a single axis read?

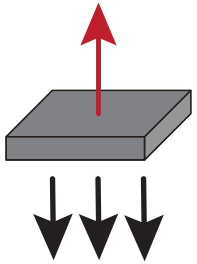
- Sitting flat on a table?



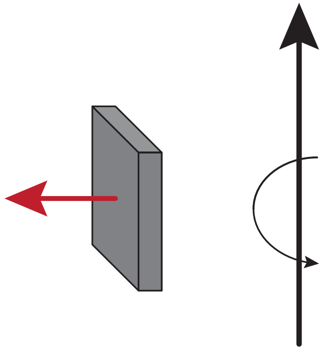
- Sitting on the side?



- Free-fall?



- Spinning off axis?

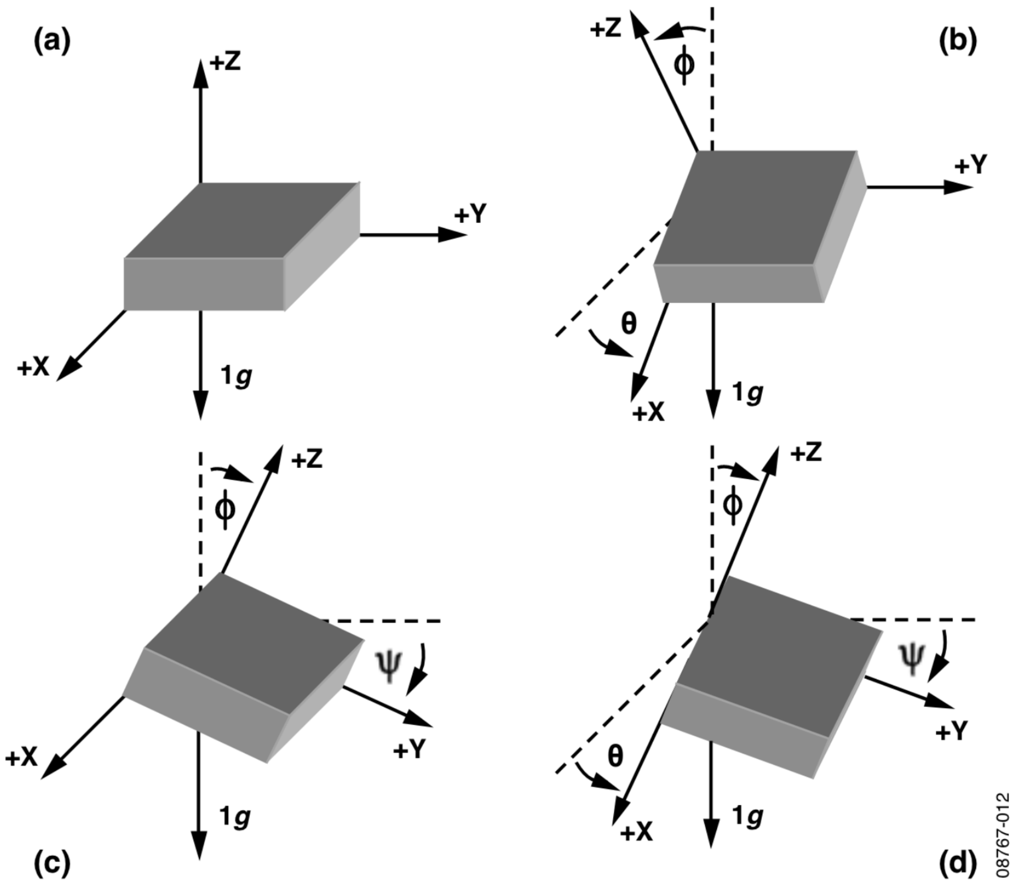


Orientation from Acceleration Readings

$$\theta = \tan^{-1}\left(\frac{A_{X,OUT}}{\sqrt{A^2{}_{Y,OUT} + A^2{}_{Z,OUT}}}\right)$$

$$\psi = \tan^{-1}\left(\frac{A_{Y,OUT}}{\sqrt{A^2{}_{X,OUT} + A^2{}_{Z,OUT}}}\right)$$

$$\phi = \tan^{-1}\left(\frac{\sqrt{A^2{}_{X,OUT} + A^2{}_{Y,OUT}}}{A_{Z,OUT}}\right)$$



Rate Gyroscope

- Vibrating mass experiences Coriolis effect
- Velocity of rotation is measured
- Susceptible to **bias** which changes with temperature

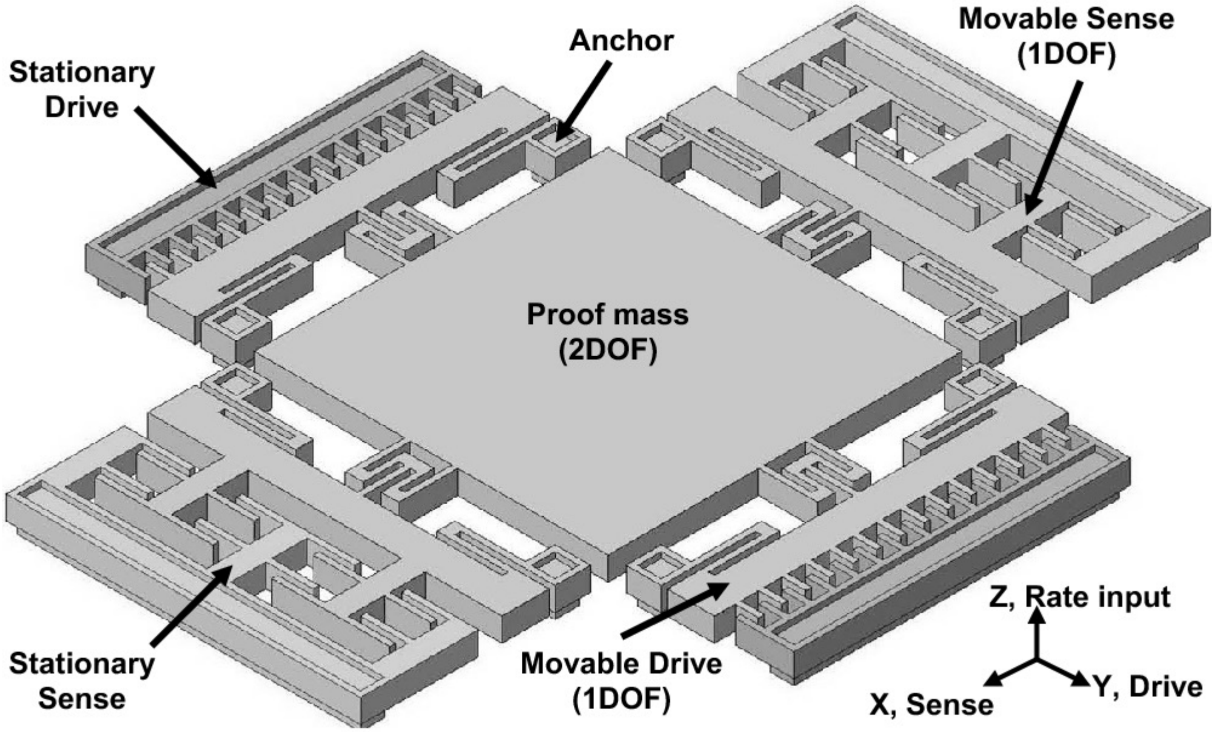
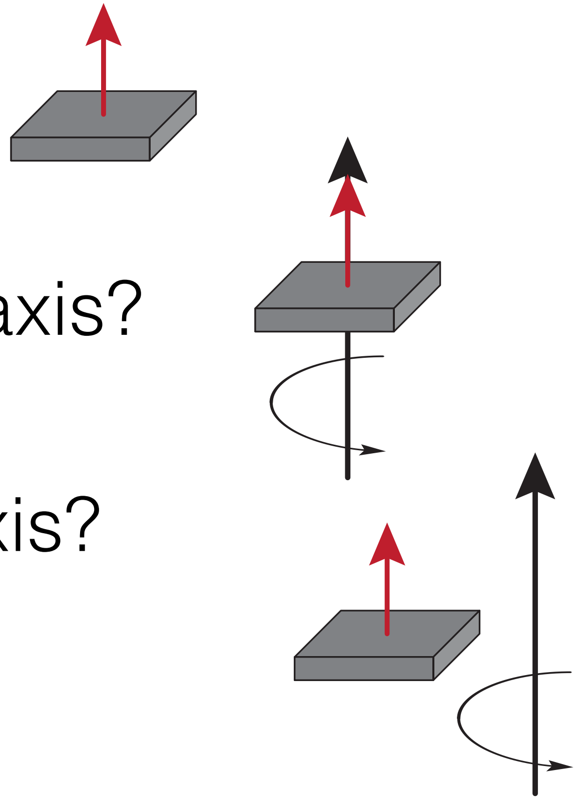


Figure 1: Perspective view of the improved symmetrical and decoupled gyroscope.

What will a single axis read?

- Sitting still?
- Spinning on axis?
- Spining off axis?

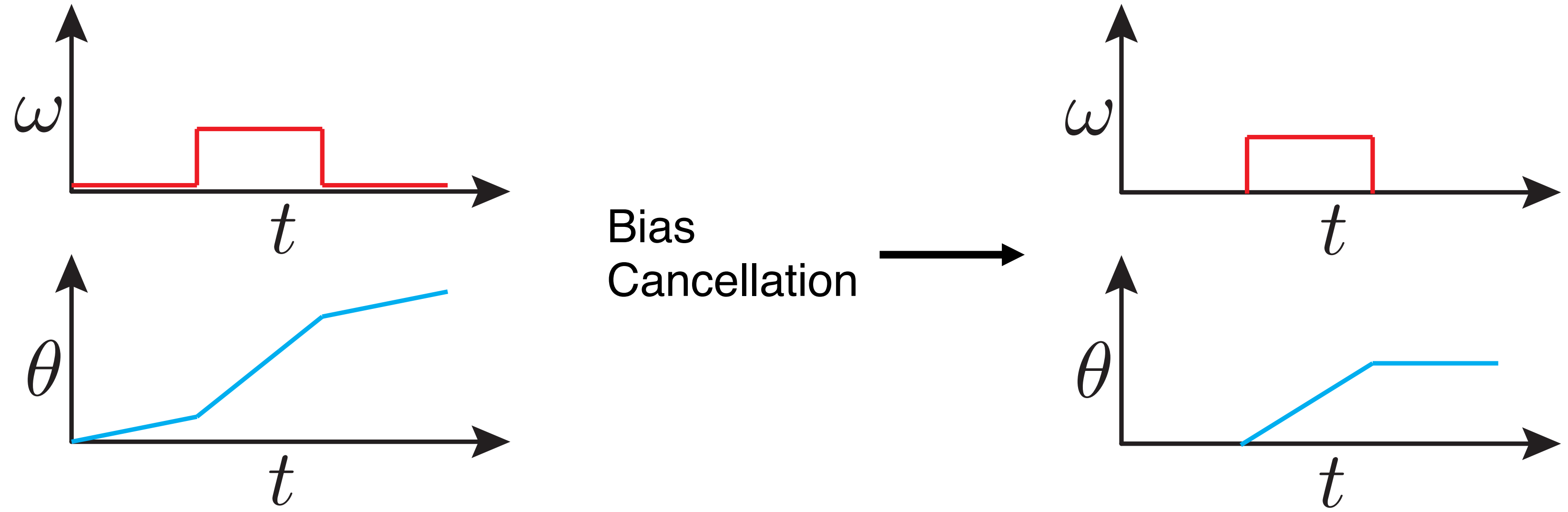


Orentation with Rate Gygo

- Integrating rate of turning gives an orientation relative to starting reference

$$\theta = \theta_0 + \int_{t=0} \omega \, dt$$

- Integrating bias causes drift
- Drift correction by subtracting bias



Orientation with Data Fusion

- Lower noise, stable orientation estimate can be found by fusing gyro and accelerometer data
- Complementary or Kalman filters can unbias the gyro with the accelerometer, while maintaining lower noise orientation estimate from the gyro
- Only works for PITCH and ROLL
- YAW must be unbiased using Magnetometer (mostly useless indoors)

Structure of rc_mpu_data_t

```
typedef struct rc_mpu_data_t{
  /* base sensor readings in real units */
  double accel[3]; ///< accelerometer (XYZ) in units of m/s^2
  double gyro[3]; ///< gyroscope (XYZ) in units of degrees/s
  double mag[3]; ///< magnetometer (XYZ) in units of uT
  double temp; ///< thermometer, in units of degrees Celsius
  /* 16 bit raw adc readings and conversion rates */
  int16_t raw_gyro[3]; ///< raw gyroscope (XYZ)from 16-bit ADC
  int16_t raw_accel[3]; ///< raw accelerometer (XYZ) from 16-bit ADC
  double accel_to_ms2; ///< conversion rate from raw accelerometer to m/s^2
  double gyro_to_degs; ///< conversion rate from raw gyroscope to degrees/s
  /* fused DMP data filtered with magnetometer */
  double fused_quat[4]; ///< fused and normalized quaternion
  double fused_TaitBryan[3]; ///< fused Tait-Bryan angles (roll pitch yaw) in radians
  double compass_heading; ///< fused heading filtered with gyro and accel data
  double compass_heading_raw; ///< unfiltered heading from magnetometer
} rc_mpu_data_t;
```