EE105 Drone Project part 1 by Eddy Zhang, Douglas Lily

Goal of the project:

The goal of this project is to balance a 1D fan arm using a PID-controlled propeller that simulates a part of the drone we are building in the next steps.

Math documentation of the model:

$$I\ddot{\theta} = F \cdot r - m \cdot g \cdot r \sin\theta$$

$\ddot{\theta}$ acceleration

F thrust

l length

m mass

g 9.8

when $\theta \approx 0^\circ$

$\Rightarrow \sin\theta \approx 1$

$$I\ddot{\theta} = k_v \cdot V_{input} \cdot l - mgl\theta$$

$$\ddot{\theta} + \frac{mgl}{I}\theta = \frac{k_v \cdot l}{I} V.$$

$$\Downarrow$$

$$\ddot{\theta} = \frac{l}{I}(kV - m \cdot g\,\theta)$$

damping

$$\begin{bmatrix}\ddot{\theta}\\ \dot{\theta}\end{bmatrix} = \underbrace{\begin{bmatrix}0 & -\frac{lmg}{I}\\ 1 & 0\end{bmatrix}}_{A}\begin{bmatrix}\dot{\theta}\\ \theta\end{bmatrix} + \underbrace{\begin{bmatrix}\frac{lk}{I}\\ 0\end{bmatrix}}_{B}$$

$$y = Cx = \theta$$

$$C = \begin{bmatrix}1\\0\end{bmatrix} \qquad D = 0$$

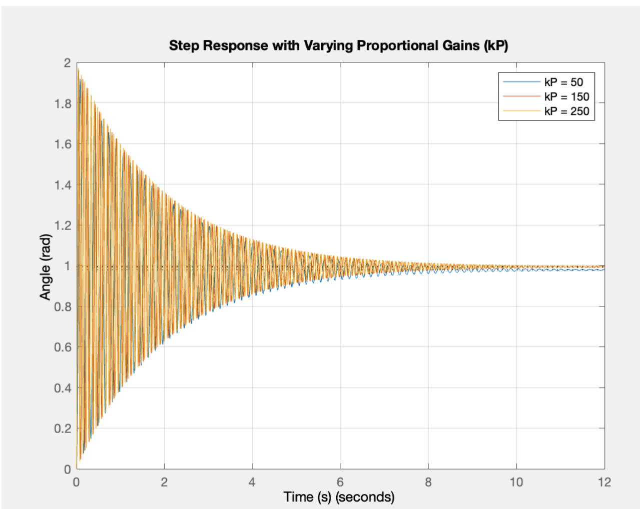
EE105 Drone Project part 1 by Eddy Zhang, Douglas Lily

Matlab code:

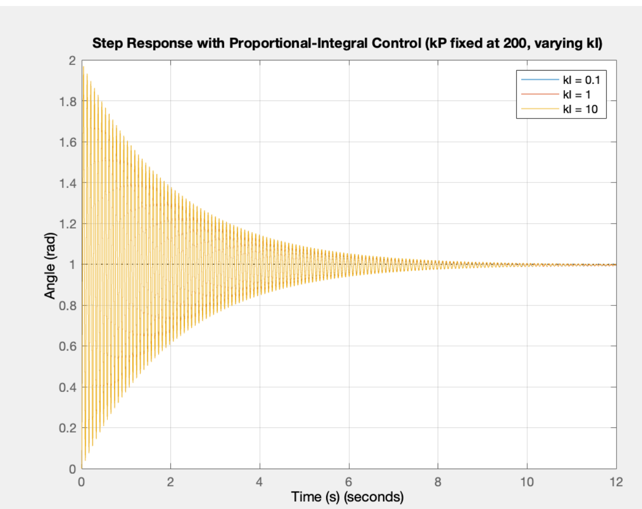
```
% Define Constants
m = 0.01; % measured mass in kilograms
r = 0.3; % arm length in meters
I = m * r^2; % moment of inertia
g = 9.81; % gravitational constant
kv = 0.087; % thrust constant
% Damping coefficient
damping = 1;
% State-Space Model
A = [0, 1; -m * g * r / I, -damping];
B = [0; kv * r / I];
C = [1 0];
D = 0;
sys = ss(A, B, C, D);
sys_tf = tf(sys);
% Proportional Control Effect figure;
hold on;
for kp = [50, 150, 250] % Test different kp values
   controller_P = pid(kp, 0, 0);
   sys_cl_P = feedback(controller_P * sys_tf, 1);
   step(sys_cl_P);
end
title('Step Response with Varying Proportional Gains (kP)');
xlabel('Time (s)'); ylabel('Angle (rad)');
legend('kP = 50', 'kP = 150', 'kP = 250'); grid on;
% Proportional-Integral Control Effect figure;
hold on;
kp = 200;
for ki = [0.1, 1, 10] % Test different ki values
   controller_PI = pid(kp, ki, 0);
   sys_cl_PI = feedback(controller_PI * sys_tf, 1);
   step(sys_cl_PI);
end
title('Step Response with Proportional-Integral Control (kP fixed at 200,
varying kI)');
xlabel('Time (s)'); ylabel('Angle (rad)');
legend('kI = 0.1', 'kI = 1', 'kI = 10'); grid on;
% Proportional-Integral-Derivative Control Effect figure;
hold on;
ki = 1; % Fixed integral gain
for kd = [0.5, 1.5, 3] % Test different kd values
   controller_PID = pid(kp, ki, kd);
   sys_cl_PID = feedback(controller_PID * sys_tf, 1);
   step(sys_cl_PID);
end
title('Step Response with Proportional-Integral-Derivative Control (kP and kI
fixed, varying kD)');
```

```
xlabel('Time (s)'); ylabel('Angle (rad)');
legend('kD = 0.5', 'kD = 1.5', 'kD = 3'); grid on;
% Combined Step Response with Tuned PID
controller_tuned = pid(200, 1, 3); % Tuned PID gains
sys_cl_tuned = feedback(controller_tuned * sys_tf, 1); figure;
step(sys_cl_tuned);
title('Step Response with Tuned PID Control (kP = 200, kI = 1, kD = 3)');
xlabel('Time (s)'); ylabel('Angle (rad)'); grid on;
```

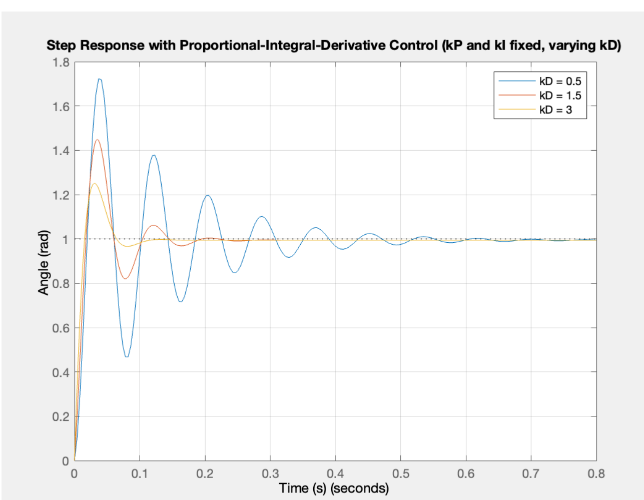
Plots:



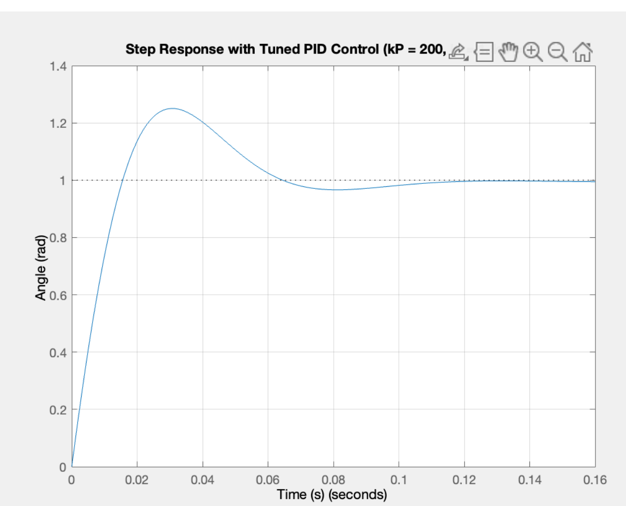
Step Response with only P control, varying Kp



Step Response with PI control, varying Ki



Step Response with PID control, varying Kd



Step Response with tuned PID control

EE105 Drone Project part 1 by Eddy Zhang, Douglas Lily

ESP32 Code:

```
#include "freertos/FreeRTOS.h"
#include "driver/ledc.h"
#include "driver/i2c_master.h"
#include "math.h"

#define SCL_IO_PIN 5
#define SDA_IO_PIN 4
#define ON_SWITCH_PIN 9 // Enable motor when low

#define REG_POWER_MGMT_1 0x6B

// PID constants (these need to be tuned)
#define KP 200
#define KI 1
#define KD 0.5

// PWM maximum duty for 13-bit resolution
#define PWM_MAX_DUTY 8191

void init_single_pwm(int gpio_pin_number)
{
   // Set up the PWM controller for the motor
   ledc_timer_config_t timer_config = {
       .speed_mode = LEDC_LOW_SPEED_MODE,
       .duty_resolution = LEDC_TIMER_13_BIT,
       .timer_num = LEDC_TIMER_0,
       .freq_hz = 200, // Hopefully something that isn't annoying to hear
       .clk_cfg = LEDC_AUTO_CLK,
   };
   ESP_ERROR_CHECK(ledc_timer_config(&timer_config));

   ledc_channel_config_t pwm_config = {
       .speed_mode = LEDC_LOW_SPEED_MODE,
       .channel = LEDC_CHANNEL_0,
       .timer_sel = LEDC_TIMER_0,
```

```
        .intr_type = LEDC_INTR_DISABLE,
        .gpio_num = gpio_pin_number,
        .duty = 0,
        .hpoint = 0
    };
    ESP_ERROR_CHECK(ledc_channel_config(&pwm_config));
}

/* Duty cycle is out of 13 bits (0-8191)*/
void pwm_set_duty(int duty)
{
    ESP_ERROR_CHECK(ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, duty));
    ESP_ERROR_CHECK(ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0));
}

int clamp(int value, int min, int max){
    if(value < min){
        return min;
    }else if(value > max){
        return max;
    }
    return value;
}

void app_main() {

    vTaskDelay(100); // wait for the IMU to wake up

    // Configure the I2C bus
    i2c_master_bus_config_t i2c_bus_config = {
        .clk_source = I2C_CLK_SRC_DEFAULT,
        .i2c_port = 0,
        .scl_io_num = SCL_IO_PIN,
        .sda_io_num = SDA_IO_PIN,
        .glitch_ignore_cnt = 7,
    };
    i2c_master_bus_handle_t bus_handle;
    ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_bus_config, &bus_handle));
```

EE105 Drone Project part 1 by Eddy Zhang, Douglas Lily

```
    i2c_device_config_t mpu_i2c_config = {
        .scl_speed_hz = 400000,
        .device_address = 0x68
    };
    i2c_master_dev_handle_t mpu_handle;
    ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &mpu_i2c_config,
&mpu_handle));

    // Set POWER_MGMT_1 register to all 0 to enable sampling
    uint8_t tx[2] = {REG_POWER_MGMT_1, 0};
    i2c_master_transmit(mpu_handle, tx, 2, -1);

    tx[0] = 0x1a; //REG_CONFIG;
    tx[1] = 0x06; // Set low-pass filter to 5Hz bandwidth
    i2c_master_transmit(mpu_handle, tx, 2, -1);

    // Read from the WHOAMI register, expect 0x68 (decimal 104)
    //uint8_t reg = 0x6B;
    //uint8_t whoami = 0xFF;
    //ESP_ERROR_CHECK(i2c_master_transmit_receive(mpu_handle, &reg, 1, &whoami,
1, -1));
    //printf("whoami (register 0x6B): 0x%x\n", whoami);

    uint8_t reg = 0x3B;
    uint8_t rx[6]; // Accelerometer data

    init_single_pwm(9);

    int iteration = 0;
    float cum_error = 0;
    float last_error = 0;

    // PID variables
    float dt = 0.01; // Time interval for each control cycle (10ms)


    while(1) {
```

```
        // Read the raw accelerometer values
        reg = 0x3b;
        ESP_ERROR_CHECK(i2c_master_transmit_receive(mpu_handle, &reg, 1, rx, 6,
-1));

        int16_t acc_x = (rx[0] << 8) + rx[1];
        int16_t acc_y = (rx[2] << 8) + rx[3];
        int16_t acc_z = (rx[4] << 8) + rx[5];

        // To calculate the angle, we just need to look at Y and Z (depending on
mounting)
        // Doesn't matter what the full-scale value is as long as it's the same!
        float angle = atan2f(-acc_y, acc_z) * 57.296f; // atan2 result is
radiants, convert to degrees
        float target = 0.0f; // Degrees (straight out, parallel to the ground)

        // Ok, this is where the magic happens.  Figure out what power to set
the motor to!
        // Calculate PID control
        float error = target - angle;
        cum_error += error * dt;
        float rate_of_change = (error - last_error) / dt;

        // PID formula
        float control_signal = KP * error + KI * cum_error + KD *
rate_of_change;

        // Convert control signal to PWM duty cycle
        int power = (int)(control_signal * PWM_MAX_DUTY / 1000.0);

        // So that we don't have negative PWM
        power = clamp(power, 0, PWM_MAX_DUTY); // Clamp power within PWM range

        pwm_set_duty(power);

        // Print out data, but only once in a while
        if(iteration % 50 == 0){
```

```
            printf("X: %d, Y: %d, Z: %d   angle: %f, power: %d\n", acc_x, acc_y,
acc_z, angle, power);
        }
        iteration++;

        last_error = error; // Update error for the next cycle

        vTaskDelay(1); // Sleep about 10ms, feel free to change this
    }
```

Reflection:

In our model, when applying only proportional or only proportional & integral control, Matlab predicts long-term (close to 10 seconds) and fast oscillation, indicating that our system has too much Proportional Gain, which is true since we set the Kp to 200; however, the reason for doing so is because in the real-life testing with our drone model we needed very fast response since the free falling of the arm can happen within 0.1 second and the overshoot can also happen rapidly. We wanted to make sure our system responded to the environmental changes as fast as possible, so we prioritized large proportional gain. However, we did find a way to smooth out the crazy oscillation by applying the derivative gain. After careful tuning of all three variables using Matlab PIDTuner, we constructed a model with a short response time (<0.1s) with only a 20% overshoot. During our final tests, the model performed well in real life.

However, I did discover that even without derivative control, the model would not oscillate non-stop for 10 seconds before settling down like predicted in the initial models, my guess would be I underestimated the damping value. We also plan to deal with the noise from sensor reading, which can also be a cause, by applying Kalman filtering.