Inverted Pendulum Control using a BeagleBone Blue

Erik Sandström

December 17th 2017

Introduction

The objective of this project is design and implement a complete control system for an inverted pendulum. Control will be applied to the angle of the robotic body and to the position relative to the ground. Two nested control loops will be used in order to achieve this. MatLab is used as a tool to design the concept and C is used to implement the design on the eduMIP platform provided by the UCSD Coordinated Robotics Lab.

Inner loop design

The first task is to design an inner loop controller to stabilize the body angle denoted θ around a given setpoint θ_{ref}. This process begins with finding the system transfer function G1 from the motor input duty cycle u to the body angle θ. The transfer function is given by,

$$G1 = \frac{\theta}{u} = \frac{-882.7s}{s^3 + 44.15s^2 - 192.8s - 2299} \tag{1}$$

The Bode plot of G1 is given by figure 1.

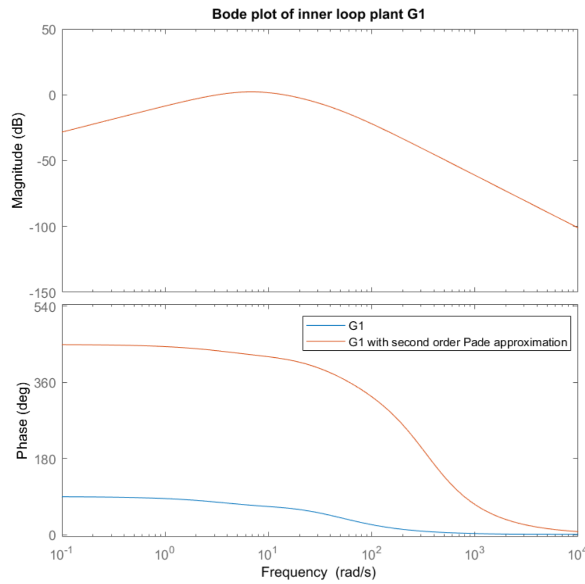

Figure 1: Bode plot of the inner loop plant with and without second order Pade approximation for delay coming from zero order hold of the discrete inner loop controller.

An analysis of the pole and zero placement of G1 reveals that it has poles located at $-47.2061,\ 8.6696,\ -5.6165$ and one zero at the origin. To get this system stable, a lag compensator that cancels the zero at the origin is necessary to bring the right hand pole to the left half plane. Cancelling a zero at the origin can be tricky business, but since it is known at this stage, that an outer loop will be implemented, we can assure that stability is kept, even though we might miss by a small amount when cancelling. The lag controller $D1_{lag}$ is of the form

$$D1_{lag} = \frac{s + 5.6165}{s}, \tag{2}$$

where a cancellation of the left half plane pole of the plant G1 is a safe cancellation since it contributes in the event of a missed cancellation to an exponentially decaying term in the expression of the output in the time domain. When the controller $D1_{lag}$ is tuned with some proportional gain, stability can be achieved, but the system is not robust enough as can be seen by the low phase margin and oscillatory behavior of figure 2.

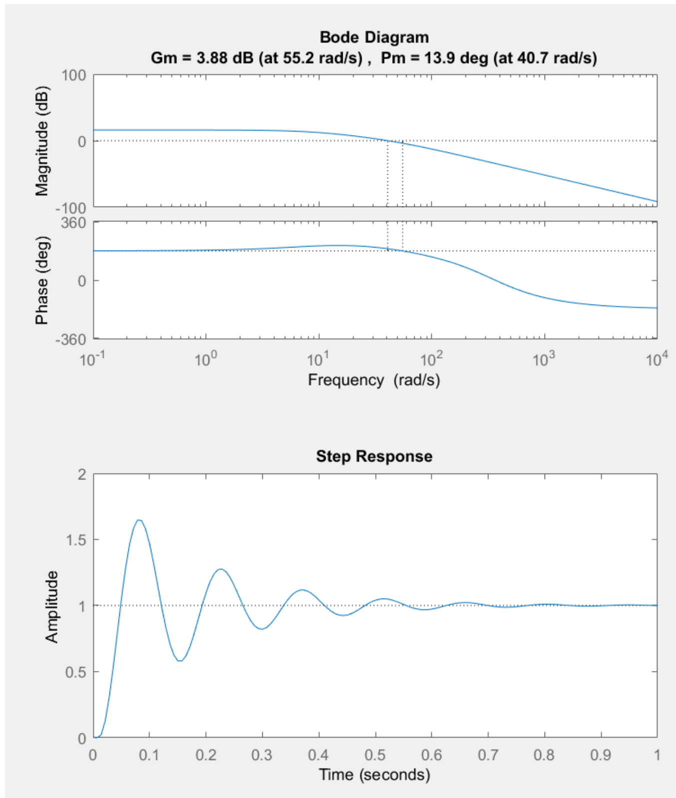

Figure 2: Bode plot of the open loop system $D1_{lag}G1$ tuned with some proportional gain to achieve stability.

To get a more stable system, a lead compensator $D1_{lead}$ is added. The lead compensator will be centered to have its maximum phase gain at the desired cross-over frequency ω_{c1} of the open loop system $L1 = D1G1$ to increase robustness. A good starting point for the the cross-over frequency ω_{c1} is to set it to a tenth of the Nyquist frequency where the Nyquist frequency is half of the sampling frequency of the IMU, which is 100 Hz. This yields a cross-over frequency

$$\omega_{c1} = 10\pi \ \ rad/sec. \tag{3}$$

One can also evaluate the chosen cross-over frequency according to the tuning guideline ($\omega_n = 1.8/t_r$) that specifies the cross-over frequency when the rise time to a step input is given. A rise time of around 0.05 seconds yields the specified cross-over frequency. This should reflect the natural time period of the pendulum when it is not inverted, but a simply experiment shows that the real world period time is closer to 0.5 seconds. This means that the inputs u will likely vary around this critical frequency during operation and it is thus important that stability is preserved for these frequencies. Using a cross-over frequency higher than the natural frequency means that we are on the safe side of operation since the closed loop system will not reject the important inputs frequencies.

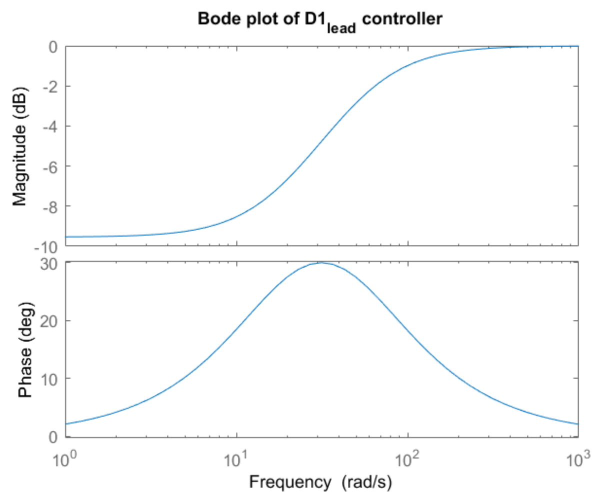

Figure 3: Bode plot of the controller $D1_{lead}$ centered at ω_{c1}.

$$D1_{lead} = \frac{s + 18.14}{s + 54.41} \tag{4}$$

with a Bode plot given by figure 3. Using $D1_{lead}$ together with $D1_{lag}$ in a cascade formation together with an appropriate gain constant yields the complete inner loop controller determined by

$$D1 = K_{D1} D1_{lead} D1_{lag} = \frac{-2.94s^2 - 69.84s - 299.5}{s^2 + 54.41} \tag{5}$$

The root locus of the system displaying the placement of the closed loop system poles is plotted together with the step response of the closed loop system, open loop $L1$ Bode plot and inner loop controllers in figure 4

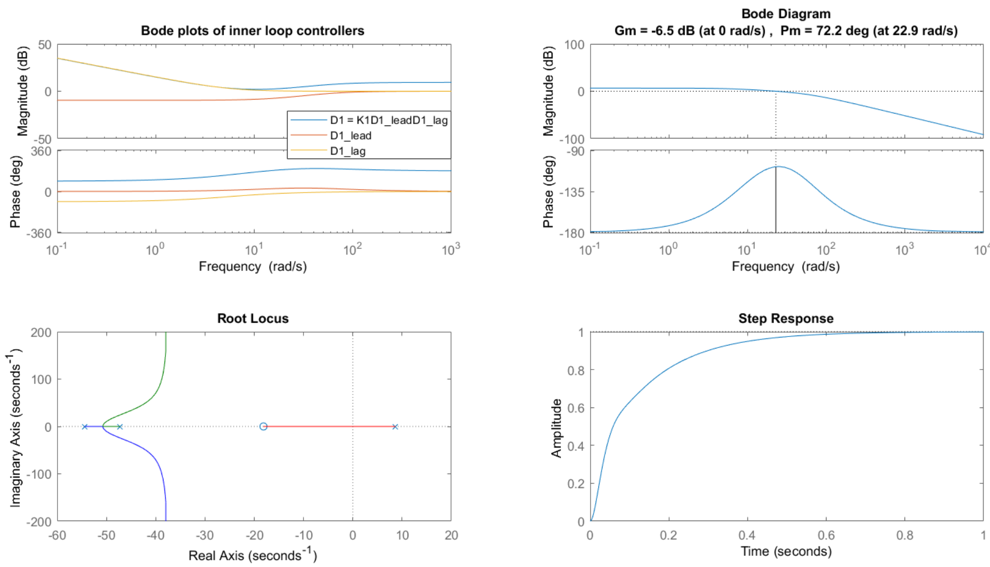

Figure 4: The root locus of the system displaying the placement of the closed loop system poles is plotted together with the step response of the closed loop system, open loop $L1$ Bode plot and inner loop controllers.

The discrete equivalent controller to $D1(s)$ is found by transforming the controller with Tustin's approximation with prewarping around the crossover frequency. The result is

$$D1(z) = \frac{u}{\theta_e} = \frac{-2.589z^2 + 4.602z - 2.037}{z^2 - 1.569z + 0.5695}, \tag{6}$$

where $\theta_e = \theta_{ref} - \theta$, the body angle error and u is the duty cycle which is the input to the PWM signal going to the motors.

As a final check of the inner loop controller, a second order Pade approximation of a delay can be applied to the plant G1. This is tested due to the fact that there will inherently be a delay of the form e^{-ds} from the DAC (Digital to Analog Converter) when the discrete controller delivers it output to the continuous plant G1. The sampling period time of the IMU is h = 0.01 s and the delay from the ZOH is $d = h/2$. The plant using a second order approxiamation is given by figure 1 and a corresponding analysis as given in figure 4 can be found in figure 5. As can be seen, the phase margin has droppen by around 7 degrees and is therefore still far from instability.

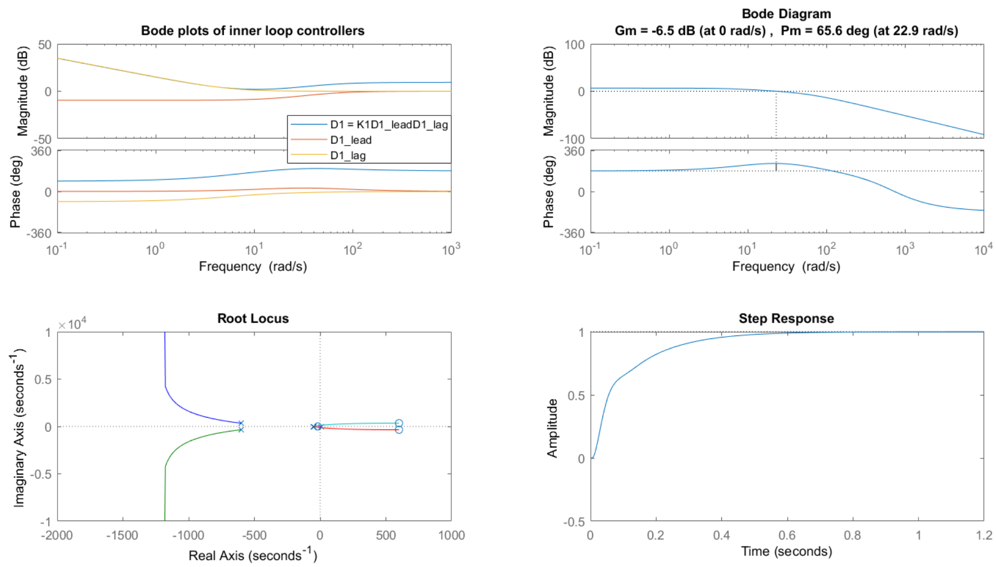

Figure 5: Bode plot of the controller $D1_{lead}$ centered at ω_{c1}.

Outer loop design

First assuming that the outer loop has much slower dynamics compared to the inner loop, the outer loop can be designed on the assumption that the closed inner loop will be very close to unity for all frequencies of interest for the outer loop. Therefore, a crossover frequency ten times smaller than the crossover frequency of the inner loop is used and the design of the outer loop controller is first made with the inner closed loop assumed to be one. From Newtons equations of motion the relationship between the body angle θ and the position ϕ can be calculated in the Laplace domain given by a transfer function

$$G2 = \frac{\phi}{\theta} = \frac{-1.476s^2 + 2.622 \times 10^{-15} + 128.9}{s^2}. \tag{7}$$

The Bode plot of the outer loop system is given by figure 6.

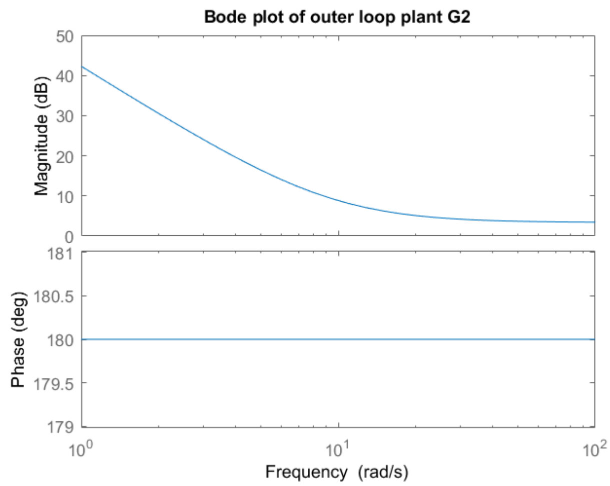

Figure 6: Bode plot of the outer loop plant G2.

Looking at the phase response which is -180 degrees over all frequencies, it is obvious that a lead compensator is needed. Designing a lead compensator $D2_{lead}$ centered at $\omega_{c2} = \omega_{c1}/10$ yields

$$D2_{lead} = \frac{s + 0.5736}{s + 17.21} \tag{8}$$

A lag compensator $D2_{lag}$ was also added to bump up the gain at lower frequencies. The robot works absolutely fine without this compensation, but the robot behaves more smoothly with it than without.

$$D2_{lag} = \frac{s + 0.06283}{s + 0.01571} \tag{9}$$

With an appropriate gain factor the complete outer loop controller then becomes

$$D2 = K2D2_{lead}D2_{lag} = \frac{0.1738s^2 + 0.1106s + 0.006263}{s^2 + 17.22s + 0.2703} \tag{10}$$

The corresponding plot to figure 4 but for the outer loop is given by 7

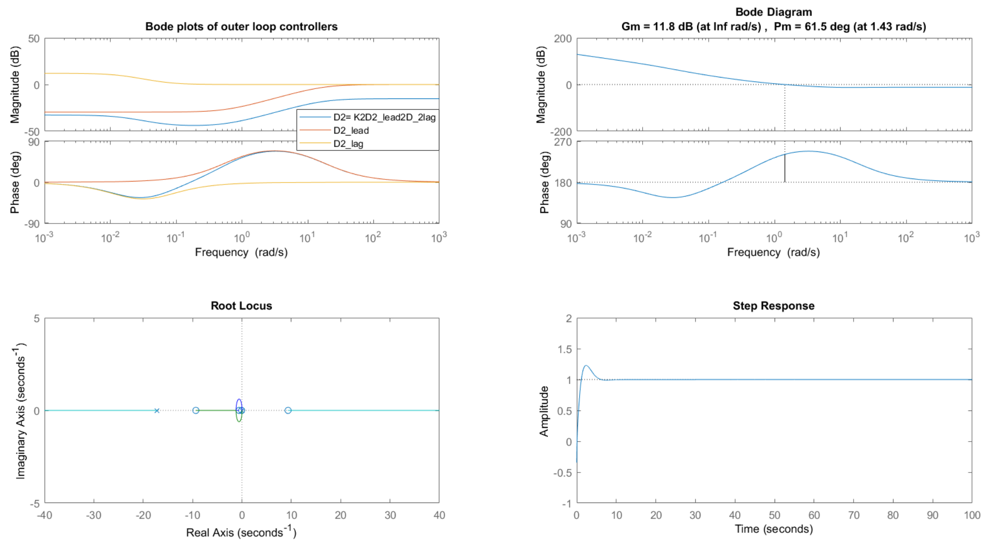

Figure 7: Bode plot of outer loop controllers, Bode plot of open loop $L2 = D2G2$ system, root locus and step response of the closed loop system of $L2$.

Adding a Pade approximated delay of e^{-ds} of fourth order with $d = h/2$ with $h = 0.05\ \ s$ yields the following result given by figure 8

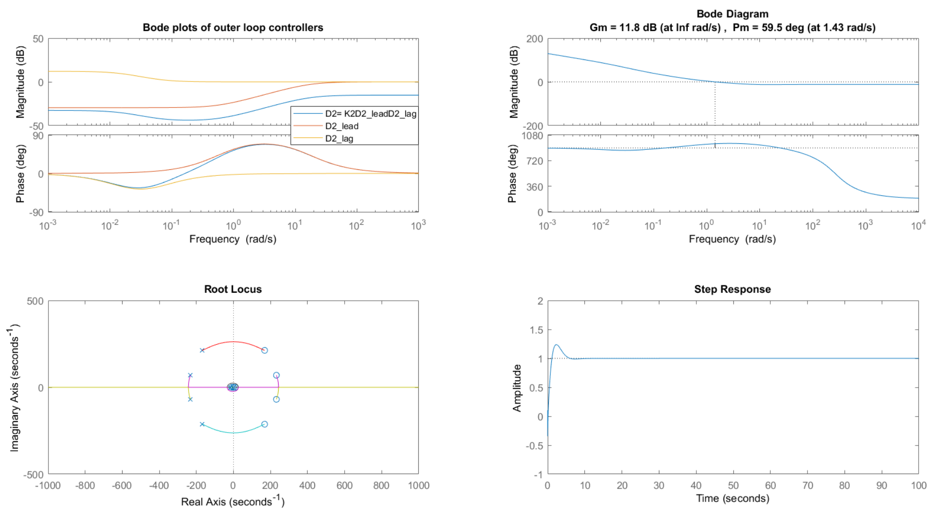

Figure 8: Bode plot of outer loop controllers, Bode plot of open loop $L2 = D2G2$ system, root locus and step response of the closed loop system of $L2$. The only difference to figure 7 is the addition of a the delay approximation to G2.

The delay occurs due to the zero order hold of the DAC in D2. A phase loss of only 2 degrees can be seen in figure 8 and thus the system is stable.

The final step is to include the closed loop transfer function $T1 = \frac{L1}{1+L1}$, where $L1 = D1G1$ to the outer loop system. The results, using a second order Pade approximation in G1 and a fourth order Pade approximation in G2 yields the final result as seen in figure 9

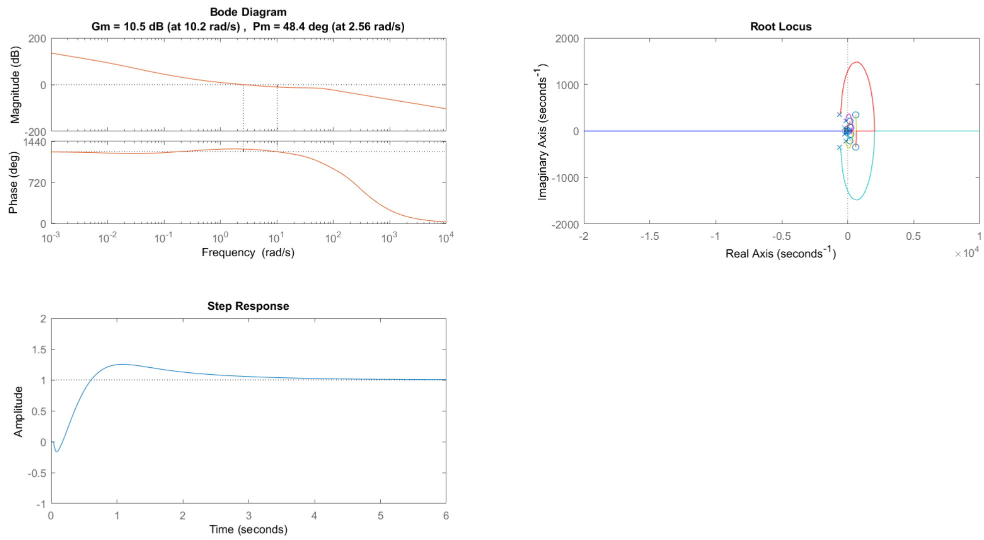

Figure 9: System analysis with the characteristics of the inner loop included. Root locus, step response and Bode plot all show stable behaviors.

Note the nonminimum phase behavior as can be seen in the step response of figure 9. This is expected and is a natural behavior of a MiP.

Finally the discrete version of D2 is found by utilizing Tustin's approximation around the crossover frequency ω_{c2} and the result is

$$D2(z) = \frac{0.1233z^2 - 0.2428z + 0.1195}{z^2 - 0.397 + 0.3972} \tag{11}$$

Code

```

/*******************************************************************************
* FINAL_PROJECT.c
*
* This program retrieves the angle estimates from the IMU and passes the values
* through a complementary filter to for a more accurate description of the angle
* and then uses a body angle reference set point to balance around. Two controllers
* are arranged in a successive loop closure to make the robot stay in the same
```

```c
* position. The sampling rate of the IMU is 100 Hz and the inner loop runs at 100 Hz
* while the outer loop runs at 20 Hz. The program prints to the screen at
* 10 Hz.
******************************************************************************/

// usefulincludes is a collection of common system includes
#include <rc_usefulincludes.h>
// main roboticscape API header
#include <roboticscape.h>

//Define inner and outer loop timesteps
#define INNER_TIMESTEP 0.01
#define OUTER_TIMESTEP 0.05

//Define D1 gain tuning factor, overload and reset factor
#define D1_GAIN 1.02 //Tuned for good performance
#define D1_OVERLOAD_TIMEOUT 50
#define D1_RESET_TIMEOUT 200

//Define D2 gain tuning factor
#define D2_GAIN 1.0 //No tuning needed

#define STEERING_GAIN 0.25
#define ANGLE_OFFSET 0.31
#define START_ANGLE_RANGE 0.2
#define MAX_LEAN                1.2

//Motor and encoder controls
#define L_MOTOR_POLARITY 1
#define R_MOTOR_POLARITY -1

#define R_MOTOR_CHANNEL 2
#define L_MOTOR_CHANNEL 3

#define R_ENCODER_CHANNEL 2
#define L_ENCODER_CHANNEL 3
#define L_ENCODER_POLARITY 1
#define R_ENCODER_POLARITY -1

#define GEARBOX 4*15*35.555


//Function declarations
void on_pause_pressed();
void on_pause_released();
void* print_thread_func();
void* outer_loop_thread_func();
void inner_loop();
void reset_controller();


//Define an armstate enum, used to turn various loops and checks on/off.
typedef enum arm_state_t{
```

```
        DISARMED,
        ARMED
}arm_state_t;

arm_state_t armstate=DISARMED;
//Initialize inner loop body angle variables.
 double theta_a_raw = 0;
 double theta_g_raw = 0;
 double theta_g_raw_last = 0;
 double theta_a = 0;
 double theta_f = 0;
 double theta_g = 0;

//Initialize body angle error variables
double theta_e = 0;
double theta_e_last = 0;
double theta_e_last_last = 0;

//Initialize body angle reference
double theta_ref = 0;
double theta_ref_last = 0;
double theta_ref_last_last = 0;

//Initialize inner loop overload and reset timer
int inner_overload_timer = 0;
int inner_reset_timer = 0;
//Initialize duty cycle variables
double u = 0;
double u_last = 0;
double u_last_last = 0;

//Initialize outer loop setpoint variable
double phi_ref = 0;
double phi = 0;
double phi_R = 0;
double phi_L = 0;
double phi_diff = 0;
double steerinput = 0;

//Initialize outer loop setpoint errors
double phi_e = 0;
double phi_e_last = 0;
double phi_e_last_last = 0;


//Struct to hold imu data
rc_imu_data_t imudata;

/*
    ****************************************************************************
* int main()
*
* This template main function contains these critical components
* - call to rc_initialize() at the beginning
* - initialize IMU and threads
* - main while loop that checks for EXITING condition
```

```c
* - rc_cleanup() at the end
*****************************************************************************/
int main(){
        //Always initialize cape library first
        if(rc_initialize()){
                fprintf(stderr,"ERROR: failed to initialize rc_initialize(), are you root?\n");
                return -1;
        }

        //Connect buttons to functions
        rc_set_pause_pressed_func(&on_pause_pressed);
        rc_set_pause_released_func(&on_pause_released);

        //Configure the IMU
        rc_imu_config_t conf = rc_default_imu_config();

        //Set up the imu for interrupt operation
      if(rc_initialize_imu_dmp(&imudata, conf)){ //Points at the data struct where we store the imu-values and to the configurations of the imu stored in conf
                fprintf(stderr,"rc_initialize_imu_failed\n");
                return -1;
        }
        if(rc_read_accel_data(&imudata)<0){
                printf("read accel data failed\n");
        }
        printf("|Angle|Motor input|Position|\n");

        theta_a = atan2(-imudata.accel[2],imudata.accel[1]); //Set the output of low pass filter to the current accelerometer value to account for steady state

         //Create thread for printing angle estimates
        pthread_t print_thread;
        pthread_create(&print_thread, NULL, print_thread_func, (void*) NULL);

        //Set priority of thread
        struct sched_param params_print_thread;
        params_print_thread.sched_priority = 1;
        pthread_setschedparam(print_thread, SCHED_FIFO,&params_print_thread);

        //Make thread for outer loop (setpoint control)
        pthread_t outer_loop_thread;
        pthread_create(&outer_loop_thread, NULL, outer_loop_thread_func, (void*) NULL);

        //Set priority of thread
        struct sched_param params_outer_loop_thread;
        params_outer_loop_thread.sched_priority = 10;
        pthread_setschedparam(outer_loop_thread, SCHED_FIFO,&params_outer_loop_thread);

        //Give the IMU an interrupt function
```

```
        rc_set_imu_interrupt_func(&inner_loop);

        rc_set_state(RUNNING);


        // Keep looping until state changes to EXITING
        while(rc_get_state() != EXITING){
                if(armstate==ARMED) {
                        rc_set_led(GREEN,ON);
                        rc_set_led(RED,OFF);
                }

                else if(armstate==DISARMED) {
                        rc_set_led(GREEN,OFF);
                        rc_set_led(RED,ON);
                }
              rc_usleep(1000000);
        }

        //Exit cleanly
        printf("Waiting for print thread to join \n");
        pthread_join(print_thread, NULL);
        printf("Print thread joined \n");
        printf("Waiting for setpoint thread to join \n");
        pthread_join(outer_loop_thread, NULL);
        printf("Outer loop thread joined \n");

        rc_power_off_imu();
        rc_cleanup();
        return 0;
}

//The interrupt function is called everytime the IMU has new data available (100 Hz) and filters the data that is collected by the imu
void inner_loop() {
        //Complementary filter cross-over frequency as a static variable which means that it is only instantiated once.
        static double wcfilter = 0.5;

        theta_a_raw = atan2(-imudata.accel[2],imudata.accel[1]);
        theta_g_raw_last = theta_g_raw;
    theta_g_raw = theta_g_raw + imudata.gyro[0]*PI/180*INNER_TIMESTEP;

        theta_a = -(wcfilter*INNER_TIMESTEP-1)*theta_a + wcfilter*INNER_TIMESTEP*theta_a_raw;

        theta_g = -(wcfilter*INNER_TIMESTEP-1)*theta_g + theta_g_raw  - theta_g_raw_last;

        theta_f = theta_a + theta_g+ANGLE_OFFSET;

        // INNER LOOP CONTROLLER
        theta_e_last_last = theta_e_last;
        theta_e_last= theta_e;
        theta_e = theta_ref-theta_f;
```

```

    //Evaluate inner loop difference equation
    u_last_last = u_last;
    u_last = u;
    u = D1_GAIN*(1.569*u_last -0.5695*u_last_last -2.589*theta_e +4.602*theta_e_last -2.037*theta_e_last_last);

    //Prevent windup by setting max and min values for the motor input
    if(u>=1){
          u=1;
    }
    if(u<=-1){
          u=-1;
    }

    /*************************
    Perform checks on status of system, and react accordingly
    *************************/

    //Turn off motor if exiting
    if(rc_get_state()==EXITING){
    rc_disable_motors();
    return;
    }

    /*nitiate a reset timer that checks if the robot is lying down, ready to be picked up to balance.
    This is used to prevent the motors from running when the robot is picked up */
    if(theta_f > -1.7 && theta_f < -1.5){
            inner_reset_timer++;
   }

    if(fabs(theta_f) < START_ANGLE_RANGE && armstate == DISARMED) {
             if(inner_reset_timer > D1_RESET_TIMEOUT){
                    reset_controller();
                    rc_enable_motors();
                armstate = ARMED;
                    inner_reset_timer = 0;
            }
    }

    //If the robot is disarmed, do nothing
    if(armstate == DISARMED) {
    return;
    }


    //If robot has tipped past point of no return, stop trying
    if(fabs(theta_f)>MAX_LEAN){
            printf("\r I have fallen over. Need help! \n");
            reset_controller();
            rc_disable_motors();
            armstate = DISARMED;
    }
```

```

        //Initiate check if the robot has gone stuck, to prevent motors from breaking
        if(fabs(u)>0.95){
                inner_overload_timer++;
        }
    else{
                inner_overload_timer=0;
        }
        if(inner_overload_timer > D1_OVERLOAD_TIMEOUT){
                printf("\r I have gone stuck. Rescue me! \n");
                reset_controller();
                rc_disable_motors();
                armstate = DISARMED;
                inner_overload_timer = 0;
                return;
        }


        //Proportional steering controller to keep robot pointing in approximately the right direction
        steerinput=STEERING_GAIN*phi_diff;

        rc_set_motor(L_MOTOR_CHANNEL, L_MOTOR_POLARITY*(u-steerinput));
        rc_set_motor(R_MOTOR_CHANNEL, R_MOTOR_POLARITY*(u+steerinput));

        return;
        }

void* print_thread_func(){
    while(rc_get_state() != EXITING) {
        printf("\r");
        printf("|%6.3f |%6.3f |%6.3f |", theta_f, u, phi);
        fflush(stdout);
        rc_usleep(100000);
        }
        return NULL;
}

void* outer_loop_thread_func(){
        while(rc_get_state() != EXITING){
        phi_R = (R_ENCODER_POLARITY*rc_get_encoder_pos(R_MOTOR_CHANNEL)*2*PI)/(GEARBOX);
        phi_L = (L_ENCODER_POLARITY*rc_get_encoder_pos(L_MOTOR_CHANNEL)*2*PI)/(GEARBOX);

        //Take average of left and right encoder and add body angle
        phi = (phi_R+phi_L)/2 +theta_f;

        //Difference between wheel positions for use in steering controller
        phi_diff = phi_L-phi_R;

        phi_e_last_last = phi_e_last;
        phi_e_last = phi_e;
```

```
        phi_e = phi_ref - phi;

        theta_ref_last_last = theta_ref_last;
        theta_ref_last = theta_ref;

        //Outer loop difference equation
        theta_ref = D2_GAIN*(1.397*theta_ref_last - 0.3972*
    theta_ref_last_last + 0.1233*phi_e - 0.2428*phi_e_last +
    0.1195*phi_e_last_last);
        rc_usleep(OUTER_TIMESTEP*1000000);
        }
        return NULL;
}

/*
    ******************************************************************************
* void on_pause_pressed()
*
* If the user holds the pause button for 2 seconds, set state to
    exiting which
* triggers the rest of the program to exit cleanly.
******************************************************************************
    */
void on_pause_pressed(){
        int i=0;
        const int samples = 100;        // check for release 100
    times in this period
        const int us_wait = 2000000; // 2 seconds

        // now keep checking to see if the button is still held
    down
      for(i=0;i<samples;i++){
                rc_usleep(us_wait/samples);
                if(rc_get_pause_button() == RELEASED) return;
        }
        printf("long press detected, shutting down\n");
        rc_set_state(EXITING);
        return;
}

void on_pause_released(){
        // toggle betewen armed and disarmed modes
        if (armstate==ARMED){
                 reset_controller();
                 rc_disable_motors();
                armstate = DISARMED;
        }
        else if (armstate==DISARMED){
                if(fabs(theta_f) < START_ANGLE_RANGE) {
                 reset_controller();
                rc_enable_motors();
                armstate = ARMED;
                }
        }
       return;
}
```

```c

void reset_controller(){
        //Reset encoders and all variables to reset controller
        rc_set_encoder_pos(R_MOTOR_CHANNEL,0);
        rc_set_encoder_pos(L_MOTOR_CHANNEL,0);

        phi_e = 0;
        phi_e_last = 0;
        phi_e_last_last = 0;

        theta_ref = 0;
        theta_ref_last = 0;
        theta_ref_last_last = 0;

        u = 0;
        u_last = 0;
        u_last_last = 0;

        theta_e = 0;
        theta_e_last = 0;
        theta_e_last_last = 0;
}
```