Matthew Hui (mmh257)
Filipe Branco (fad28)
Edward Gu (elg227)

PID Control of 1D Helicopter

Introduction

For this lab, the objective was to simulate the movement of a helicopter to one degree. In our example, instead of the helicopter being able to freely move around, instead we simulated its movements to a vertical plane of a sphere. (ie. if you slice a sphere vertically, the shape of our setup would move in a shape similar to that circle's circumference.) To create our simulation environment, we fixated a drone motor to one end of a plastic arm. This plastic arm was then fixated to a hinge attached on a wooden block such that the hinge opens up horizontally. The desired output of this lab was to have a motor like contraption that would quickly snap to an input angle.

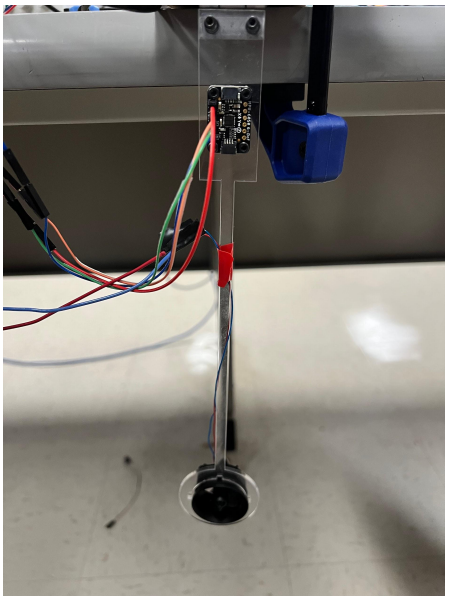


Figure 1: Motor set up attached to the plastic arm, with the top screws attached to a metal hinge.

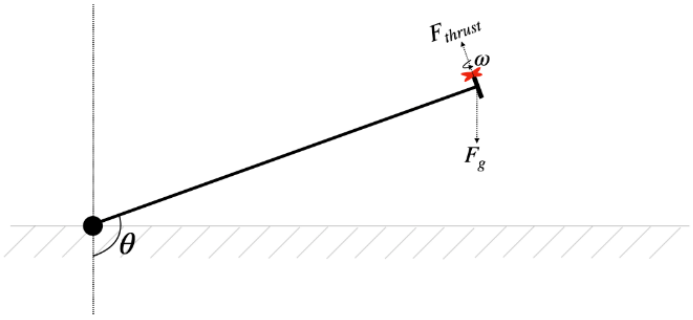


Figure 2: Free body diagram demonstrating the degree of motion and forces involved looking at our contraption sideways. (Source: course website)

In addition to the physical setup, we had many other components that needed to be properly connected on our breadboard and integrated before we could proceed with designing any logic corresponding to it. To obtain sufficient measurements to calculate angles and other parameters, we needed an IMU (inertial measurement unit) in addition to the Pico, and the chosen IMU was the MPU-6050 which had both gyroscope and accelerometer capabilities. (It is also depicted attached on the arm above). As with lab 2, we also integrated the VGA driver and had to set up its according circuit on our board in order to provide us with visual representations

of the calculations and data we were collecting. Finally, we also had to set up the proper motor circuit to provide the proper voltage input into our motor that we could control.

Hardware Component Setup

As briefly mentioned and depicted above, the entire setup consisted of multiple components each having its own specifications to set it up. Below is an image of the final circuit layout we had.

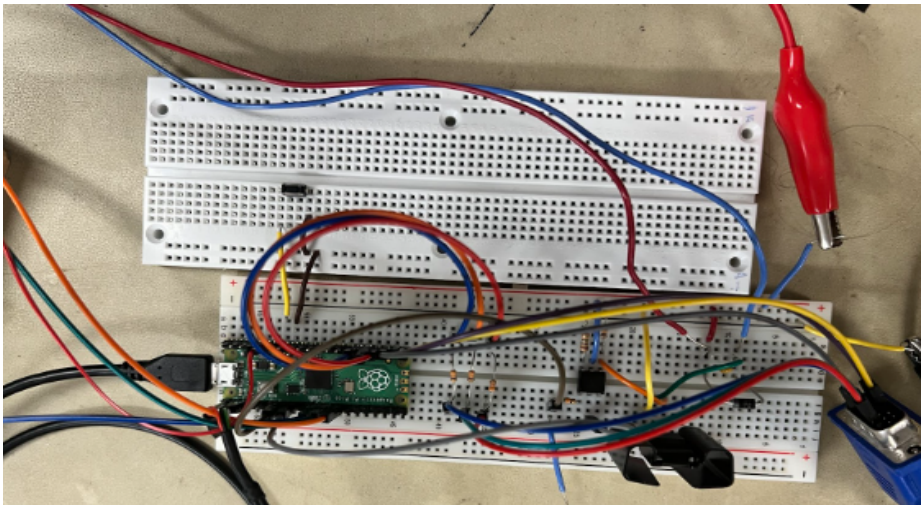


Figure 3: Breadboard containing circuit components

Motor Circuit:

At a higher level, we essentially need to generate a PWM signal (pulse width modulation) which looks like a square wave. With a longer duty cycle of the square wave (a longer time the square wave has a nonzero value), the motor is able to rotate faster relative to its maximum duty cycle. Increasing the overall voltage applied increases this maximum speed the motor can rotate at. (We have 2 basic parameters we are able to control of the motor, its max speed, and its % of the max speed it wants to operate at).

For the motor circuit, there were two main subcircuits that each had a function related to the operation of our lab. The first contained the PWM signal output coming from the MCU (microcontroller unit), and the second was circuitry to drive our actual motor. It is very important to note that it was extremely important for us to isolate both subcircuits from each other since the MCU can be very sensitive to the large voltage spikes that can arise from the second subcircuit. The circuit elements used were: an optoisolator to translate the PWM signal from the MCU to the rest of the circuit, a MOSFET, resistors, a diode, capacitor, nodes corresponding to the + and - of a motor, and a voltage supply. Below is an image of the sample circuit diagram we had:

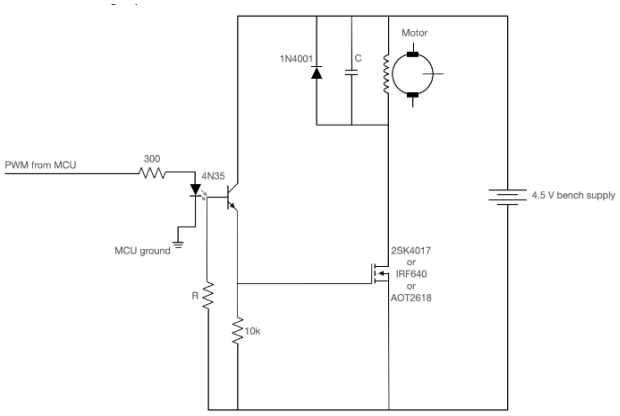


Figure 4: Motor circuit diagram.

Source: course website https://vanhunteradams.com/Pico/Helicopter/MotorCircuit.html

We ended up using a value of R = 1M Ohm, C = 0.1uF, and the IRF6040 MOSFET in our final circuit.

To generate the PWM signal, the input node was connected to GPIO 4 as it was configured to be the PWM signal output generated by the Pico. The 4.5V bench supply was provided using a voltage supply. (We were able to adjust the voltage to values up to 7V, also providing our circuit with a ground node to reference.) In the circuit diagram, although hard to point out, most of our circuit components were located in the bottom right of figure 3. The big black object was a heatsink placed on top of our MOSFET since it was reaching very high temperatures due to the large amounts of current drawn through our circuit relative to its comfortable levels.

VGA Screen:

The setup to the VGA screen was extremely similar to the setup of lab 2, connecting various GPIO output ports to its corresponding VGA Plug connections and including resistors for the ports that controlled the color.

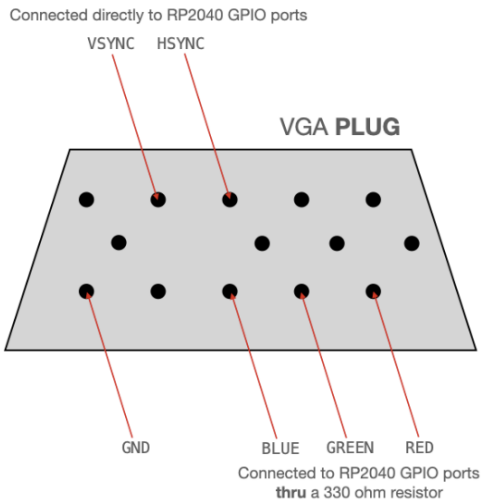


Figure 5: VGA Screen port connections
Source: course website

MPU Circuit Setup

Since we are using the MPU6050, to determine how to connect everything up, we referenced the datasheet and port explanation of the 6050 and connected it to the corresponding GPIO pins on the Pico.

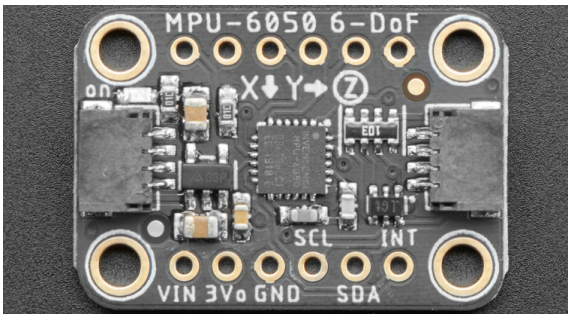


Figure 6: MPU-6050 pinout diagram zoomed in

Source: https://learn.adafruit.com/mpu6050-6-dof-accelerometer-and-gyro/pinouts

With respect to our Pico, the following pins from the GPIO were connected and used:

1. $3*V_o$: Pin 36, the 3V3(OUT) functionality
2. gnd: Any Pico ground pin is sufficient, Pin 8 was used
3. SDA: GPIO 8
4. SCL: GPIO 9

In the hardware setup image, in the bottom left corner there are 4 wires leaving the screen, and in figure 1 we see 4 wires entering the SDA. These wires represent the connections just listed above to ensure that the MPU has the proper channel to communicate with the Pico and actually collect data.

Motor Setup:

For our motor, plastic arm frames were already given to us, and all we needed to do was attach the motor and the IMU on it. For our motor, we had to attach additional wires and use a heat shrink to make sure the longer wire was properly created. The motor was also fixed to the acrylic arm with a rubber band, however, it was removed later on upon finding out that the pressure from the rubber band really limited the speed at which the motor was capable of spinning at. On the other end of the acryling arm, the IMU was fastened onto it using nuts and bolts and the resulting arm was then attached to a block of wood using a metal hinge. The final product was fastened onto the work bench with a clamp.

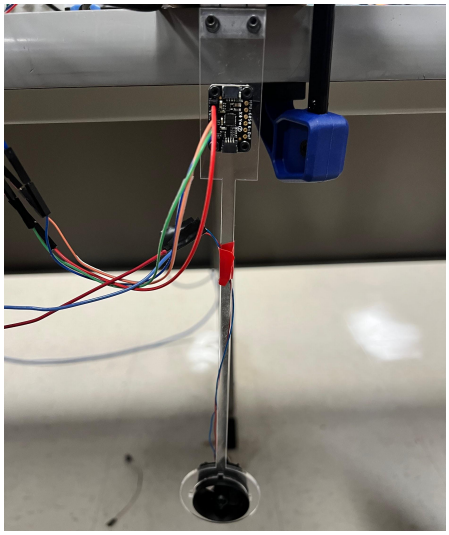


Figure 7: Arm set up after being clamped down onto the bench top.

Button Routine Test circuit

In addition to building out the hardware corresponding to the actual functionality of the motor, we also had to set up an additional circuit to implement our testing routine suite in the end. (When a button was pressed, our motor would pop up to 4 different angles, hovering at them for 5 second intervals.) We modeled our button as a simple switch and followed the following circuit diagram to set it up:

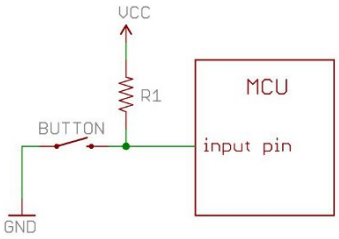


Figure 8: Button Pull Up circuit
Source: https://cdn.sparkfun.com/assets/6/f/b/c/7/511568b6ce395f1b40000000.jpg

However, instead of having an external resistor, we took advantage of the Pico's internal pull up resistor functionality, which incorporated the Vcc and R1 circuit part into the MCU. In order to accomplish this, we needed to add additional setup calls in the main method to properly initialize our GPIO pin to be responsive to the button. The following code was added into the bottom of our main method:

```
//Button for routine configuration
gpio_init(28); //Using pin 34 as the pull up
gpio_set_dir(28, GPIO_IN); //Setting it to an input
```

```
gpio_pull_up(28); //Turning on the internal pull up
```

Here, we used pin 34 on the board, corresponding to GPIO 28, as the gpio tracking the button inputs to our code. This pin was chosen purely for convenience on our circuit. (In figure 3, the button logic was added to a separate bread board due to lack of space and avoidance of interfering too close to other components). Here, all we needed to do was set GP28 as an input pin and a pull up resistor. Then, to properly have our program actively check for a button press, we added an additional thread onto one of the cores that contained logic using `(gpio_get(28) == 0)` to check whether or not the pin was being pulled to ground with the button on. (More detail on this thread later on.)

Additional background algorithms & concepts

Angle Measurement Logic - Complementary Filters:

To calculate our angles, we have two sets of measurements that we can collect to help us determine these values : data from the gyroscope and data from the accelerometer. Both sets of measurements alone are not sufficient enough to get us accurate results, and we will need to use information from both. We will use a similar model to figure 2 as our system under consideration.

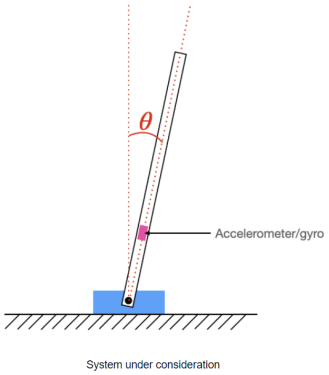


Figure 10: Diagram of the lever system & angle
Source: course website

1. Accelerometer angle measurements: With the accelerometer, we can measure the acceleration of the arm in 3 degrees, a_x, a_y, and a_z. In the system shown above, we only need the two values in the plane of the angle, in the example of the figure a_x and a_y, and we can find a reasonable estimate of the angle using the formula:

   $$\text{angle} = \arctan(a_x / a_y)$$

   However, these measurements of acceleration can be extremely noisy due to the very precise nature that we are operating within and especially within the presence of other mechanical noise from our motor. However, since the accelerometer has a fixed

reference vector, gravity, it eliminates any sort of measurement biasing error because all of our measurements with it will reference the same gravity.

2. Gyroscope measurements: The gyroscope operates differently from the accelerometer as it measures the rotation rate of our system. We can take advantage of this because we can simply integrate that value over time to determine the angle in radians of how much the arm has rotated. (ie: synonymous to integrating the velocity over time to determine the distance an object has moved.) This does, however, have a downside that overtime, we accumulate random biasing that can easily affect our angle measurements and calculations. (Although the noise here is much lower than the accelerometers.)

Using these two measurements, together and applying a complementary filter where we high pass one set of measurements and low pass another will allow us to take the best of both worlds. (We obtain the good stability of the accelerometer over time, and the nice preciseness and behavior of the gyroscope over short periods of time.) The following diagram demonstrates the flow of information in our complementary filter we implemented:

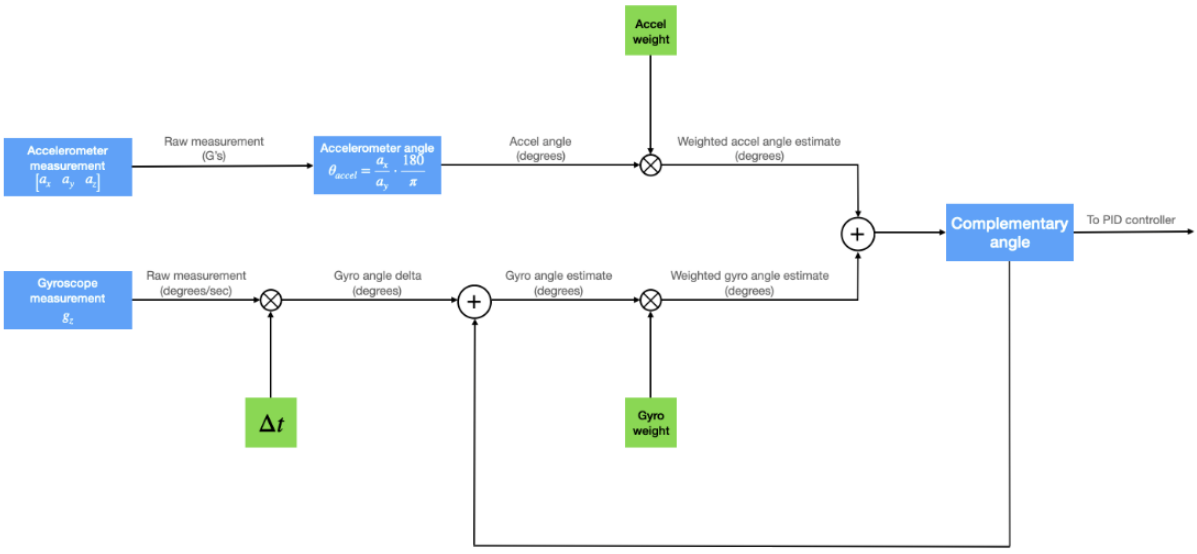


Figure 11: Complementary filter diagram
Source: course website

Each of the tunable values we used were: 0.001 for delta t since our system operated at 1kHz, 0.999 and 0.001 for the accel_weight and gyro_weight respectively which seemed to produce reasonable angle measurements.

Low pass filter:

In a similar vein to the complementary filter, some measurements needed to be low-passed due to the high noise nature and the amount of extra noise that could impact our system. In order to

low pass our measurements, we simply perform a bit shift of the difference in measurements of our data and add it onto the original value. The following formula demonstrates this:

```
motor_disp = motor_disp + ((pwm_on_time - motor_disp)>>6) ;
```

Figure 12: Low pass filter of our data
Source: course website

In our case, the motor_disp value can refer to anything we are trying to low-pass filter, and the pwm_on_time is the “measured” value that is actively changing. Graphically, when we bit shift, say the desired duty cycle we want to operate our system on, the duty cycle response has a gradual and nice shape to get the system up to the desired value.

PID Controller:

One of the main algorithms we used for our control over the motion of the arm was the PID controller - proportion, integral, and derivative controller. This controller operates on a closed loop feedback basis, where we take an input, in our case a fixed angle error and a constantly changing angle measurement, perform some operations on it to generate a behavior response, then recycle whatever output values and continue to draw more measurements from a sensor or some other sensing element.

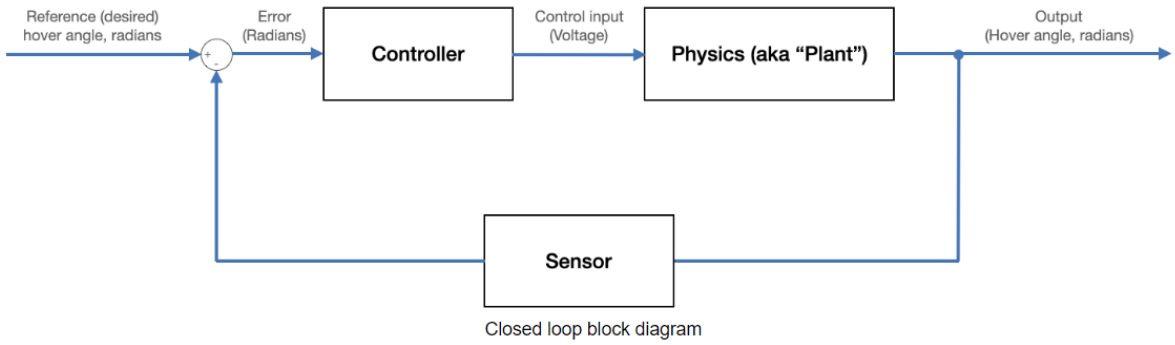


Figure 9: Closed loop block diagram of how the PID controller generally operates
Source: course website

Each of the components, proportion, integral, and derivative has a different effect on the general behavior of the system and the effect of each can be fine tuned using a coefficient value associated with it.

1. Proportion: The proportion term at a higher level, gets us the majority of the way to our angle. It essentially operates by fluctuating how fast our motor needs to spin (duty cycle) based on the magnitude of the current error; the difference between our current calculated angle and the actual target angle. However, one issue that does typically

arise with just this term is the fact that it is unable to fully reach our target angle since it operates on the difference between two angles (if it were at the target angle, the motor would shut off) and that it reaches a steady state error due to the nature of the motor trying to counteract the effects of gravity. In this error, we observe a constant up and down oscillation that may tend towards a state of instability as the error grows in magnitude.

2. Integral: This term works to counteract the steady state error observed by the proportion term, eliminating the up and down oscillations observed. This term operates on the sum of the measured angle errors overtime - the steady state error will continue to accumulate error, and the integral term “notices this” and scales it to add back to the duty cycle from the proportional component effectively eliminating the error. These two terms alone, however, will not reach our target angle and will need a third term.
3. Derivative: This final term looks at the rate of change of our error and uses past data to counteract the effects of the integral and proportional term. In doing so, it gives us extra room for our proportional term to “overshoot” the desired angle, however in reality this overshoot brings us closer to the actual angle.

Design & Testing Methods

For this lab, we had to incorporate many different parts with each other and each phase required different levels of testing.

The physical setup had its own series of testing. Since we had already used the VGA screen, we recycled the same connections from lab 2 to the VGA screen. The more complicated testing came from the motor circuit where it cascaded multiple smaller circuits on top of each other and we had to step through several individual tests. Referring to figure 4 from above, the first thing we did in testing this circuit was to simulate a PWM cycle using the function generator instead of starting off with the PWM signal from the MCU. The motor operates at a frequency of around 1kHz, so our simulated PWM signal was centered around 1kHz, and we aimed for a target voltage of 3.3V. (Which is the voltage output of the MCU). Once we verified that the functional generator was working, the oscilloscope was then used to probe at several spots. First, before the diode in the optoisolator circuit was tested. When probed with the oscilloscope, we saw a signal with amplitude around 1.3V, which is expected as we have a resistor in series with the input, providing a voltage drop spot. The next part we tested was above the 10k resistor. Here we needed to see a similar signal to the input, however amplified to the value set by the voltage supply. (The same PWM cycle behavior and characteristics were observed.) Once the motor was verified to work, we could then substitute in the MCU pin out in place of the functional generator.

For testing the validity of our work, we decided it would be best to have some sort of visual component displaying the angle and duty cycle/motor spin rate on the VGA screen. To get started with everything, we first utilized the sample code under MPU6050_IMU_Demo which

contained starter code and basic accelerometer and gyroscope measurements. This demo also contained basic VGA display information. The first thing we did was add in our own section in the upper left corner of the VGA screen containing information about the calculated angle our mechanical lever arm was currently held at, and the current duty cycle we were passing through our MPU to the motor. More detail and both calculations will be included later on in this lab report.

In addition to this, we also modified the information that was displayed in the two graphs on the VGA contained in the sample code. The upper graph was converted to displaying the angle on the range of [-180, 180] degrees for ease of calculations, and the bottom graph displayed the current duty cycle we were operating our motor at. On the bottom graph, in addition to the overall duty cycle of the motor, we also displayed the contributions from each component to the duty cycle. More detail on this graph and examples can be found later on.

With the necessary testing components developed, we were then able to add more to our lab and work towards the desired final product. We took a very modularized approach to the lab, compounding small features and progress on top of each other. Starting with the sample code, we then imported logic from another sample, the PWM_Demo, to add in code that would activate and control the PWM signal to a fixed angle. From here, we decided that we would use 90 degrees as a good reference to work with - this was the target test angle used when building the lab logic to move the lever arm to a given desired angle input. Since this lab had a bit more open-endedness in terms of making our lever arm reach the desired angle as quickly as possible and as “snappy”, we were given a lot of freedom and control on how to tune the values of the coefficients we chose. To tune our coefficients, we developed another testing suite using the serial terminal where we could, in real time, change the values of the coefficients and observe its effects on the responsiveness of our lever arm.

Software Design & Components

We first have two fix15 arrays that store the raw measurements of the acceleration and gyro. Among the other instantiated global variables, most notably we have two fix15 variables holding the old and new duty cycles, two fix15 variables keep track of the low pass filter, a fix15 variable containing the target angle, variables holding values for the Kp, Ki, and Kd constant parameters that we tune, a counter variable for Kd set to 0, a limit variable for Ki set to 5000, and two fix15 modifiers for Kd and Ki. (Shown below)

```
//volatile variables to keep track of low pass filter stuff
volatile fix15 accel_0 = int2fix15(0);
volatile fix15 accel_1 = int2fix15(0);

volatile fix15 motor_display = int2fix15(0);

volatile fix15 target_angle = int2fix15(-10);
volatile float angle_error = 0;
volatile float prev_error = 0;
volatile float error_delta = 0;
volatile float angle_delta_rad = 0;
volatile float total_error = 0;

volatile int k_p = 14000;
volatile int k_d = 1400000;
volatile float k_i = 20;

volatile int k_d_counter = 0;
volatile fix15 k_d_modifier = int2fix15(0);
volatile fix15 k_i_modifier = int2fix15(0);
volatile int k_i_limit = 5000;
```

Figure 13: Initialized variable code
Source: our own code we wrote

In our interrupt service routine, we read the raw measurements (acceleration and gyro) from our MPU6050 IMU and then apply the complementary filter algorithm which uses the short-term accuracy of the gyro angle estimates and the long-term stability of the accelerometer angle estimates. From the project website, we implement a low pass filter on the motor signal in order to stabilize the oscillations caused by our very large derivative term. We implement the provided equation where we subtract the motor display from the pwm on time, then right shift by 4 (filter has a cutoff frequency at 16 PWM units), and then add this onto the existing motor acceleration data on the y and z axis. Without small angle approximation (since the motor is tested on angles as large as 120 degrees), we compute the accelerometer angle using the provided equation, then compute the gyro angle delta which was also provided (multiply the raw measurement with a fixed timestep). Finally, we compute the "new complementary angle estimate by performing a weighted average of the accelerometer angle and the sum of the previous complementary angle estimate and the change in angle measured by the gyro" (lab project website).

To implement the proper logic for our interrupt service routine, we followed the pseudo code for calculating the angle using the accelerometer and gyroscope measurements. However, instead of referencing the x and y components for the accelerometer and gyroscope, we had to change it since our lever is operating in a different plane.

```
//Applying the low pass filtering on the raw acceleration data, using the Y and Z axis
accel_0 = accel_0 + ((acceleration[1] - accel_0)>>4);
accel_1 = accel_1 + ((acceleration[2] - accel_1)>>4);

// NO SMALL ANGLE APPROXIMATION
accel_angle = multfix15(float2fix15(atan2(accel_0, accel_1) + PI_2), oneeightyoverpi);

// Gyro angle delta (measurement times timestep) (15.16 fixed point), Adjusted the gyro to act on t
//it is rotating about the x axis
gyro_angle_delta = multfix15(-gyro[0], zeropt001);

// Computing true angle estimate
arm_angle = multfix15(arm_angle - gyro_angle_delta, zeropt999) + multfix15(accel_angle, zeropt001);
```

The accelerometer is moving the y and z directions, while our gyroscope rotates along the x axis.

The PID controller includes the proportional term which is proportional to the current error between the measured and desired output, the derivative term which is proportional to the rate of change of the error, and finally the integral term which is proportional to the integral of the error. (Note: the PID controller lies within the same ISR as the angle measurements since we utilize every new angle measurement to calculate the next duty cycle we need to set out data to.) If the target angle is greater than or equal to our lower bound of -10 degrees, we

calculate the proportional term each iteration, which is simply the target angle minus the current arm angle, converted to radians, and then multiplied by the Kp term which we tuned.

```
//Adding Kp logic to replace the code block above for setting the PWM channel
if (target_angle >= int2fix15(-10)) {
    angle_delta_rad = fix2float15(target_angle - arm_angle) * pi_180;
    angle_error = angle_delta_rad * k_p;
    new_duty_cycle = float2fix15(angle_error);
```

Note: since we are using radians as our measured angle, the values for all of our coefficients that we are tuning are at really large magnitudes.

For every 4 iterations of the interrupt (4 because that was what was recommended in lecture to reduce noise), we execute the inner if statement that calculates the derivative and integral terms (execute when Kd counter is equal to 4; Kd increments by 1 every iteration). The logic to calculate the derivative term is to divide the change in the error by the change in time, which we made to be 1, and then multiplied by the Kd term. The Kd counter is then set to 0. The integral term is calculated by getting the total error, which is just the target angle minus arm angle result added onto the total error, and then multiplied by the Ki term.

```
//k_d logic
if (k_d_counter == 4) {
    error_delta = angle_delta_rad - prev_error;
    k_d_modifier = float2fix15(error_delta * k_d);
    prev_error = angle_delta_rad;
    k_d_counter = 0;
    // new_duty_cycle = new_duty_cycle + k_d_modifier;

    //Adding k_i logic as well, should be consistent with d
    total_error = total_error + angle_delta_rad;
    k_i_modifier = float2fix15(total_error * k_i);
    if (k_i_modifier > int2fix15(5000)) {
        k_i_modifier = int2fix15(5000);
        total_error = divfix(k_i_modifier, float2fix15(k_i));
    }
    if (k_i_modifier < int2fix15(-5000)) {
        k_i_modifier = int2fix15(-5000);
        total_error = divfix(k_i_modifier, float2fix15(k_i));
    }
```

For our calculations, due to the extremely precise nature of the measurements, we had to operate on floating point values instead of fix15 since we were experiencing some overflow with our data values. This did not impact the speed of our system since we did not need to optimize the speed at which we calculated data.

Another error that we had to account for was the large accumulation by the integral term. If, say, our desired angle is 90 degrees and we hold our system to 45 degrees for extended periods of time, the accumulation of the error will tend to infinity. This is not our desired response since we cannot deal with values of infinity and instead we created a threshold maximum and minimum values for the duty cycle contributions of the Kd and Ki terms. If the integral term is greater than our upper limit of 5000 or less than our bottom limit of -5000, we set

the integral term to be 5000 or -5000 respectively and then divide this resulting term by Ki to be our total error. The new duty cycle is then just the sum of the proportional, derivative, and integral terms. Once the duty cycle is clamped, we signal the VGA to draw the duty cycle.

A separate thread, **protothread_button**, checks the button inputs, where if the button is pressed, then the switch is closed and the voltage reads from ground which is 0 (otherwise the gpio 28 reads 1). When the button is pressed the target angle for the motor first goes to 90 degrees, waits 5 seconds, goes to 120 degrees, waits 5 seconds, goes to 60 degrees, waits 5 seconds, and then finally remains to 90 degrees until the user stops.

```
while(true) {
    if (gpio_get(28) == 0) {
        //Beginning the testing routine
        target_angle = int2fix15(90);
        PT_YIELD_usec(5000000);
        target_angle = int2fix15(120);
        PT_YIELD_usec(5000000);
        target_angle = int2fix15(60);
        PT_YIELD_usec(5000000);
        target_angle = int2fix15(90);
    }
    PT_YIELD_usec(1); //Added this in to force exit the thread
```

Another thread, **protothread_vga**, draws to the VGA display a bottom plot containing the motor display and its individual contributions from the 3 PID terms, and a top plot showing the angle the 1D helicopter is currently at. The top plot scales the angle from -180 degrees to 180 degrees by first scaling the value to between 0 and 150 inclusive, and then offsetting by 80. (This ensures that we plot the values onto the right section of the VGA screen.) The bottom graph ranges from 0 to 5000 (consistent with our boundary values set at the beginning) and contains 4 colored lines, the green line shows the duty cycle, the white line shows the proportional term contribution, the red line shows the derivative term contribution, and the yellow line represents the integral term contribution. A 1 millisecond delay was added to draw a picture containing drops between all of the angle shifts. In addition to the plots, we display the current angle and the motor display at the top left of the VGA screen.

Finally we have an input thread where the user is able to interface with PuTTy to change the following parameters: press a to select the angle, press 1 to change the Kp value, press 2 to change the Kd value, press 3 to change the Ki value, press l to set the Ki limit, and press r to activate the routine.

```
//Logic to display a small menu
sprintf(pt_serial_out_buffer, "Select parameter to change \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "a: select angle \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "1: change k_p \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "2: change k_d \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "3: change k_i \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "l: set the k_i limit \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "r: activate routine \n\r");
serial_write;
```

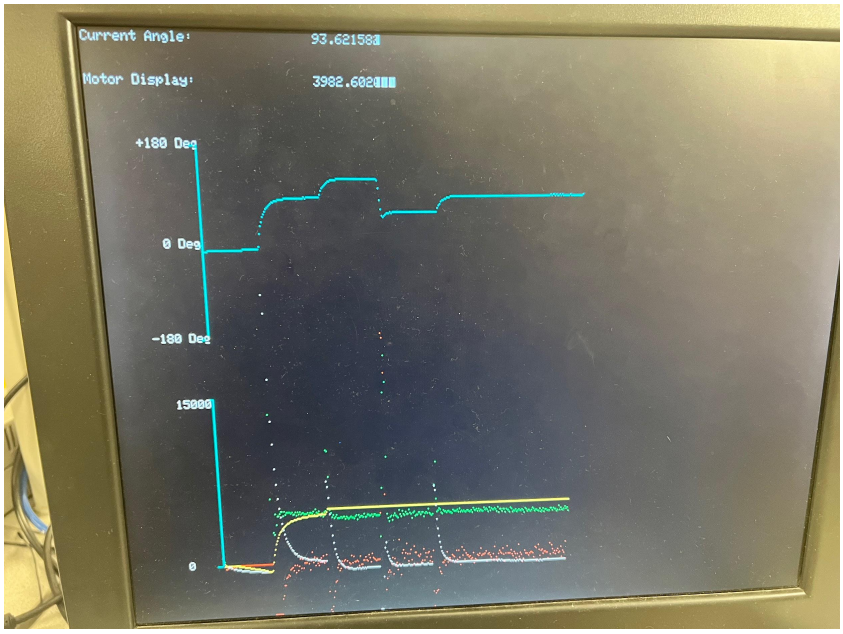


Figure 14: VGA Screen display showing the response of our system when changing our angles

In the above figure, it displays the response and values calculated/set when changing the arm angle from 0 -> 90 -> 120 -> 60 -> 90. The upper graph displays the trace of the angles we calculate that the arm is currently held at. We can observe that the angle does not immediately change, this is in part due to the filtering we applied to our angle measurements. The bottom graph displays the duty cycle responses of our system. The green line represents the overall duty cycle (the value our motor is set to), the yellow represents the contributions from the Ki term, the red represents the contributions from the Kd term, and the white represents the contributions from the Kp term.

*****Choosing Kp, Kd, Ki explanation included in results section below*****

Results

After accounting for the initial angle offset, we were able to get incredibly close to the real-life desired angle for the helicopter arm. To be precise, we consistently got to within 0.5 degrees of the real reading from the desired angle we imputed in the PuTTy. Here are three examples of values we measured(in degrees):

Desired Angle(as inputted in PuTTy)	Angle Reading from Sensor
60	~60.145
90	~89.968

120	~119.971

The reason for the approximations in the angle reading was that the sensor on the helicopter arm read up to 6 decimal points, so it would constantly oscillate around a small and finite range of values around the desired angle. With a reading that accurate, it is extremely difficult and almost impossible(at least for the scope of this lab) to achieve perfect accuracy for the robotic arm. However, we would venture to say that we got pretty close. In order to achieve this level of accuracy in our measurements, we had to play around with countless values for Kp, Kd, and Ki that would give us the best results. For this, we employed a sort of “educated trial-and-error” method, in which we somewhat knew what order of magnitudes each value needed to be, and tried to activate the helicopter arm with different values for each parameter within the appropriate range.

For Kp we knew it had to be the largest value since the proportion term was the part that got us closest to the target angle. We started out with very small values, starting out with 5000. While it was providing us with an angle, it was not getting us as close as we could to our target test value of 90. We continued to play around with the values until we reached a point where the angle we would get purely from the Kp term began to cap off without having an unstable steady state response. We found this current value to be around 7500. We then implemented the Kd term, which helped us reduce the amount of instability observed in the system. With the Kd term implemented, we could then go back to increase the Kp term to reach higher initial angles. We went back and forth between increasing and decreasing both respective values to reach the highest angle we could find to be: Kp = 14000, and Kd = 1400000. This behavior got us to around 80 degrees without having any Ki term. To reach the final 10 degrees of our system, we implemented the Ki term and played around with that value until our system was able to accurately reach our target test angle of 90 degrees. (The value of Ki was determined to be 20, we did not really observe much effect from Ki when increasing it any further.)

Max/Min angles to produce stable behavior for our PID gains (+- 1 degrees):

- Max angle: 153 Degrees
- Min angle: -20 degrees
- Resting angle: -7 degrees

Final Tuned PID Gains:

Kp	14,000
Kd	1,400,000
Ki	20

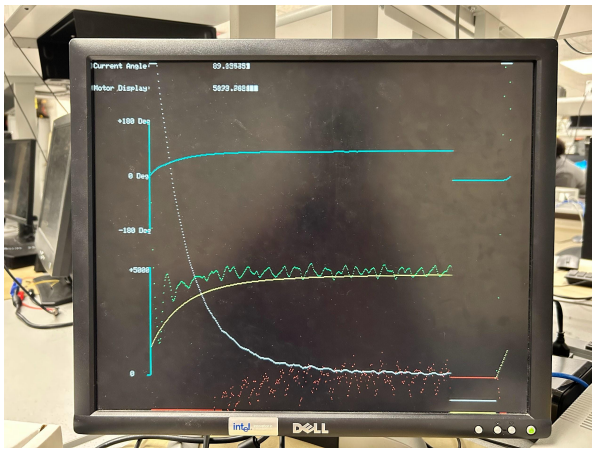
Setting times for different angles:

- Starting at -10 degrees because our set up when the motor is off starts at -10 degrees
- Measured using a stopwatch: Timed when we reached within +- 1 degrees

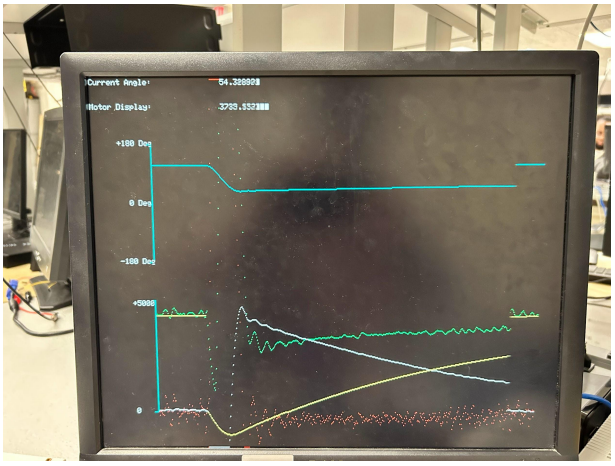
Matthew Hui (mmh257)
Filipe Branco (fad28)
Edward Gu (elg227)

0 to 30 deg	5.25 seconds	**60 to 120 deg**	3.3 seconds	**90 to 120 deg**	6.62 seconds
0 to 60 deg	4.52 seconds	**60 to 90 deg**	5.8 seconds	**90 to 60 deg**	9.37 seconds
0 to 90 deg	2.83 seconds	**60 to 30 deg**	8.55 seconds	**90 to 30 deg**	9.57 seconds
0 to 120 deg	1.93 seconds	**60 to 0 deg**	1.35 seconds	**90 to 0 deg**	0.98 seconds

- Shortest amount of time is from 90 to 0 degrees (0.98 seconds)
- Longest amount of time is from 90 to 30 degrees (9.57 seconds)
- Trend: time it takes typically increases when the change in degrees is lower, probably because of the way PID works.

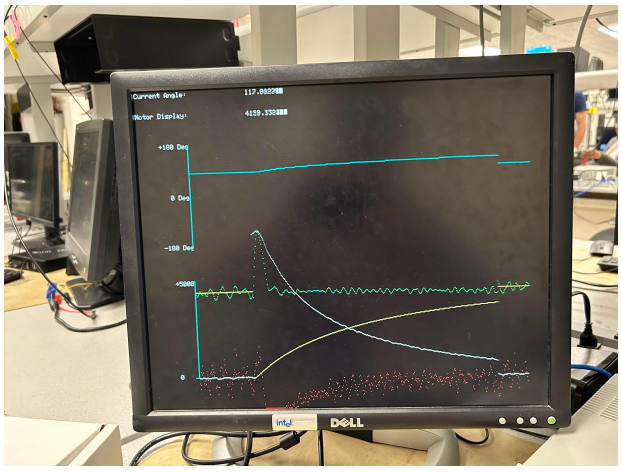


1:



2:

3:



4:

Images from left to right:

1. Response for setting the angle from 0 to 90 degrees
2. Response for setting angle from 90 to 60 degrees
3. Behavior in the “null”/off state - deadhanging
4. Transition from 90 to 120 degrees

These bottom graphs depict an isolated and more in depth view of the responses of our system and the different contributions from the PID values. These images are taken from before we changed the scaling of the y axis for the bottom one, which is why some of the components extend over the limits. However, it is interesting to note that the Ki term almost always reaches its maximum values when we transition to a new angle. In terms of how much each term contributes, the Kp term has the most prominent effect initially, as it is trying to adjust the arm as much as possible within reasonable effects to get as close as it can to the new angle. We see spikes in the white lines on the graphs representing this behavior. The derivative term fluctuates a lot as it is trying to “counteract” the instability we observe from the Kp term alone which can be seen with a relatively flatter shape in the green line, the overall duty cycle, as time goes on. As the Kp term decreases, our Ki term also increases to ensure that we maintain sufficient duty cycle and more to reach our desired angle.

For the top graphs, once the duty cycle plateaus at approximately the desired angle, it maintains the angle stably.

<u>Overall Graph Legend:</u>
Top Graph: Shows angle
Bottom Graph: Motor display + components

- Green line: duty cycle
- White line: k_p
- Red line: k_d
- Yellow line: k_i

Difficulties & Design Challenges:

Throughout the lab we encountered a few challenges that took extended debugging time to resolve. The first major difficulty we countered was the physical setup of the system. When we had set the coefficients to large magnitudes, our lever arm was capping off at very low angles; around 45 degrees. Upon some hardware analysis, we identified that the physical configuration and routing of our wires added additional downwards force onto the arm, limiting how high it was able to reach. To resolve this problem, we simply re-routed all of the wiring to the side of our wooden block and bread board in an attempt to isolate it as much as possible.

An error we encountered with our code was the structuring of the Ki and Kd on our duty cycle. Initially, we had our Kp and Ki only impact the duty cycle on every 4 interrupts, the frequency at which we updated our Kd and Ki values. However we noticed that with this setup, the contributions from Ki were near negligible and Kd was not behaving in the expected format. To get around this, we spent many hours debugging and thoroughly looking through our code

and found that even though the Kd and Ki terms get updated every 4 interrupts, our system should still “feel” Kd and Ki at every interrupt : these values do have to update on every interrupt, the every 4 cycle logic was only in place to help reduce the noise between measurements.

Conclusions

The purpose of this lab was to implement a PID controlled hardware implementation of a 1D helicopter (motor on a shaft hanging down from a block of wood) where it would only be able to move in a plane to a desired angle. We had to set up a motor circuit to provide the correct voltage to the helicopter’s motor. We observed how each of the three PID terms (along with a low pass filter) contributes to the overall correction of the helicopter’s angle error (target angle minus current angle), which depends on the measurement of the acceleration and gyro (from the IMU). We found that the Kd term that performs well is two orders of magnitude compared to the Kp term that performs well and the Ki term can be very low compared to both Kp and Kd, all of which was to our expectation. We were able to display both the angle of the helicopter as well as the motor display with each of the individual contributions from the Kp, Kd, and Ki terms.

Overall, we found the lab very interesting and learning about the use of PID controllers in a real application helped visualize its importance to us. The difference in the response of our system from an untuned system to a tuned system was incredible to see as it was very snappy and felt almost magical. We definitely had a longer time with understanding how the overall PID controller fully worked, however I think the resources and the information mentioned throughout the lectures was more than enough to help us get through the different parts of the lab. It was very fun having a lot of variability on how to alter certain aspects of the system and we think that the flow of development of the different parts of the lab was well thought out and thorough.

Code

```
/**
 * V. Hunter Adams (vha3@cornell.edu)
 *
 * This demonstration utilizes the MPU6050.
 * It gathers raw accelerometer/gyro measurements, scales
 * them, and plots them to the VGA display. The top plot
 * shows gyro measurements, bottom plot shows accelerometer
```

```
 * measurements.
 *
 * HARDWARE CONNECTIONS
 *  - GPIO 16 ---> VGA Hsync
 *  - GPIO 17 ---> VGA Vsync
 *  - GPIO 18 ---> 330 ohm resistor ---> VGA Red
 *  - GPIO 19 ---> 330 ohm resistor ---> VGA Green
 *  - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
 *  - RP2040 GND ---> VGA GND
 *  - GPIO 8 ---> MPU6050 SDA
 *  - GPIO 9 ---> MPU6050 SCL
 *  - 3.3v ---> MPU6050 VCC
 *  - RP2040 GND ---> MPU6050 GND
 */

// Include standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
// Include PICO libraries
#include "pico/stdlib.h"
#include "pico/multicore.h"
// Include hardware libraries
#include "hardware/pwm.h"
```

```
#include "hardware/dma.h"

#include "hardware/irq.h"

#include "hardware/adc.h"

#include "hardware/pio.h"

#include "hardware/i2c.h"

// Include custom libraries

#include "vga_graphics.h"

#include "mpu6050.h"

#include "pt_cornell_rp2040_v1.h"


// Arrays in which raw measurements will be stored

fix15 acceleration[3], gyro[3];


// character array

char screentext[40];


// draw speed

int threshold = 10;


// Some macros for max/min/abs

#define min(a, b) ((a < b) ? a : b)

#define max(a, b) ((a < b) ? b : a)

#define abs(a) ((a > 0) ? a : -a)


// semaphore
```

```
static struct pt_sem vga_semaphore;


// Some paramters for PWM
#define WRAPVAL 5000
#define CLKDIV 25.0
uint slice_num;


//PWM Duty cycles
volatile fix15 new_duty_cycle = int2fix15(0);
volatile fix15 old_duty_cycle = int2fix15(0);


#define PI 3.14159
#define PI_2 1.57079
#define pi_180 0.017453


//Accelerometer angle
fix15 arm_angle = int2fix15(-10);
fix15 gyro_angle_delta;
fix15 accel_angle;


//volatile variables to keep track of low pass filter stuff
volatile fix15 accel_0 = int2fix15(0);
volatile fix15 accel_1 = int2fix15(0);


volatile fix15 motor_display = int2fix15(0);
```

```
volatile fix15 target_angle = int2fix15(-10);
volatile float angle_error = 0;
volatile float prev_error = 0;
volatile float error_delta = 0;
volatile float angle_delta_rad = 0;
volatile float total_error = 0;

volatile int k_p = 14000;
volatile int k_d = 1400000;
volatile float k_i = 20;

volatile int k_d_counter = 0;
volatile fix15 k_d_modifier = int2fix15(0);
volatile fix15 k_i_modifier = int2fix15(0);
volatile int k_i_limit = 5000;

// Interrupt service routine
void on_pwm_wrap()
{

    // Clear the interrupt flag that brought us here
    pwm_clear_irq(pwm_gpio_to_slice_num(5));

    // Read the IMU
```

```
    // NOTE! This is in 15.16 fixed point. Accel in g's, gyro in deg/s

    // If you want these values in floating point, call fix2float15() on

    // the raw measurements.

    mpu6050_read_raw(acceleration, gyro);


    //Applying the low pass filtering on the raw acceleration data, using
the Y and Z axis

    accel_0 = accel_0 + ((acceleration[1] - accel_0)>>4);

    accel_1 = accel_1 + ((acceleration[2] - accel_1)>>4);


    // NO SMALL ANGLE APPROXIMATION

    accel_angle = multfix15(float2fix15(atan2(accel_0, accel_1) + PI_2),
oneeightyoverpi);


    // Gyro angle delta (measurement times timestep) (15.16 fixed point),
Adjusted the gyro to act on the x axis since

    //it is rotating about the x axis

    gyro_angle_delta = multfix15(-gyro[0], zeropt001);


    // Computing true angle estimate

    arm_angle = multfix15(arm_angle - gyro_angle_delta, zeropt999) +
multfix15(accel_angle, zeropt001);


    motor_display = motor_display + ((new_duty_cycle - motor_display) >>
6);


    //Adding Kp logic to replace the code block above for setting the PWM
channel
```

```
    if (target_angle >= int2fix15(-10)) {
        angle_delta_rad = fix2float15(target_angle - arm_angle) * pi_180;
        angle_error = angle_delta_rad * k_p;
        new_duty_cycle = float2fix15(angle_error);

        //k_d logic
        if (k_d_counter == 4) {
            error_delta = angle_delta_rad - prev_error;
            k_d_modifier = float2fix15(error_delta * k_d);
            prev_error = angle_delta_rad;
            k_d_counter = 0;
            // new_duty_cycle = new_duty_cycle + k_d_modifier;

            //Adding k_i logic as well, should be consistent with d
            total_error = total_error + angle_delta_rad;
            k_i_modifier = float2fix15(total_error * k_i);
            if (k_i_modifier > int2fix15(5000)) {
                k_i_modifier = int2fix15(5000);
                total_error = divfix(k_i_modifier, float2fix15(k_i));
            }
            if (k_i_modifier < int2fix15(-5000)) {
                k_i_modifier = int2fix15(-5000);
                total_error = divfix(k_i_modifier, float2fix15(k_i));
            }
            // new_duty_cycle = new_duty_cycle + k_i_modifier;
```

```
        }


        new_duty_cycle = new_duty_cycle + k_d_modifier;

        new_duty_cycle = new_duty_cycle + k_i_modifier;


        k_d_counter = k_d_counter + 1;




        //clamp the duty cycle

        if (new_duty_cycle < int2fix15(0)) {

            new_duty_cycle = int2fix15(0);

        }

        pwm_set_chan_level(slice_num, PWM_CHAN_B,
fix2int15(new_duty_cycle));

    }


    // Signal VGA to draw

    PT_SEM_SIGNAL(pt, &vga_semaphore);

}


//Thread to check the button inputs

static PT_THREAD(prototheread_button(struct pt *pt))

{

    PT_BEGIN(pt);

    while(true) {

        if (gpio_get(28) == 0) {
```

```
            //Beginning the testing routine
            target_angle = int2fix15(90);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(120);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(60);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(90);
        }
        PT_YIELD_usec(1); //Added this in to force exit the thread
    }
    PT_END(pt);
}

// Thread that draws to VGA display
static PT_THREAD(protothread_vga(struct pt *pt))
{
    // Indicate start of thread
    PT_BEGIN(pt);

    // We will start drawing at column 81
    static int xcoord = 81;

    // Rescale the measurements for display
    static float OldRange = 500.; // (+/- 250)
```

```
    static float NewRange = 150.; // (looks nice on VGA)
    static float OldMin = -250.;
    static float OldMax = 250.;


    // Control rate of drawing
    static int throttle;


    //Values for drawing out the arm angle
    char angletext[40];


    // Draw the static aspects of the display
    setTextSize(1);
    setTextColor(WHITE);


    // Draw bottom plot
    drawHLine(75, 430, 5, CYAN);
    drawHLine(75, 280, 5, CYAN);
    drawVLine(80, 280, 150, CYAN);
    writeString(screentext);
    sprintf(screentext, "15000");
    setCursor(50, 280);
    writeString(screentext);
    sprintf(screentext, "0");
    setCursor(50, 425);
    writeString(screentext);
```

```
// Draw top plot
drawHLine(75, 230, 5, CYAN);
drawHLine(75, 155, 5, CYAN);
drawHLine(75, 80, 5, CYAN);
drawVLine(80, 80, 150, CYAN);
sprintf(screentext, "0 Deg");
setCursor(50, 150);
writeString(screentext);
sprintf(screentext, "+180 Deg");
setCursor(35, 75);
writeString(screentext);
sprintf(screentext, "-180 Deg");
setCursor(35, 225);
writeString(screentext);

//Static Text to keep track of angle
sprintf(angletext, "Current Angle: ");
setCursor(0, 0);
setTextSize(1);
setTextColor(WHITE);
writeString(angletext);

//Static Text to keep track of motor display
sprintf(angletext, "Motor Display: ");
```

```
    setCursor(0, 30);
    setTextSize(1);
    setTextColor(WHITE);
    writeString(angletext);

    while (true)
    {
        // Wait on semaphore
        PT_SEM_WAIT(pt, &vga_semaphore);
        // Increment drawspeed controller
        throttle += 1;
        // If the controller has exceeded a threshold, draw
        if (throttle >= threshold)
        {
            // Zero drawspeed controller
            throttle = 0;

            // Erase a column
            drawVLine(xcoord, 0, 480, BLACK);

            //Drawing the top plot, scaling the angle from -180 to 180
            //First scale value to between [0, 150], then offset it by 80
            drawPixel(xcoord, 230 - (int)((fix2float15(arm_angle) +
180)*0.416666) , CYAN);
```

```
            //Drawing the bottom plot, showing the motor display, ranging
from 0 to 5000

            drawPixel(xcoord, 430 -
(int)(fix2float15(motor_display)*0.01), GREEN); //duty cycle



            //Also plotting the k_d and k_p addition terms

            //k_p ranging from [0, 5000]

            //k_d ranging from [0, 5000]

            drawPixel(xcoord, 430 - (int)(angle_error*0.01), WHITE); //k_p

            drawPixel(xcoord, 430 - (int)(fix2float15(k_d_modifier)*0.01),
RED); //k_d

            drawPixel(xcoord, 430 - (int)(fix2float15(k_i_modifier)*0.01),
YELLOW); //k_i



            // Update horizontal cursor

            if (xcoord < 609)

            {

                xcoord += 1;

            }

            else

            {

                xcoord = 81;

            }



            //Additional logic to display the angle

            fillRect(175, 0, 50, 50, BLACK);

            sprintf(angletext, "%3f", fix2float15(arm_angle));
```

```
            setCursor(175, 0);
            setTextSize(1);
            setTextColor(WHITE);
            writeString(angletext);

            //Logic to display the motor disp
            sprintf(angletext, "%f", fix2float15(motor_display));
            setCursor(175, 30);
            setTextSize(1);
            setTextColor(WHITE);
            writeString(angletext);
        }
        PT_YIELD_usec(10000);
    }
    // Indicate end of thread
    PT_END(pt);
}

// User input thread. User can change draw speed
static PT_THREAD(protothread_serial(struct pt *pt))
{
    PT_BEGIN(pt);
    static char classifier;
    static int test_in;
    static float float_in;
```

```
//Adding in static local variables for duty cycle
static int duty_cycle_input;
static int target_angle_input;
static int k_p_input;
static int k_d_input;
static float k_i_input;
static int k_i_limit_input;

//Logic to display a small menu
sprintf(pt_serial_out_buffer, "Select parameter to change \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "a: select angle \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "1: change k_p \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "2: change k_d \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "3: change k_i \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "l: set the k_i limit \n\r");
serial_write;
sprintf(pt_serial_out_buffer, "r: activate routine \n\r");
serial_write;
```

```
    while (1)
    {
        sprintf(pt_serial_out_buffer, "input a command: ");
        serial_write;
        // spawn a thread to do the non-blocking serial read
        serial_read;
        // convert input string to number
        sscanf(pt_serial_in_buffer, "%c", &classifier);

        // num_independents = test_in ;
        if (classifier == 't')
        {
            sprintf(pt_serial_out_buffer, "timestep: ");
            serial_write;
            serial_read;
            // convert input string to number
            sscanf(pt_serial_in_buffer, "%d", &test_in);
            if (test_in > 0)
            {
                threshold = test_in;
            }
        }

        //Adding in another functionality for obtaining duty cycle
```

```
        if (classifier == 'd')
        {
            sprintf(pt_serial_out_buffer, "duty cycle [0,5000]: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%d", &duty_cycle_input);
            if (duty_cycle_input > 5000)
                continue;
            else if (duty_cycle_input < 0)
                continue;
            else
                new_duty_cycle = int2fix15(duty_cycle_input);
        }


        //Functionality to take in an angle:
        if (classifier == 'a')
        {
            sprintf(pt_serial_out_buffer, "Desired Angle [-20, 180]: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%d", &target_angle_input);
            if (target_angle_input > 180)
                continue;
```

```
            else if (target_angle_input < -20)
                continue;
            else {
                target_angle = int2fix15(target_angle_input);
                k_i_modifier = int2fix15(0);
                total_error = 0;
                angle_error = 0;
                prev_error = 0;
                error_delta = 0;
                angle_delta_rad = 0;
            }
        }

        //Functionality to find the target desired values: Kp value
        if (classifier == '1')
        {
            sprintf(pt_serial_out_buffer, "Desired k_p value: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%d", &k_p_input);
            if (k_p_input < 0)
                continue;
            else
                k_p = k_p_input;
```

```
        }
        //K_d value tuning next
        if (classifier == '2')
        {
            sprintf(pt_serial_out_buffer, "Desired k_d value: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%d", &k_d_input);
            if (k_d_input < 0)
                continue;
            else
                k_d = k_d_input;
        }
        if (classifier == '3'){
            sprintf(pt_serial_out_buffer, "Desired k_i value: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%f", &k_i_input);
            if (k_i_input < 0)
                continue;
            else
                k_i = k_i_input;
        }
```

```
        if (classifier == 'l') {
            sprintf(pt_serial_out_buffer, "Desired k_i_limit value: ");
            serial_write;
            serial_read;
            //Obtaining the duty cycle
            sscanf(pt_serial_in_buffer, "%d", &k_i_limit_input);
            if (k_i_limit_input < 0)
                continue;
            else
                k_i_limit = k_i_limit_input;
        }
        if (classifier == 'r') {
            //Beginning the testing routine
            target_angle = int2fix15(90);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(120);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(60);
            PT_YIELD_usec(5000000);
            target_angle = int2fix15(90);
        }

    }
    PT_END(pt);
}
```

```
// Entry point for core 1

void core1_entry()

{

    pt_add_thread(protothread_vga);

    pt_add_thread(prototheread_button);

    pt_schedule_start;

}


int main()

{


    // Initialize stdio

    stdio_init_all();


    // Initialize VGA

    initVGA();



////////////////////////////////////////////////////////////////////////

    ////////////////////////// I2C CONFIGURATION
/////////////////////////////

    i2c_init(I2C_CHAN, I2C_BAUD_RATE);

    gpio_set_function(SDA_PIN, GPIO_FUNC_I2C);

    gpio_set_function(SCL_PIN, GPIO_FUNC_I2C);

    gpio_pull_up(SDA_PIN);
```

```
    gpio_pull_up(SCL_PIN);


    // MPU6050 initialization
    mpu6050_reset();
    mpu6050_read_raw(acceleration, gyro);


    //Button for routine configuration
    gpio_init(28); //Using gpio 34 as the pull up
    gpio_set_dir(28, GPIO_IN); //Setting it to an input
    gpio_pull_up(28); //Turning on the internal pull up




////////////////////////////////////////////////////////////////////////////

    ////////////////////////// PWM CONFIGURATION
/////////////////////////////


////////////////////////////////////////////////////////////////////////////
    // Tell GPIO's 4,5 that they allocated to the PWM
    gpio_set_function(5, GPIO_FUNC_PWM);
    gpio_set_function(4, GPIO_FUNC_PWM);


    // Find out which PWM slice is connected to GPIO 5 (it's slice 2, same
for 4)
    slice_num = pwm_gpio_to_slice_num(5);


    // Mask our slice's IRQ output into the PWM block's single interrupt
line,
```

```
    // and register our interrupt handler
    pwm_clear_irq(slice_num);
    pwm_set_irq_enabled(slice_num, true);
    irq_set_exclusive_handler(PWM_IRQ_WRAP, on_pwm_wrap);
    irq_set_enabled(PWM_IRQ_WRAP, true);

    // This section configures the period of the PWM signals
    pwm_set_wrap(slice_num, WRAPVAL);
    pwm_set_clkdiv(slice_num, CLKDIV);

    // This sets duty cycle
    pwm_set_chan_level(slice_num, PWM_CHAN_B, 0);
    pwm_set_chan_level(slice_num, PWM_CHAN_A, 0);

    // Start the channel
    pwm_set_mask_enabled((1u << slice_num));


//////////////////////////////////////////////////////////////////////////

    ////////////////////////////// ROCK AND ROLL
//////////////////////////////


//////////////////////////////////////////////////////////////////////////
    // start core 1
    multicore_reset_core1();
    multicore_launch_core1(core1_entry);
```

```
    // start core 0

    pt_add_thread(protothread_serial);

    // pt_add_thread(prototheread_button);

    pt_schedule_start;

}
```