ECE 4760 - Digital Systems Design Using Microcontrollers

Lab 3: PID Control of an Inverted Pendulum with a Reaction Wheel

[redacted]

Introduction:

In Lab 3, we worked on controlling and stabilizing an inverted pendulum with a reaction wheel using a Raspberry Pi Pico board (RP2040). The overall goal was to keep the inverted pendulum balanced at the zero angle (vertical position) by accelerating the reaction wheel as by conservation of angular momentum, accelerating the wheel one way will spin the pendulum the opposite way. We achieved this goal by using a PID controller to control the speed of the motor that the wheel is attached to and the torque generated by the wheel while it is trying not to fall to either side. In order for the inverted pendulum to be balanced, we needed to determine the pendulum's tilt angle, which was calculated by the complementary filter of the accelerometer and gyro measurement. We also considered other factors such as the proportional control, integral control, dithering, derivative control, and PID control algorithm to tune the PID controller. Tuning the PID controller would result in a better balance to the pendulum when there is an external force put upon the spinning wheel.

Hardware:

In order to begin this lab, we first needed to assemble the mechanical components. There were three major hardware components in this lab: the setup of the mechanical components of the inverted pendulum with a reaction wheel, wiring the H-bridge motor circuit, and connecting the VGA display.

The setup for the mechanical components was straightforward. We followed the instructions under the course webpage for the mechanical construction of an inverted pendulum. With the materials listed in figure 1, we were able to finish the mechanical construction of the system by screwing, attaching the components, building the lego platforms, and soldering pins on the IMU board for the pendulum.

- Lasercut acrylic arm (x1) (x1)
- 3D printed reaction wheel - designed by Smith Charles
- Hobby gearmotor (x1)
- MPU-6050 IMU breakout board (x1)
- Clamps (x2)
- Lego bricks
- 10-32 screw (x1)
- 10-32 nut or wingnut (x1)
- M2.5 screw (x4)
- M3 nuts (x4)
- M2.5 screw (x2)
- M3 nut or wingnut (x2)

Figure 1. List of materials used for the mechanical construction of the inverted pendulum with a reaction wheel

The motor and the IMU are both attached to the arm. The IMU pinout connection with RP2040 is shown in figure 9. The reaction wheel is attached to the gear motor. The whole attachment of the arm is screwed between the legos for it to be able to move left and right. The lego bricks are stabilized by attaching the components to the edge of the table. With the mechanical setup, the whole design should look like the following in figure 2.

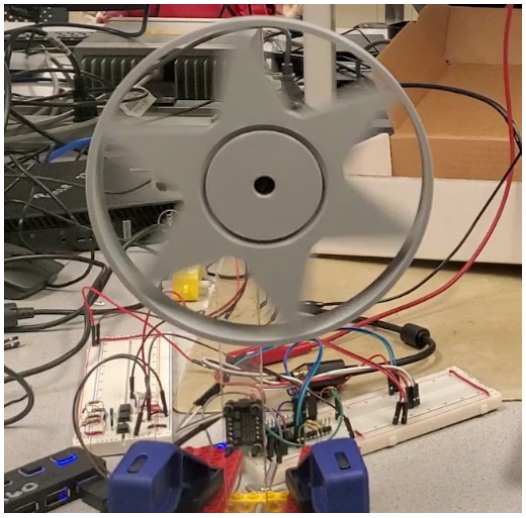


Figure 2. The inverted pendulum with a reaction wheel.

Connecting the Raspberry Pi Pico (RP2040) to the VGA cable was straightforward. The VGA display displays the accelerometer angle, gyro angle, complementary, and the PID computed control input. We used the pinout in figure 3 to connect to the VGA cable using the GPIO layout specified in figure 10.

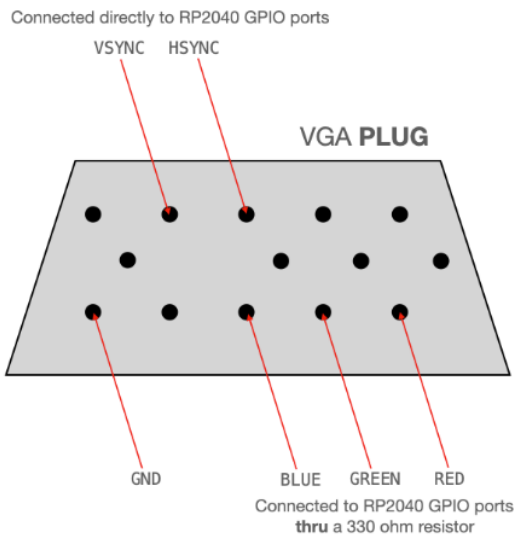


Figure 3. VGA pinout (from VGA driver page on 4760 site)

To control the pendulum reaction wheel, a motor must be controlled driving the wheel. Since the motor can produce back current, an H-bridge circuit was created using an L9110H chip. In this way, the circuit takes in 2 inputs and turns the motor one way or the other depending on which input is on and how strong the input is, shown in figure 5 using the pinout in figure 4.

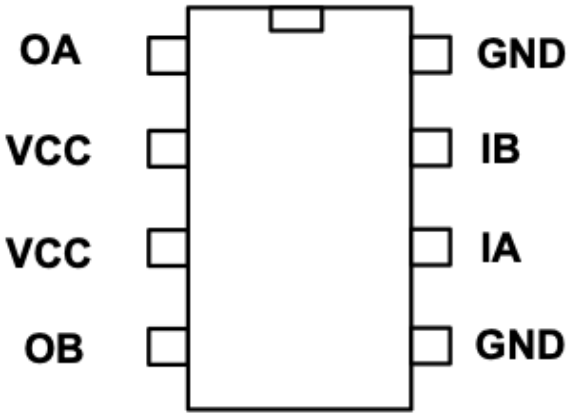


Figure 4. L9110 pinout (4760 site)

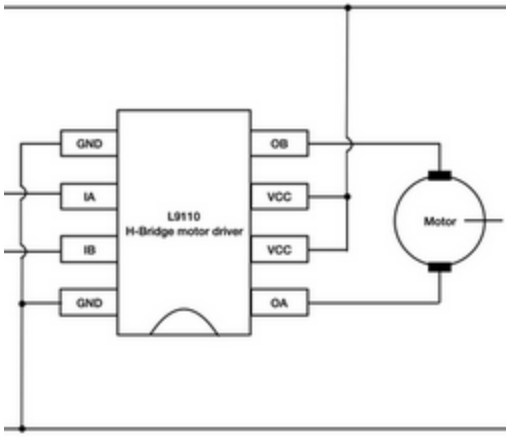
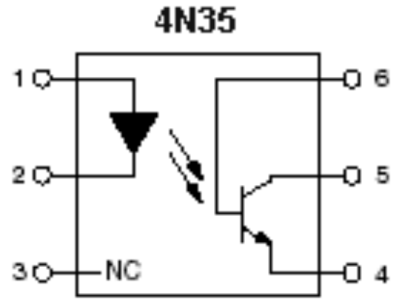


Figure 5. Schematic of H-bridge circuit (4760 site)

The IMU sends data to the RP2040 using an I2C connection, but the problem is that the motor controlling the wheel on the pendulum produces a great amount of noise. To reduce the effect of this noise, an optocoupler 4N35 chip was used in order to send our PWM signal to the H-bridge circuit without having to have a common ground between the two shown in figure 7 using the 4N35 pinout in figure 6.



PIN 1. ANODE
2. CATHODE
3. NO CONNECTION
4. EMITTER
5. COLLECTOR
6. BASE

Figure 6. 4N35 pinout (4760 site)

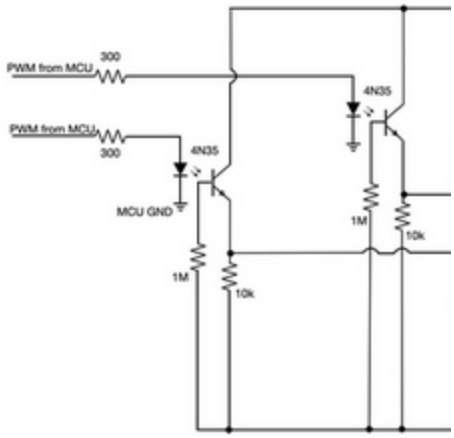


Figure 7. Schematic of optocoupler circuit (4760 site)

To test the entire circuit, we started by building the optocoupler circuit. We tested this circuit by using a function generator and producing a square wave at the input. If the input received a square wave, then the output should be a square wave as well tested with an oscilloscope probe. If any wires were placed incorrectly, this square wave would not show up. Once this circuit was tested, we connected the optocoupler circuit to the H-bridge circuit creating the motor driver circuit in figure 8. This was also tested by making sure the inputs of the L9100H were consistent with the square wave input of the function generator. When these inputs were correct. The circuit worked.

The motor driver circuit takes in analog inputs to control the motor, so using PWM signals from the RP2040 is the closest option to producing an 'analog' signal. Mixing these circuits, a total motor driver circuit was created taking in the RP2040 PWM output as inputs and using a bench power supply for the 5V supply.

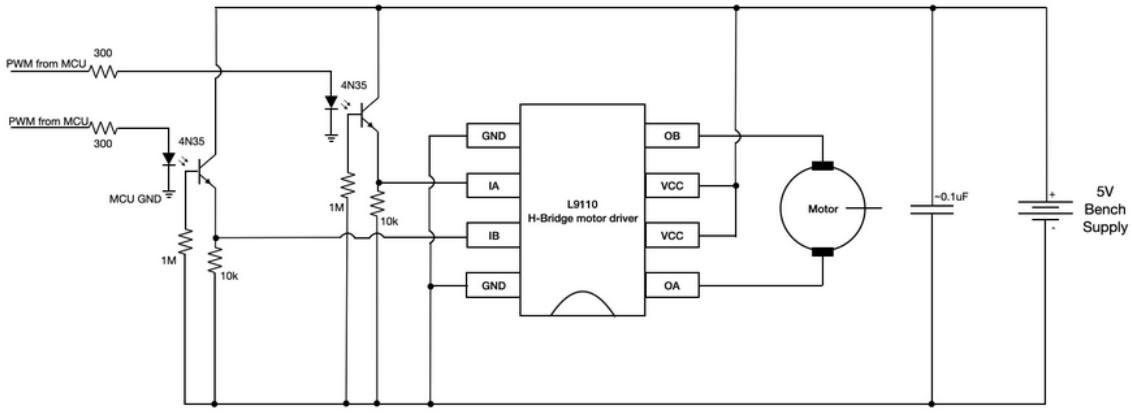


Figure 8. Schematic of motor driver circuit (from the ECE 4760 website)

With circuits built for connecting to the motor, IMU, and VGA screen, we created the final circuit for the entire lab shown in figure 9 using pin connections shown in figure 10. The top right off screen is the connection to the bench power supply as well as the vga connection offscreen.

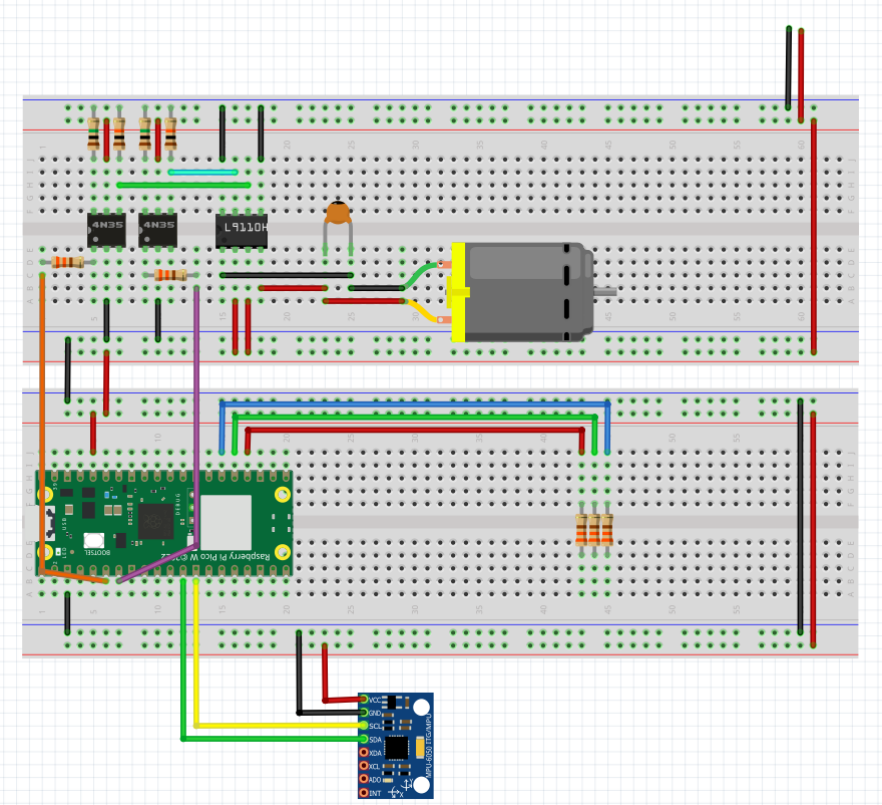


Figure 9. Breadboard layout (VGA connection and power supply off screen including resistors for RGB connections)

```
10    * HARDWARE CONNECTIONS
11    *   - GPIO 16 ---> VGA Hsync
12    *   - GPIO 17 ---> VGA Vsync
13    *   - GPIO 18 ---> 330 ohm resistor ---> VGA Red
14    *   - GPIO 19 ---> 330 ohm resistor ---> VGA Green
15    *   - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
16    *   - RP2040 GND ---> VGA GND
17    *   - GPIO 8 ---> MPU6050 SDA
18    *   - GPIO 9 ---> MPU6050 SCL
19    *   - 3.3v ---> MPU6050 VCC
20    *   - RP2040 GND ---> MPU6050 GND
21    *    - GPIO 4 ---> PWM1 output
22    *    - GPIO 5 ---> PWM2 output
```

Figure 10. RP2040 side VGA and IMU pinout from imu_demo.c (line 11-22)

As can be seen, a capacitor is placed in between the power supply leads in the circuit schematic, but our final design had the capacitor in between the motor leads. This is to reduce sparks, so the capacitor will clean the power supply while also reducing sparking from the DC motor. To reduce the mentioned noise from the motor that would break the I2C communication between the IMU and RP2040, we put the H-bridge and IMU connections on two different breadboards in order to separate the circuits as much as possible to reduce noise.

Software:

In our implementation, we had a main function, a thread for the user interface, another thread for the VGA display, and a PWM interrupt service routine function. The main function performs all the initializations for communication to the various devices and starts the protothreads running the continuous code.. The two threads (*protothread_vga(struct pt *pt)* and *protothread_serial(struct pt *pt)*) manage the user input from the serial interface and display the accelerometer angle, gyro angle, complementary angle, and control input on the VGA display. With the parameters being modified in the user interface thread, the VGA display responds to the change in the parameters and reflects the modified data on the screen.

The main function completed various tasks. It called stdio_init_all(), initVGA() and i2c_init() to initialize all necessary peripherals. Then, the SDA and SCL pins for use in the I2C connection to the IMU were set to the I2C function using gpio_set_function(pin#, GPIO_FUNC_I2C). Functions were given to use with the IMU (mpu6050). In order to get an initial measurement on the approximated angle of the accelerometer (later explained in the **Calculating Pendulum Angle** section).

Various things were done at the beginning of main to set up the PWM outputs that would control the motor. First, the gpio_set_function(pin#,GPIO_FUNC_PWM) was used to set pins 4 and 5 to outputs. Next, the PWM configuration functions configure “slices” but not gpio ports themselves as they are not part of the PWM peripheral. This meant, pwm_gpio_to_slice_num had to be used to get the slice number of the gpio pins. Next, pwm_clear_irq, pwm_set_irq_enabled, irq_set_exclusive_handler, and irq_set_enabled were all used to the set an interrupt that would occur at the end of a PWM period using “on_pwm_wrap” as the function to use for the interrupt handler. Many tasks are completed in the interrupt handler and these values should be updated at 1kHz, so the PWM outputs had to have a frequency of 1kHz. To do this, pwm_set_clkdiv was used with a clock div of 25. With this value, the 125MHz clock was divided by 25 to 5MHz when entering the PWM peripheral. With a 5MHz clock, to make the period of the PWM signal 1kHz, the wrap value was set using pwm_set_wrap to 5000. With this, the period of the PWM signal was 5000cycles, so 5MHz/5000cycles = 1kHz thus achieving the desired PWM frequency.

Calculating Pendulum Angle:

The sensors at our disposal were an accelerometer and a gyroscope that had a permanent bias that would constantly lower the gyroscope measurement and the accelerometer would produce an unbiased but inconsistent measurement. In order to get the best of both of these measurements, we needed to low pass the accelerometer measurement to avoid the high-frequency inconsistent measurements while high pass the low-frequency biased measurement of the gyroscope.

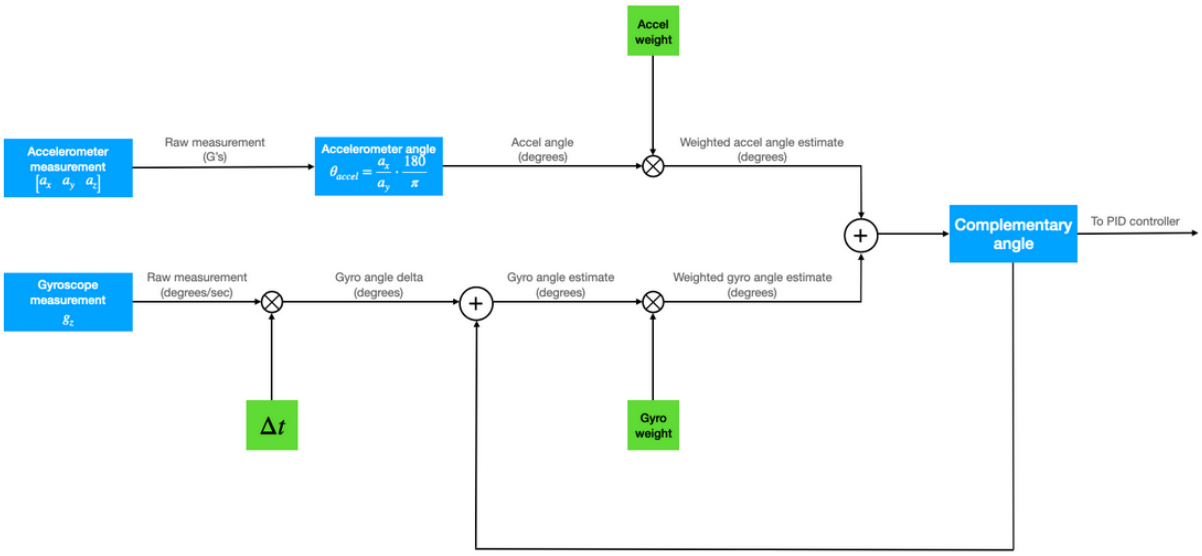


Figure 11. Complementary filter calculation flowchart (from the ECE 4760 website)

To implement the calculation flowchart shown in figure 11, we first needed to use *mpu6050_read_raw* to put data into acceleration and gyro buffers. Once we got this, we calculated the angle that the accelerometer gives using figure 12. We divided ax and ay or in the data buffer, acceleration[0] and [1]. We then multiplied this by 180/pi in order to get the accelerometer given angle in degrees.

$$\theta \approx \frac{a_x}{a_y}$$

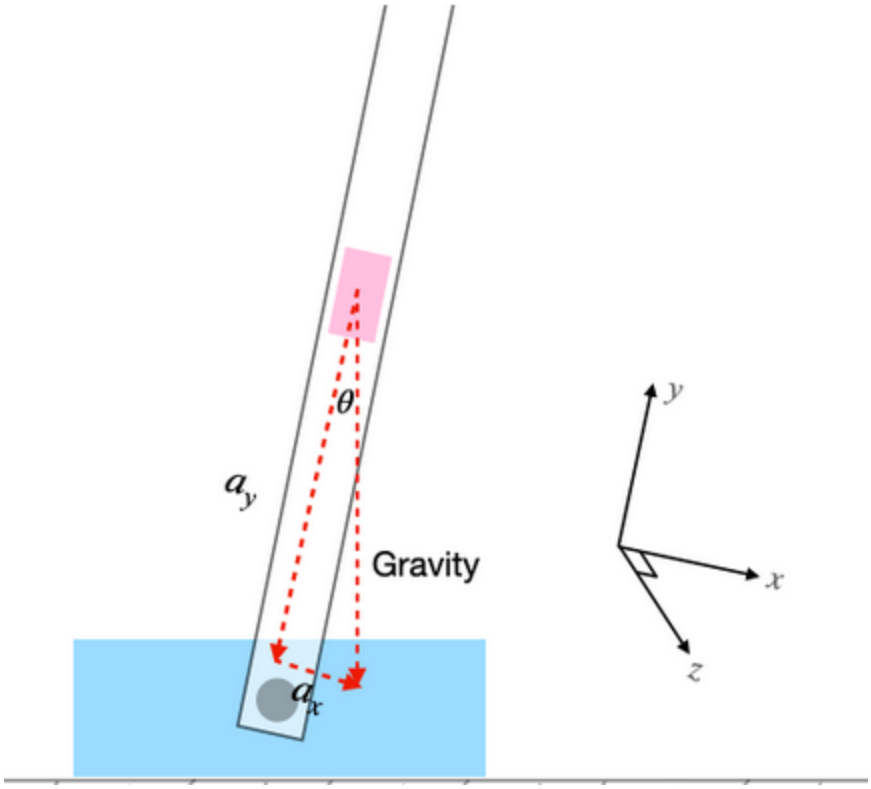


Figure 12. Acceleration calculation (from the ECE 4760 website)

As shown in the flowchart in figure 11, updating the complementary angle only requires the change in angle or gyro[2]*.001. Gyro[2] because this is the rotation rate and .001 is the rate at which this expression is evaluated since the IRQ handler runs at 1kHz. So rate*time = distance, so this gives the approximate change in angle. After getting the accelerometer calculated angle and gyroscope calculated change in angle, we did a weighted addition of the current complementary angle - the calculated change in angle given by the gyroscope and the accelerometer angle. The accelerometer was given a weight of about .001 and the gyroscope angle a weight of about .999 as this would reduce the fluctuations of the accelerometer while also preventing a complete unbiased measurement from the gyroscope thus low passing the accelerometer measurement while high passing the gyroscope measurement.

As can be seen, the complementary angle is calculated using the previous complementary angle. As previously stated, at the beginning of main, the accelerometer data is grabbed and the complementary is just given as the calculated accelerometer angle as this measurement is unbiased and gives a good enough approximation of the pendulum angle to start from.

An issue we had was that the filter didn't completely get rid of the bias of the gyroscope, so when the pendulum was vertical, the complementary angle would read about 2 degrees. To counteract this we subtracted a constant by the complementary angle until the vertical was about +-.02 degrees.

Controlling the Motor:

Controlling the motor and thus the wheel on the pendulum required modifying the duty cycle of the PWM output. If the duty cycle was higher on one input, the motor would spin faster as the duty cycle is equivalent to the average voltage of the output. Also to turn the motor one way or the other, one PWM input had to be low while the other was high since the H-bridge does not output anything when both inputs are high. In order to do this, we had a variable, duty_cycle. Duty_cycle would be the duty cycle we want the motor to spin at. Negative values are for spinning the motor the opposite direction. We would do this using the pwm_set_chan_level method. We would set channel A or B to the control value based on which direction we wanted the motor to go. The other channel would be set to 0 to not disrupt the H-bridge. We had to switch which channels went high for a given sign of the duty cycle as this changed based on which direction we wanted the motor to go. This would be updated in the on_pwm_wrap function as this would update the duty_cycle at 1kHz which was ideal.

PID Controller:

With the angle of the pendulum as well as control of the motor achieved, the best way to balance the pendulum would be to use a PID (proportion, integral, derivative controller). This controller would provide a consistent way to try to keep the complementary angle at 0 degrees (vertical) using proportionality, integration, and the derivative of the error between the current angle of the pendulum vs the desired angle of 0 degrees.

The proportional control tries to spin the motor in the direction that the pendulum is tilting in order to make the pendulum swing the opposite direction. This was done by defining a proportionality constant, Kp at a value of 1500. This would be multiplied by the error between the current pendulum angle vs the desired angle. The duty cycle of the motor would then be assigned to this error multiplied by Kp in order to provide an adequately large duty cycle for the motor to use to correct the pendulum tilt. We tested if the value should be positive or negative by testing the direction the wheel should spin in order to counteract tilt. We would set a Kp value and by holding the pendulum, if the wheel's direction of rotation made a force pulling the pendulum towards vertical, then the sign on Kp was correct. We found that the wheel should spin in the direction of tilt, so clockwise when tilting right. This used a positive Kp value, but if we switched the PWM channels, this value would have to be negative. Proportional control added a resistance to tilt, but it didn't spin the motor fast enough to balance.

The next and most effective control was integral control. To do this, an error_accumulation variable was created. This variable would be incremented by the difference between the current and ideal angle of the pendulum each PWM cycle. A variable, Ki (originally

value 8) was created. The duty cycle of the motor would then be the error_accumulation*Ki and then be added to the previously discussed proportional control to get a final duty cycle using proportional and integral control. The integral control would thus make the wheel spin according to how much the pendulum has changed in order to make the wheel spin more if the pendulum started to tilt more than usual. Another variable had to be created, Imax. This was the maximum value that error_accumulation could take as if this integral value was not capped. Then if the pendulum started to spin the opposite way after being tilted a certain way for a while, the integral value would be too large to change sign and the wheel would think that it's still tilting the wrong way and never correct.

The next addition was dithering. To do this, the ideal angle (usually 0 degrees) would be incremented or decremented by a value angle_increment (initially value 0.0001) every cycle to push the desired angle away from the current angle of the pendulum thus increasing the error and thus the corrective duty cycle of the motor. This was helpful in more quickly correcting the pendulum, but was slightly unstable as it was possible for the motor to overcorrect if angle_increment was increased too high. This value was enough to get the system to balance somewhat, but it was very unstable.

The final value that gave stability to the system was derivative control. This involved making a variable prev_error. This would be the error between the desired angle and current angle from the last PWM cycle. A variable, error_deriv, was then created as the difference between the error and prev_error. This would approximate the derivative of the angle error of the pendulum. This derivative would then be multiplied by a value Kd (initially 10000) and then added to the calculated duty cycle from the proportional and integral control, therefore finishing the PID controller. With the optimization explained later, this addition was able to get the pendulum to consistently balance even with outside pushes from even our feet.

All of these PID parameters were initially fix15 types from Lab 2. These were used in order to reduce the computation costs since these calculations were all done inside of an on_pwm_wrap. In an interrupt handler, calculations must be as fast as possible in order to not mess up timing. Duty_cycle was an integer in order to be used in the pwm_set_chan_level method, so all duty cycle calculations were converted using fix2int15. These calculations as well as the duty cycle update are all done in on_pwm_wrap to update at 1kHz.

Optimization:

Getting the pendulum balanced required changing all of the variables previously discussed in the PID controller to specific values so that the pendulum would correctly change the motor's duty cycle to counteract the tilt of the pendulum. To do this we started with proportional control and worked our way up to the derivative control looking at how each parameter changed the pendulum. First we optimized Kp. When the pendulum would tilt, it would simply not tilt as fast as it should have been for the amount it was tilting. To fix this, we raised Kp by 300 to 1800. Looking at Ki, we found that the pendulum correctly sped up after tilting for too long, so Ki was unchanged. One thing about integral control that was changed was

Imax. When testing the pendulum initially, the pendulum would correct itself once and then fall in the other direction because it had overcorrected. By lowering Imax to about 1500 from 2500, the pendulum was able to realize the direction it was going in more correctly and correct itself after overcorrecting. Two things were done with derivative control that immensely helped the system balance. Using the VGA screen (discussed later) we printed out the individual PID variable calculated duty cycles instead of just the sum of the three calculations. Looking at Kd*error_deriv, we saw that this value was negligible compared to other values. This was because of the approach we took. Since we were using fixed point calculations for complementary angle, and its various error calculations, Kd*error was really multfix15(Kd,error_deriv). Since Kd is a large value (10000), this value overflowed or something went wrong and resulted in a small fix15 value. Since the entire product was converted to an int, the fix was simply to make Kd an integer and convert error_deriv to an integer before multiplying the two numbers. This way, the value wouldn't overflow before being casted since an integer had 2 more bytes of space. Kd was also multiplied by 10 to make the contribution of the product as significant as the proportional and integral control. Using this fix, the pendulum stability increased tremendously as it could react to the speed at which the pendulum was falling thus giving a much more accurate motor reaction. Another change was multiplying angle_increment by 10 in order to speed up the rate of correction. This made the system more jittery, but it was better able to correct after being hit with a pencil.

User Interface:

Throughout the lab, we were able to control the inverted pendulum system through the user interface by changing the speed of the motor, angle, and PID parameters (Kp, Ki, and Kd values). We were able to observe the immediate effect of these changes after finishing setting up the user interface, and thus better understand how each parameter plays different roles in the system's behavior.

In week 1, we started with the most basic feature of the user interface: controlling the motor speed according to the input value. We restricted the range for the motor speed from -500 to +500 to safely drive the motor; moreover, the range was set to be big enough to see the dramatic changes when the input value's magnitude was big. We used *sprintf* to display the range of the motor speed that had been input by the user; the magnitude represented the speed at which the motor spins and the positive or negative sign indicated the direction. For example, a positive value with a large magnitude, such as 450, will make the reaction wheel turn clockwise at almost the maximum speed. Similarly, if the input value were -300, the wheel would rotate counterclockwise at a slower speed. This is also described later in the week 1 part of the **Displaying the VGA** section. We set the rotation directions in this way so that when the wheel is falling to one side, it would try to restore its stability by turning in the correct direction. When the input value is too low, around the range between -30 and 30, the wheel would stop turning since the voltage was too low to overcome the kinetic friction of the motor so the motor would simply make a stalling sound.

Throughout weeks 2 and 3, we added a feature to the user interface so that the user can change the PID controller parameters. The thread *protothread_serial(struct pt *pt)* takes in two static variables called *user_input and float_input.* The *user_input* variable takes in the letters or values corresponding to what parameters the user wants to change. We used the characters *a, b, c, d, and e* to change Kp, Ki, Kd, and the desired angle. After the user chooses which parameter to modify by inputting a character corresponding to the options displayed on the terminal, the variable *float_input* takes in user input as a float value for the chosen parameter, and the parameter would be set to the new value. Any integer inputs would simply be casted to an integer before being assigned to the variable. By changing the parameters through the user interface, we were able to observe the immediate effect of changing the PID parameters on the reaction wheel and the graphs on the VGA display as the wheel moved.

Displaying the VGA:

The VGA display was mainly for debugging purposes. Being able to see the changes for each value and how it is affecting the wheel was critical for testing and debugging. The changes in the motor speed and the PID controller parameters could be made through the user interface terminal, as explained above in the **User Interface** section. These changes had immediate effects on the VGA display, so we did not have to reboot our system every time we made a change. The VGA display showed two plots, one placed on top of another. The usage of the plots changed in each week of this lab, to make the testing easier.

We used many static variables that kept track of specific positions on the display, such as where to start drawing from and how the values would need to be updated. We included static variables called *xcoord, OldMotorRange, OldMotorMin, OldMotorMax, OldCompRange, OldCompMin, OldCompMax,* and *NewRange* to rescale the displayed data so that any change to the plots can be more visible to us. The "Old" prefix is just legacy from when we were printing out the demo code's "old_control" variable. Another static variable that we used was called *throttle*, which acted as a counter and controlled the rate for drawing. Then, we drew the plots starting at column 81 and included the y-axis labels of +/- 500 and +/- 9 (we manipulated these values as we rescaled the magnitudes) for the motor speed and tilt angle, respectively. In our code, there is a semaphore signal that indicates when the VGA can be updated; we decided to have this for the synchronization of objects. The drawing speed, *threshold,* was set to an integer value of 10 so it wouldn't get updated too frequently but rather shown as a smooth update on the plots. Once *throttle* exceeds or is equal to the value of *threshold*, the variable *throttle* would be reset and the drawing of the plot gets updated. We erased the previous columns by covering them with black pixels and drew new pixels for the updated values at the corresponding position on the screen. The position of each plot was updated pixelwise through the x-coordinate and y-coordinate. The x-coordinate position, *xcoord,* is the horizontal cursor that starts from an integer value of 81 and gets updated if *throttle* is greater than or equal to *threshold* and *xcoord* is less than 609*,* which is where the plot ends. *xcoord* would be reset to 81 if the line reaches the

end of the plot length. The y-coordinate of the line gets updated as the PID controller parameters change; this will be discussed more later in the weeks 2 and 3 section.

In week 1, the VGA display was used to show the speed and the actual tilt angle. The top plot was used to show the complementary angle in green, the gyro angle in red, and the accelerometer angle in white while the bottom plot was used to depict the speed of the motor/pendulum wheel in blue. The top plot has a range from -9 to +9 to indicate the angle degrees. The bottom plot has a range from 500 to -500 to indicate the motor speed. We can see the changes when the user modifies the speed. Let's say the user inputs +300, the bottom plot will show a line around the middle between 0 and 500. When the input is 0, it will bring the motor to stop and the line on the top plot would be zero as well. When we manually move the pendulum to the left or right, the top plot generates a sinusoidal shape depending on how fast or slow we move the pendulum from its position. Moving from one side to the other will bring the wave from positive to negative angle values and vice versa. The tilt angle was calculated based on the complementary filter shown above in figure 11. The tilt angles are constantly changing whenever we manually move the pendulum wheel, the tilt angle will change. This is shown by the complementary angle line which is depicted in the color green. The other two lines are for the accelerometer angle and gyro angle in color red and white, respectively. These values were read from the IMU. The accelerometer angle is the angle that is being pressured by gravity. The gyro angle measures the angle rotated within a certain time. These values will change depending on the parameters and the position of the pendulum wheel.

After being able to control the wheel and have the right setup for the pendulum, we were able to control the PID controller parameters. Throughout weeks 2 and 3, it depicts the same plots with additional arguments for debugging purposes. We added two lines on the screen to depict the angle value whether it is positive or negative value and the error accumulation. We were able to see when the wheel turned left or right the error accumulation values changed dramatically. In these two weeks, we mainly worked on the PID controller parameters which affect the duty cycle. We created four variables called *duty1, duty2, duty3,* and *duty_cycle*, to indicate the duty cycle for each PID parameter. *duty1, duty2, duty3,* and *duty_cycle* correspond to the duty cycle for Kp, Ki, Kd, and the sum of the duty cycles for each parameter. *Duty1* is the product of Kp with error, *duty2* is the product of Ki with *error_accumulation*, and *duty3* is the product of Kd and *error_deriv* (error derivative). The duty cycle changes when the values of the parameter changes and when the error changes depending on the position of the pendulum. (PID controller was explained above) With the value of the duty cycle, the speed changes correspondingly. Since duty cycle is affected by the PID parameters, it can change depending on the user input for the parameters. This affects the speed that the pendulum wheel is spinning at, since to update the speed is calculated by NewRange*(duty_cycle/10-OldMotorMin) /OldMotorRange. Subtracting 430 by that value gives the y-coordinate for the plot that shows the motor control. It is 430 due to the horizontal height that we set for the bottom plot. Also, the NewRange is a constant to have a better depiction on the VGA, and the calculation provides is to update the motor speed. Another example is the complementary angle gets updated in the same

way but it does not need to be multiplied by the duty cycle, since duty cycle only plays a role in the speed. The complementary angle gets updated by NewRange* ((complementary_angle - OldCompMin) / OldCompRange) and subtracting 230 by that value would be the y-coordinate of the line in a certain x-coordinate. The value 230 is the horizontal height set for the top plot. With the updates on the PID controller that affects the value of the duty cycle. The VGA display would update the new values to show how different parameters play a role in speed on certain angles when seeking for stability. With the values discussed above for the balanced point of the pendulum, the VGA display then only shows the motor control value and the complementary angle. Thus, depicting the line changes on the VGA display, it can clearly see the effects and debug to find the right value for Kp, Ki, and Kd to keep the pendulum stable at a certain point (which was discussed in the PID controller section).

Results:

Compared to the previous lab, where the results were displayed mainly in a quantitative way, the goal of this lab required both quantitative and qualitative approaches in analyzing our outcome. We used the complementary angle, constants Ki, Kp, and Kd, and calculations for error accumulation and duty cycle to quantitatively analyze and debug our system. We were also able to physically interact with the system through the mechanical setup; one of our testing methods included hitting the wheel with a pencil and observing if it could remain stable.

The VGA screen played an important role in testing and debugging our system, as it displayed the angles and speeds which were helpful to know if our reaction wheel was behaving as desired. While testing our system, we printed out the reference angle measured from the vertical position on the VGA display to make it easier to observe how tilted the wheel is. In addition to the angle, we also printed out the error accumulation. Both of them helped us reposition and balance the wheel when it fell to either side.

We spent most of our time trying to make our reaction wheel effectively exert torque on the inverted pendulum and stay balanced in the vertical position. We manipulated each K parameter, and changed the voltage to values within the range from 5 to 7V, to observe if these attempts affected the stability of our system in any way. After a few different troubleshooting, we found that setting the voltage to approximately 6.5V was the most effective for our system as it gave the motor enough torque to correct itself effectively, so we kept it throughout the rest of our testing process.

While testing, we often found that the reaction wheel wouldn’t stop moving even when it was placed at the vertical position, and it was inefficient to turn off and back on the entire power every time this problem occurred. When the wheel kept turning, it was harder for us to find the right vertical position to stabilize it because of the momentum created by the wheel. To fix the problem, we thought of a way to effectively reset the system that takes less effort and time, and decided to add a pushbutton to our circuit. The button acted as a reset button in the sense that it stopped the movement of the wheel by zeroing the error_accumulation variable and set the desired angle to the current pendulum angle no matter what angle it is currently at. With this

button, we were able to easily reset the system when the wheel would go out of control and keep spinning even when the wheel is at the vertical position; it allowed us to test our inverted pendulum at a much faster rate than before.

When we finally got our reaction wheel balanced, we realized that the changes on our graphs were hard to see, so we increased the magnitude of each value displayed on the graphs by multiplying each datapoint by a static float variable called *NewRange* with a value of 150.0 to make the changes more visible. We also performed more tests by changing the K values to make the system even more stable, but we soon found out that there was a certain range for the values in which the system works without failing, so we readjusted the values (in discussion of optimization) to Kp=1800, Ki=8, and Kd=10000.

As shown in figure 13, the top plot that shows the complementary angle fluctuated when an external force was applied; that was when we hit the reaction wheel with a pencil to test if it could remain balanced. However, after a short period of time, the plot quickly goes back to the angle near 0 degree, rather than tilting towards +/-9 degrees. This shows that our wheel was able to restore its vertical position and remain stable without falling to either side. The bottom plot in figure 14 shows the motor control discussed in the **User Interface** section of this report; it shows the range between -500 and +500 that indicates the direction of rotation and the speed at which the wheel is turning.

At first, our goal was to keep the wheel stay balanced for at least one minute, but we were surprised that our wheel was able to balance for a much longer time, at least a few hours possibly unless an external force was applied to it. Even if we applied an external force by hitting the wheel lightly with a pencil, it would restore its balance quickly, as long as the force was not too large. It could even stay up with light taps from a finger or even a foot. We were satisfied with our outcome, and took one step further to improve our system. The process for this improvement is described in the **Conclusion** section of this report.

In terms of error, the pendulum stayed between about -1 and 1 degrees from the vertical when idle, which isn't ideal, but was an acute enough angle to keep the pendulum steady. Obviously a perfectly vertical pendulum would be impossible, so this error range was good. In conclusion, the pendulum was very good at dealing with external forces, but lacked in a smooth stability that would make the pendulum look 'vertical'.

We think what sets our project apart is how we used dithering rather than getting rid of it. Many groups took away dithering since it was unstable, but we increased its effect on the pendulum to get a very reactionary wheel that could resist large external forces.

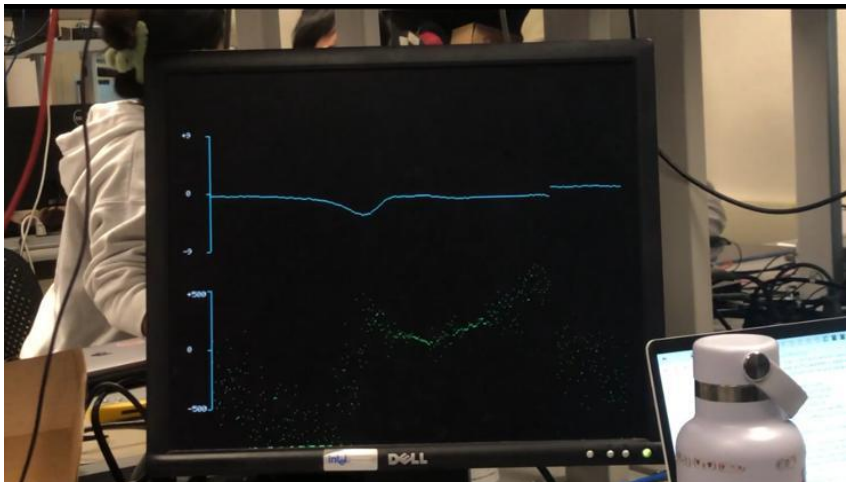


Figure 13. Depicting the complementary angle plot (top) and the motor control value plot (bottom) when the wheel is hit by a pencil (external force)

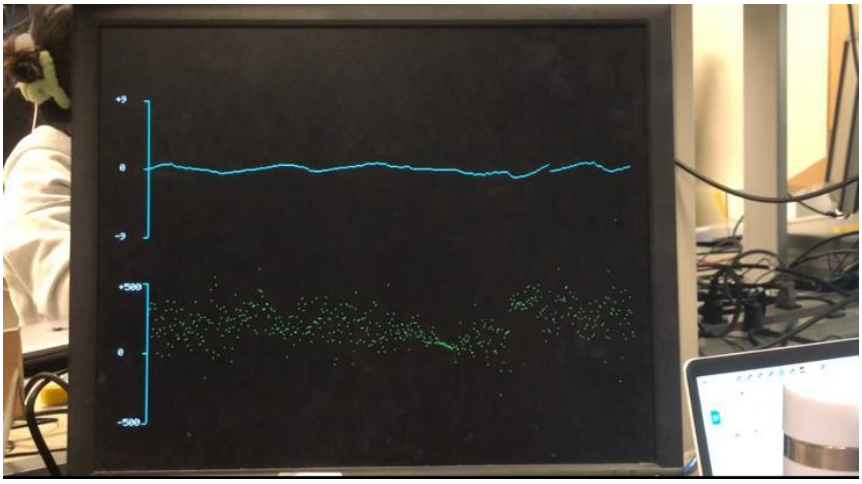


Figure 14. Depicting the complementary angle plot (top) and the motor control value plot (bottom) when the wheel is stabilized

Conclusion:

Compared to the two previous labs that we worked on throughout this semester, this lab was particularly unique because we worked on not only the ECE part–building the circuit and writing the code to operate the system–but also the mechanical construction that allowed us to interact with the system.

We also struggled comparatively less when debugging our system, as we have already been familiarized with building a circuit by following a schematic diagram, setting up the VGA screen and creating the user interface from the previous lab experience. With the experience, we were able to focus mainly on improving our system, rather than spending time on figuring out other problems.

Although this lab was shorter than the previous labs, we were able to accomplish our goal early, so we worked on adding more features to enhance the system. Our goal was to make the wheel to restore its vertical position on its own when it happens to fall to either side. To achieve the goal, we tried changing the K values, adjusted the tightness of the screw connecting the acrylic arm to the lego blocks, and added another wheel on the other side of the reaction wheel to add more weight. We spent the rest of the lab session trying to make the wheel bounce back at more extreme angles, but unfortunately, we were not able to finish adding this feature. We hope to successfully implement it in the future. We were also occupied with adding this feature as we did not implement the low pass filter of the *duty_cycle* measurement; the *duty_cycle* measurement can thus be seen fluctuating drastically on the VGA screen giving an unclear pattern.

Balancing the wheel was the biggest challenge, as it was the ultimate goal of this lab. We revised our choice for the K values multiple times until we observed the desired behavior. However, as we ran more tests, we were able to develop a sense of predicting how changing each parameter would affect the behavior, and the debugging process was done much faster towards the end of the lab.

One of the major issues that we faced was that the reaction wheel would constantly be detached from the motor. We spent a decent amount of time on re-engage the wheel back to the motor throughout the lab sessions, and switched out the wheel four times. In the last lab session, we decided to replace the motor itself, as other groups seemed to have no problem with their wheels, and the wheels were all identically 3D-printed. After we replaced the motor, the wheel stayed on the motor in a more stable way, and we were able to focus on testing without worrying about the wheel flying off and hitting one of us.

Overall, we believe that we accomplished all our tasks in this lab without falling behind. Our result was much more stable than we had originally expected it to be, and we gained a better understanding of the digital filtering algorithms, PID controller, and mechanism of an inverted pendulum. With this experience, we hope to apply our knowledge to other projects in the future.

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
 *   - GPIO 4 ---> PWM1 output
 *   - GPIO 5 ---> PWM2 output
 */


// 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"

#define RST_BTN 2
// Arrays in which raw measurements will be stored
fix15 acceleration[3], gyro[3];
//PID variables
```

```
fix15 error_deriv; //derivative of pendulum angle
int duty_cycle; //duty cycle of motor
fix15 duty1; // Proportional duty cycle
fix15 duty2; // Integral duty cycle
fix15 duty3; // Derivative duty cycle
fix15 error; // error between angle and desired angle
fix15 accel_angle; // accelerometer calculated angle
fix15 gyro_angle_delta; // gyroscop calculated change in angle
fix15 complementary_angle; // corrected angle using accelerometer and gyroscope
fix15 desired_angle = int2fix15(0); //desired angle for pendulum to hover around 0 = vertical
int Kp = 1800;  //Proportional gain constant. Integer to avoid overflow
fix15 Ki = float2fix15(8); //integral gain constant
int Kd = 100000; //derivative gain constant. Integer to avoid overflow
fix15 angle_increment = float2fix15(.001); //rate at which desired angle changes for dithering
fix15 error_accumulation = 0; // amount of error accumulated
fix15 prev_error = 0;
fix15 Imax = int2fix15(1500); //clamp value for error_accumulation
// 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 ;
// PWM wrap value and clock divide value
// For a CPU rate of 125 MHz, this gives
// a PWM frequency of 1 kHz.
// 5MHz / 5000 cycles/pwm-wrap = 1KHz
#define WRAPVAL 5000
// 125MHz / 25 = 5MHz
#define CLKDIV  25.0f
// Variable to hold PWM slice number
uint slice_num ;

// PWM 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);
        // Accelerometer angle (degrees - 15.16 fixed point)
        accel_angle = multfix15(divfix(acceleration[0], acceleration[1]), oneeightyoverpi) ;

        // Gyro angle delta (measurement times timestep) (15.16 fixed point)
        gyro_angle_delta = multfix15(gyro[2], zeropt001) ;

        // Complementary angle (degrees - 15.16 fixed point) + constant to correct the vertical to
0 degrees
        complementary_angle = multfix15(complementary_angle - gyro_angle_delta, zeropt999)
+ multfix15(accel_angle, zeropt001) - float2fix15(.006) ;
        //reset button to reset desired angle and error accumulation when reseting position of
pendulum
        if(gpio_get(RST_BTN)){
        desired_angle = complementary_angle;
        error_accumulation = 0;
        }
        // Compute the error
        error = desired_angle - complementary_angle  ;
        //Dithering
        if (error < 0) {
        desired_angle -= angle_increment ;
        }
        else {
        desired_angle += angle_increment ;
        }

        // Integrate the error
        error_accumulation += error ;

        // Clamp the integrated error (start with Imax = max_duty_cycle/2)
```

```
        if (error_accumulation>Imax) error_accumulation=Imax ;
        if (error_accumulation<(-Imax)) error_accumulation=-Imax ;
        //comput error derivative
        error_deriv = error - prev_error;
        // Compute duty cycle with PI controller
        //proportional control
        duty1 = Kp * fix2float15(error);
        //integral control
        duty2 = fix2int15(multfix15(Ki,error_accumulation));
        //derivative control using integer multiplication to avoid overflow
        duty3 = Kd * fix2float15(error_deriv);
        //total control
        duty_cycle =  duty1 + duty2 + duty3 ;
        //assign previous error for calculating derivative
        prev_error = error ;
        //minimum duty cycle if needed
        #define duty_min 0
        //change motor duty cycle to controlled value
        if(duty_cycle < 0 && abs(duty_cycle) > duty_min){
        pwm_set_chan_level(slice_num, PWM_CHAN_B, duty_cycle);
        pwm_set_chan_level(slice_num, PWM_CHAN_A, 0);
        }
        //reverse direction if needed
        else if(abs(duty_cycle) > duty_min){
        pwm_set_chan_level(slice_num, PWM_CHAN_A, duty_cycle);
        pwm_set_chan_level(slice_num, PWM_CHAN_B, 0);
        }
        // Signal VGA to draw
        PT_SEM_SIGNAL(pt, &vga_semaphore);
}

// 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
```

```
//motor display corrective values
static float OldMotorRange = 1000;
static float OldMotorMin = -500;
static float OldMotorMax = 500;
//complementary angle corrective values
static float OldCompRange = 18;
static float OldCompMin = -9;
static float OldCompMax = 9;
static float NewRange = 150. ; // (looks nice on VGA)

// Control rate of drawing
static int throttle ;

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

// Draw bottom plot
drawHLine(75, 430, 5, CYAN) ;
drawHLine(75, 355, 5, CYAN) ;
drawHLine(75, 280, 5, CYAN) ;
drawVLine(80, 280, 150, CYAN) ;
sprintf(screentext, "0") ;
setCursor(50, 350) ;
writeString(screentext) ;
sprintf(screentext, "+%d", (int)OldMotorMax) ;
setCursor(50, 280) ;
writeString(screentext) ;
sprintf(screentext, "%d", (int)OldMotorMin) ;
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") ;
setCursor(50, 150) ;
writeString(screentext) ;
```

```
        sprintf(screentext, "+%d", (int)OldCompMax) ;
        setCursor(45, 75) ;
        writeString(screentext) ;
        sprintf(screentext, "%d", (int)OldCompMin) ;
        setCursor(45, 225) ;
        writeString(screentext) ;



        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) ;

        // Draw bottom plot (duty cycle of motor) (scale from -500 to 500 )
        drawPixel(xcoord, 430 -
(int)(NewRange*((float)(duty_cycle/10-OldMotorMin)/OldMotorRange)), GREEN) ;
        // Draw top plot (complementary angle -9 - 9 degrees)
        drawPixel(xcoord, 230 -
(int)(NewRange*((float)((fix2float15(complementary_angle))-OldCompMin)/OldCompRange)),
CYAN) ;
        // Update horizontal cursor
        if (xcoord < 609) {
                xcoord += 1 ;
        }
        else {
                xcoord = 81 ;
        }
        }
        }
        // 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 int user_input ;
        static float float_input ;
        while(1) {
        sprintf(pt_serial_out_buffer, "\na: change Kp\nb: change ki\nc: change kd\nd: desired
angle:");
        // non-blocking write
        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", &user_input) ;
        user_input = (char)user_input;
        if(user_input == 'a'){
        sprintf(pt_serial_out_buffer, "input a value for Kp, any integer number (current value
%d): ",Kp);
        serial_write;
        serial_read;
        sscanf(pt_serial_in_buffer,"%f", &float_input) ;
        //reassign Kp value
        Kp = (int)float_input;
        }
        if(user_input == 'b'){
        sprintf(pt_serial_out_buffer, "input a value for Ki, any decimal or integer number (current
value %f): ",fix2float15(Ki));
        serial_write;
        serial_read;
        sscanf(pt_serial_in_buffer,"%f", &float_input) ;
        //reassign Ki value
        Ki = float2fix15(float_input);
        }
        if(user_input == 'c'){
        sprintf(pt_serial_out_buffer, "input a value for Kd, any integer number (current value
%d): ",Kd);
        serial_write;
```

```
        serial_read;
        sscanf(pt_serial_in_buffer,"%f", &float_input) ;
        //reassign Kd value
        Kd = (int)(float_input);
        }
        if(user_input == 'd'){
        sprintf(pt_serial_out_buffer, "input a value for desired angle, any decimal or integer
number (current value %f): ",fix2float15(desired_angle));
        serial_write;
        serial_read;
        sscanf(pt_serial_in_buffer,"%f", &float_input) ;
        //reassign desired angle value
        desired_angle = float2fix15(float_input);
        }
        }
        PT_END(pt) ;
}

// Entry point for core 1
void core1_entry() {
        pt_add_thread(protothread_vga) ;
        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) ;
        //set reset button as gpio in
        gpio_init(RST_BTN);
        gpio_set_dir(RST_BTN, GPIO_IN);
```

```
        // MPU6050 initialization
        mpu6050_reset();
        //get acceleration measurement
        mpu6050_read_raw(acceleration, gyro);

        // set initial complementary_angle as current accelerometer angle
        complementary_angle = multfix15(divfix(acceleration[0], acceleration[1]),
oneeightyoverpi) ;
        ////////////////////////////////////////////////////////////////////
        ////////////////////// 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_schedule_start ;

}
```