Embedded Control: Lab 5

Bench-Top Testing of Blimp Gondola Hardware and Software

Gabriel Medina, Emily Nelson, Paul Sicard & Adam Steinberger
ENGR 2350 – Section 3
May 4, 2010
Grading TA: Alexey Gutin

Table of Contents

Introduction

The following is an outline of the hardware and software employed in Autonomous Blimp and Smart Car systems. It represents a culmination of the knowledge and programming skills acquired in the Embedded Control course. The gondola code employs various functions on the C8051 microcontroller to control the heading and altitude of the blimp. Through experimentation, proportional and derivative gains for the rudder fan and thrust fans have been optimized to achieve the desired heading and altitude efficiently.

The purpose of the *Blimp Gondola Project* is to develop code that will allow a Smart Car or Blimp to maintain a user-defined altitude and heading. The smart car and blimp employ similar hardware and software to function so the software from the car has been modified to control the blimp. The C8051 microcontroller communicates with the compass and ranger peripheral devices through the SMBus to determine the direction and speed/altitude of the car/blimp. Using the programmable counter array and interrupt service routines a pulse width modulated signal is sent to the servo and dc motors. In the blimp code, this signal is controlled using proportional and derivative algorithms. Analog to digital conversion is used to monitor the battery voltage. The hardware components and software along with the results of various derivative and proportional gains will be discussed in detail in the following sections.

System Description and Development

Hardware Description

The hardware for the gondola system consists of different components including switches, resistors, input devices, and output devices. There are switches wired to control the thrust fans, the rudder fan, and one to clear the LCD screen, as well as eight DIP switches for use. The input devices are the Electronic Compass, and the Ultrasonic ranger. The output devices are a speed controller, a DC motor, and a servo motor. There is also a serial bus, and buffers used in the gondola. All of the components can be seen in the circuit diagram in Appendix A: Hardware, figure 1. The wiring in the gondola is set up similarly to that of the smart car.

The speed controller is wired to a twelve volt source, while the compass, the ranger, and the key pad are all wired to a five volt source. The speed controller and servo motor are outputs based on the cross bar initialization. A 74F365 buffer chip is wired as an interface between the EVB connector and the speed controller, DC motor and servo motor. The compass is wired to a calibration switch, and there is a drive switch and a steering switch. The drive and steering switch are connected from five volts through a resistor to specified ports based on the cross bar initialization. The other end of the switch is also grounded to turn the fans off when the switches are flipped. The calibration switch of the compass is connected to ground. The seven volt battery is connected through two resistors, one 22 k ohm and one 6.8 k ohm which is connected to ground.

The electronic compass is used to control the steering of the smart car and the heading of the blimp. The compass uses a magnetic field detector to detect the current direction with respect to magnetic north. It is wired to a five volt power source in pin one, and a ground at pin nine. It is also wired to the SCL and SDA in pins two and three, respectively. Pin six is used to wire the calibration switch. The compass must be connected to the EVB through a buffer. The compass needs the buffer to act as a current source since the compass typically draws 10mA. It however does not need any voltage amplification. The compass is used to control the steering of the gondola system.

The servo motor is used to adjust steering in the smart car and the thrust fan angle on the gondola. For the gondola it is connected through a buffer to port 0 pin 5 of the EVB. The servo motor consists of a potentiometer, a comparator, and a clock. This internal circuitry allows the motor to correctly function and vary depending on the pulse width of the driving signal, in this case the signal from the electric compass. The driving signals' pulse width is determined by the amount of time the signal is high during the period of the signal. There is a potentiometer is connected to vary the pulse with of the internal clock. The servo motor turns the wheels to the left if the comparator returns a value of the driving signal being less than that of the clock. If the signal retuned indicates that the driving signal is greater than that of the clock the wheels will turn to the right. The potentiometer is connected to the rotating shaft of the servo motor. As this rotates the potentiometer varies in turn varying the pulse width of the clock. The rotation will continue until the potentiometer has adjusted the internal clock pulse width to be equal to the driving pulse width as determined by the comparator.

The altitude of the blimp will be controlled by a two thrust fans. The heading is controlled by a third rudder fan. These three fans are controlled by DC motors. The DC motor is connected to a twelve volt source through a speed controller. The EVB outputs a pulse width modulated signal to the motors. The pulse width duty cycle is utilized to determine the power applied to the drive motor. This variation of power controls the speed at which the motor turns. The pulse width can be used if a variable voltage source is not available. The internal components of the motor include an inductor in series with some resistance. This acts as a low pass filter which removes the frequency spikes allowing an average input signal to the motor. The output voltage is essentially controlled by the duty cycle, which is the time the signal is high divided by the total period. As the duty cycle increases towards 100% the power also increases to a maximum at 100%. In order to get the motor to turn at a certain voltage the pulse width must be mapped to a corresponding voltage.

The DC motors are all connected to a speed controller in order to have the output mapped to a voltage. The speed controller receives an input from the EVB and maps it to a duty cycle which will allow the DC motor to have the correct voltage. This process allows for the variation in spin speed on the DC motor. The motor must have a second to warm up and be set in neutral so there is no output. The pulse width is refreshed every 20 milliseconds. If it is not refreshed enough or refreshed too often, it will confuse the program. The DC motor is used to control the speed at which each fan spins. This is used to adjust the altitude and have the rudder fan spin, which will allow heading adjustment.

In order to know where the blimp is, the Ultrasonic Ranger sensor must be used. The ranger sends out a ping of high frequency sound waves. The waves reflect off the closest object in their path back to the ranger. Once the echo is received the ranger calculates the distance based upon time. The ranger is wired to a five volt source, ground, and to the SCL and SDA lines. The distance from the ranger is used to calculate the pulse width sent to the speed controller. In order to ensure that the close proximity of the ranger to the compass does not affect reading returned from the compass, a 100 micro farad capacitor is wired from the power to te ground pin connections of the ranger. The ranger and the compass are both considered outputs from the EVB.

The LCD display is an input which allows users to interface with the gondola's settings. The LCD screen also includes a key pad where users are prompted to configure the gondola's set up. The display has a total of four lines, each 20 characters long. The LCD and keypad are connected to power, ground, the SDA and SCL lines in pins one, four, two, and three, respectively.

In order for the car and blimp to be able to maintain a specified heading and direction/altitude it is necessary for the microprocessor to interface with peripheral devices, namely the compass and ranger. This was accomplished using serial communication through the System Management Bus. Because serial ports transmit one bit of data at a time along a single wire it is necessary for the master and slaves to be synchronized so that the receiver knows how to interpret the data being transferred. This synchronization is accomplished by sending the data with a clock signal along the SCL line. As shown in Appendix A: Hardware,

Figure 2, the SCL clock is set to 100 kHz. The SDA and SCL lines are pulled high using pull-up resistors because while the master can pull a line high, not all slaves can pull a line high but all can ground the line.

For the smart car and blimp, the master (C8051) reads data from the slave devices (compass and ranger). To read data from a slave the master must first send a start signal and then send the 7 bit slave address with the read/write bit set high to indicate that the master is going to read from the slave. In addition to declaring the slave address the master must also indicate which register to start at. The slave acknowledges the master and sends a byte of data. When the master has received the byte it sends an acknowledgement signal and the slave writes the next byte of data. This continues until the master sends a read and stop signal which informs the slave to send the last byte of data.

For the blimp the crossbar (CEX0) was set to 0x25. This enables UART0EN, SMB, and CEX0 through CEX3. According to the crossbar priority of the C8051, UART0 uses port 0 pins 0 and 1 while the serial communication (SDA and SCL) uses port 0 pins 2 and 3. The SDA and SCL pin connections were initialized to high impedance and set to open drain output mode to allow communication between the C8051 and the compass and ranger. Because UART1EN is not used in this lab, CEX0 is assigned to port 0 pin 4. Port 0 pins 4-7 were set to push-pull output mode to send pulse width modulated signals to the dc and servo motors. CEX0 was used to control the rudder fan, CEX1 was used for the servo motor which controls the angle of the thrust fans and CEX2 and 3 control the left and right thrust fans respectively. Both the smart car and gondola hardware use a 74F365 Hex Buffer to protect the C8051 from power surges as well

as allowing the engineer to treat the protoboard circuit independent of the internal circuitry of the C8051. This is reflected in the circuit schematic in Appendix A: Hardware, Figure 1.

To read the battery voltage port 0 pin 3 was used; it was set to analog input mode and set to high impedance so that voltage could be read by the microprocessor. Because the voltage of the battery is decreased using a 22 kΩ and 6.8kΩ resistor, the digital result of the ADC conversion must be multiplied by a factor to calculate the current voltage of the battery. In addition to the required components, three switches on the blimp were used for clearing the LCD screen, setting the thrust fans to neutral and turning on/off the rudder fan. The switches were set to open drain input mode and set to high impedance. These switches were controlled by the microprocessor using port 3 pin0 (LCD), port 3 pin 65 (thrust fans) and port 3 pin 7 (rudder fans). The use of these switches are extremely helpful in running the program because it allows the user to choose different proportional and derivative gains from the LCD screen while the blimp is stationary. It is also a useful debugging tool during the program testing.

Software Description

The software for the Blimp Gondola Project is written in the C language, and controls the operations of several fans installed on the Blimp Gondola. Besides hardware initializations and prototype functions, the software operates in five main blocks. These blocks are for warming up the motors, configuring the Blimp Gondola, driving the motors, displaying the Blimp Gondola statistics, and checking the battery voltage. The C8051 embedded controller is configured in the Programmable Counter Array 0 (PCA0) Initialization to use SYSCLK/12, which

operates at 1.843200 MHz. The peripherals used by the software all need to operate at the same frequency, so the System Management Bus (SMBus) Clock (SCL) is set to 100 kHz. The PCA0 Interrupt Subroutine (ISR) sets the PCA0 start count at 28672, which means that PCA0 will overflow once approximately every 20 ms. The PCA0 ISR is setup this way because the DC and Servo Motors used by the Blimp Gondola need to be driven once every 20 ms. After the motors have been warmed up and the Blimp Gondola has been configured, the statistics need to be printed to the LCD Display and the SecureCRT screen once every second. Also the Electronic Compass needs to be read about once every 40 ms, and the Ultrasonic Ranger needs to be read once every 80 ms. PCA0 overflow counters PCA_count, h_count and r_count are used to regulate this. If the Blimp Gondola's battery voltage is too low, the embedded controller will lose control of the Blimp. The software Analog/Digital Converter 1 monitors the battery voltage to make sure it isn't too low, and if it is the fans will be directed to steer the Blimp downwards towards a safe landing.

After the Blimp Gondola's hardware is initialized, the DC Motors must be set to neutral and warmed up for a second. To do this, the rudder and thrust pulse widths are set to PW_NEUT (2750), which are used to set the Capture/Compare Modules (CCM) associated with each motor. The CCMs are used to compare the digital values stored within them to the PCA counter value. For the Blimp Gondola Project, CCM0 through CCM3 are configured as 16-bit counters, which means that PCA0 will count from 0 to 65535, overflow and then start again from 0. By setting the PCA0 start count to 28672 in the PCA0 ISR, the counter will start counting again at 28672 after each overflow. The CCM values are stored so that each I/O line CEX0 through CEX3 outputs a digital 0 before the PCA0 count reaches $65535 - \mathrm{CCM}n$, and then

outputs a digital 1 until the PCA0 overflows at 65535. This generates a pulse train whose duty cycle is $\mathrm{CCM}n/36863$. The duty cycle for PW_NEUT is 7.5%, so the CEX*n* I/O lines will output a digital 1 only 7.5% of the time. Since SYSCLK/12 operates at 1.843200 MHz, PCA0 will count at this frequency. With a PCA start count of 28672, PCA0 overflow rate is $(1\ \mathrm{sec}/1843200\ \mathrm{counts}) \cdot (36863\ \mathrm{counts}/1\ \mathrm{overflow}) \approx 0.020\ \mathrm{sec/overflow}$. With a duty cycle of 7.5%, the CEX*n* I/O lines will output a digital 1 for approximately 1.49 ms every 20 ms. The DC Motor Speed Controllers will have no output when the input pulse width is 1.5 ms, and a pulse width of 1.49 ms should be close enough to start the motors so they are not spinning. (The hardware is configured so that CEX0 drives the rudder fan DC Motor used to control the Blimp Gondola's heading, CEX1 drives the thrust fan Servo Motor used to control thrust angle, CEX2 drives the left thrust fan DC Motor, and CEX3 drives the right thrust fan DC Motor.) To wait 1000 ms for the motors to warm up, the software enters a while loop that exits once $\mathrm{PCA\_count} \geq 50$. This loop contains a blank printf statement, which is used because printing text to a computer screen usually takes a considerable amount of time. Additionally, the LCD screen and keypad need some time to initialize, so a delay time is set at 100,000 before the program continues onto the second block of code.

Now that everything has been warmed up, the user must configure the Blimp Gondola. This routine is completed before the fans will start spinning, and can also be run during flight when the user presses the 0 key on the keypad. The LCD screen and keypad communicate with the C8051 embedded controller via the Serial Data (SDA) line. Printing text to the LCD screen is very similar to printing text to the SecureCRT screen, except the lcd_print command is used and the carriage return character is used by itself to start a new line. Also, the lcd_clear() command

is used to clear the LCD screen. The Blimp Configuration Menu will clear the screen, print instructions for modifying the Blimp Gondola, store a random invalid character as keypad input to clear any previous presses stored in the variable, and then wait for user input from the keypad. The keypad is read using the read_keypad() command. Because the keypad operates at a baud rate of 9,600 bits/sec, which is must faster than it takes for a human to press a single key on the keypad, the embedded controller must wait for some time before each time it checks for a valid input. The Menu asks for the user to adjust the thrust fan angle, set a desired heading, choose rudder fan proportional and derivative gains, and choose thrust fan proportional and derivative gains. Data is sent from the keypad to the C8051 as ASCII characters, so the Blimp Configuration Menu simply waits for input as one of the characters displayed in the instructions. The wait function loops for 25534 counts, so it will wait for about 13.9 ms.

The next block of code drives the rudder and thrust motors once every 20 ms. Rudder pulse width is controlled by readings from an Electronic Compass, which are returned in tenths of a degree between 0 and 3599. Thrust pulse width is controlled by readings from an Ultrasonic Ranger, which are returned in centimeters. Readings from the compass need to be taken once every 40 ms, and readings from the ranger need to be taken once every 80 ms. The PCA0 ISR uses counters PCA_count, h_count and r_count to keep track of PCA0 overflows for different operations that need to be timed like taking readings from the two sensors. When h_count is 2, the new_heading flag is set to 1 and h_count is reset to 0. When r_count is 4, the new_range flag is set to 1 and r_count is reset to 0. The new_heading and new_range flags tell the C8051 to take new readings from the sensors. After each reading is taken, the new reading

flag is then reset to 0. The drivers for the rudder and thrust fans use proportional and derivative gains, so error values must be found to determine the thrust or steering required for the Blimp to reach the desired heading or altitude. For heading error, corrections need to be made so the error value is between ±180°. Proportional and Derivative Control algorithms are used for all three DC Motors, which take the form of PW_NEUT + K_P*error + K_D*(error - previous error). This control algorithm becomes the pulse width that is sent via CEX*n* to the motors. The current error is then stored as the previous error for the next time the sensors are read. Corrections are then made to prevent pulse widths outside of the PW_MIN (2000) to PW_MAX (3500) range from reaching the motors. Pulse widths outside of this range may damage the motors. DC Motor controls also set the pulse widths to neutral if STEER_SWITCH or THRUST_SWITCH are off. If the battery voltage is below 5%, the rudder pulse width will be set to neutral and the thrust pulse width will be set to PW_MIN. The Blimp is filled with helium, which means that if the computer loses control the Blimp will float up to the ceiling. The thrust motor control system sets the pulse width to PW_MIN to drive the Blimp down to the floor before the computer loses control. Lastly, the proper CCM lines are set equal to the new pulse widths for all three motors.

Taking readings from the Electronic Compass and the Ultrasonic Ranger requires the Inter IC bus (I^2C). Each sensor has a unique address that can be used by an embedded controller to communicate back and forth between the sensors. The i2c_read_data function is used to locate the sensor and read a specific number of bytes from it, starting at some desired register. Peripherals may have multiple registers that are used for controlling different parts of the device. Once data has been read from the device, the software will combine each byte and then

return this value as either the heading or the range. An added step for the Ultrasonic Ranger is to start a new data ping request and write this to the ranger. The ranger will receive this ping and will return the range 80 ms later. This means that the first reading from the ranger will be wrong, but all readings thereafter should be accurate (ignoring electromagnetic interference).

The Blimp Gondola statistics are displayed on the LCD screen and the SecureCRT screen once every 1000 ms. To time this correctly, the software waits until PCA_count = 50 to display text on the screens. Percent battery voltage, heading and range are all displayed on the LCD screen. If the battery voltage is less than 10%, the screen will display “WARNING: BATTERY LOW”. Otherwise, it will instruct the user how to enter the Blimp Configuration Menu. The PCA_count is then reset to 0. Also heading, desired heading, heading error, range, desired range, range error and battery voltage are all displayed on the SecureCRT screen.

The last block of code for the Blimp Gondola Project checks the battery voltage left. The Analog/Digital Converter 1 (ADC1) is used to obtain this data. Resistors are used in the hardware for the Blimp Gondola to lower the voltage by a calculated amount so that the voltage entering ADC1 is at or below the reference voltage (V_{REF}) of 2.4 V multiplied by the gain set in the ADC initialization. ADC1 then compares this voltage to V_{REF} and converts it to a one byte integer between 0 and 255. If the battery voltage is 5% or less, the rudder pulse width will be set to neutral and the thrust pulse width will be set to drive the blimp downwards. These pulse widths are sent to the three DC Motors, and then the program ends. This is a safety net because the C8051 does not respond well to low battery voltage and may not control the Blimp correctly. Most likely, the Blimp will have landed before the battery voltage reaches 5% or less.

A few additional notes about the software used for the Blimp Gondola Project. To setup the different Port Pins for digital or analog input or output, bit-masking is required. To configure a Digital Input pin as Open-Drain mode, the pin must be set to zero using P*n*MDOUT. To then set the pin to High Impedance mode, the pin must be set to one using P*n*. Setting Digital Output pins to one using P*n*MDOUT will configure them to Push-Pull mode. For Analog Input, the pin must be set to zero using P*n*MDIN. Bit-masking can be performing using bitwise operations to mask bits in an 8-bit port that are not to be changed while changing other bits. Also, the software configures a CLEAR_SCREEN slide switch to clear the LCD screen when enabled. This feature is useful because turning off the Blimp Gondola while running code sometimes causes the LCD screen to get stuck outputting the information it last displayed. Lastly, several variables used by the software are declared as xdata. Due to the large number of variables used for the Blimp Gondola Project, the RAM required to run the code exceeds the 256 byte limit. Therefore, it is necessary to store a few of these variables in external memory.

The complete code can be found in Appendix B. For a logical guide through the software operation, please see the flowchart found on the next page.

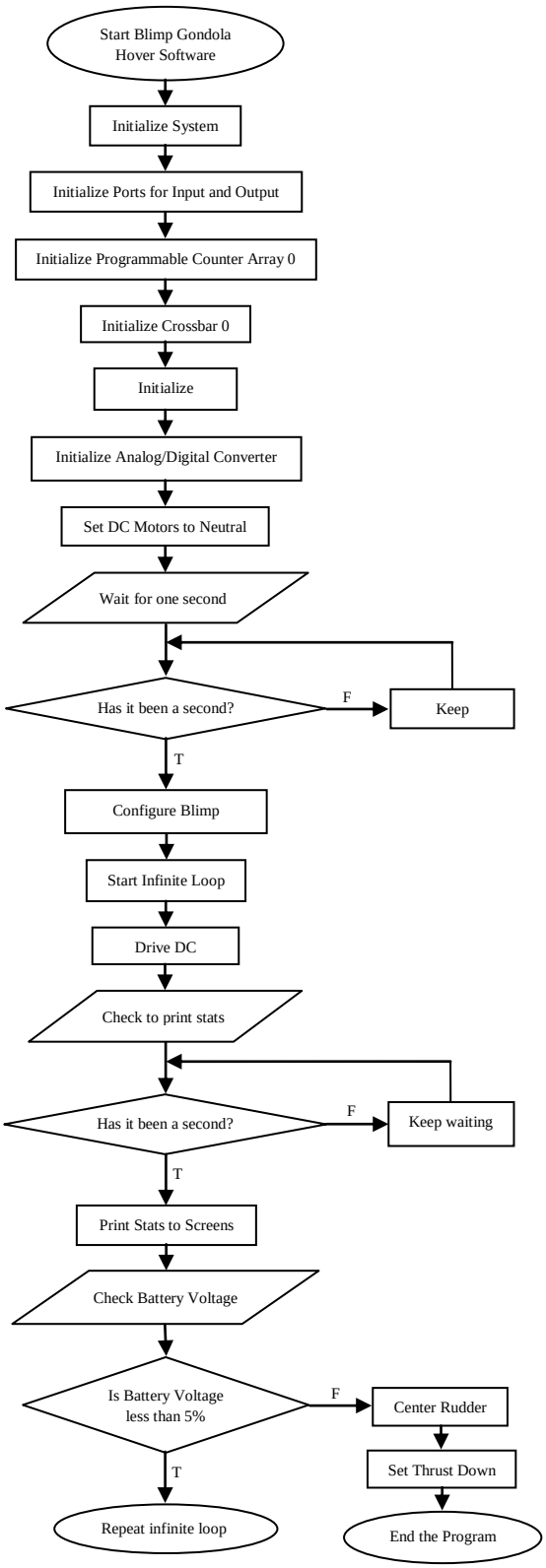


Figure 1 - Logical Software Operation Flowchart

Results and Conclusion

The Blimp Gondola Project modified the code used in the smart car to control the steering and altitude for the blimp. The gondola operates through the execution of six main steps i.e.: 1. initialization of all system parts 2. motor warm up, 3. blimp configuration 4. motor operation 5. stat display 6. battery voltage display.

Configuration of the blimp is done by the user through the use of an LCD control pad. Major subsystems include a steering servo control function, speed control function, and LCD display function.

Proper control of the steering servo and speed controller was necessary for the gondola to achieve the desired heading and the desired altitude quickly and efficiently. This was accomplished by implementing proportional and derivative control into both motor control functions. The derivative gain provides damping and allows larger proportional gains to be used for a quicker control response. Since the blimp is not naturally damped, this method was best suited to solve the control problems.

To test this modified control method, the complete code was uploaded to the gondola. Several gains for steering were then tested. A desired direction of magnetic North, equal to a desired heading of 360°, and desired altitude of 50 cm were set using the LCD keypad. To find the desired proportional and derivative gains, k_P and k_D, for steering control, six different cases were tested. These test cases can be seen in Appendix B: Results, Table 1. The graphical data for each test can be found in the Appendix C. Test Case 4 from Table 1 was found to use the most desirable gains.

After the desired gains for steering control were found, they were compared with three other case tests. One case only implemented a proportional control. This was accomplished by setting the derivative gain to zero. Another case used poor control gains. The last case used almost desirable control gains. These cases were tested with the same desired altitude and a desired direction of magnetic South, which is equivalent to a desired heading of 180°. Compiled graphical data is shown below, individual graphs for each case are found in Appendix C.

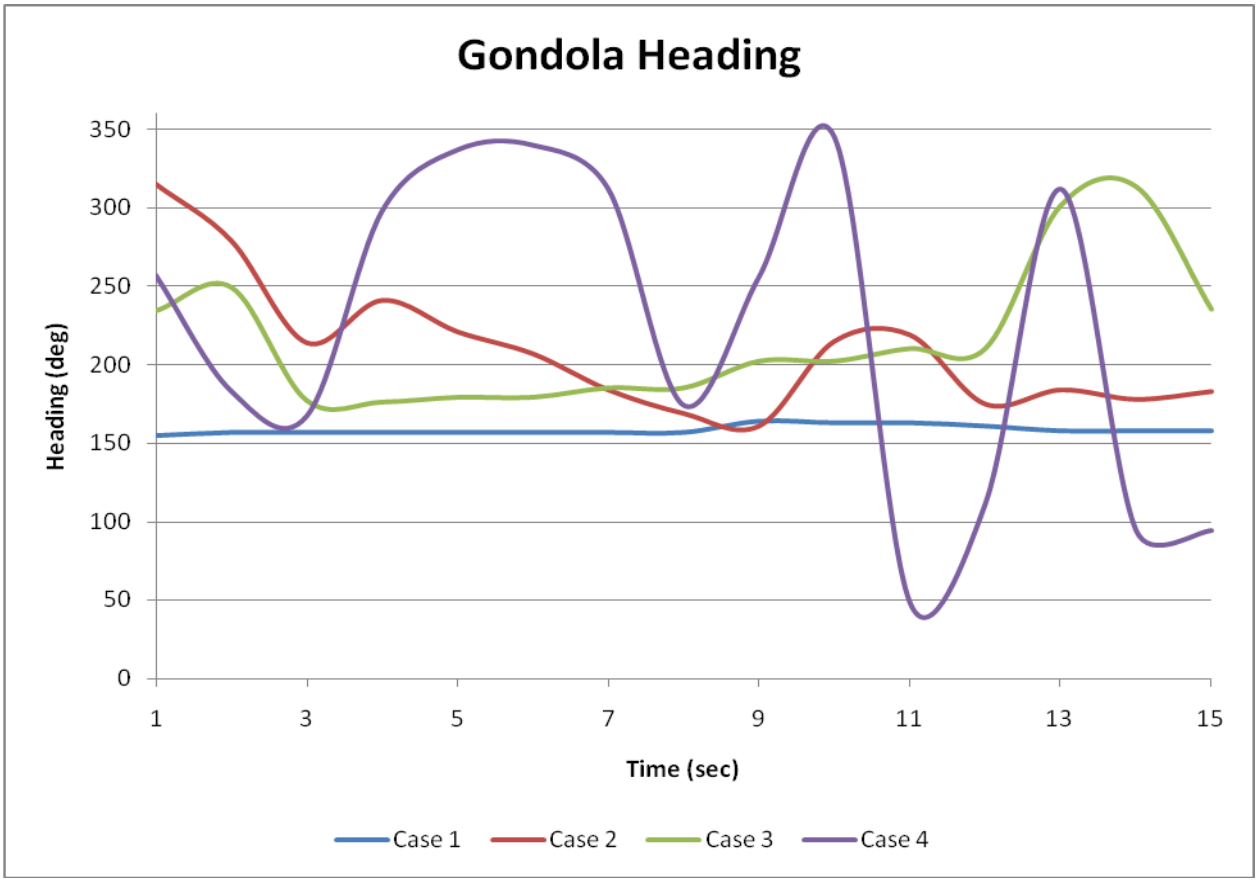


Figure 2 - Blimp Gondola Heading

Case 1 represents an over damped system. This is due to the fact that the k_P and k_D control gains do not cause the system to overshoot when approaching the desired heading. Furthermore, as shown by the graph, case 1 takes a long time to reach the desired heading, or steady state, and as such, has a long settling time. The settling time is defined as the time it

takes for the amplitude to reach and stay within the settling band. The settling band is the region around the desired output in which it does not matter if the system oscillates or not. k_P and k_D for this case were equal to 0.5 and 70 respectively. The results of case 1 on the gondola caused it to take a long time to reach the desired heading.

Case 2 represents the desired case, being a critically damped system. This is the most desired because a critically damped system will overshoot just slightly and quickly settle into the steady state with no further oscillations. Constants k_P and k_D for this case were equal to 3 and 100 respectively. The results of case 2 on the gondola caused it to reach the desired heading both quickly and efficiently, with minimal overshoot error.

Case 3 represents an under damped system. Such a system overshoots the desired heading, but gradually corrects to try and achieve the steady state. Depending on how big the overshoots are, the system may oscillate for quite a long time before coming into a steady state. k_P and k_D for this case were equal to 12 and 70 respectively. The results of case 3 on the gondola caused it to overshoot a lot. Because of this, the gondola kept oscillating back and forth trying to reach the desired heading. It took too long to reach the desired heading and thus was deemed to be inefficient in its control task.

Case 4 represents a control system that only uses proportional control. This is accomplished by setting k_D to zero. As shown by the graph, this system is unstable and causes huge overshoots and multiple oscillations. k_P for this case was equal to 30. The results of case 4 on the gondola caused it to overshoot too much. This eventually resulted in it spinning around fast and unable to stop.

To find a suitable proportional and derivative gain for the altitude control, some trial and error was done. It was found that a proportional gain of 3 and a derivative gain of 100 were suitable for proper altitude control. This caused minimal overshoot and a quicker response of the thrust fans. Due to this, the gondola was able to attain any desired altitude both quickly and efficiently with minimal error. Other gain combinations tested caused the thrust fans to overshoot a bit too much and multiple oscillations occurred afterward. This resulted in the fans having to decelerate for a time being and then start up again in order to achieve the desired altitude.

After analyzing and comparing each case, it was concluded that the best method for controlling the steering of the gondola was to implement a small proportional gain and a large derivative gain. The results showed that these gains were equal to 3 and 100 respectively. A proportional control gain constant lower than 3 would cause a slower response of the system and a higher one would cause the system to severely overshoot its desired heading. The derivative gain constant had to be modified from 180 to 100. The higher gain caused large directional fluctuations of the system during operation. Since the derivative gain provides system damping and makes it possible to use larger proportional gains, a derivative gain of 180 would require a higher proportional gain to minimize overshoot. In fact, when tested the system with a large derivative gain and low proportional gain, the system oscillated too much. Lowering the derivative gain constant fixed the problem.

During the testing and coding for the gondola, a few problems occurred temporarily preventing the task of full blimp control. Problems occurred with both the software and hardware.

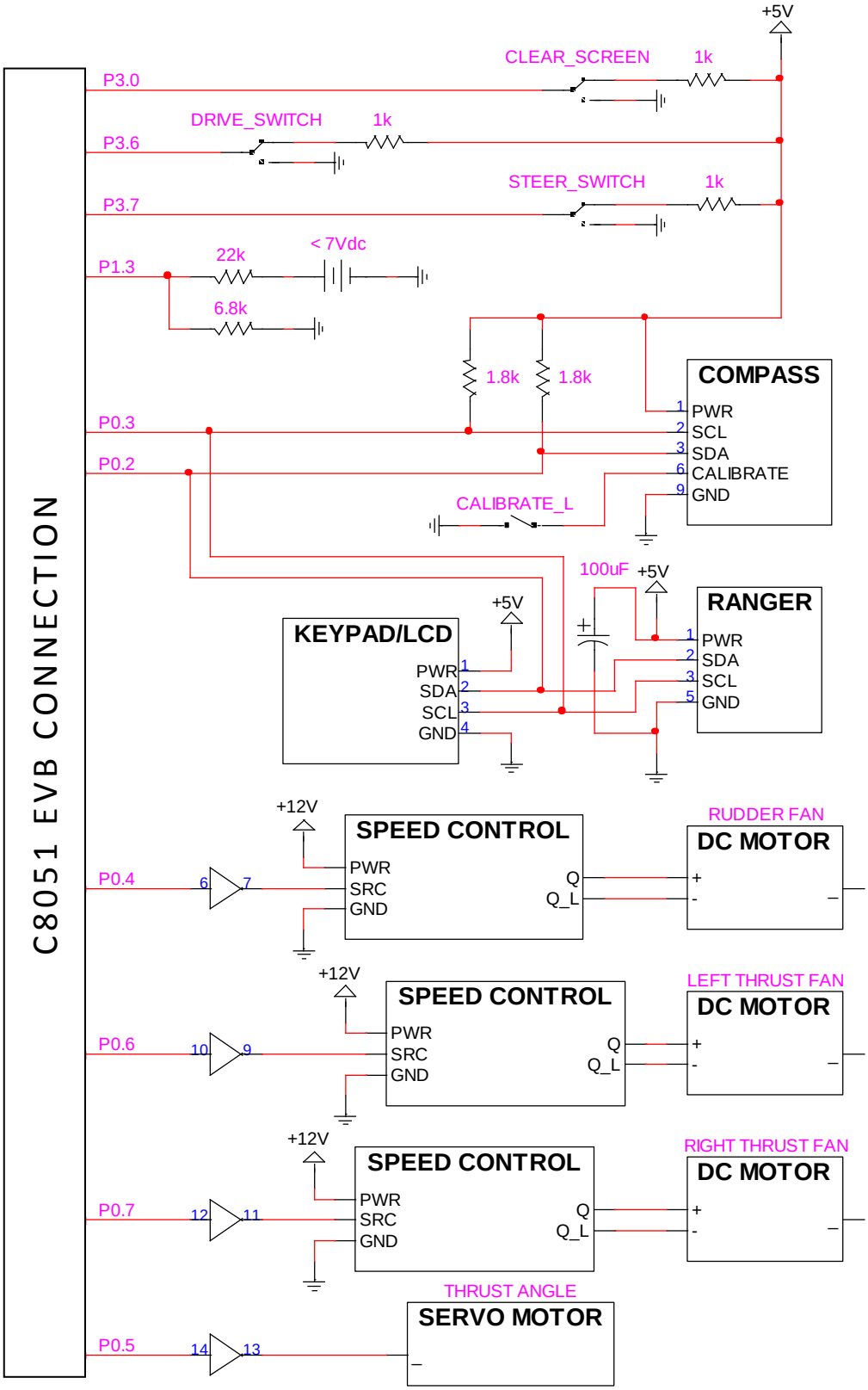
During the case testing for the gondola, the gondola got stuck after a few seconds of the code execution. It was unable to completely move to the desired direction since the interrupt service routine was too long. The interrupt service routine is meant to be small so it can be executed quickly. The code however was too long causing delays and problems when implemented on the gondola. To fix this issue, portions of the code that flagged for readings from the electronic compass and ultrasonic ranger were moved into the main part of the code.

When running the code for the ideal case, the thrust fans and rudder fan fluctuated unpredictably. The gondola moved either ever so slightly, or not at all on multiple resets of both the gondola and the software. Our A low battery voltage was causing this erratic behavior. After swapping out batteries, the code worked perfectly and the desired results were achieved.

One final note is that there was a lot of noise in the laboratory. This may have been caused by our gondola picking up other signals from other gondola teams or because the room is mostly made of steel and there are many wires and pipes around it. This noise caused our team's data collection to vary at times and give strange readings at others. This can be seen a bit in the graph above. The unusual data points were eliminated from the graph.

In the end our team was successful in developing a functioning control algorithm that was able to steer the gondola in a desired direction as well as keep it at a desired altitude. The addition of proportional and derivative control helped to achieve the desired results. In addition, our team learned how to program and use a microcontroller to control these functions by taking readings from an electronic compass and ultrasonic ranger.

Appendix A: Schematic



Appendix B: Source Code

```
#include <c8051_SDCC.h>       // Include C8051 SDCC Header
#include <stdio.h>            // Include Standard Input/Output Header
#include <stdlib.h>           // Include Standard Library Header
#include <i2c.h>              // Include I2C Bus Header

#define PW_MIN  2000
#define PW_NEUT 2750
#define PW_MAX  3500

void Port_Init(void);                 // Initialize Ports for Input and Output
void PCA_Init(void);                  // Initialize Programmable Counter Array 0
void XBR0_Init(void);                 // Initialize Crossbar 0
void SMB_Init(void);                  // Initialize SMBus
void ADC_Init(void);                  // Initialize Analog/Digital Converter 1
void PCA_ISR(void) interrupt 9;       // Programmable Counter Array 0 Interrupts Subroutine
int ADC_input(void);                  // Read Analog/Digital Converter 1 Input
void Steer_Rudder(void);              // Steer Rudder Fan
void Drive_Thrust(void);              // Drive Thrust Fans
int Read_Compass(void);               // Read Compass
unsigned int Read_Ranger(void);       // Read Ranger
void config_blimp(void);              // Configure Blimp
void keypad_wait(void);               // Wait before reading user input from keypad

sbit at 0xB0 CLEAR_SCREEN;    // Clear Screen Slideswitch connected to Port 3 Pin 0 --> input
sbit at 0xB6 DRIVE_SWITCH;    // Drive Slideswitch connected to Port 3 Pin 6 --> input
sbit at 0xB7 STEER_SWITCH;    // Steer Slideswitch connected to Port 3 Pin 7 --> input

unsigned int h_count;                 // PCA0 Timer overflow counter for compass
unsigned int r_count;                 // PCA0 Timer overflow counter for ranger
unsigned int new_heading;             // Flag for reading car heading data from compass
unsigned int new_range;               // Flag for reading driving speed data from ranger
unsigned int ANGLE_PW = 2700;         // Actual servo motor steering pulsewidth
unsigned int PCA_start = 28672;       // PCA period used for approximately 20ms
unsigned int desired_heading = 900;   // Desired heading value
unsigned int PD_case = 2;             // Case number for steering control
xdata long RUDDER_PW;                 // Actual rudder fan dc motor steering pulsewidth
xdata long THRUST_PW;                 // Actual thrust fans dc motor driving pulsewidth
xdata int kd_thrust = 50;             // Derivative Gain for Thrust Fans
xdata float kp_thrust = 0.5;          // Proportional Gain for Thrust Fans
int heading;                          // Car heading in tenths of a degree between 0 and 3599
int range;                            // Ultrasonic Ranger reading
int error;                            // Steering needed to reach desired heading
int range_error;                      // Thrust Error
int prev_error = 0;                   // Previous Error
int prev_range_error = 0;             // Previous Range Error
int battery = 100;                    // Percent battery charge left
int PCA_count = 0;                    // PCA0 Timer overflow counter
int wait = 0;                         // Wait counter
char keypad;                          // User input from keypad

// Autonomous Blimp
void main(void) {

        Sys_Init();     // Initialize System
        Port_Init();    // Initialize Ports for Input and Output
        PCA_Init();     // Initialize Programmable Counter Array 0
        XBR0_Init();    // Initialize Crossbar 0
        SMB_Init();     // Initialize SMBus
        ADC_Init();     // Initialize Analog/Digital Converter 1
```

```
putchar(' ');  // The quote fonts may not copy correctly into SiLabs IDE

RUDDER_PW = PW_NEUT;   // Set Rudder Fan DC Motor to center
THRUST_PW = PW_NEUT;   // Set Thrust Fans DC Motor to neutral

PCA0CPL0 = 0xFFFF - RUDDER_PW;        // Set low byte of CCM0  --> rudder fan heading
PCA0CPH0 = (0xFFFF - RUDDER_PW) >> 8; // Set high byte of CCM0 --> rudder fan heading
PCA0CPL1 = 0xFFFF - ANGLE_PW;         // Set low byte of CCM1  --> thrust fans angle
PCA0CPH1 = (0xFFFF - ANGLE_PW) >> 8;  // Set high byte of CCM1 --> thrust fans angle
PCA0CPL2 = 0xFFFF - THRUST_PW;        // Set low byte of CCM2  --> left thrust fan power
PCA0CPH2 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM2 --> left thrust fan power
PCA0CPL3 = 0xFFFF - THRUST_PW;        // Set low byte of CCM3  --> right thrust fan power
PCA0CPH3 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM3 --> right thrust fan power

// Wait one second for DC Motor to warm up
while (PCA_count < 50)
        printf("");

delay_time(100000);    // Wait for the keypad and display to initialize
config_blimp();        // Configure Blimp

printf("\n\rheading");
printf("\t\tdesired heading");
printf("\tcompass error");
printf("\taltitude");
printf("\tdesired altitude");
printf("\trange error");
printf("\tbattery voltage\n\r");

keypad_wait(); // Wait for data to display

while (1) {

        Steer_Rudder();        // Steer Rudder Fan
        Drive_Thrust();        // Drive Thrust Fans

        battery = ADC_input() * 40 / 74;      // Calculate percent battery charge left

        // Check for user input of 0
        if (read_keypad() == '0')
                config_blimp();        // Configure Blimp


        // Update display every 1000 ms
        if (PCA_count % 50 == 0) {

                lcd_clear();   // Clear display
                lcd_print("Battery: %d percent\r", battery);
                lcd_print("Heading: %u deg\r", heading / 10);
                lcd_print("Range: %d counts\r", range);

                if (battery > 10)
                        lcd_print("PRESS 0 TO CONFIGURE");

                else
                        lcd_print("WARNING: BATTERY LOW");

                PCA_count = 0; // Clear PCA0 Timer overflow counter

                printf("%u\t\t", heading / 10);
                printf("%u\t\t", desired_heading / 10);
                printf("%d\t\t", error);
```

```
                printf("%u\t\t", range);
                printf("50\t\t");
                printf("%d\t\t", range_error);
                printf("%u\n\r", battery * 74 / 1000);

        }

        // Read compass every 2 PCA0 Timer overflows (40 ms)
        if (h_count >= 2) {
                new_heading = 1;       // Set compass read flag to 1
                h_count = 0;
        }

        // Read ranger every 4 PCA0 Timer overflows (80 ms)
        if (r_count >= 4) {
                new_range = 1; // Set ranger read flag to 1
                r_count = 0;
        }

        // Stop running software if battery almost dead
        if (battery <= 5) {

                lcd_clear();   // Clear display
                lcd_print("\r BATTERY LOW!\r");
                lcd_print(" PLEASE RECHARGE!");

                RUDDER_PW = PW_NEUT;   // Center rudder fan if battery low
                THRUST_PW = PW_MIN;    // Set thrust fans down if battery low

                PCA0CPL0 = 0xFFFF - RUDDER_PW;        // Set low byte of CCM0 --> rudder fan heading
                PCA0CPH0 = (0xFFFF - RUDDER_PW) >> 8; // Set high byte of CCM0 --> rudder fan heading
                PCA0CPL2 = 0xFFFF - THRUST_PW;        // Set low byte of CCM2 --> left thrust fan power
                PCA0CPH2 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM2 --> left thrust fan power
                PCA0CPL3 = 0xFFFF - THRUST_PW;        // Set low byte of CCM3 --> right thrust fan power
                PCA0CPH3 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM3 --> right thrust fan power

                return;

        }

        // Check if clear screen switch enabled
        if (CLEAR_SCREEN)
                lcd_clear();   // Clear display

        wait = 0;      // Clear wait counter

        // Loop until wait counter reaches 10
        while (wait < 10) {

                wait++; // Increment wait counter
                printf("");

        }

    }

}

// Initialize Ports for Input and Output
void Port_Init(void) {

        P0MDOUT |= 0xF0;       // Set Port 0 Pins 4, 5, 6 and 7 to Push-Pull output mode
```

```
        P0MDOUT &= ~0x0C;      // Set Port 0 Pins 2 and 3 to Open-Drain input mode
        P0 |= 0x0C;            // Set Port 0 Pins 2 and 3 to High Impedance mode
        P1MDIN &= ~0x08;       // Set Port 1 Pin 3 to Analog input mode
        P1MDOUT &= ~0x08;      // Set Port 1 Pin 3 to Open-Drain input mode
        P3MDOUT &= ~0xD1;      // Set Port 3 Pins 0, 6 and 7 to Open-Drain input mode
        P3 |= 0xD1;            // Set Port 3 Pins 0, 6 and 7 to High Impedance mode

}

// Initialize Programmable Counter Array 0
void PCA_Init(void) {

        PCA0MD = 0x81;         // Use SYSCLK/12, enable CF interrupts, suspend when idle
        PCA0CPM0 = 0xC2;       // Set CCM0 to 16-bit compare mode
        PCA0CPM1 = 0xC2;       // Set CCM1 to 16-bit compare mode
        PCA0CPM2 = 0xC2;       // Set CCM2 to 16-bit compare mode
        PCA0CPM3 = 0xC2;       // Set CCM3 to 16-bit compare mode
        PCA0CN |= 0x40;        // Enable PCA0 counter
        EIE1 |= 0x08;          // Enable PCA0 interrupts
        EA = 1;                // Enable global interrupts

}

// Initialize Crossbar 0
void XBR0_Init(void) {

        XBR0 = 0x25;   // XBR0 enables TX0, RX0, SDA, SCL, CEX0, CEX1, CEX2 and CEX3

}

// Initialize SMBus
void SMB_Init(void) {

        SMB0CR = 0x93; // Set SCL to 100kHz
        ENSMB = 1;     // Enable the SMBus

}

// Initialize Analog/Digital Converter 1
void ADC_Init(void) {

        REF0CN = 0x03;         // Set Vref to use internal reference voltage (2.4 V)
        ADC1CN = 0x80;         // Enable Analog/Digital Converter (ADC1)
        ADC1CF |= 0x01;        // Set Analog/Digital Converter gain to 1

}

// Programmable Counter Array 0 Interrupts Subroutine
void PCA_ISR(void) interrupt 9 {

        if (CF) {

                PCA_count++;   // Increment PCA0 Timer overflow counter
                h_count++;     // Increment PCA0 Timer overflow counter for compass
                r_count++;     // Increment PCA0 Timer overflow counter for ranger

                PCA0L = PCA_start;            // Set low byte of PCA0 start count
                PCA0H = PCA_start >> 8;       // Set high byte of PCA0 start count
                CF = 0;                       // Clear interrupt flag

        } else
                PCA0CN &= 0xC0;        // Handle all other type 9 interrupts
```

```
}

// Read Analog/Digital Converter 1 Input
int ADC_input(void) {

        AMX1SL = 3;                     // Set Port 1 Pin 3 as Analog Input for ADC1
        ADC1CN = ADC1CN & ~0x20;        // Clear the Conversion Completed flag
        ADC1CN = ADC1CN | 0x10;         // Initiate Analog/Digital conversion

        while ((ADC1CN & 0x20) == 0x00);      // Wait for conversion to complete

        return ADC1;   // Return digital value in ADC1 register

}

// Steer Rudder Fan
void Steer_Rudder(void) {

        if (new_heading) {

                heading = Read_Compass();     // Read Compass

                // Determine steering needed to reach desired heading
                error = desired_heading - heading;

                // Correct error for degrees less than -180
                if (error < -1800)
                        error = 3600 + error;

                // Correct error for degrees greater than 180
                if (error > 1800)
                        error = error - 3600;


                // Get Servo Motor pulsewidth
                if (PD_case == 1)
                        RUDDER_PW = (long) PW_NEUT + (long) (0.5) * (long) (error) + (long) (70) *
(long) (error - prev_error);  // Case 1

                else if (PD_case == 2)
                        RUDDER_PW = (long) PW_NEUT + (long) (2) * (long) (error) + (long) (100) *
(long) (error - prev_error);  // Case 2

                else if (PD_case == 3)
                        RUDDER_PW = (long) PW_NEUT + (long) (20) * (long) (error) + (long) (70) *
(long) (error - prev_error);  // Case 3

                else if (PD_case == 4)
                        RUDDER_PW = (long) PW_NEUT + (long) (30) * (long) (error);  // Case 4

                prev_error = error;    // Store previous error

                // Stop car from steering too far to the right
                if (RUDDER_PW > PW_MAX)
                        RUDDER_PW = PW_MAX;

                // Stop car from steering too far to the left
                else if (RUDDER_PW < PW_MIN)
                        RUDDER_PW = PW_MIN;

                new_heading = 0;       // Clear compass read flag
```

```
        }

        // Center rudder fan if Steer Slideswitch in STOP position or battery low
        if (STEER_SWITCH || battery <= 10)
                RUDDER_PW = PW_NEUT;

        PCA0CPL0 = 0xFFFF - RUDDER_PW;        // Set low byte of CCM0 --> rudder fan heading
        PCA0CPH0 = (0xFFFF - RUDDER_PW) >> 8; // Set high byte of CCM0 --> rudder fan heading

}

// Drive Thrust Fans
void Drive_Thrust(void) {

        if (new_range) {

                range = Read_Ranger(); // Read Ranger

                // Determine thrust needed to reach desired altitude
                range_error = 50 - range;

                // Get DC Motor pulsewidth
                THRUST_PW = (long) PW_NEUT + (long) (kp_thrust) * (long) (range_error) + (long)
(kd_thrust) * (long) (range_error - prev_range_error);

                prev_range_error = range_error;       // Store previous error

                // Stop blimp from driving too much in the upward direction
                if (THRUST_PW > PW_MAX)
                        THRUST_PW = PW_MAX;

                // Stop blimp from driving too much in the downward direction
                else if (THRUST_PW < PW_MIN)
                        THRUST_PW = PW_MIN;

                new_range = 0; // Clear ranger read flag

        }

        // Stop thrust fans if Drive Slideswitch in STOP position
        if (DRIVE_SWITCH)
                THRUST_PW = PW_NEUT;

        // Set thrust fans down if battery low
        if (battery <= 10)
                THRUST_PW = PW_MIN;

        PCA0CPL2 = 0xFFFF - THRUST_PW;        // Set low byte of CCM2 --> left thrust fan power
        PCA0CPH2 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM2 --> left thrust fan power
        PCA0CPL3 = 0xFFFF - THRUST_PW;        // Set low byte of CCM3 --> right thrust fan power
        PCA0CPH3 = (0xFFFF - THRUST_PW) >> 8; // Set high byte of CCM3 --> right thrust fan power

}

// Read Compass
int Read_Compass(void) {

        unsigned char addr = 0xC0;            // Address of compass sensor
        unsigned char Data[2];                // Data array of length 2
        int heading;                          // Car heading in tenths of a degree
        i2c_read_data(addr, 2, Data, 2);      // Read two bytes from compass, start at register 2
```

```
        heading = (((int) Data[0] << 8) | Data[1]);  // Combine the two bytes read

        return heading;        // Return car heading

}

// Read Ranger
unsigned int Read_Ranger(void) {

        unsigned char addr = 0xE0;            // Address of ranger sensor
        unsigned char Data[2];                // Data array of length 2
        i2c_read_data(addr, 2, Data, 2);      // Read two bytes from ranger, start at register 2

        range = (((unsigned int) Data[0] << 8) | Data[1]);   // Combine the two bytes read

        Data[0] = 0x51;                       // Start new ping
        i2c_write_data(addr, 0, Data, 1);     // Read one byte from ranger, start at register 0

        return range;  // Return ranger data

}

// Configure Blimp
void config_blimp(void) {

        lcd_clear();   // Clear display
        keypad_wait(); // Wait before reading user input from keypad

        lcd_print(" Adjust Thrust Fans\r");
        lcd_print("       2 = Up\r");
        lcd_print("    5 = Calibrate\r");
        lcd_print("      8 = Down");

        keypad = '?';  // Clear keypad value

        // Wait for valid user input
        while (keypad != '5') {

                keypad = read_keypad();       // Get user input from keypad

                if (keypad == '2')
                        ANGLE_PW = ANGLE_PW + 20;     // Increase the steering pulsewidth by 20

                else if (keypad == '8')
                        ANGLE_PW = ANGLE_PW - 20;     // Decrease the steering pulsewidth by 20

                PCA0CPL1 = 0xFFFF - ANGLE_PW;         // Set low byte of CCM1 --> thrust fans angle
                PCA0CPH1 = (0xFFFF - ANGLE_PW) >> 8;  // Set high byte of CCM1 --> thrust fans angle

                keypad_wait(); // Wait before reading user input from keypad

        }

        keypad_wait(); // Wait before reading user input from keypad
        lcd_clear();   // Clear display

        lcd_print("   2 = North\r");
        lcd_print("4 = West 6 = East\r");
        lcd_print("   8 = South");

        keypad = '?';  // Clear keypad value
```

```
	// Wait for valid user input
	while (keypad != '2' && keypad != '4' && keypad != '6' && keypad != '8') {

		keypad = read_keypad();		// Get user input from keypad

		if (keypad == '2')
			desired_heading = 3600;		// North

		else if (keypad == '4')
			desired_heading = 2700;		// West

		else if (keypad == '6')
			desired_heading = 900; // East

		else if (keypad == '8')
			desired_heading = 1800;		// South

		keypad_wait(); // Wait before reading user input from keypad

	}

	keypad_wait(); // Wait before reading user input from keypad
	lcd_clear();   // Clear display

	lcd_print(" Rudder Gain Kd/Kp\r\r");
	lcd_print("1: 70/.5  2: 100/2\r");
	lcd_print("3: 70/20  4: 0/20\r");

	keypad = '?';  // Clear keypad value

	// Wait for valid user input
	while (keypad != '1' && keypad != '2' && keypad != '3' && keypad != '4') {

		keypad = read_keypad();		// Get user input from keypad

		if (keypad == '1')
			PD_case = 1;   // Case 1

		else if (keypad == '2')
			PD_case = 2;   // Case 2

		else if (keypad == '3')
			PD_case = 3;   // Case 3

		else if (keypad == '4')
			PD_case = 4;   // Case 4

		keypad_wait(); // Wait before reading user input from keypad

	}

	keypad_wait(); // Wait before reading user input from keypad
	lcd_clear();   // Clear display

	lcd_print("Thrust Gain:5=exit\r");
	lcd_print("  1: kd +   3: kd -\r");
	lcd_print("  4: kp +   6: kp -\r");
	lcd_print(" kd=%u kp=%u.%u", kd_thrust, (int) kp_thrust, (int) ((kp_thrust - (int)
kp_thrust) * 10));

	keypad = '?';  // Clear keypad value
```

```
        // Wait for valid user input
        while (keypad != '5') {

                keypad = read_keypad();       // Get user input from keypad

                if (read_keypad() == '1')
                        kd_thrust = kd_thrust + 20;   // Increase derivative thrust gain by 20

                if (read_keypad() == '3')
                        kd_thrust = kd_thrust - 20;   // Decrease derivative thrust gain by 20

                if (read_keypad() == '4')
                        kp_thrust = kp_thrust + 0.3;  // Increase proportional thrust gain by 0.3

                if (read_keypad() == '6')
                        kp_thrust = kp_thrust - 0.3;  // Decrease proportional thrust gain by 0.3

                if (keypad == '1' || keypad == '3' || keypad == '4' || keypad == '6') {

                        lcd_clear();   // Clear display
                        lcd_print("Thrust Gain:5=exit\r");
                        lcd_print("  1: kd +   3: kd -\r");
                        lcd_print("  4: kp +   6: kp -\r");
                        lcd_print(" kd=%u kp=%u.%u", kd_thrust, (int) kp_thrust, (int) ((kp_thrust
- (int) kp_thrust) * 10));

                }

                keypad_wait(); // Wait before reading user input from keypad

        }

        lcd_clear();   // Clear display

}

// Wait before reading user input from keypad
void keypad_wait(void) {

        wait = 0;      // Clear wait counter

        // Loop until wait counter overflows
        while (wait < 25534) {

                wait++; // Increment wait counter
                printf("");

        }

}
```

Appendix C: Data and Observations

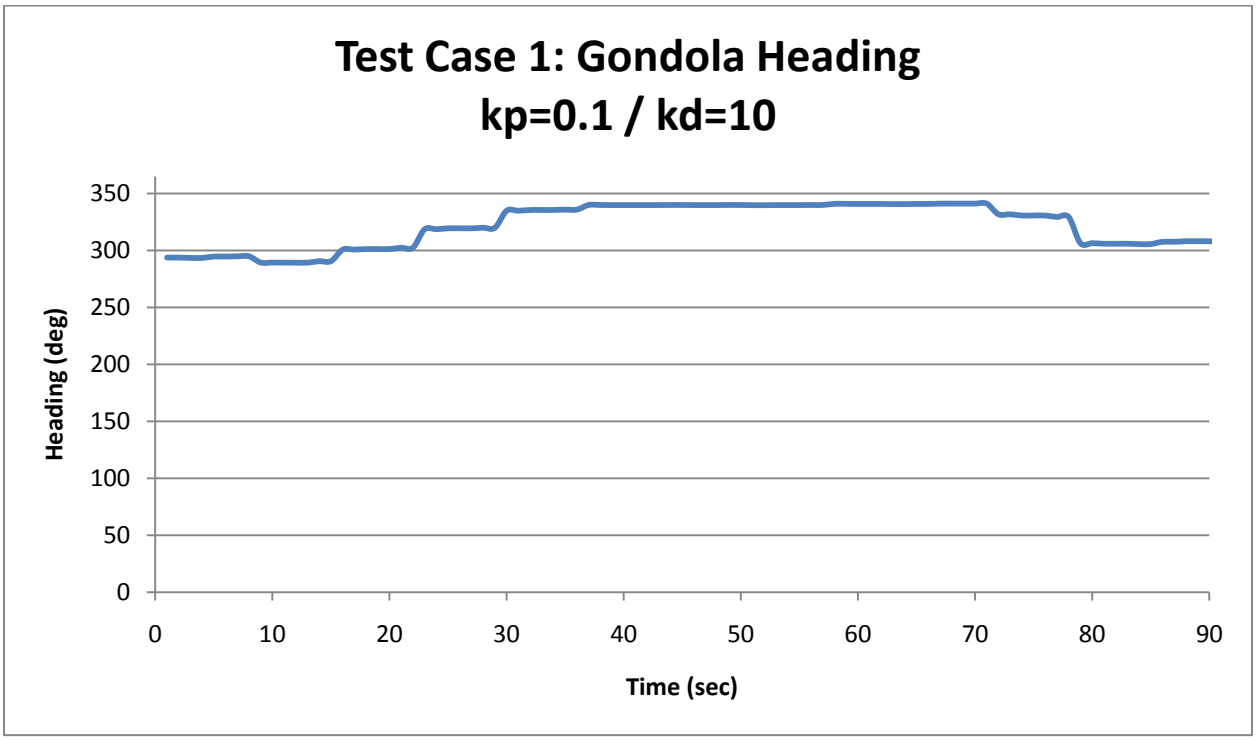


Figure 3 - Test Case 1: Blimp Gondola Heading

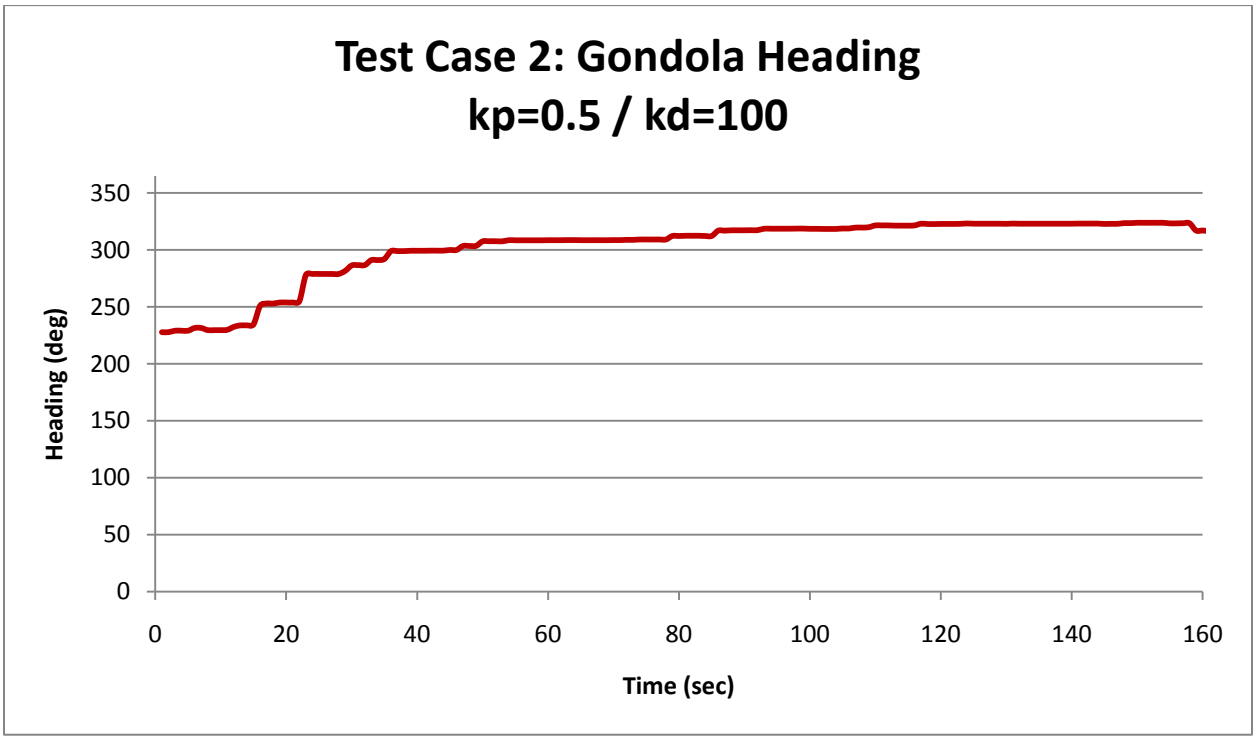


Figure 4 - Test Case 2: Blimp Gondola Heading

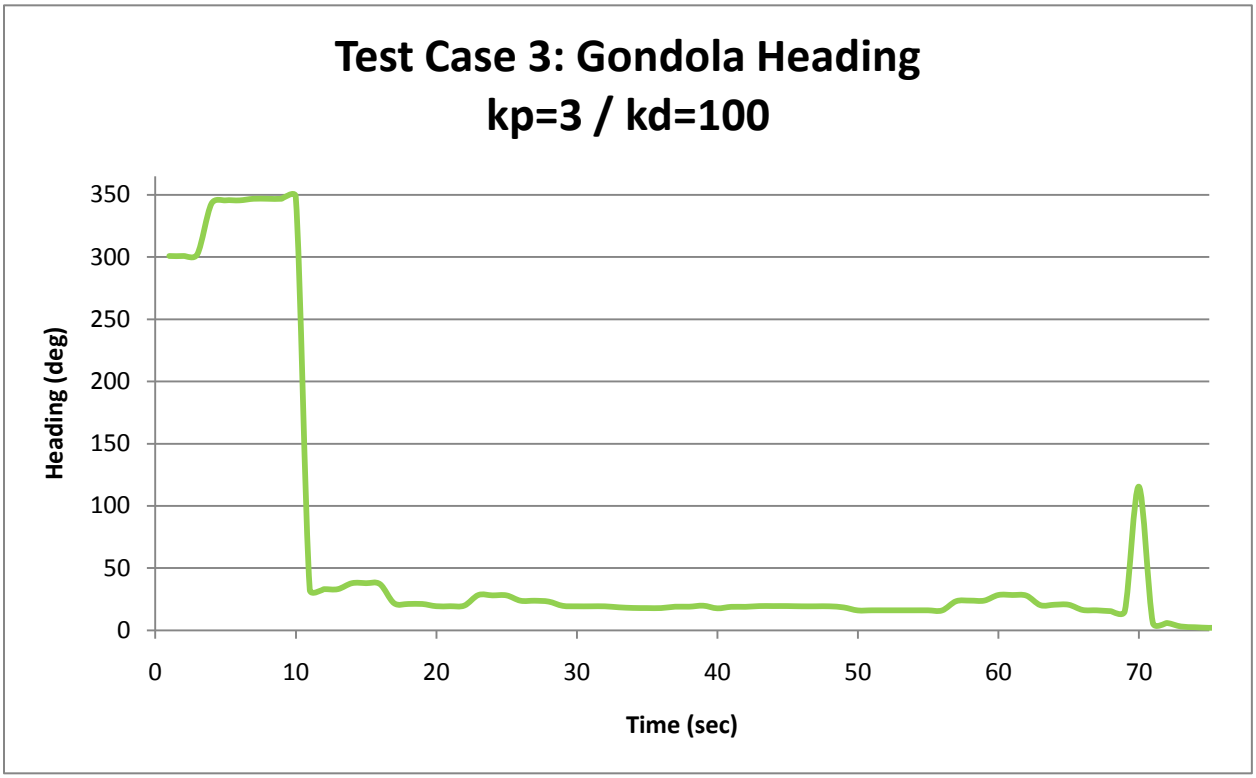


Figure 5 - Test Case 3: Blimp Gondola Heading

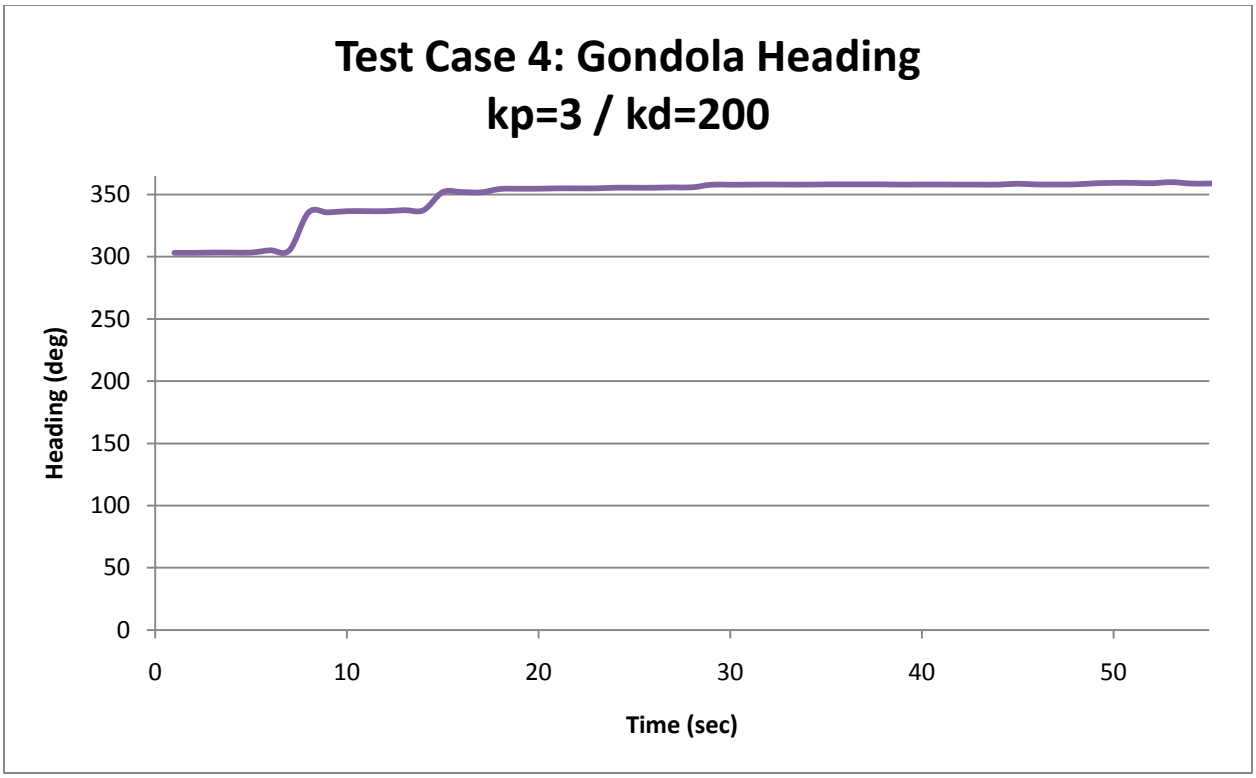


Figure 6 - Test Case 4: Blimp Gondola Heading

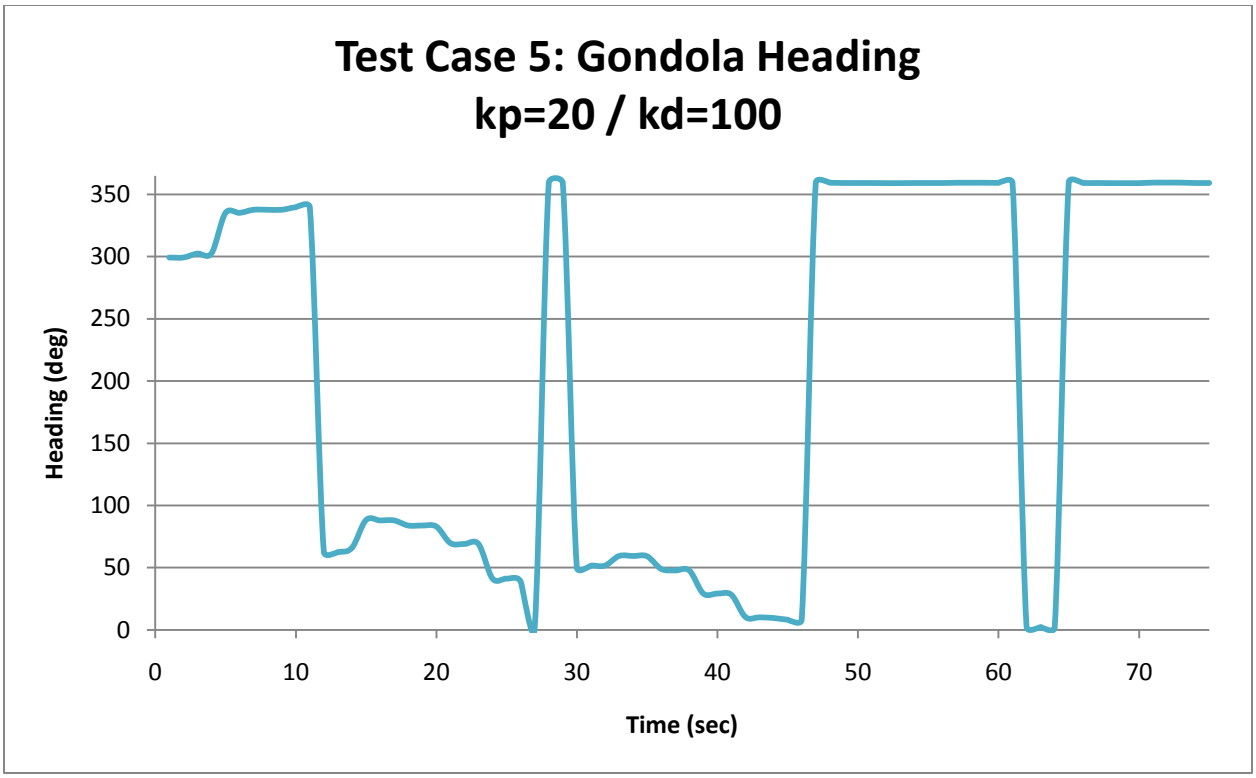


Figure 7 - Test Case 5: Blimp Gondola Heading

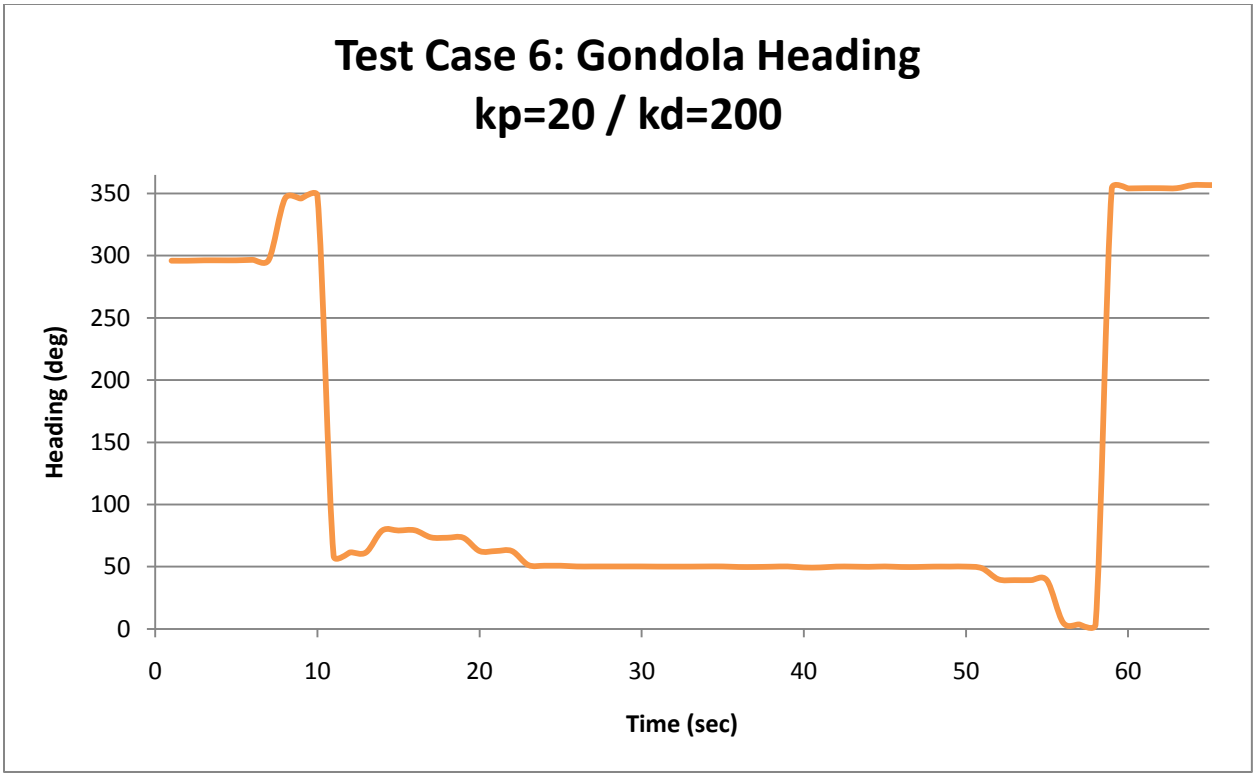


Figure 8 - Test Case 6: Blimp Gondola Heading

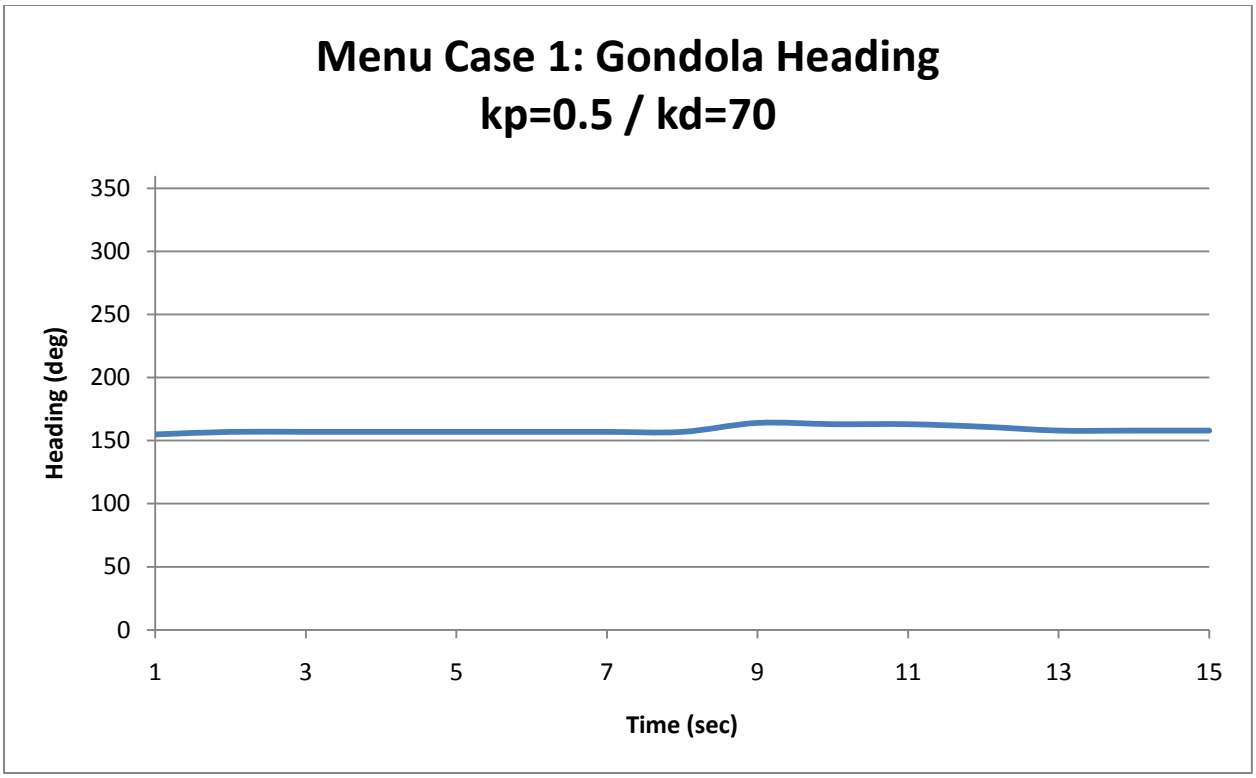


Figure 9 - Menu Case 1: Blimp Gondola Heading

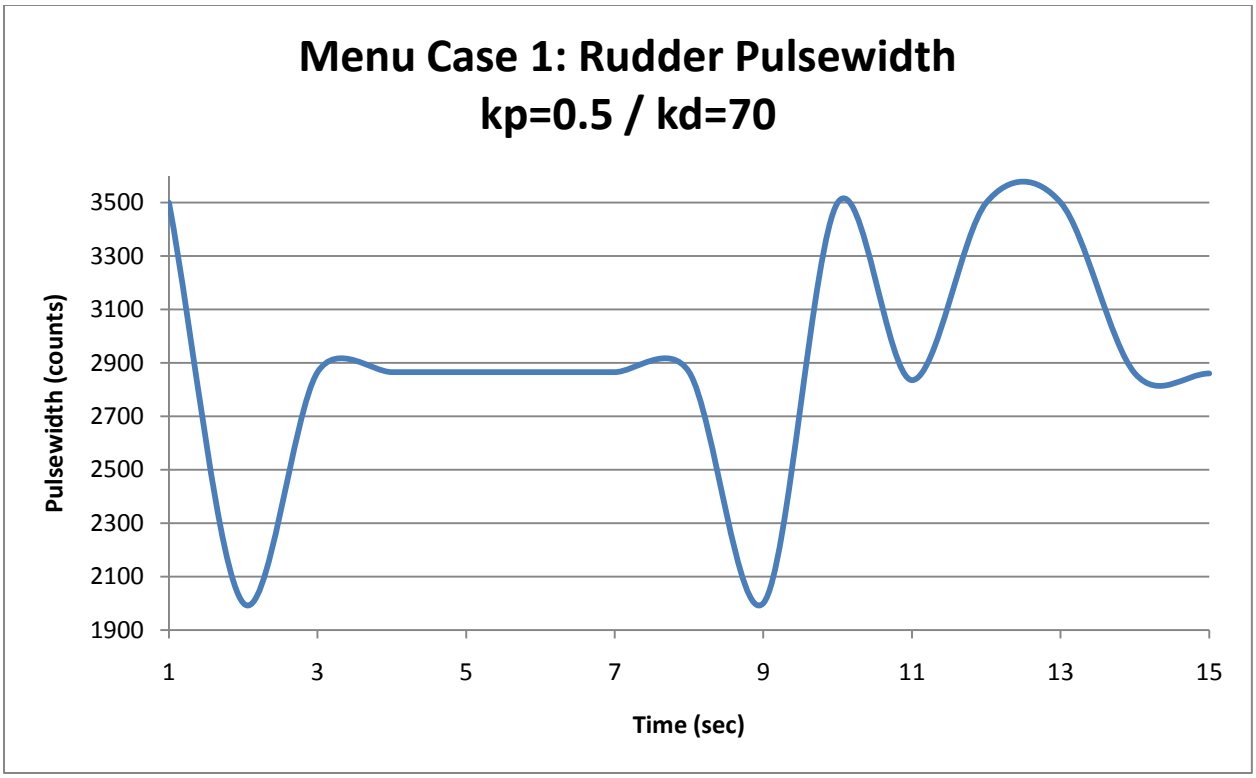


Figure 10 - Menu Case 1: Rudder Motor Pulsewidth

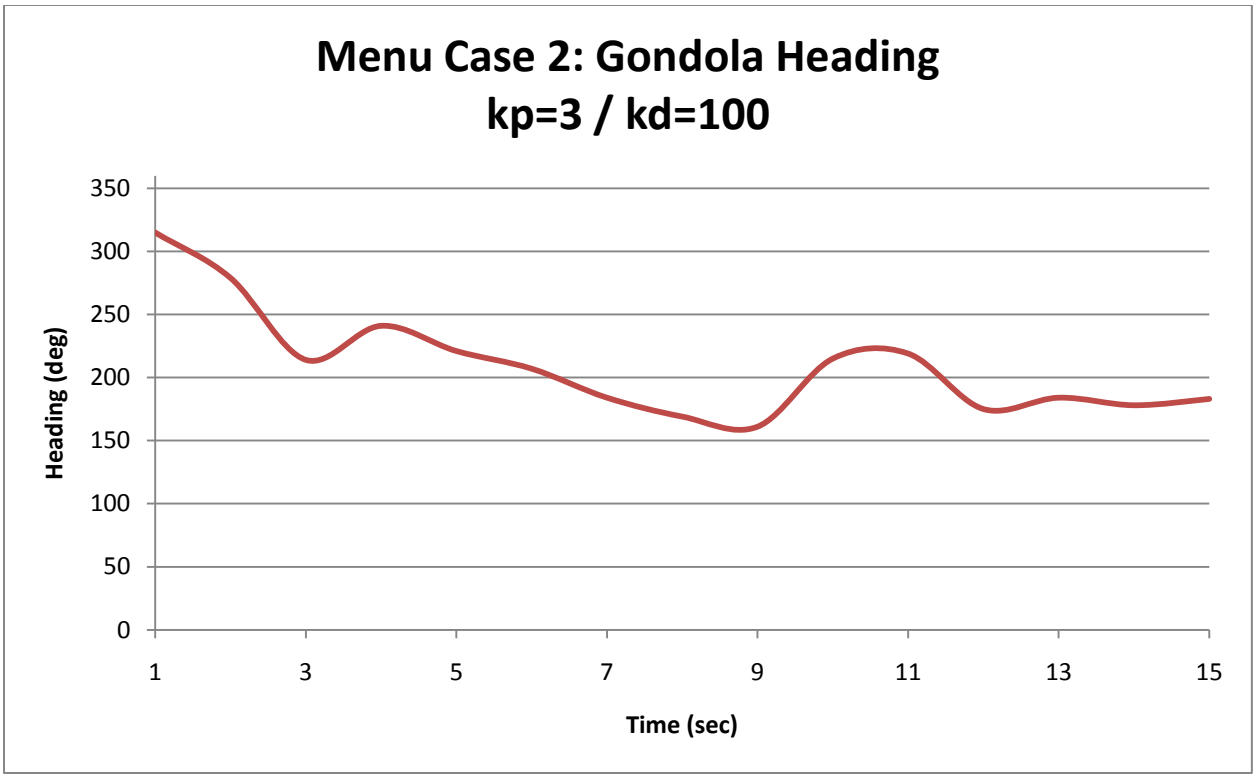


Figure 11 - Menu Case 2: Blimp Gondola Heading

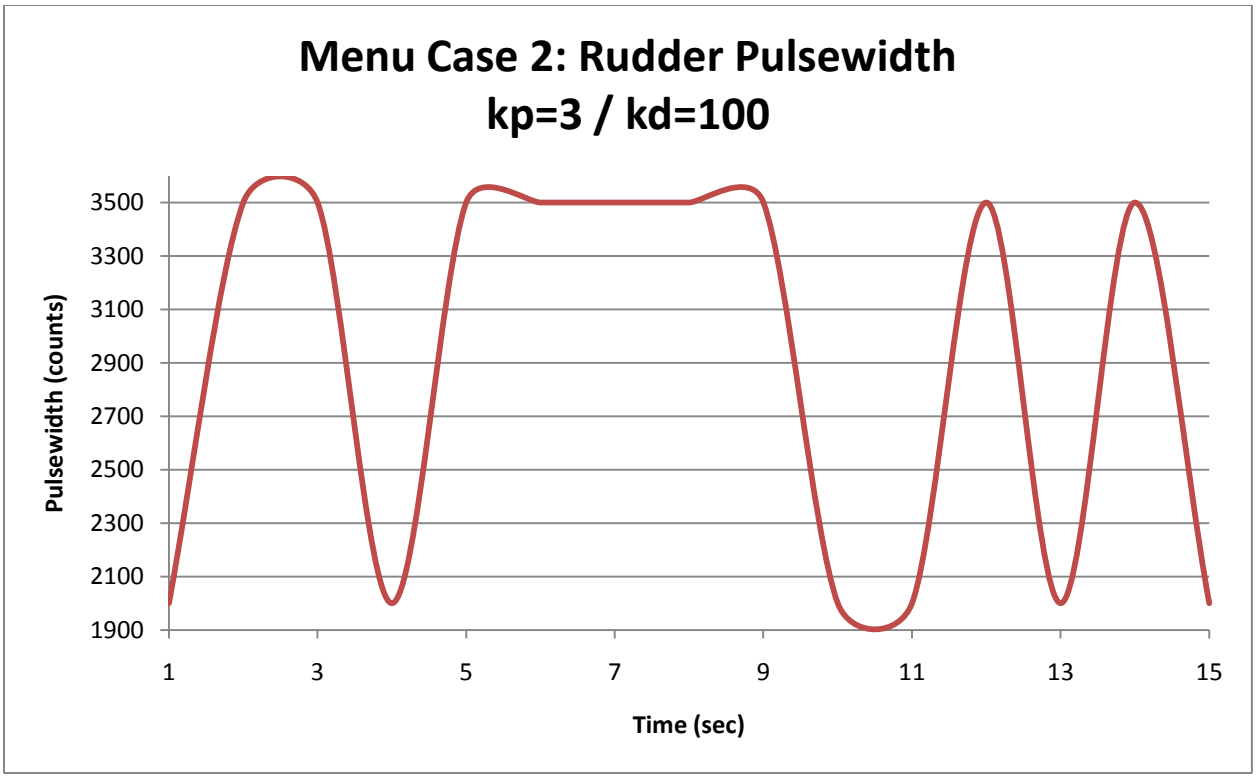


Figure 12 - Menu Case 2: Rudder Motor Pulsewidth

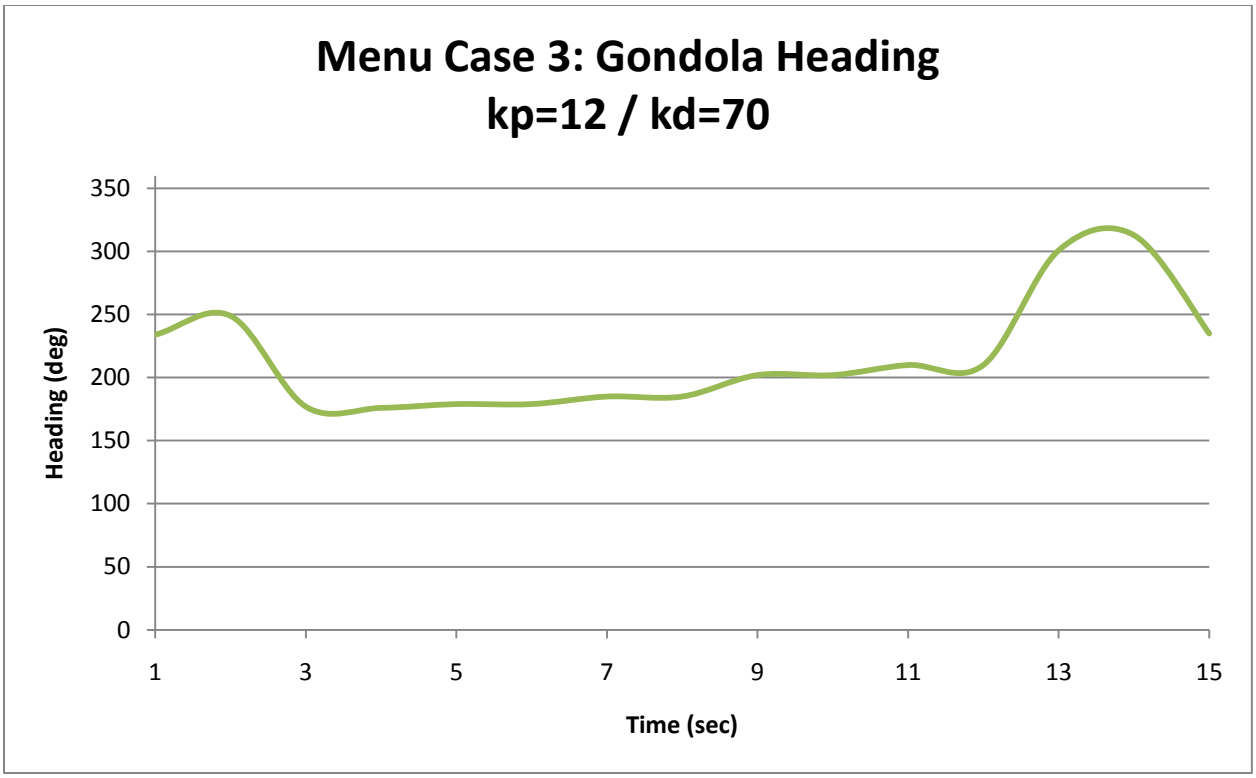


Figure 13 - Menu Case 3: Blimp Gondola Heading

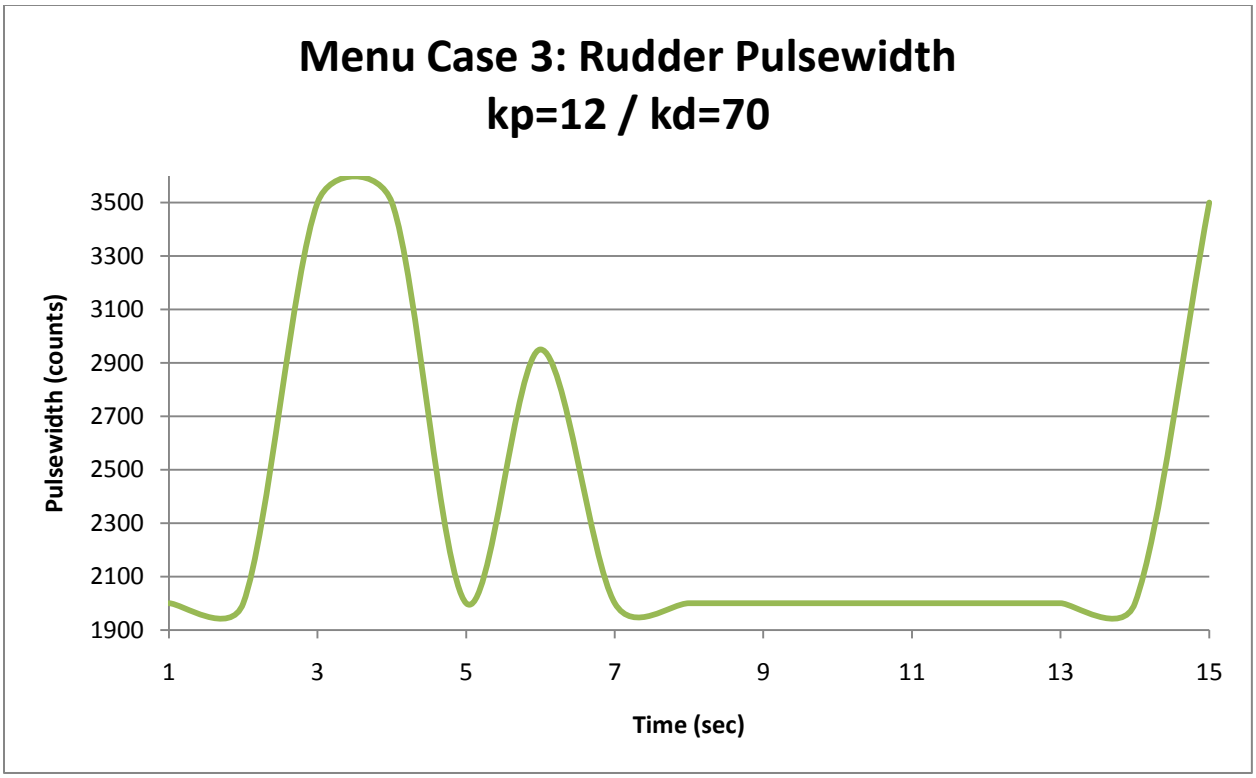


Figure 14 - Menu Case 3: Rudder Motor Pulsewidth

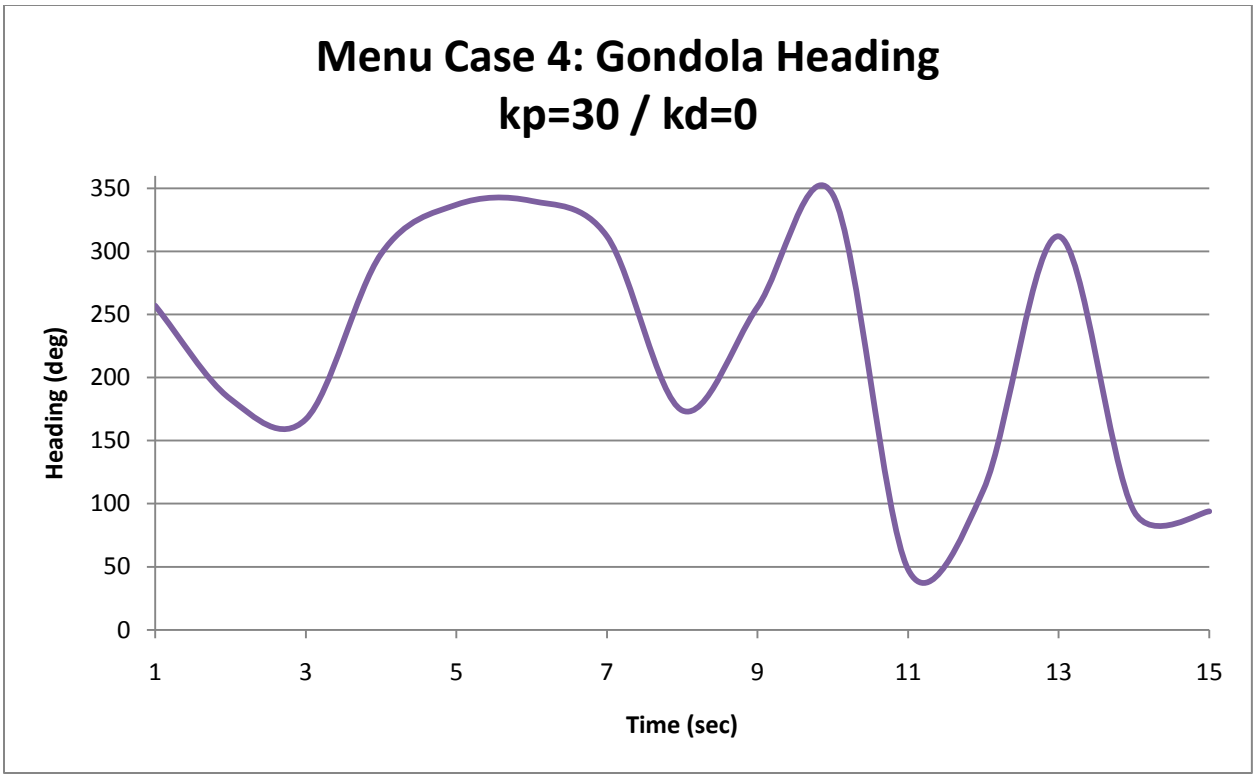


Figure 15 - Menu Case 4: Blimp Gondola Heading

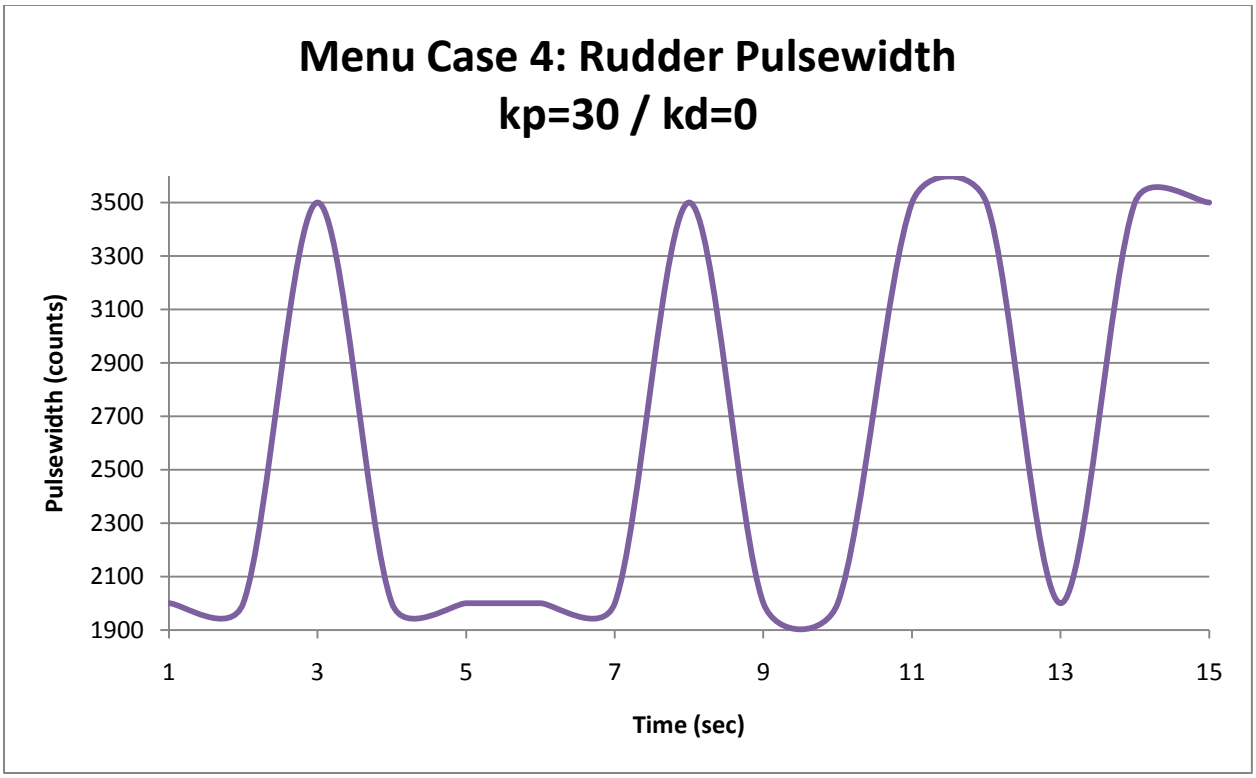


Figure 16 - Menu Case 4: Rudder Motor Pulsewidth

Test Case	Proportional Gain	Derivative Gain
1	0.1	10
2	0.5	70
3	3	70
4	3	180
5	12	70
6	12	180

Table 1 - Pre-Laboratory Test Case Gain Constants

Case	Proportional Gain	Derivative Gain
1	0.5	70
2	3	100
3	12	70
4	30	0

Table 2 - Blimp Gondola Configuration Menu Case Gain Constants

Participation

The chart below indicates each task involved in completing this lab, and who did what.

Task	Completed By
Pseudocode	Sicard
Software Development	Medina / Steinberger
Software and Hardware Troubleshooting	Medina / Sicard / Steinberger / Nelson
Data Collection and Compilation	Medina / Steinberger
Introduction	Sicard
Hardware Description	Nelson / Sicard
Software Description	Steinberger
Results and Conclusion	Medina
Flowchart	Steinberger
Schematic	Steinberger
Source Code	Steinberger
Data and Observations	Medina / Steinberger
Formatting	Nelson / Steinberger

Gabriel Media Date

Emily Nelson Date

Paul Sicard Date

Adam Steinberger Date

Works Cited

Schoch, P., Kraft, R., Gutin, A., Lee, S., & Braunstein, J. (2010). *Laboratory Introduction to Embedded Control: Lab Manual v13.1.* Troy, New York.

Silicon Laboratories Inc. (2003). *C8051F020/1/2/3-DS14.* Austin, Texas.