APPLICATION OF KALMAN FILTERING AND PID CONTROL FOR DIRECT INVERTED PENDULUM CONTROL

_____

A Project

Presented

to the Faculty of

California State University, Chico

_____

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

in

Electrical and Computer Engineering

Electronic Engineering Option

_____

by

José Luis Corona Miranda

Spring 2009

APPLICATION OF KALMAN FILTERING AND PID CONTROL FOR DIRECT INVERTED PENDULUM CONTROL

A Project

by

José Luis Corona Miranda

Spring 2009

APPROVED BY THE DEAN OF THE SCHOOL OF GRADUATE, INTERNATIONAL, AND INTERDISCIPLINARY STUDIES:

_____
Susan E. Place, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

_____
Adel A. Ghandakly, Ph.D.
Graduate Coordinator

_____
Dale Word, M.S., Chair

_____
Adel A. Ghandakly, Ph.D.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

ABSTRACT

APPLICATION OF KALMAN FILTERING AND PID CONTROL FOR DIRECT INVERTED PENDULUM CONTROL

by

José Luis Corona Miranda

Master of Science in Electrical and Computer Engineering

Electronic Engineering Option

California State University, Chico

Spring 2009

Robotic mobility technologies over the past few years have gain popularly in both commercial and government sectors. There been a variety of techniques suggested to increase robotic mobility on dynamic environments. One such popular technique used to provide greater mobility to a robotic platform is based on the inverted pendulum model. The presented document will demonstrate the techniques involved in balancing an unstable robotic platform. The objective is to design a complete discrete digital control system that will provide the needed stability. The platform will be an ideal test bed for the implementations of both PID digital control and Kalman filter algorithms. Both algorithms will provide the necessary control for the system. Therefore the presented

project will investigate the performance of both PID digital control and Kalman filter algorithms.

Test software was written to gather performance results for both the PID controller and Kalman filter. The control system performance is directly dependent on Kalman filter and PID controller input parameters. The results clearly show how the adjustable parameters on the control system directly affected the overall system performance. The results also demonstrate the performance and the need of the Kalman filter to remove sensor noise. The almost reliable sensor data increases PID controller performance to drive the robotic platform to vertical equilibrium. The gathered results for the Kalman filter were compared against the raw noisy sensor data. The plots for such comparison are shown on the Kalman filter results section. PID controller output response data was also collected and plotted. The PID output response results were used in the controller tuning process.

CHAPTER I

INTRODUCTION

"The research on balancing robots has gained momentum over the last decade in a number of robotics laboratories around the world" [7, p. 1]. The research has inspired some robotic enthusiasts to develop both private and public industry products. Such robots are characterized by the ability to balance on two wheels and spin on the spot. This additional maneuverability allows easy navigation on various terrains. These capabilities have the potential to solve a number of challenges in industry and society. The balancing robot platform technologies will eventually emerge as a new way of maneuverability and mobility in robotic applications. For example, a motorized wheelchair utilizing this technology would give the operator greater maneuverability and thus access to places most able-bodied people take for granted. Small carts built utilizing this technology allow humans to travel short distances in a small area or factories as opposed to using cars or buggies.

The presented project document will investigate a two wheel balancing robot, which will be used as a test bed to examine the use of a digital control algorithm and a Kalman filter for sensor fusion. The self balancing robot will be model after the inverted pendulum problem. The digital control algorithm that will be investigated as part of the self balancing robot project is the Proportional-Integral-Derivative controller. Over the past few years microcontrollers have become faster, cheaper and more reliable. A

microcontroller will be the choice to implement the filter and digital control algorithms. Both algorithms will be programmed using a high level programming language such as C. The suitability and the performance of the control algorithm will be examined. The PID control algorithm will be used on the balancing robot to provide system stability. Another very important addition to the digital control system on the robot is the use of the Kalman filter. The filter is an estimation algorithm that is popular among the embedded control community. The Kalman filter will be used as part of the project to provide sensor fusion between the accelerometer and gyroscope. The digital filter will provide the reliable sensor data that will be used by the robot to get tilt angle information.

The control and filter algorithms will all be written in software and implemented on a PIC32 microcontroller. The main use of the microcontroller is to serve as the digital controller for the robot. Measurable data will be logged onto a micro-SD memory card. The data collected from the self balancing robot will then be transferred to a personal computer, where data will be plotted to show control and system performance. The maneuverability for the autonomous self balancing robot will be achieved though the use of two dc brushed motors. Function of the self balance robot digital control algorithm, Kalman filter, microcontroller, sensors, and motors will be described in the following sections.

Purpose of the Project

The purpose of the project is to enhance the understanding of digital control algorithms and how the algorithms can be used to balance a robot on two wheels. The goal of the project is to provide the means of implementing a Kalman filter and PID

controller on a microcontroller. Provide a student or control system researcher means of testing the various digital control algorithms on the self balancing robot platform. The self balancing robot can also be used as a demonstration platform in a control systems course setting to inspire future control engineers.

Limitations of the Project

The major constraint that the autonomous two wheel robot will have will be the sampling time that will be used to execute the control algorithms. The noisy measurements from the feedback sensors can impact the performance of control algorithm. Noisy sensor measurements give inaccurate results which will not allow PID control performance. Another major issue on the robot will be the computation time that will be used on the microcontroller to run the control algorithms. If there is an extensive delay, the robot will not be able to correct the angle in time to keep the robot stable. Due to the cost of some essential components for this project, the overall cost of putting this project together can be a great limitation.

CHAPTER II

REVIEW OF RELATED LITERATURE

Conducting initial review research is very critical in understanding self balancing robot control techniques. The review of research related literature conducted for this project will summarize some of topics related to the techniques used for the balancing of robot based on the inverted pendulum model. Comparisons between the present master's project and the related topics of existing information will also be discussed. Abstracts of the related literature on the balancing robot topic will provide the needed information on the technology that is available. The methodologies and the techniques used by other researchers around the globe on the two wheel balancing robots topic will also be discussed.

The inverted pendulum problem has been a topic of high interest among the control engineering community. The uniqueness and complexity of the inverted pendulum problem has made it an ideal control engineering problem that can be used for both commercial and military applications. In recent years, researchers and engineers have applied the idea of a mobile inverted pendulum model to various problems, some which include walking gaits for humanoid robots, personal transport systems and robotic wheelchairs. Applications of the inverted pendulum control problem can be applied to both commercial and government uses. For example the Segway, used for commercial purposes and the prototype of the VECNA B.E.A.R robot project can be used for

government military operations. Humanoid like robots that gain there mobility thru two wheels have in the past few years become popular in commercial and government. The control problem can be simulated and implemented on a classroom setting to teach control engineering students the need for control in the unstable two wheeled robot. The following research literature abstracts summarize some of the popular balancing robot platforms and technologies that are used by researchers and engineers around the world.

BallBot

Ralph Hollis is a research professor at Carnegie Mellon University; he has developed a totally unique balancing robot that balances on top of a bowling ball. He calls his robot design “Ballbot” [2, p. 72]. Mr. Hollis and his research associates believe that robots in the future will play a vital role in the daily lives of humans. He believes that in order for robots to be productive in our daily lives, some key problems need to be solved first. One the important problem he states in his article about mobile self balancing robots is the overall structure of the robot itself. As stated by Ralph Hollis,” Robots tall enough to interact effectively in human environments have a high center of gravity and must accelerate and decelerate slowly, as well as avoid steep ramps, to keep from falling over. To counter this problem, statically stable robots tend to have broad bodies on wide wheelbases, which greatly restricts their mobility through doorways and around furniture or people” [2, p. 74]. The size of the robots will ultimately affect its mobility of the robot. In order to solve the problem, Hollis came up with a new design that improved the robot’s overall structure and mobility. Hollis and his associates have built a five foot tall, agile, and skinny robot. The robot’s design is to balance itself on top

of a spherical wheel. Hollis compares his robot structure design much like a giant ball pen or a circus clown trying balance on top of a ball [2].

Aside from acknowledging and solving the structure and mobility difficulty, Hollis also faced a major challenge when it came to keeping the self-balancing robot on a stable vertical position. Hollis found that the best way to solve this new issue was by implementing into his design advance sensors and control algorithms. The sensors that he incorporated included a gyroscope and an accelerometer. They were set and placed orthogonal to each other. Hollis implemented a Linear Quadratic Regulator (LQR); a control algorithm technique used to keep the Ballbot in a stable vertical state. The LQR is based on optimal control theory. The main objective when using optimal control techniques on a system is to minimize the effort to stabilize the Ballbot in the vertical position.

The Ballbot's incorporates optimal control algorithms. These control algorithms helped increase stability and system robustness. Ballbot major strength is the inertial measurement units used to provide the tilt angle information. The Ballbot's size and inability to climb staircases were obvious weaknesses. Hollis's article "Ballbot" is informative piece of literature that has presented a technological innovation in robotics. It has inspired parts of this master's project. Just like Hollis Ballbot, this master's project two wheeled balancing robot also incorporates the control techniques in order to achieve a vertical stability. After acquiring additional information on both the gyroscope and accelerometer sensors, it was determined that it was the best choice of sensors to be implemented in the self-balancing two wheel robot.

As there are similarities between Hollis's Ballbot and self-balancing two wheel robot, there are also differences. The structures and control algorithms used were a major difference. As stated before, the Ballbot uses optimal control theory to minimize the robot's effort to stabilize. The self-balancing two wheel robot used for this master's project will use classical control theory.

Battlefield Extraction Assist Robot

Self-balancing robots are starting to emerge as a new technology in the area of the military applications. Applications include defusing bombs in the battlefields and extracting wounded soldiers from hostile areas. For the past few years, the U.S. government has contributed millions of dollars towards the research of life size humanoid robots. An emerging company named VECNA promotes the research and development humanoid like robots for the battlefields of the future. VECNA has successfully developed a prototype humanoid like robot called BEAR. BEAR stands for Battlefield Extraction Assist Robot. The BEAR project is proof that the research and development VECNA has provided towards this technology has been fruitful. In the near future, robots will assist humans in battlefields. As stated by Tom Atwood from Robot Magazine, "The Battlefield Extraction-Assist Robot, or BEAR, is an extremely strong, extremely agile robot roughly the size and shape of an adult male human. It is designed to safely lift humans, carry them, and put them down. Specifically, it is built to rescue human casualties from dangerous areas and take them back to safety. More generally, it is designed to lift heavy objects, carry them for long distances as needed, over obstacles such as stairs or rough terrain, and set them down safely " [3]. Tom Atwood also states

that, “ With initial funding from TATRC, the Army's Telemedicine and Advanced Technology Research Center, the BEAR's primary mission was straightforward, if not simple: Enter a battle zone, find wounded soldiers unable to rescue themselves, and bring them to safety. As the BEAR is developed, the elite team of researchers and engineers building it, as well as representatives from the government and military-medical colleagues at TATRC and elsewhere, are discovering many important new applications for the BEAR” [3]. VECNA does not want to limit this technology to battlefields; they also envision the humanoid robot BEAR in assisting hospital staff to safely move patients. BEAR has been modeled like the Ballbot; as a classical inverted pendulum. The advanced controls algorithms that are implemented into BEAR are a company trade secret. The magazine only briefly describes some advanced features that BEAR possesses. Some of the BEAR’s important features include; Motion control systems, gyroscope & accelerometers to enable dynamic balancing, and the use of hydraulics to lift heavy loads. The hydraulic systems allow the BEAR to lift up to 260 pounds on each arm. Common characteristics that are shared by the BEAR and the autonomous self-balancing two wheel robot are that they are both driven on two wheels and they both maintain their balance by incorporating gyroscopes and accelerometers into an advanced control system. The only difference that was determined from the available information is that the BEAR and the two wheeled robot operate on different control techniques.

Major strengths the BEAR possesses are its rigid body structure and its mobility making it an ideal humanoid like robot to be used in hostile environments. The major weaknesses of the BEAR are its size and weight of the robot. If for any reason the

BEAR malfunctions and loses its up right stability, the person being carried by BEAR can get seriously injured.

nBot and Legway

Two wheel balancing robots have also gain popularity among hobbyists and engineering students. Examples of such popular two wheeled balancing include the nBot and the Legway. The two wheeled robot platforms have drawn high interest from the robot enthusiast communities. An example, “nBot is a two-wheeled balancing robot built by David P. Anderson. This robot uses commercially available off the shelf inertial sensors and motor encoders to balance the system” [7, pg. 3]. Such inertial sensors that are used on nBot are an accelerometer and a gyroscope.

“Steven Hassenplug has successfully constructed a balancing robot called Legway using the LEGO Mindstorms robotics kit. Two Electro-Optical Proximity Detector sensors are used to provide the tilt angle information” [5, pg. 3]. The controller is programmed in high level programming language specifically created for LEGO Mindstorms. Legway uses its two optical proximity detectors to balance the two wheel LEGO robot.

Major strengths of the both the nBot and the Legway are the accessibility and availability of parts and the lower building cost. The fact that these two designs use off the shelf parts with no custom parts make them easier to build and in turn bring down the price. With that in mind, the autonomous self-balancing two wheel robot was also designed to accommodate commercially available parts. The autonomous self balancing two wheel robot presented on this master’s project report will have almost similar design

structure as that of David P. Anderson nBot. A weakness of both the Legway and nBot is the limited environment and terrain that both robots can travel.

The articles on the nBot and Legway briefly explained the control algorithms used to balance and keep the both robots in a stable state. The nBot created by David P. Anderson and the Legway created by Steven Hassenplug are both two wheeled balancing robots can be made from little control theory knowledge. Both robots are modeled after the inverted pendulum.

Segway

In recent years, the use of personal human transport vehicles have gained popularity. The Segway PT is a popular personal vehicle that is available to the public. Invented by Dean Kamen, the Segway PT's dynamics are identical to the inverted pendulum. For added mobility, the Segway is also based on the two wheel platform design. The advanced control algorithms behind the Segway transporter are a company trade secret. The basics of a Segway are computers that process the control algorithms, two tilt sensors, five gyroscopes, and two electric motors. Only three of the five gyroscopes are used to balance the Segway. The remaining two gyroscopes are used as backup. These critical components that make up a Segway are important to keep the vehicle in perfect balance. Current models of the Segway personal transporter can achieve top speeds of 12.5 mph. The Segway is able to navigate thru rough terrain, while successfully carrying a human onto of the platform. The Segway is typically found in urban settings; used for guided tours and city government officials.

The strengths of the Segway are that the personal transporter can be used in outdoor recreation. It is an alternative for people that are unable to walk long distances or ride a bike to enjoy the outdoors without the use of a vehicle. Since the Segway runs on rechargeable batteries, it is environmental friendly. A disadvantage of Segway is its cost which can run in the few thousands of dollars.

In contrast to this master's project on the autonomous self balancing two wheel robot, it has only a cost of few hundred dollars. The autonomous self balancing two wheeled robot presented on this master's project report can implement basic control algorithms similar to the Segway.

Kalman Filter

R.E Kalman published a paper in the early 1960s titled, "A New Approach to Linear Filtering and Prediction Problems." R.E Kalman published his famous paper describing a solution to the discrete data linear filtering problem. Since the paper was published in the early 1960's, the Kalman filter has become widely used in areas of embedded control systems and assisted navigation systems. As stated by both Greg Welch and Gary Bishop, "The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown." [4, p. 1]. That is one of the reasons the Kalman filter was used throughout the NASA Apollo program

back in the 1960s. The filter was used in Apollo spacecraft navigation computers to provide an exact estimate of the position of the spacecraft.

Since its publication the Kalman filter has also gain popularity in others areas of engineering. One area in particular that the Kalman filter is often used is in digital control engineering. The filter is used in control engineering to remove measurement noise that can affect the performance of system under control. It also provides an estimate of the current state of the process or system. As stated in the article, “Kalman Filtering” written by Dan Simon, “The Kalman filter is a tool that can estimate the variables of a wide range of processes. In mathematical terms we would say that a Kalman filter estimates the states of a linear system. The Kalman filter not only works well in practice, but it is theoretically attractive because it can be shown that of all possible filters, it is the one that minimizes the variance of the estimation error. Kalman filters are often implemented in embedded control systems because in order to control a process, you first need an accurate estimate of the process variables” [5, p. 72].

After reviewing the articles on the Kalman filter, one can point out a major advantage of using the filter in the autonomous self balancing two wheel robot, that it can be used to provide a good estimate of vertical angle to control and maintain the robot balance. It can also be used to remove any measurement noise from the gyroscopes and accelerometers. A disadvantage to using the Kalman filter is that there is not a standard methodology or notation for the equations used for the filter; making the use of the filter more complex. Since the original article that first introduced that Kalman filter, various authors on the topic have expressed the filter equations in different ways. This makes it difficult to learn and implement Kalman filter into a project. When that is the case, many

are turned away from using the filter and possibly limit the potential of their project. Another disadvantage in implementing the Kalman filter on an embedded control system is that it tends to load the microcontroller's computing features and performance. This is due to its complex matrix manipulations that are required to run the filter efficiently.

Additional details on the Kalman filter and its implementation in the autonomous self-balancing two wheeled robot will be discussed on the Kalman filter section on this master's project paper.

PID Control System

Control system development is an imperative process to guarantee the success of stabilizing the two wheeled robot. While there is variety of control techniques that can be applied to stabilize the robot, the main objective is to control the robot system effectively and at a low cost without limiting the strength and performance of the controller. The elements that define how a balance control algorithm will be implemented depend on how the system will be modeled and how the tilt sensor data is obtained. A common approach that is often used by two wheeled robot designers is to separate the balancing and position control from the mobile robot.

Control techniques for a control system can be divided into two distinct categories. First technique being a linear control model of the system. While the second category being the nonlinear controller model. A linear control method models the process about a desire operating point. The linear method is usually very sufficient in balancing the system and bringing it to a stable vertical position. On the other hand, a nonlinear controller uses the unrealistic dynamics model of the system in order to design

a controller. Nonlinear controllers would provide a more robust system implementation. The implementation and complexity difficulty associated with the nonlinear method causes most control researchers to utilize the linear controller approach.

The method that will be used to control the self-balancing two wheeled robot will be a linear controller. It will be applied through a Proportional, Integral, and Derivative also refer to as the PID. The PID has proven to be popular among the control engineering community. As stated by the author of article Vance J. VanDoren, “For more than 60 years after the introduction of Proportional-Integral-Derivative controllers, remain the workhorse of industrial process control” [6, p. 1].

CHAPTER III

BALANCING ROBOT SYSTEM OVERVIEW

The self balancing two wheeled robot system built as part of the master's project requirement for Kalman filter and PID controller experimentation. The design of the system will be kept as very straightforward as possible without affecting the final goal of the project to achieve stability. A microcontroller, DC motors, and inertial sensors will be used to meet the objective of balancing a two wheel robot.

Robot Physical Structure

The two wheeled robot structure is a very simple design. The robot chassis design is based on three 6.25in. x 7.25in Plexiglas sheets. The sheets are stacked on top of one another with treaded spacers that are adjustable. These spaces in between each Plexiglas allow the electrical components and hardware to be arranged for easy access. The height of the overall robot chassis including the wheels is about 12.5in. Threaded steel rods serve as standoffs to separate the Plexiglas sheets. The two independent driven DC motors are screwed on to L- aluminum brackets. The aluminum brackets are screwed on to the lower plexiglass sheet and attach the motors to the robot chassis. The two wheels attached to the motors shafts are made of high friction coefficient black rubber material to prevent accidental slipping. The high traction wheels make it possible for the

robot to have higher friction coefficients. The general physical robot platform is shown on Figure 1.

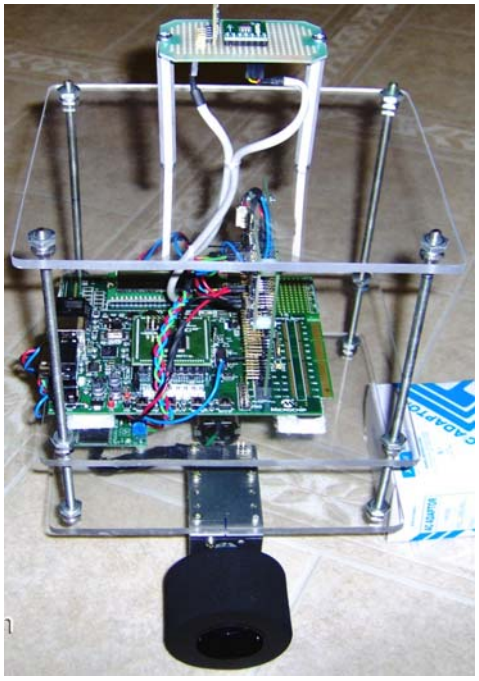

Fig. 1. Two wheeled robot platform.

PIC32 Microcontroller

The intelligence behind the two wheeled robot system is implementing the popular PIC32 microcontroller. The PIC32 is a 32-bit programmable microcontroller that has many unique onboard features and peripherals. The PIC32 can be programmed in either assembly or C. The PIC32 microcontroller programmed in C and serve as the digital controller to control the two wheeled robot. The specific PIC32 model that was used for this master's project is the PIC32MX360F512L. The PIC32 development board designated "Explorer 16" from Microchip technologies. The development board is

attached to the middle Plexiglas sheet as shown on Figure 1. Some of the features and peripherals onboard the PIC32MX360F512L include;

- MIPS32® M4K™ 32-bit Core with 5-Stage Pipeline
- 80 MHz Maximum Frequency
- 512K Flash Memory (User Program ROM)
- 32K SRAM Memory
- Two USART Modules
- Five 16-bit Timers/Counters
- Five PWM Outputs
- Up to 16-Channel 10-bit Analog-to-Digital Converters

The PIC32 microcontroller will be set to operate at a clock speed of 80 MHz. The PIC32 microcontroller supports double-precision floating-point numbers. The robot will use two of the 10-bit analog to digital converters to extract analog data from both the gyroscope and accelerometer. PWM outputs will also be used to drive both of the DC motors. A 16-bit timer will be used to create the require sampling time to run the control algorithm. Finally, a onboard USART will be used to log critical system data onto Micro-SD card module. A general block diagram showing how the components of the robot are interconnected to the PIC32MX360F512L is shown on Figure 2.

Sensors

The two important sensors that were implemented into the autonomous self balancing two wheel robot system include the gyroscope and accelerometer. Both of these sensors will provide the two wheel robot angular and angular rate data. This data

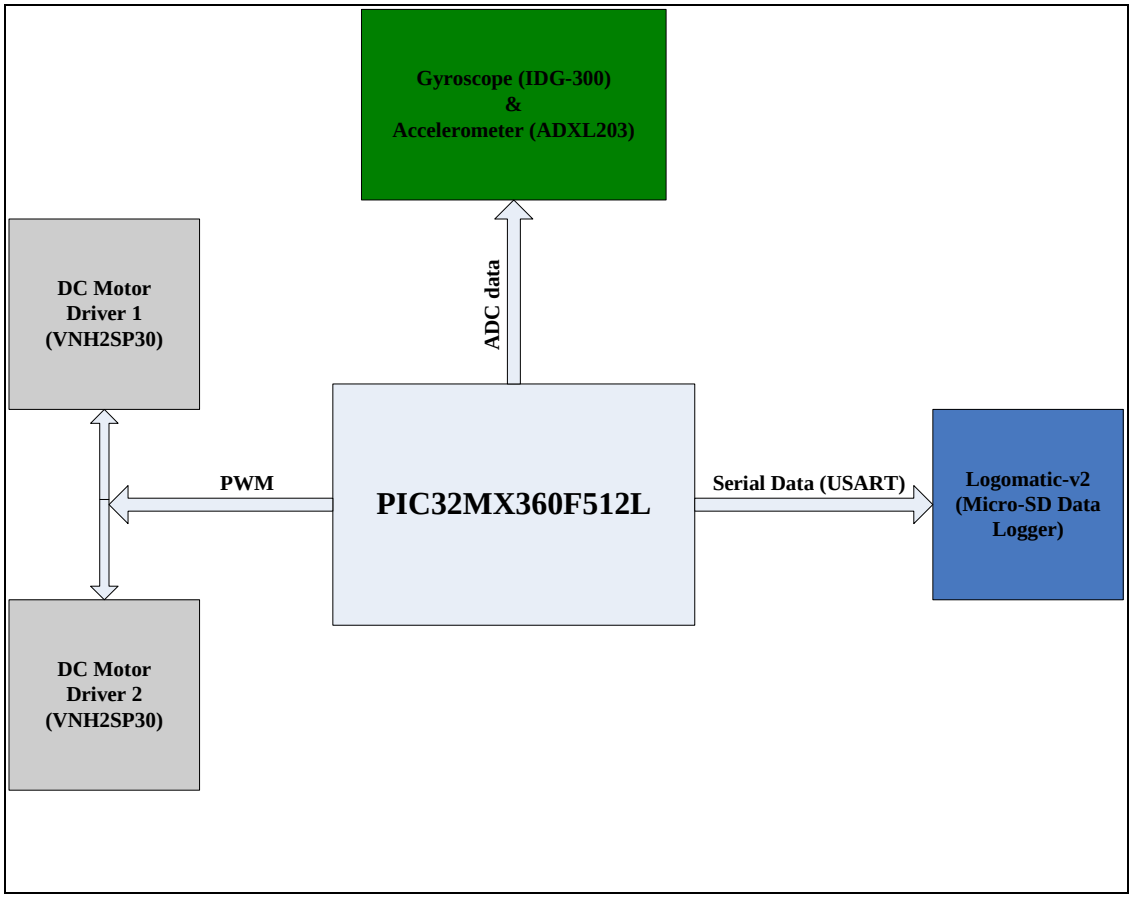

Fig. 2. Two wheeled robot general system block diagram.

will then be fed into the control algorithms to process. The data from the sensors will give the robot a sense of angle displacement from true vertical zero angle. Both sensors will be fused together thru the use of a Kalman filter to remove measurement noise and provide an excellent estimate of the angle. The estimated angle after passing thru the Kalman filter will be used by the digital control system to keep the two wheel robot stable.

Accelerometer

The accelerometer is a sensor will be used to measure both dynamic and static accelerations. For this master's project, the accelerometer of choice is the ADXL-203

from Analog Devices. The X_{out} voltage output on the ADXL-203 will be the only output that will be used on this project. The ADXL203 is a high precision, low power, dual axis accelerometer with analog voltage outputs. The voltage outputs of the ADXL-203 accelerometer will be connected to an analog to digital converter to transmit the data onto the microcontroller. The ADXL-203 will be used to measure static acceleration that in turn will be used as a tilt sensor to report angles. The static acceleration measure is referenced to 1g ($9.8m/s^2$). A block image of the accelerometer sensor is shown on Figure 3.

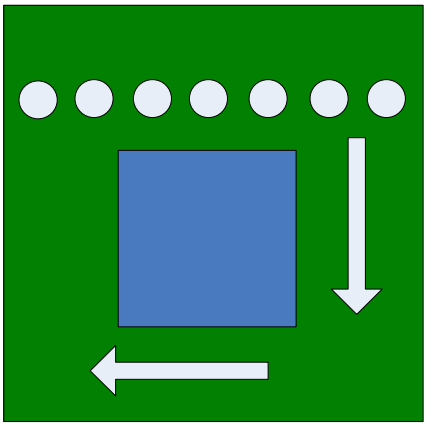

Fig. 3. ADXL-203 accelerometer.

When the ADXL-203 accelerometer is used as a tilt sensor, it's most sensitive to tilt when the sensor is perpendicular to the force of Earth's gravity. The output provided by the ADXL203 is an analog voltage, proportional the tilt. When the accelerometer is at level, its output is half of its input voltage (V_{cc}=5V). As its tilt angle varies, the ADXL-203 voltage varies proportionally with the angle. Figure 4 shows how

the accelerometer voltage output varies with tilt and orientation. Figure 4 demonstrates how the accelerometer voltage output varies with tilt and orientation.

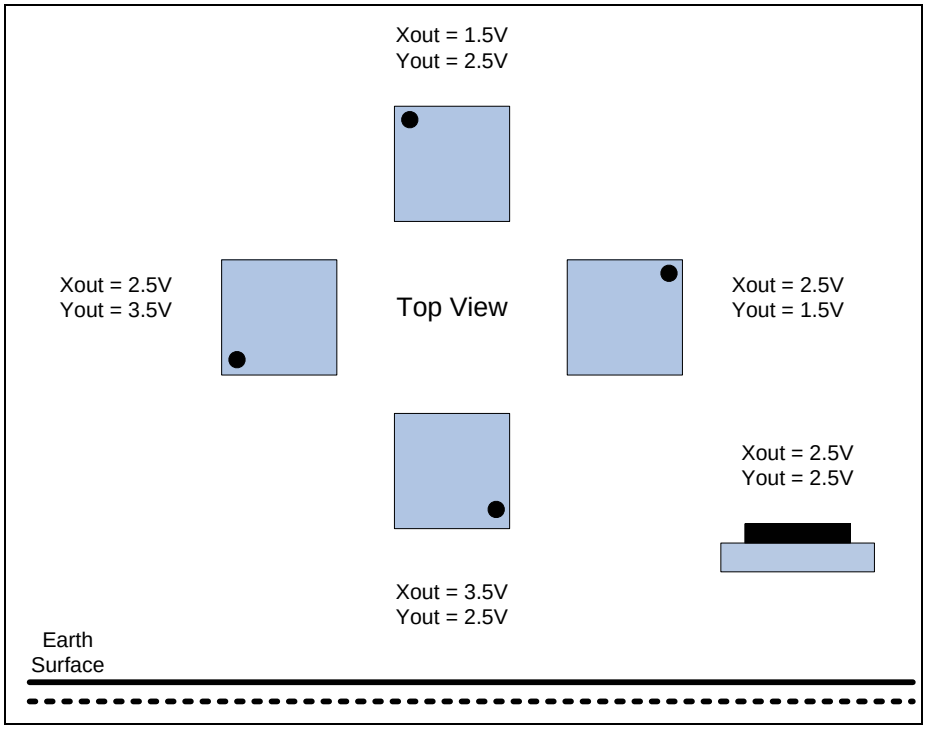

Fig. 4. ADXL-203 Output voltage response vs. orientation.

Some of unique features of the ADXL-203 accelerometer include;

- Precision: +/- 1.5g single/dual axis
- Sensitivity: is 1000 mV/g
- 0g Offset Voltage at X_{out}/Y_{out}= 2.5V
- Operating Voltage: V_{cc}=5V

To calculate the tilt angle from the ADXL-203 voltage output, the following steps need to be executed. Steps 1-5 are coded into the PIC32 microcontroller software to get the measured angle from the accelerometer.

Assume: V_{cc}=3.3V, A=1g, nbits=10, V_{offset}=2.5V

$$(1)\ \frac{(\mathrm{Vcc})}{2^{\mathrm{nbits}}} = \frac{(3.3\mathrm{V})}{2^{10}} = 0.003223\ \mathrm{mV}/\mathrm{bit} = ADC_{resol}$$

$$(2)\ \frac{(1g)}{1000mV}\left(ADC_{resol}\right) = 0.003223\ g/bit = Accelerometer_{resol}$$

$$(3)\ \frac{(V_{offset} \bullet 2^{nbitd})}{Vcc} = \frac{(2.5V \bullet 1024)}{3.3V} = 776 = Zero\_g\_offset$$

$$(4)(ADC_{result} - Zero\_g\_offest) \bullet (Accelerometer_{resol}) = A_X$$

$$(5)\ \sin^{-1}\left(\frac{A_X}{A}\right) = pitch(radians)$$

Gyroscope

The gyroscope is another important inertial sensor that is needed to achieve successful balancing control of two wheel robots. The gyroscope measures angular rate and determines how fast the two wheel robot is falling in either side. The units of angular rate can be expressed as either degrees/sec or radians/sec. For the autonomous self balancing two wheel robot, the gyroscope of choice that was implemented onto the project was the IDG-300. Figure 5 shows the IDG-300 gyroscope. When the IDG-300 gyroscope is not moving the analog outputs will output a voltage value of 1.5V. Only one of the analog outputs (Y_{out}) of the IDG-300 was connected to one of the onboard analog to digital converters on the PIC32 microcontroller. The microcontroller will read the analog voltage from gyroscope output. The features on the IDG-300 include;

- Dual Axis Outputs X_{out}/Y_{out}
- Full Scale Range: +/- 500 °/s
- Sensitivity: 2.0 mV/°/s

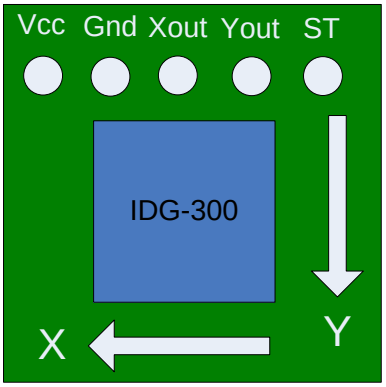

Fig. 5. IDG-300 gyroscope.

- Static Output (Bias): 1.5V
- Operating Voltage: V_{cc}=3.3V

To calculate the angle rate from the IDG-300 voltage output, the following steps need to be executed. Steps 1-5 are coded into the PIC32 microcontroller software to get the measured angle rate from the gyroscope.

Assume: V_{cc}=3.3V, nbits=10, V_{static}=1.5V

$$(1)\ \frac{(\mathrm{Vcc})}{2^{\mathrm{nbits}}} = \frac{(3.3\mathrm{V})}{2^{10}} = 0.003223\ {}^{\mathrm{mV}}\!/_{\mathrm{bit}} = ADC_{resol}$$

$$(2)\ \frac{(ADC_{resol})}{Gyro\_Sensitivity} = \frac{(0.003223\ {}^{\mathrm{mV}}\!/_{\mathrm{bit}})}{2mV\ /^{o}/\ s} = 1.61133\ {}^{\mathrm{deg}\,ree}\!/_{\mathrm{sec}} = Gyro_{resol}\left({}^{\mathrm{deg}}\!/_{s}\right)$$

$$(3)Gyro_{resol}\left(\frac{\pi}{180}\right) = \left(1.61133\ {}^{\mathrm{deg}\,ree}\!/_{\mathrm{sec}}\right)\left(\frac{\pi}{180}\right) = 0.028123\ {}^{radians}\!/_{\mathrm{sec}} = Gyro_{resol}\left({}^{rad}\!/_{s}\right)$$

$$(4)\ \frac{(V_{static} \bullet 2^{nbitd})}{Vcc} = \frac{(1.5V \bullet 1024)}{3.3V} = 396 = Zero\_gyro\_offset$$

$$(5)(ADC_{result} - Zero\_gyro\_offest) \bullet \left(Gyro_{resol}\left({}^{rad}\!/_{s}\right)\right) = Angular\_rate\left({}^{rad}\!/_{s}\right)$$

DC Motors

In order for the autonomous self balancing two wheel robot to remain in a vertical stable state, selection of good DC motors is important. DC motors with high torque output and fast RPM make them ideal to be used on a two wheel robot system. Selection of proper DC motors was an important great consideration for the robot system used as part of this master's project. The motors used for the autonomous self balancing two wheel robot are the popular Tamiya Gear head 380k75 DC motors. Tamiya motors are very popular in control remote RC cars, because of there high torque output and high RPM. Figure 6 displays the motor with the foam wheel attached. Characteristics of the Tamiya gear head 380k75 DC motors include:

- Gear Ratio: 75:1
- Max. Voltage Supply: 7.2V
- RPM at no Load: 246 RPM
- Torque at Best Efficiency: 0.49N·m (0.4949kg·m)
- Running Speed: 1m per 1.35sec
- Torque Constant: K_{torque}=0.0739 kg·m/A

Logomatic-v2

The Logomatic v2 is a module that allows data to be saved onto a micro-SD memory card. Logomatic serial data logger will be used to capture and save real time control data from the two wheeled robot. The data logger is connected to one of PIC32 microcontroller USART transmit pin. The communication protocol between the data logger module and the PIC32 is standard serial port setup (8 data bits, one stop bit, no

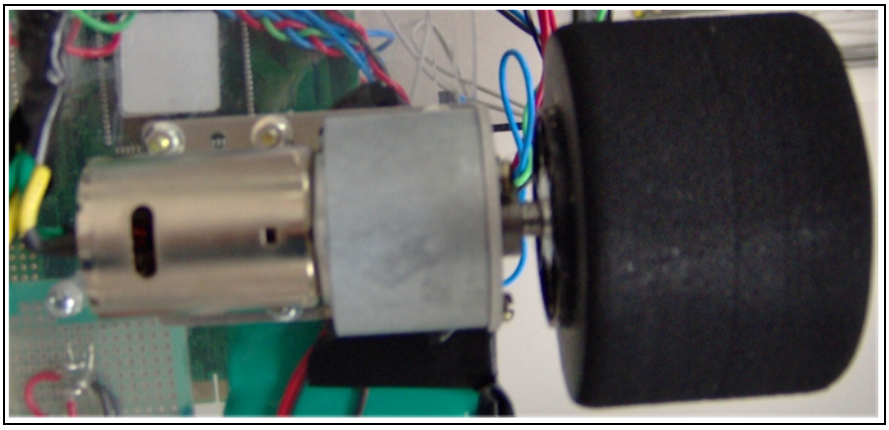

Fig. 6. Tamiya DC Motor with foam wheel.

parity, and data rate). The Logomatic module can handle micro-SD memory cards up to 2 GB. Once the data is saved onto the micro-SD memory card, the card will be placed in a micro-SD carder reader. The data can also be downloaded from the module to a personal computer via USB cable. When the USB cable is attached to the data logger, the computer will see it as a removed mass storage device. The data saved on the memory card will then be used to plot the control system output response using either Microsoft Excel or MATLAB. Figure 7 shows an image of the Logomatic v2 serial micro-SD data logger module. The module can also be used to charge a single cell lithium polymer battery. Sparkfun's website, where the module was purchased, provides a sheet that explains how to properly configure the Logomatic v2 module. The only limitation that comes with using the Logomatic module is the write speed to the micro-SD card. The longest write cycle speed to the micro-SD is 42.5 ms. If logging occurs faster than 42.5 ms, there is a risk for data loss. This can be a major issue if the module is set to communicate with the PIC32 microcontroller at baud rate speeds of 115Kbps. To resolve

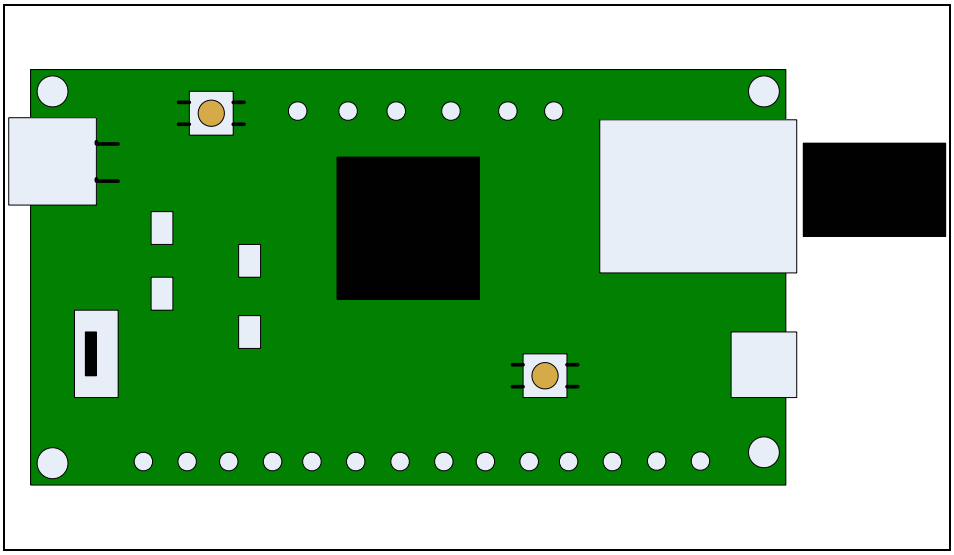

Fig. 7. Logomatic serial data logger.

this write speed issue, the data logger will communicate with the PIC32 at a lower baud rate.

DC Motor Drivers

A set of DC motor drivers are needed to support the Tamiya gear head 380k 75 DC motors, to successfully work on the two wheel robot. The motor drivers are critical in getting any type motor to function properly. The drivers provide the high voltage and current levels outputs necessary to drive the motor. Some available motor drivers have inputs that allow the user to control the motors speed and direction.

The chosen DC motor driver for this master's project is the VNH2SP30 from STMicroelectronics. Figure 8 show s an image of the VNH2SP30 DC motor driver. The VNH2SP30 is a dual DC motor driver that is able to drive two independent DC motors. Each DC motor driver has inputs that vary the motors speed and direction. The motor speed is controlled by a PWM input. Directions are determined by toggling the direction

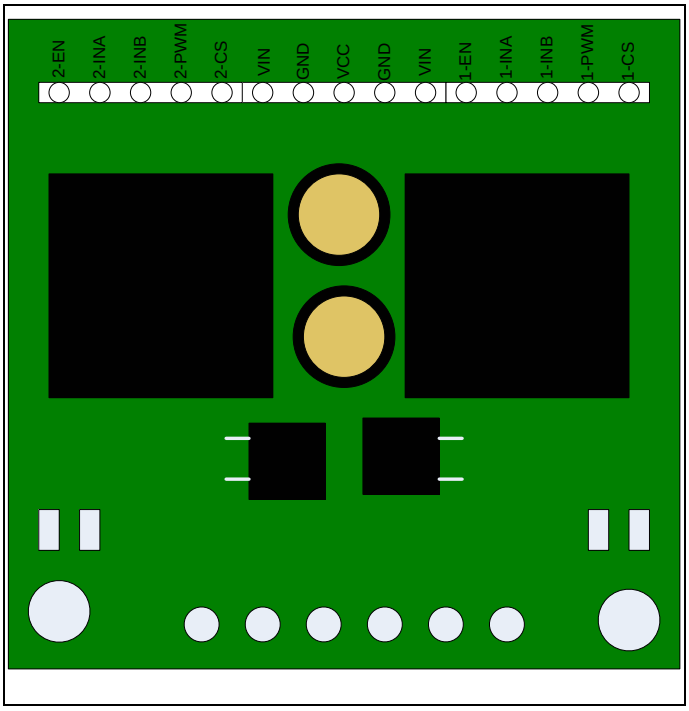

Fig. 8. VNH2SP30 DC motor driver.

inputs to either a 0 (0V) or 1 (5V). Some of the characteristics of the VNH2SP30 DC motor driver include:

- Maximum PWM Frequency: 20 KHz
- Maximum Output Current (continuous): 30A
- 5V Logic Level Compatible Inputs.
- Maximum Operating Supply Voltage: 16V
- MOSFET on-resistance (per leg): 19 mΩ

Power Supply

The power supply is another important component of this project. There are many types of batteries that have different chemical makeup. Different battery types have advantages or disadvantages over one another in terms of power capacity. For the two

wheel robot implementation on this master's project a compact yet with high power capacity battery was chosen. The battery type of choice for this project is the Nickel-Metal Hydride batteries. Ni-MH batteries will provide the required power needed to drive all of the electrical devices on the two wheel robot. The Ni-MH battery to be used on the robot has a power rating of 4.8V at 2000mAh. Physical Nickel-Metal Hydride battery is shown on Figure 9.

Fig. 9. Nickel-Metal Hydride battery.

Three Ni-MH batteries will be used and hooked in a series configuration. When the three batteries are configured in a series configuration, there will be an increase in voltage up to 14.4V. The current will remain constant at 2000mAh. Figure 10 shows an image of a series battery configuration. The two wheel robot will run from +3.3V, 5V, and 9V power supplies. These three voltage values will run the onboard electronics on the robot. The three Ni-MH battery packs will provide the total of 14.4 voltages. The onboard voltage regulators will provide the three stepped down voltage values. When

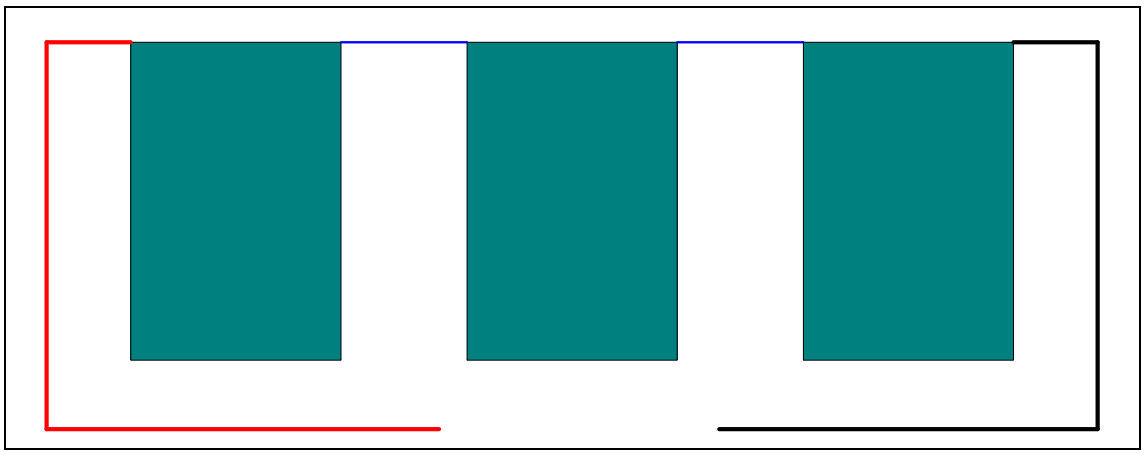

Fig. 10. Nickel-metal hydride cell batteries.

using Ni-MH batteries proper caution needs to be exercised. If not properly charged with the correct Ni-MH charger, the batteries will swell up and can catch on fire.

CHAPTER IV

METHODOLOGY

The following sections will present and provide in depth details on the methods used to implement both the Kalman filter and the PID controller. The methodology section will start talking about the Kalman Filter and its algorithm and corresponding equations. The purpose of implementing the Kalman filter is that it provides sensor fusion for the autonomous self-balancing two wheel robot. The other half of the methodology section will discuss the theory behind the PID controller and its implementation in this project.

Kalman Filter

Since its introduction in the early 1960s, the Kalman filter has being widely used in the control engineering community. The Kalman is a recursive digital filter that provides a very effective means of estimating the state of any process. The Kalman filter can be thought of being a state estimator. Kalman filtering can be used as a tool to provide a reliable state estimate of the process. Another important feature of the Kalman filter is its ability to minimize the mean of the square error and provide a solution for the least square method. The Kalman filter can be used on a control system that is exposed to noisy environments because it minimizes the square error. The filter can reduce noisy measurements from sensors' data before it's fed into any control system. Noisy

measurements from the sensors will not allow the system to reach stability nor get a reasonable output response. As stated by Rich Ooi, “the Kalman filter does not require all previous data to be kept in storage and reprocessed every time a new measurement is taken” [7, p. 13]. Having this filter characteristic, the Kalman filter can be implemented on a microcontroller and in sofware.

Discrete Kalman Filter Algorithm

On this section, both the algorithm and corresponding Kalman filter equations will be discussed. Before the Kalman filter can be used to get rid of noise from a sensor signal. The process or system that is being measured must be modeled by linear system. A linear system can best be described by the following two state space representation equations (6.0) and (6.1).

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{6.0}$$

$$z_k = Hx_k + v_k \tag{6.1}$$

Equation 6.0 represents the process state equation. *A*, *B*, and *H* represent the state matrices, the w_{k-1} represents the process noise, *k* is the time index, x_{k-1} is the state of the process, and u_{k-1} is the known input to the process. Equation 6.1 above represents the output state equation. The z_k represents the measured process output and the v_k represents the measurement noise. The sensor data for the system are taken at discrete time sample points. The measurements are dependent on the state of the process or system.

There is no association between the measurement noise (v_k) and the process noise (w_{k-1}), there are independent random noise variables. The process noise and

measurement noise can be best described by there covariance matrices Q_w and R_v. The covariance matrices are then written as

$$Q_w = E[w_k w_k^T] \tag{6.2}$$

$$R_v = E[v_k v_k^T] \tag{6.3}$$

Where T represents a matrix transpose and E represent the expected estimated value. Both the process noise covariance (Q_w) matrix and the measurement noise covariance (R_v) matrix play a vital role in the overall output performance of the Kalman filter. These two matrices values can be either calculated using advance statistics equations or can be manually tuned until the desired filter output response is achieved. The noise covariance matrices (Q_w) and (R_v) can also be adjusted dynamically. But for the autonomous self balancing two wheel robot project, the noise covariance matrices will remain constant and be adjusted manually.

The Kalman filter estimates the process by using a feedback scheme. First the filter estimates the system's state at some time step and then gets noisy measurements in the form of feedback. Therefore, the equations for the Kalman filter fall into two categories, the time update equations and measurement update equations. The time update equations can be thought as the predictor equations. The measurements update equations can also be thought as corrector equations. These two equation types form the basis for the Kalman filter algorithm, which resemble a predictor-corrector estimation algorithm. Figure 11 shows the Kalman filter predictor-corrector estimation algorithm cycle.

Since the Kalman filter algorithm is based on a predictor-corrector scheme, its equations can now be described based on that. The predict stage displays the current state

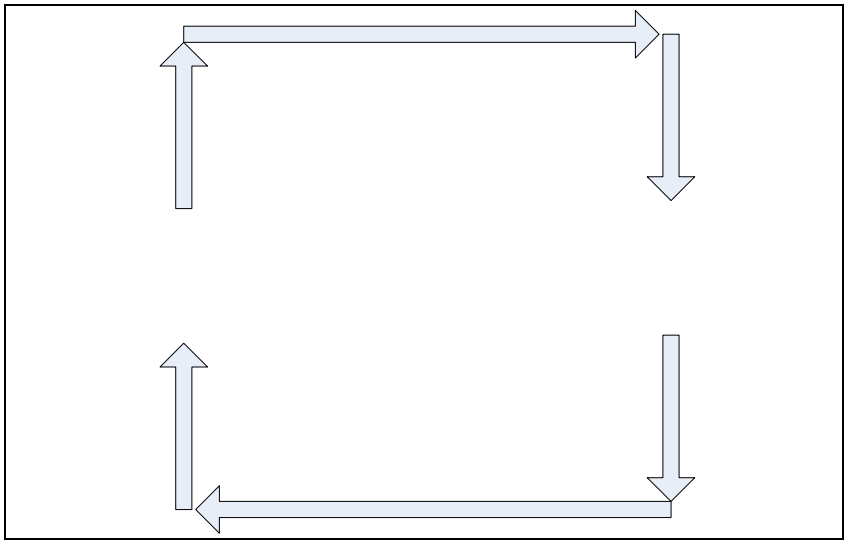

Fig. 11. Kalman filter estimation algorithm cycle.

and error covariance to the next time step. The correct stage is responsible for adjusting the expected estimate from the prediction; by incorporating a new measurement at the current time step. Figure 12 shows the complete Kalman filter algorithm with equations and how it relates to the predictor-corrector cycle.

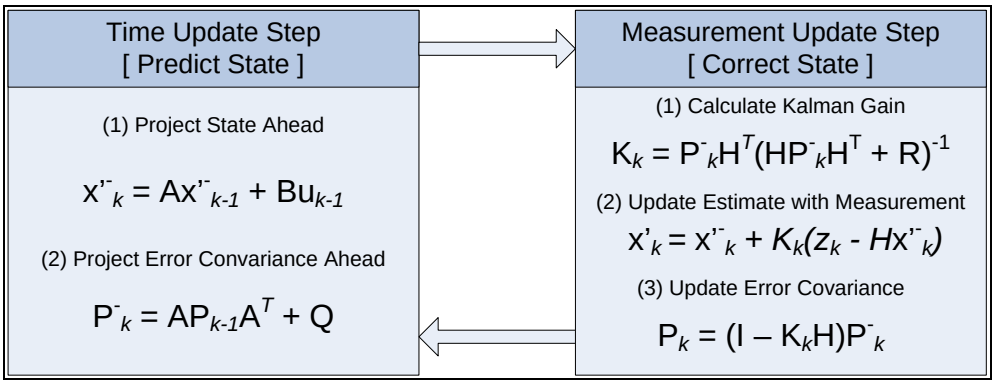

Fig. 12. Kalman filter algorithm equations.

Sensor Fusion Using the Kalman Filter

The reason for using sensor fusion is stated by Rich Ooi as follows, “the accuracy and reliability of information regarding its operating environment for these mobile robots is critical, as these systems are usually autonomous. These requirements call for highly accurate sensors which are very expensive. Sensor fusion technology, where signal from several sensors are combined to provide an accurate estimate, is the most widely used solution. The Kalman filter is used in a number of multi-sensor systems when it is necessary to combine dynamic low-level redundant data in real time” [7, p. 5].

Examples of multi-sensor systems include navigation systems and inertial measurement units. The Kalman, “filter uses the statistical characteristics of a measurement model to recursively determine estimates for fused data that are optimal in a statistical sense. The recursive nature of the filter makes it appropriate for use in systems without large data storage capabilities” [7, p. 5], such as the PIC32 microcontroller.

This section on the sensor fusion, using the Kalman filter, details on the experiment conducted on the inertial sensors used as part of the project. Figure 13 illustrates the test bed for both the IDG-300 gyroscope and ADXL-203 accelerometer. The image shows how an angle finder was used as a visual angle reference on the autonomous self-balancing two wheel robot. The figure also shows how the two wheel robot was programmed using a laptop and a Microchip ICD 2 programmer.

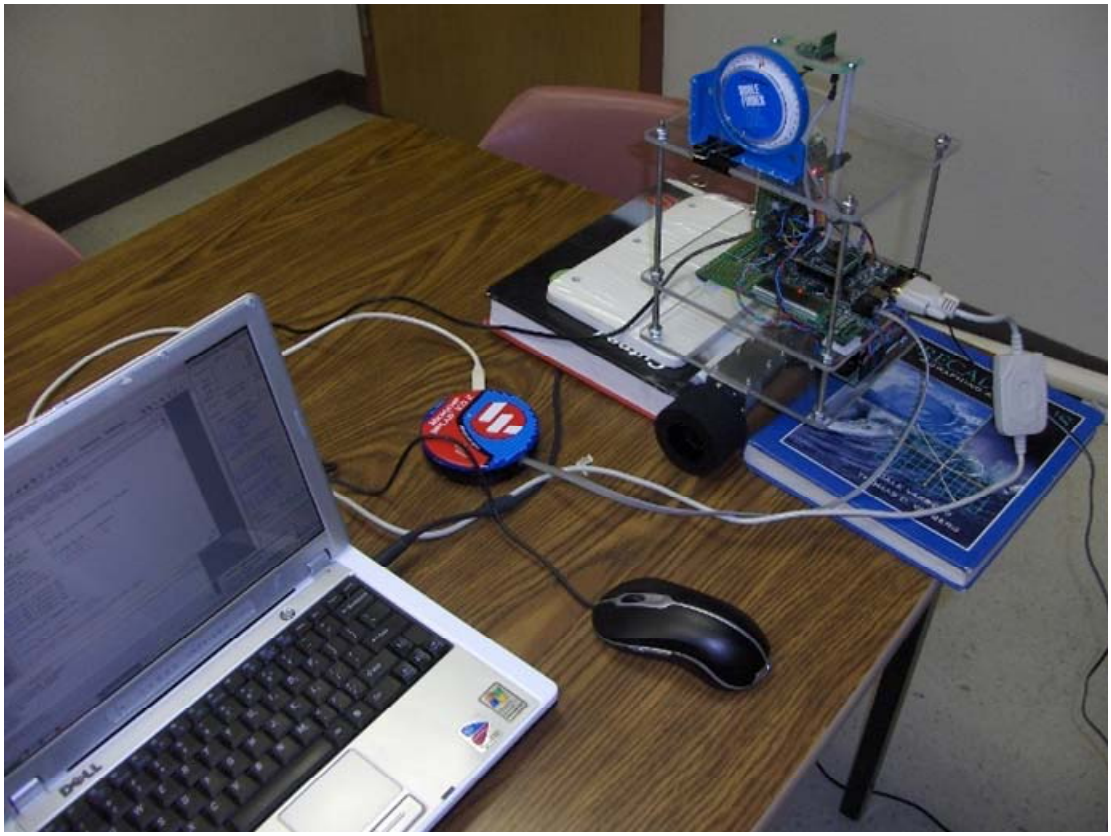

Fig. 13. Gyroscope and accelerometer sensor test setup.

The digital IDG-300 gyroscope can only provide a measure of angular change. Angular change or rate refers to how fast an object is rotating in radians per second. The output of the gyroscope can be considered the derivative of the measured output angle from the accelerometer. The gyroscope tends to have a faster reaction to change as compared to an accelerometer. A very unique feature of the gyroscope is that it has a rest average value, also referred to as a bias. The average value is given when the gyroscope is motionless. The bias has to be corrected at every measurement to get accurate velocity data. Gyroscopes do not hold onto the angular rate output change. When an object is tilted, the angular rate output changes and then quickly returns to its resting average

value. Since a gyroscope does not hold on to the output value, it can't be used as a tilt sensor. One of the major problems that most gyroscopes have is that output average value tends to drift with time when the gyroscope is motionless. The output drift can be cause by changes of device operating temperature or the internal physical properties of the gyroscope itself. This drift can introduce significant errors in the angular measurements. Figure 14 shows the plot of the raw gyroscope data.

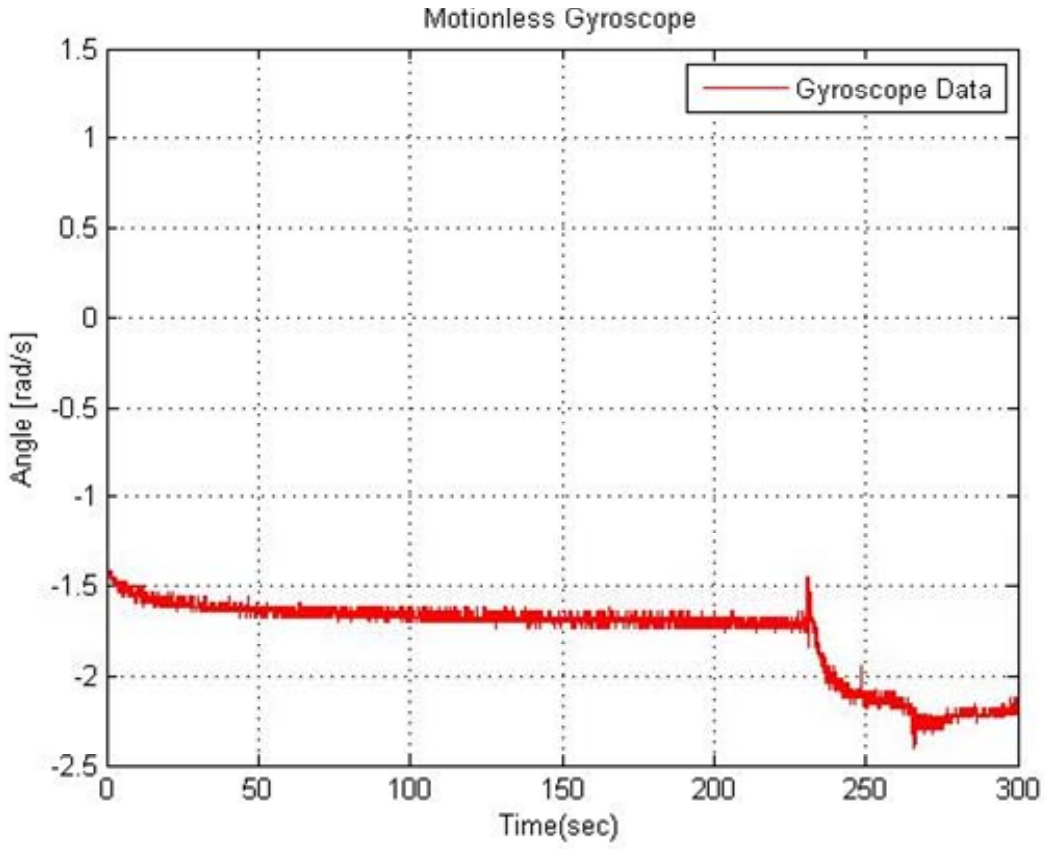

Fig. 14. Gyroscope measurements with drift.

Observed from Figure 14 that as time progresses the gyroscope starts to drift away from its average resting value of -1.45 rad/s.

The ADXL-203 accelerometer is the other sensor that is implemented on this robot project. The accelerometer is a sensor that measures acceleration in a predefined axis. The sensor can provide a handy reference point to determine which way is up versus down orientation. The ADXL-203 accelerometer mounted on the robot was chosen because of it can be used as a tilt sensor. When configured as a tilt sensor, the accelerometer can measure static accelerations. The ADXL-203 accelerometer uses earth's downward gravity force of 1G as its reference. So the sensor has a range of +/- 1G, which can be converted to a range in degrees of +/- 90^{o}. An advantage of using the ADXL-203 accelerometer as a tilt sensor is that it can hold on to its output angle value. It will remain at that angle until it gets disturbed by an external force. Since the ADXL-203 accelerometer can be used as a tilt sensor, it seems that only sensor that is needed. There are some issues that may arise if the accelerometer is only sensor used as the input sensor to the robot control system. One issue that most accelerometer sensors have is that the output angle tends to have a very slow reaction to change. Another problem is that the sensor is very sensitive to noisy environments and vibrations. The accelerometer output is usually corrupted with noise. Figure 15 shows the results of non filtered accelerometer raw measurements.

From Figure 15, it can be observed that an accelerometer used alone on the robot cannot get a reliable tilt angle data. The figure also indicates the need to implement a Kalman filter on the robot.

Since both the gyroscope and accelerometer have problems in the outputs, the sensors used alone can't provide very reliable data. If the sensor information is not reliable, the two wheel robot will have problems achieving a stable state. To overcome

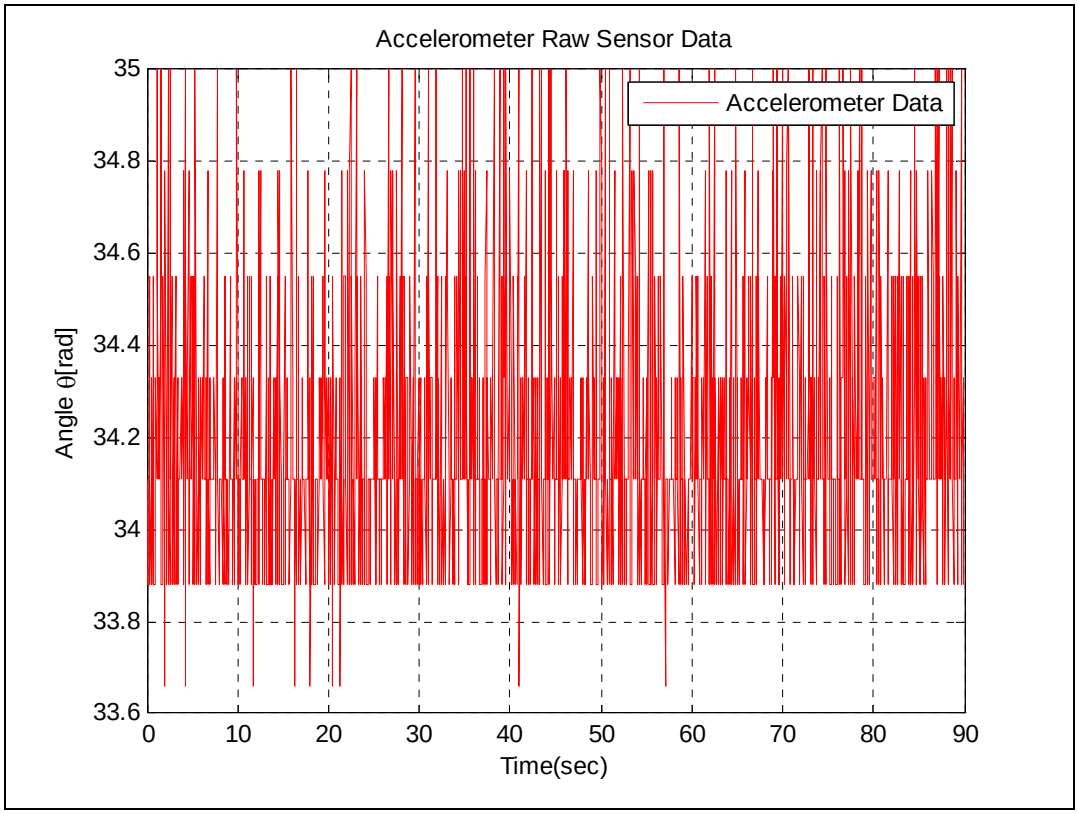

Fig. 15. Noisy accelerometer measurements.

the gyroscope and accelerometer output data problems, a signal level sensor fusion technique can be used. Sensor fusion can be achieved thru the use of a Kalman filter algorithm. This fusion technique combines the output signals of the sensors with the objective of providing an output signal in the same form as the original signal, but with better quality. The sensor fusion was used on this project to provide means to overcome some of problems associated with the sensors outputs. In this case, the ADXL-203 accelerometer is used to eliminate the drift problem from the gyroscope signal. The IDG-300 gyroscope eliminates the corrupt accelerometer output data by using its clean stable resting average value. As a result of using the sensor fusion with the Kalman filter, a proper estimate of the tilt angle and its derivative term is obtained.

Kalman Filter Process Model

In order for the Kalman filter to successfully be implemented onto the two wheel robot, an accurate model needs to be developed. The process also needs to be modeled as a linear system for Kalman filter implementation to be successfully. As mention in the previous section the Kalman filter will use the data from the accelerometer to eliminate the drift problem from the gyroscope output signal. In the process of using the filter, unwanted noise from the accelerometer will either be minimized or completely eliminated. The Kalman filter process model will be modeled as a single dimensional inertial measurement unit. For the single dimensional inertial measurement unit, a two state Kalman filter is implemented to track the angle of the two wheeled robot and gyroscope bias value. The simple process model using the gyroscope input data can be modeled in state space representation as shown on equation 6.4.

$$x_k = Ax_{k-1} + Bu_{k-1}$$

$$\begin{bmatrix} angle \\ dot\_angle \end{bmatrix}_k = \begin{bmatrix} 1 & -dt \\ 0 & 1 \end{bmatrix} \begin{bmatrix} angle \\ dot\_angle \end{bmatrix}_{k-1} + \begin{bmatrix} dt \\ 0 \end{bmatrix} u_k \qquad (6.4)$$

$$\begin{bmatrix} \theta \\ \theta* \end{bmatrix}_k = \begin{bmatrix} 1 & -dt \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \theta* \end{bmatrix}_{k-1} + \begin{bmatrix} dt \\ 0 \end{bmatrix} u_k$$

The single dimensional inertial measurement unit process model will be used to implement the Kalman filter algorithm into the two wheel robot.

PID Controller

The control algorithm that was used to maintain it balance on the autonomous self-balancing two wheel robot was the PID controller. The proportional, integral, and derivative (PID) controller, is well known as a three term controller. Since its introduction, more than sixty years ago, the PID controller has been the popular choice in industrial process control. Due to the simplicity of the controller, it has become the basis for many advanced control algorithms and strategies. The PID controller was first introduced as a purely mechanical controller. Being used as a mechanical controller, it saw its first mechanical implementation on controlling pneumatic systems. After being successfully being used to control mechanical systems, it started to be implemented in electrical analog circuits. PID implementation in analog circuits has made it possible to control systems such as house heaters to chemical process plants. As microprocessors and microcontrollers have become popular in control engineering, the PID controller has become a popular embedded software implementation. This PID controller being implemented in software has out performed the analog and mechanical versions of the controller. The controller can now be programmed onto a single integrated circuit chip.

PID Controller Algorithm

In this section the PID controller algorithm's software implementation on a microcontroller and tuning will discussed. The PID control algorithm is a very straight forward algorithm that provides the necessary output system response to control a process. One unique feature of the PID controller is that it is capable of manipulating the process inputs based on the history and rate of change of the signal. The algorithm is best suited when the process under control is modeled as a linear system. This gives more

accurate and stable control. The PID controller consists of proportional, integral, and derivative terms. The terms of the PID controller are summed up to create a controller output signal. Each term performs a different task in the control process. The controller terms also have different effects on the system output response. Figure 16 shows the classical continuous time PID controller.

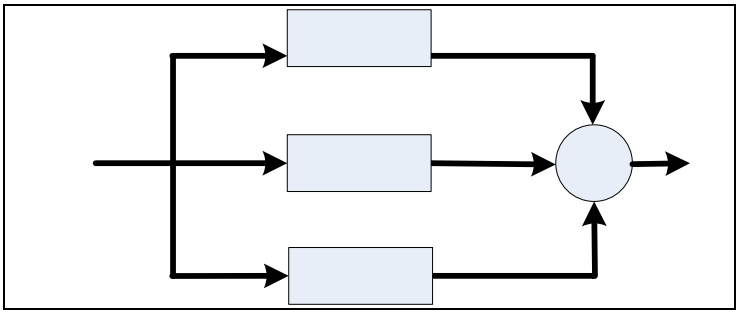

Fig. 16. Continuous PID controller.

As shown from the figure, the input to the controller is the *error* from the system and the *u* is the output controller signal. The K_P, K_I, and K_D are referred as the proportional, integral, and derivative constants. The PID controllers are widely used on closed loop control systems, where the process output measurement is fed back to the system and gets processed by the controller. Figure 17 shows the integration of the PID controller on a general close loop control system.

The closeed loop control system shown on figure 17 is also referred to as a negative feedback system. The basic idea of a negative feedback system is that it measures the process output *y* from a sensor. The measured process output gets subtracted from the reference *set-point* value to produce an *error*. The error is then fed into the PID controller, where the error gets managed in three ways. The error will be

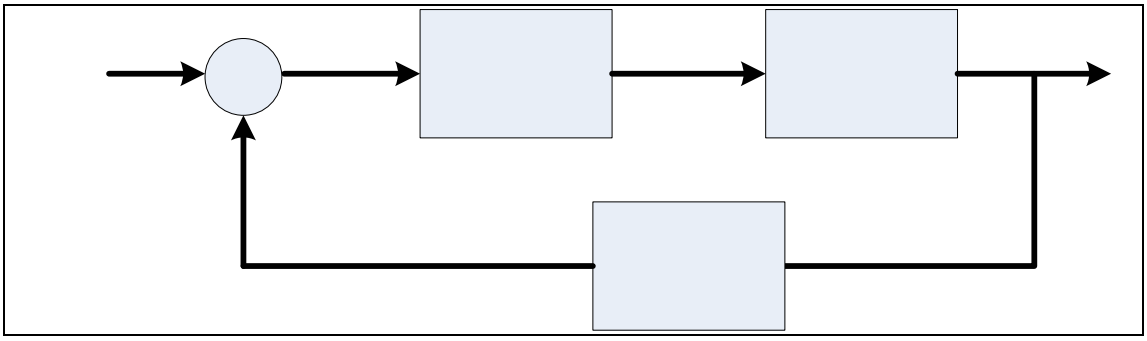

Fig. 17. Closed loop control system with PID controller.

used on the PID controller to execute the proportional term, integral term for reduction of steady state errors, and the derivative term to handle overshoots. After the PID algorithm processes the error, the controller produces a control signal *u*. The PID control signal then gets fed into the process under control. The process under PID control is the two wheeled robot. The PID control signal will try to drive the process to the desired reference set-point value. In the case of the two wheel robot, the desired set-point value is the zero degree vertical position. The PID control algorithm can be modeled in a mathematical representation. Figure 18 shows the algorithm in its time domain mathematical form.

Equation 6.7 represents the continuous time domain of the PID controller. Both the T_D and T_I represent the time constants for both the derivative and integral terms. Since the PID controller algorithm will be implemented on a PIC32 microcontroller, equation 6.7 can't be programmed in the time domain. The solution to implementing the PID controller will be discussed later in the section.

The K_P, K_I, and K_D will then be examined to show how the three term gain constants affect the system output response performance. The first gain term to be examined is the proportional K_P constant. Setting both the K_I and K_D term gains from

$$error = ref - y$$

$$u(t) = K_P\left(error(t) + \frac{1}{T_I}\int_0^t error(t)dt + T_D\frac{derror(t)}{dt} \right)$$

where

$$K_I = \frac{K_P}{T_I}$$

$$K_D = K_P T_D$$

$$u(t) = K_P error(t) + K_I\int_0^t error(t)dt + K_D\frac{derror(t)}{dt}$$

Fig. 18. PID algorithm in time domain representation.

equation 6.7 to zero, gives a proportional control algorithm. The proportional K_P gain gives a process control that is the proportional to the error. The error of the system is multiple by the K_P gain. One of the major benefits of the K_P gain is that it improves the rise time of the system. An improved rise time gives the system under PID control a faster recovery response from a disturbance. The disadvantages of using only the K_P gain is that a K_P gain that is too high of a value will cause system to be unstable A low K_P gain will cause the system to drift away and never reach the reference set point value.

The next PID gain to be examined is the integral K_I term constant. Using only the integral K_I gain in the system will never achieve the desire system output response. The system will have a slow response to system disturbances. It will also cause the system to oscillate and become unstable. In common practice the integral K_I gain is used together with the proportional K_P gain. The resulting control algorithm is the PI, where the K_D gain term in equation 6.7 is set to zero. The unique feature of the integral gain is

that it provides a way to sum or add the system errors. The summing of the process errors will continue until the output of the process reaches the desire set point value. Including the integral K_I gain along with the K_P gain helps eliminate or reduce the steady state error. The steady state error is the offset that can occur between the desired value and the process output over time. The major disadvantage of the integral K_I gain in the control algorithm is that it is subject to integral windup or integral saturation. Integral windup is caused when there is to large of an error between the reference set-point value and the measured process. A large error might come from a disturbance that lasts a long time which the system can not handle. The disturbance causes the integral gain term to add the error continuously. Integral windup prevents the error signal from reaching a steady state zero value. A quick solution to prevent integral windup on a PI or PID controller is to set the integral term between a maximum and a minimum value. Once the integral term reaches the maximum and minimum value, the integral term gets reset to a predefined value.

The derivative K_D gain term is mostly used along with the controller algorithm. The derivative K_D gain is commonly used with a proportional K_P gain which makes up the PD control algorithm. The PD algorithm is derived by setting the K_I gain term in equation 6.7 to zero. The derivative gain term can also be used on the full PID controller. When used on a PD or PID controller, the derivative term can help increase the system's rise time. The main function of the derivative term is used to improve the response to a sudden change in the system's state. The derivative term also improves the stability of the system, reduces the overshoot, and improves the transient response. The K_D gain term can be thought as a damper on the system's output. The damping effect can

help the system reach stability more rapidly. The major drawback of using the derivative term is that it is subject to signal noise and that in a control system an improper K_D value can lead to system instability. It is possible for the noise to come from noisy sensor measurements. If signal noise becomes an issue, a software or hardware filter needs to be adapted. A Kalman filter may be implemented in software to reduce the signal noise. Figure 19 shown below summarizes the effects of the proportional, integral, and derivative terms on the system output response.

Fig. 19. PID controller characteristics.

When using the PID control algorithm to control a system, the control engineering designer needs to be aware that not all parts of the system that need to be controlled need the full three term PID controller. If the system has a good output response with only the PI or PD controllers, there is no need to implement the full PID control algorithm. When using only the PI or PD controller, the implementation of control algorithm will be kept simple.

Discrete PID Controller Implementation

Implementing the PID control algorithm in the continuous time domain, as shown in equation 6.7, isn't practical in industry. Trying to develop a PID software

algorithm that follows the time domain notation is impossible. A common approach used to implement the PID algorithm in software is to approximate the PID terms in equation 6.7. The approximation of the controller terms is achieved when the PID terms are rewritten in discrete form. When the PID control algorithm is written in discrete form, software implementation of the PID controller is possible. The discrete representation of the controller makes it easier to implement the PID algorithm on either a microcontroller or microprocessor. Figure 20 shows the discrete representation of the PID control algorithm.

$$\int_0^t error(t)dt \approx T_S \sum_{k=1} error(k)$$

$$\frac{derror(t)}{dt} \approx \frac{error(k) - error(k-1)}{T_S}$$

$$u(k) = K_P\left(error(k) + \frac{T_S}{T_I}\sum_{k=1} error(k) + \frac{T_D}{T_S}(error(k) - error(k-1)) \right)$$

where

$$K_I = \frac{K_P T_S}{T_I}$$

$$K_D = \frac{K_P T_D}{T_S}$$

$$u(k) = K_P error(k) + K_I \sum_{k=1} error(k) + K_D\left(error(k) - error(k-1)\right)$$

Fig. 20. PID controller discrete representation.

Figure 20 shows how the continuous time domain PID control algorithm can be approximated and represented in discrete form. The *T* represents the sample time while the *k* represents the sample number. Both equations 6.8 and 6.9 show how the PID terms can be approximated in discrete form. The final discrete form, of the PID control algorithm that will be implemented in software is shown in Equation 6.11.

Ziegler-Nichols PID Tuning

In order for the closed loop system to work correctly, the PID controller must be correctly tuned. The constant values for the PID controller play a very vital role in the system output response. To find the PID constant gain values, a tuning procedure on the PID needs to be carried out. The sample time in the algorithm is also another important parameter when tuning the PID controller. There are numerous ways to tune the PID controller, to achieve desire output response. The widely used tuning algorithm is the Ziegler-Nichols which was introduced back in the 1940s. The Ziegler-Nichols PID tuning algorithm is the preferred choice to manually tune any PID controller used in industry today.

To carryout the procedure to tune the PID controller using the Ziegler-Nichols is very straight forward. Since the self balancing two wheel robot is classified as a closed loop system. The close loop form of the Ziegler-Nichols tuning algorithm will be used to find the controller gain values. The following steps describe the procedure needed to apply the closed loop PID tuning method.

- First both the integral K_I and the derivative K_D gains from equation 6.7 need to be set to zero. The controller will have only the proportional K_P gain action on the closed loop.

- Second carry out a set point test on the system and observe the system output response.
- Repeat the set point test by either incrementing or decrementing the proportional K_P gain, until a stable oscillation is achieved at the output.
- The stable oscillation observed at the system output needs to be of equal amplitude and period. The gain that produced the output oscillation is called the ultimate gain K_U.
- Read and record the period of the steady output oscillation. This period is the ultimate period P_U.

Once the ultimate gain K_U and the ultimate period P_U are recorded, the final step is to calculate the PID controller gain values. Figure 21 lists the Ziegler-Nichols tuning formulas used to calculate the PID controller gain values.

Controller	K_P	T_I	T_D
P	$0.50*K_U$	-	-
PD	$0.65*K_U$	-	$0.12*P_U$
PI	$0.45*K_U$	$0.85*P_U$	-
PID	$0.65*K_U$	$0.50*P_U$	$0.12*P_U$

Fig. 21. Ziegler-Nichols PID tuning parameters.

There are four different types of controller configurations that can be used with the Ziegler-Nichols tuning algorithm as shown in table 6.1. The time constants T_I and T_D are referred to as the integral and derivative time constants. Both of these time constants are used to calculate the K_I and K_D controller gains. The following example demonstrates the use of the Ziegler-Nichols tuning algorithm to find the PID controller gains. Figure 22 shows the ultimate gain Ku and ultimate period P_U values that will be used in the Ziegler-Nichols tuning example,

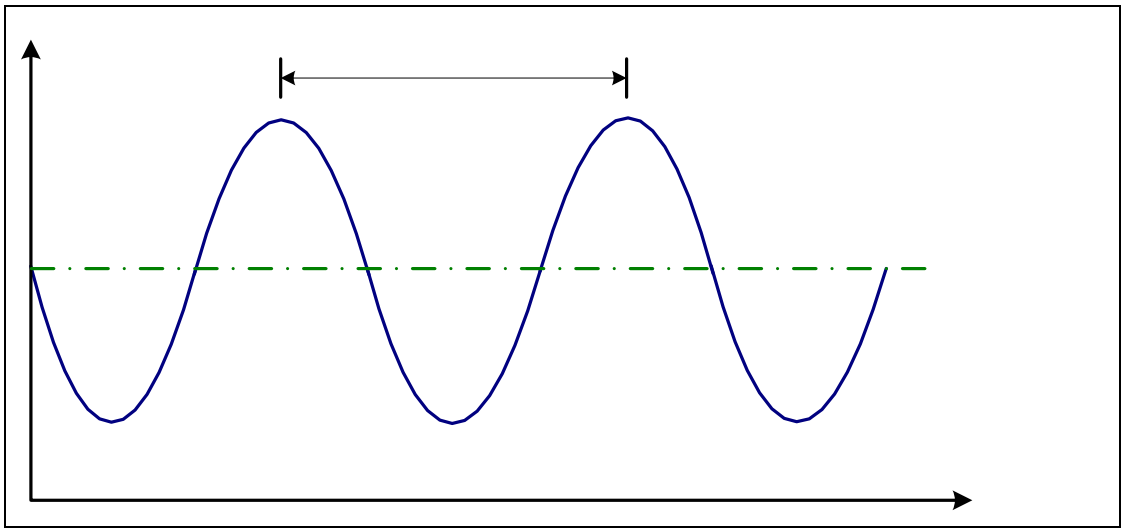

Fig. 22. Ziegler-Nichols close loop tuning test.

For the example, the controller to be designed is the PID controller. Figure 23 shows an example using the values from Figure 22.

From the example, the PID controller gains were calculated using the Ziegler-Nichols tuning method. After obtaining the PID gain values, the gains need to be plugged back into equation 6.11. The main idea behind using the Ziegler-Nichols tuning method

$$K_U = 100.0$$
$$P_U = 4.25$$
$$T_S = 0.050\,\text{sec}$$

$$K_P = 0.65K_U = 0.65 * 100.0 = 65.0$$

$$T_I = 0.50P_U = 0.50 * 4.25 = 2.125$$
$$T_D = 0.120P_U = 0.120 * 4.25 = 0.51$$

$$K_I = \frac{K_P T_S}{T_I} = \frac{(65.0)(0.050)}{2.125} = 1.529$$
$$K_D = \frac{K_P T_D}{T_S} = \frac{(65.0)(0.51)}{0.050} = 663.0$$
$$K_P = 65.0$$
$$K_I = 1.529$$
$$K_D = 663.0$$

Fig. 23. Example.

is to first force the system to be unstable in order to record both ultimate gain K_U and ultimate period P_U. Then use Figure 21 to design the PID controller.

The Ziegler-Nichols tuning method is one of many ways to tune a PID controller. When using the Ziegler-Nichols tuning method, the designer needs to setup a test plan to observe the system output response. The PID controller designer needs to understand the specifications that are part of the output response. Figure 24 shows a typical closed loop system output response waveform.

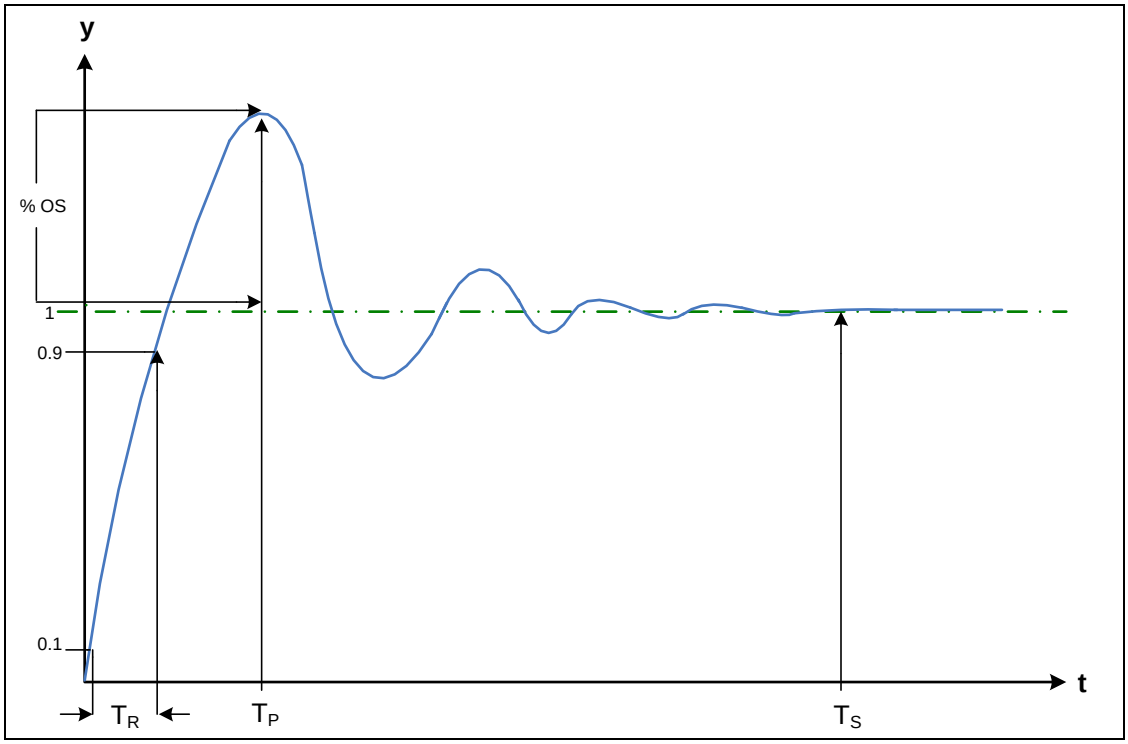

Fig. 24. Typical close loop output response.

Figure 24 shows the typical system response terms. Each of the parameters shown are close loop specifications that are taken into account when designing any control system. The response specifications allow the control engineer to analyze the performance of the system under control. The close loop specifications are defined as follows:

- Peak Time T_P: The peak time is the time it takes to reach the first maximum peak of the output response.
- Settling Time T_S: The settling time is the time required for the transient response to be damped down to a steady state value.
- Rise Time T_R: The rise time is the time needed for a waveform to from 0.10 to 0.90 of the final value.

- Present Overshoot *%OS*: The present overshoot is the amount the waveform overshoots the desire final value.

Sampling Rate

The sample rate, in a discrete PID control algorithm, plays an important role in the execution of the closed loop system. The sampling rate in a discrete time controller is both affected by the software and hardware structure of the controller. Choosing the right sampling rate in a discrete time controller determines its overall performance. Some considerations when selecting a suitable sampling rate include the following:

- First, if the sampling rate is too large, the closed loop system might experience excessive overshoot oscillations. These overshoot oscillations can lead to system instability.
- Next, if the sampling rate is too low, the discrete time system will start to resemble the same behavior as a continuous time system.
- Finally when choosing the proper sampling rate, the time should be long enough for the proper execution of the PID control algorithm. A 50 ms sample rate was chosen to run the PID control algorithm.

The goal is to have a sampling period for a discrete time system that resembles the same response as that of a continuous time system. A disadvantage of having too short of a sampling period is that it can lead a closed loop system to have signal delays.

CHAPTER V

RESULTS AND PERFORMANCE EVALUATION

This section talks about the results and the performance evaluation of the balancing robot system. Followed by discussing the implementation and testing completed in the duration of the project. The results were logged by the serial data logger and plotted using MATLAB. The Kalman filter and the PID controller were programmed in C language. A P1C32MX360F512L microcontroller was programmed to implement the filter and the closed loop PID control algorithms for the balancing robot system. A built in timer from the P1C32MX360F512L microcontroller was used to provide an accurate sample time for both of the algorithms. A system block diagram representing the implemented robot components is shown in Figure 25.

Figure 25 shows a block diagram that details the data flow between the electrical components. The block diagram also displays the actual balancing robot system implementation. The balancing robot system results were collected using the block diagram layout. The first results presented will be the Kalman filter results, followed by the PID controller results.

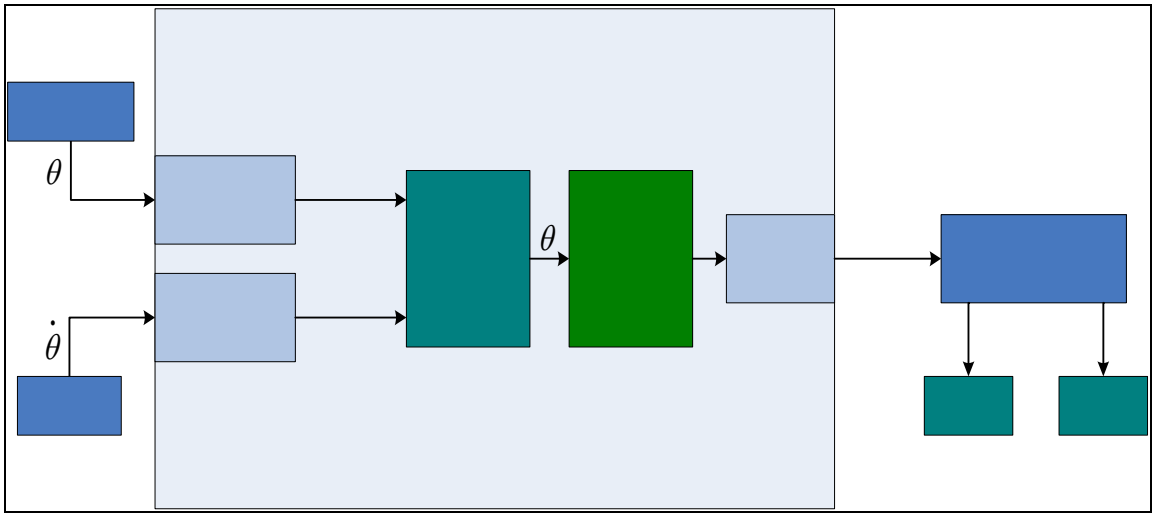

Fig. 25. Functional block diagram for balancing system.

Kalman Filter Results

As previously described, the main objective of using the Kalman filter in an application is to eliminate most of measurement noise from the accelerometer sensor. The filter also helps to reduce the drift problem that the gyroscope sensor experiences. The Kalman filter algorithm will provide a better angle estimate to the true angle from the accelerometer. For the filter to work properly, a manual tuning method needs to be carried out. The tuning filter parameters that needed to be tuned are the covariance matrices Q_w and R_v. The Q_w matrix consists of the process noise and the R_v matrix is the measurement noise. Both matrices take the form as shown on Figure 26.

Values of *Q_accelerometer* and *Q_gyroscope* are set depending on how much confidence one puts on the particular sensor. A higher value indicates that the particular sensor is trusted less as compared to the other sensor. The value of the *R_measurement* matrix indicates the amount of noise expected from the measurement, a low value indicates that the measurement is less corrupted with noise.

$$Q = \begin{bmatrix} Q\_accelerometer & 0 \\ 0 & Q\_gyroscope \end{bmatrix}$$

$$R = \begin{bmatrix} R\_measurement \end{bmatrix}$$

Fig. 26. Process and measurement noise covariance matrices.

The filter tuning method used on this project was done by trial and error. Once the tuning covariance matrices Q_w and measurement matrix R_v constant parameters provided the desire output response, the constant values were kept. The simple test platform shown in Figure 13 was used for the Kalman filter experimentation. Both the gyroscope and the accelerometer were rotated at angles between *+1-* 10 degrees. Before the Kalman filter can be implemented and tested, the accelerometer and gyroscope raw sensor data has to be scaled. The accelerometer data was scaled in radians, while the gyroscope data was scaled in radians per second. If the senor data is not scaled properly, the digital filter will not work as expected. The Kalman filter algorithm was set to run every 50 ms. The following Kalman filter waveforms demonstrate the output response when the covariance matrix Q_w values get varied and R_v remains constant. The red waveform represents the noisy accelerometer angle data and the blue waveform represents the Kalman filter angle data. Figure 27 shows the output response when the filter is not properly tuned.

From Figure 27 it is clearly shown that the filter output response is not tuned properly. The filter resembles a saw tooth waveform. The filter doesn't track the true accelerometer angle in real time. The output of the Kalman filter has a very slow

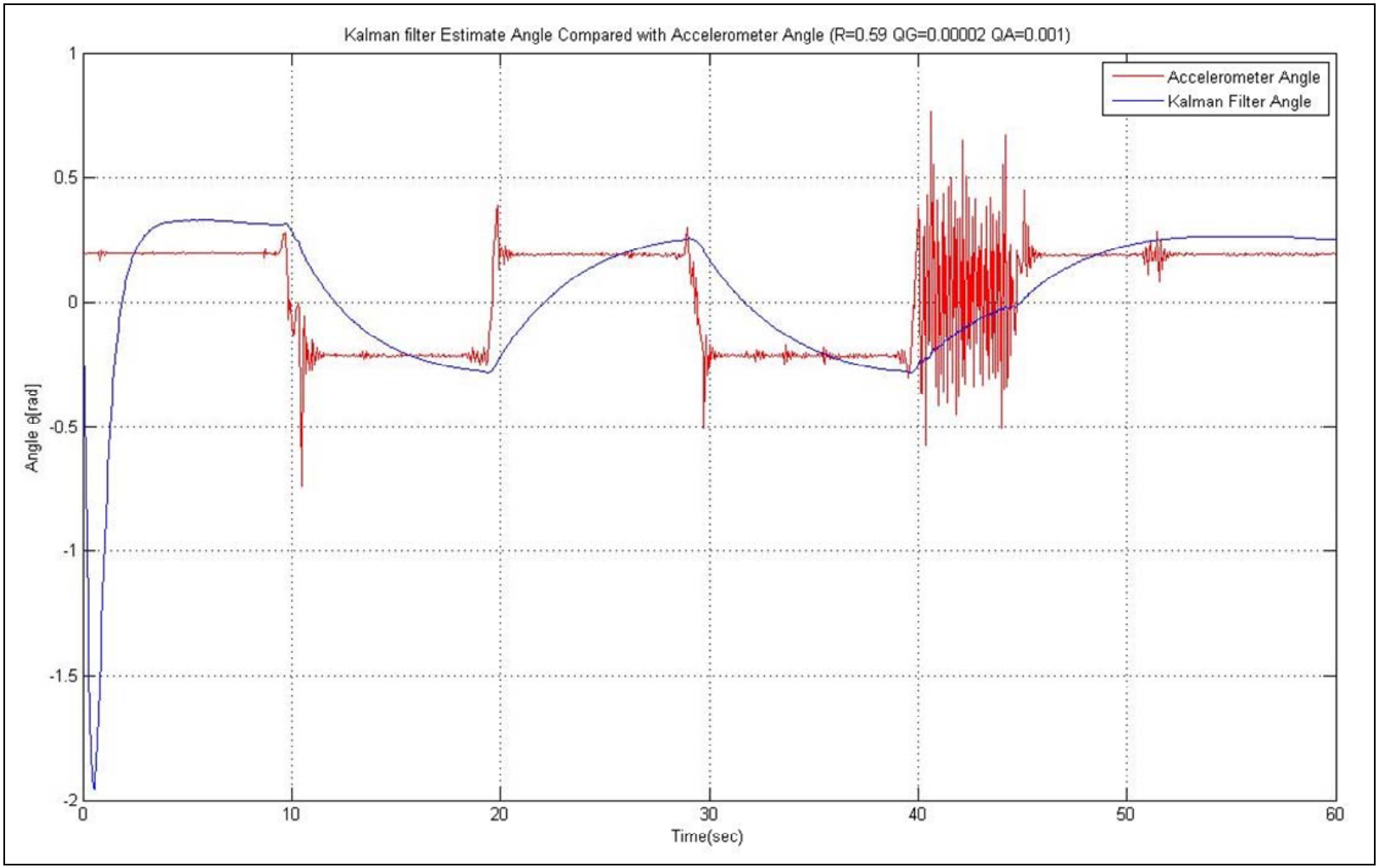

Fig. 27. Not properly tuned Kalman filter.

response to changes from the accelerometer data. This large delay in Kalman filter can be a major problem when the estimated accelerometer angle is fed into the PID controller. From figure 24, it can be observed that the process noise covariance matrix parameter *Q_gyroscope* was set to have a lower value than the *Q_accelerometer*. A lower *Q_gyroscope* value means that the gyroscope sensor data is trusted more than the accelerometer data. From the Kalman filter results it seems that the *Q_gyroscope* value controls the filter response to changes causing the saw tooth like waveforms. Also from figure 24, the disturbances towards the end of the waveform were cause by shaking the robot. The shaking disturbance shows how the Kalman responses to external forces. The same disturbance was injected to the system for the remaining Kalman waveforms below. To solve the slow output response of the Kalman filter, the *Q_accelerometer* was set to have a lower value than the *Q_gyroscope.* In this case the accelerometer sensor data will be trusted more than the gyroscope data. Figure 28 demonstrates this Kalman filter case.

From Figure 28, it is observed that the filter output response has improved. The Kalman filter tracks the accelerometer sensor data in real time. A problem was observed in Figure 28, when the filter is initialized; the filter tends to have a dramatic overshoot. After approximately 5 seconds, the filter recovers from the overshoot. The other problem that was observed, from the figure, is that every time there is a change in angle there are overshoots that follows. The Kalman filter does an excellent job eliminating the overshoots when the filter is operating. These overshoots can become an issue if the estimated Kalman angle gets fed into the PID controller.

The solution to the overshoot problem can be solved by adjusting the process noise covariance matrix parameter *Q_gyroscope.* After numerous trials and errors, the

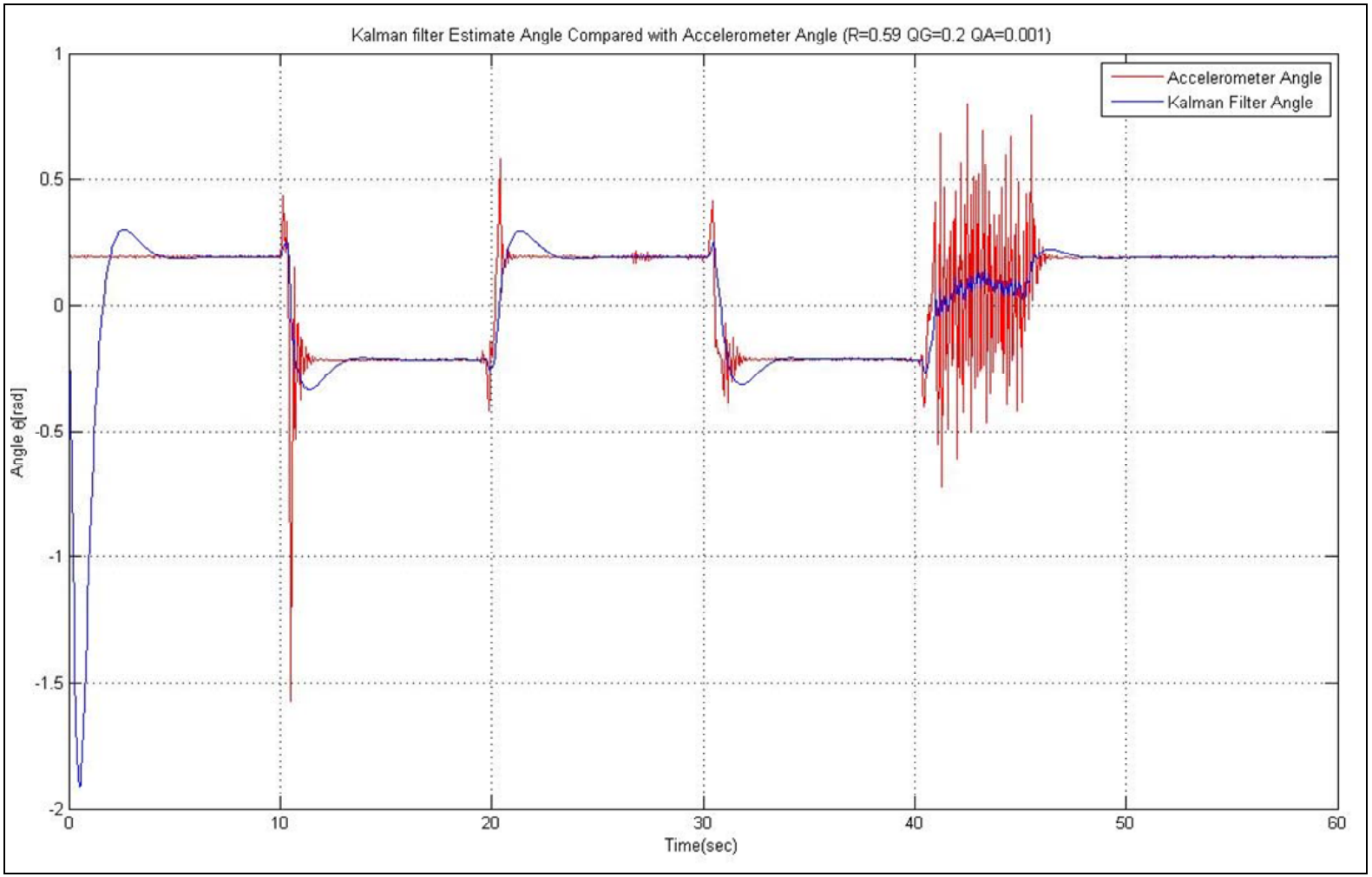

Fig. 28. Kalman filter output response varying Q matrix.

optimal covariance matrix Q_w values were set as *Q_accelerometer=0.001* and *Q_gyroscope=0.002*. These process noise covariance matrix values offer the best filter output response. Figure 29 shows the results for these matrix values.

Figure 29 shows a very clear observation that the Kalman filter output response has reduced or eliminated the overshoots. The figure also illustrates how rapidly the filter tracks the accelerometer angle data. The initial overshoot that the Kalman filter experiences gets reduced and dies off after about 3 seconds. After the initial overshoot, the filter operates normally as expected.

The measurement noise covariance matrix R_v, determines much the measurements should be trusted. Setting the covariance matrix R_v to a high value means that the sensor measurements are too noisy. Too low of a covariance matrix R_v value means that the measurements are less noisy. The Kalman filter parameter also plays a major role in the performance of the filter. Figure 30 shows when the covariance matrix R_v value is set too low.

In Figure 30, the *R_measurement* was set to 0.000059. The waveform results clearly show that the Kalman filter has an output response that is similar to the accelerometer waveform. This type of Kalman filter response has the same amount of noise as the raw accelerometer data. The noisy filter data cannot be used in the balancing robot system. To solve the noisy Kalman filter data, the measurement noise covariance matrix was set to *0.59*. Figure 29 displays the waveform when the measurement noise covariance matrix is set to *0.59*.

When using the Kalman filter, numerous trial and error need to be carried out to find the optimal values. The Kalman filter covariance matrix values that were chosen

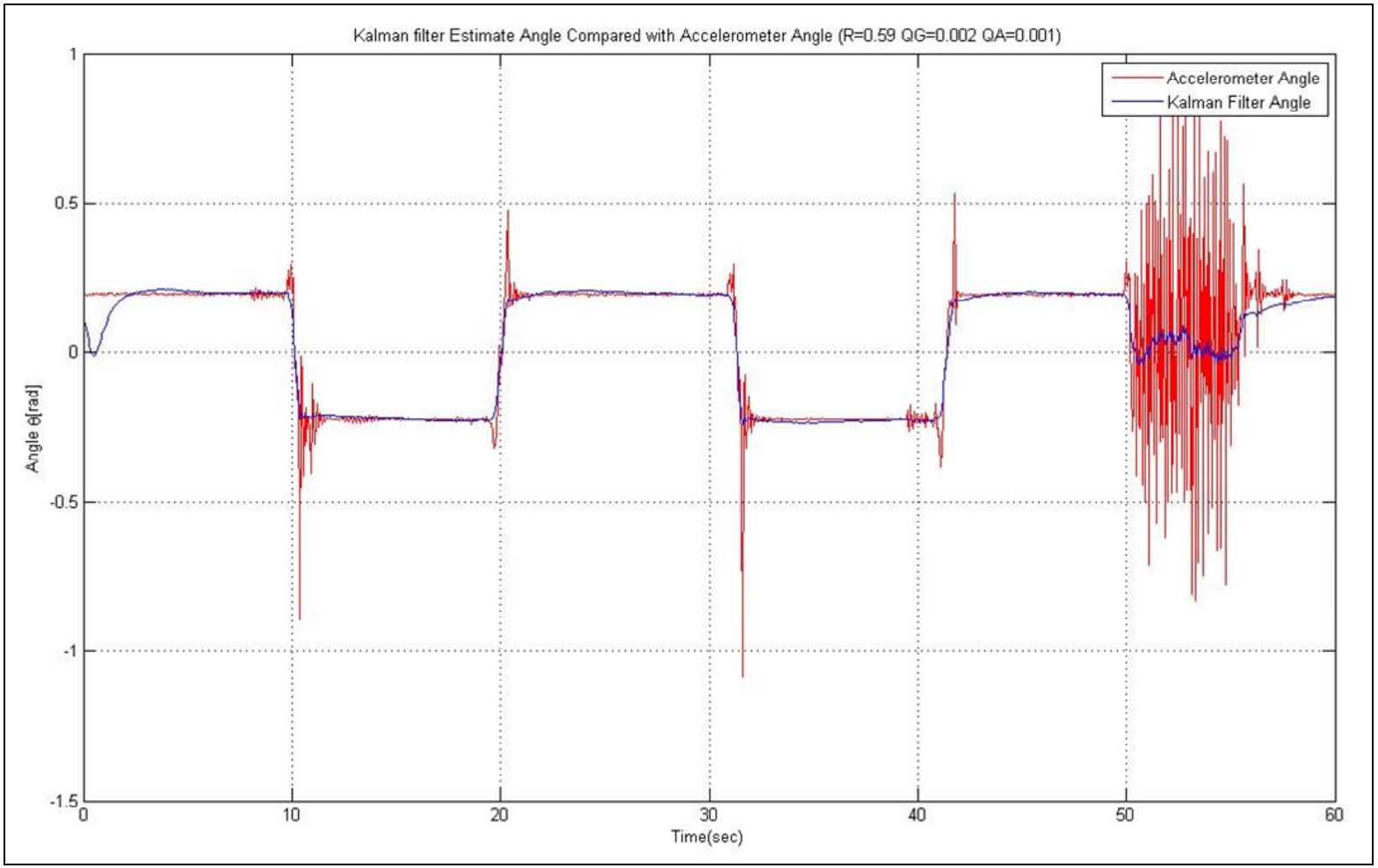

Fig. 29. Optimal Kalman filter Q matrix values.

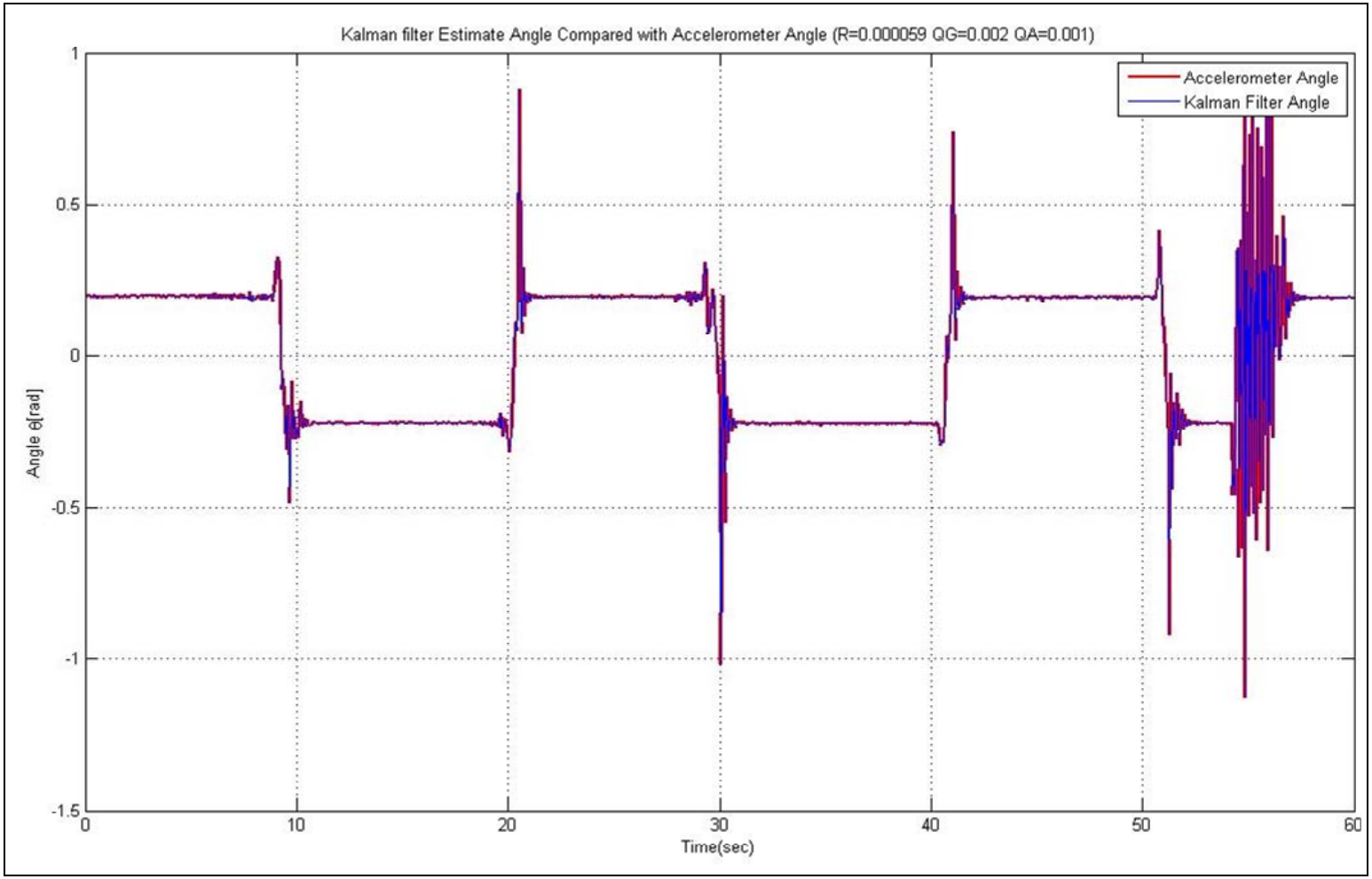

Fig. 30. Kalman filter output response varying R matrix.

to give the best results were hard coded into software. The measurement noise and the process noise covariance matrix values are listed as follows:

- *R_measurement=0.590*
- *Q_accelerometer=0.001*
- *Q_gyroscope=0.002*

These filter parameters offered the best performance and results. Figure 29 shows the Kalman filter results when using the parameters listed.

The pervious results showed how the Kalman filter helped to reduce or eliminate most of the noise from accelerometer. The other results also captured the performance of the Kalman filter in the gyroscope sensor data. As mentioned before, the gyroscope has the problem of its steady state value drifting away over time. The Kalman filter algorithm helps eliminate the drift problem on a motionless gyroscope. Figure 31 shows the results for a motionless gyroscope. The blue waveform represents the Kalman filter data while the red waveform represents the raw gyroscope sensor data. Figure 31 shows how well the filter performed in preventing the drift problem caused by the motionless gyroscope. The Kalman filter average drift value remained within 0 rad/s. The spikes in the waveforms were caused when the robot chassis was moved. After passing the gyroscope raw sensor data thru the Kalman filter, the estimated velocity can be used along with estimated angle.

PID Controller Results

The objective of the PID controller is to provide control and stability. The PID consists of three gain terms that affect the output response of the system. The

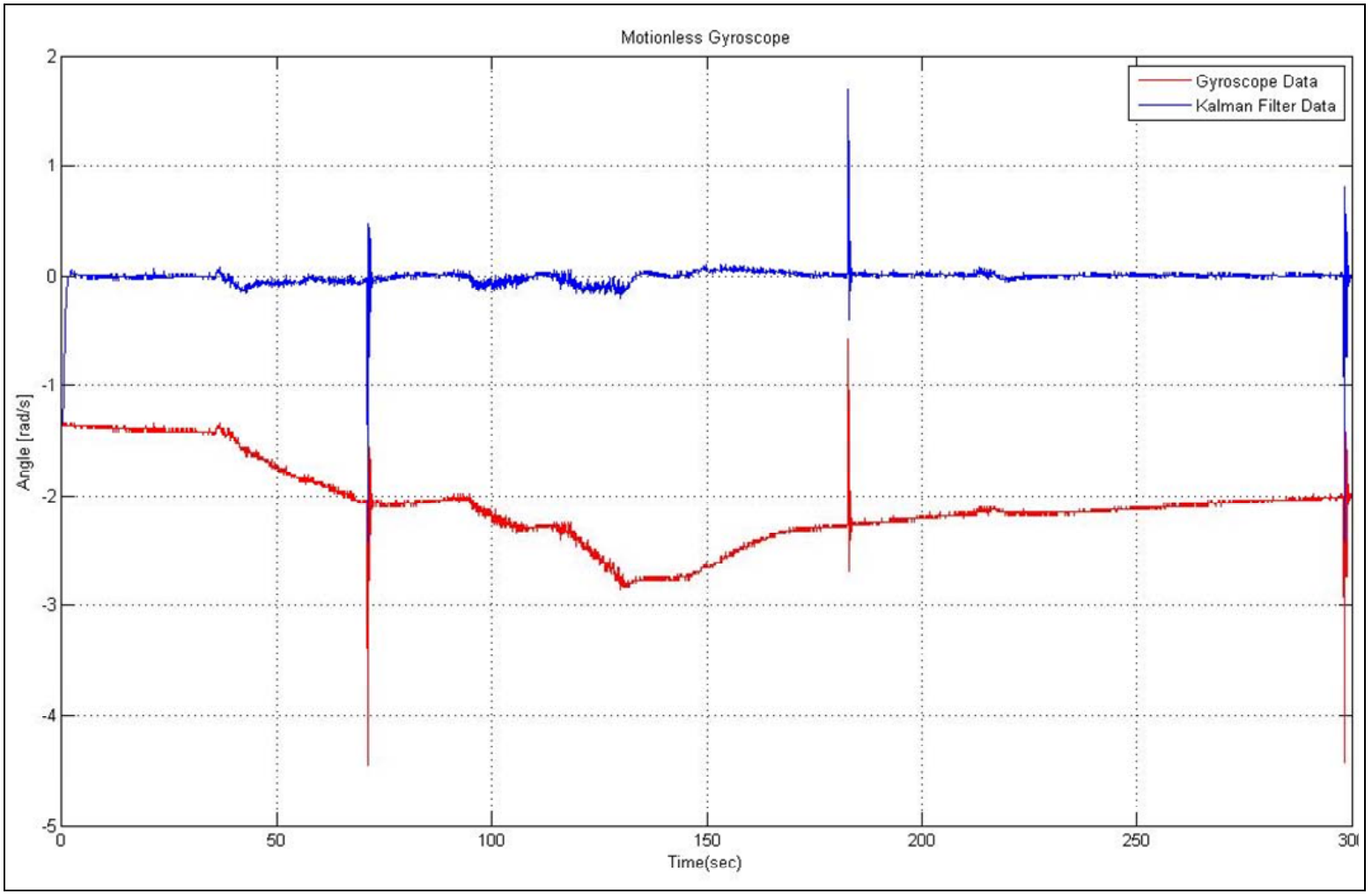

Fig. 31. Kalman filter motionless gyroscope result.

proportional K_P gain determines the rise time of the system. The integral K_I gain affects the steady state error. The derivative K_D gain controls the system response overshoot and helps with its stability. Figure 32 displays the output response result captured for the self-balancing robot.

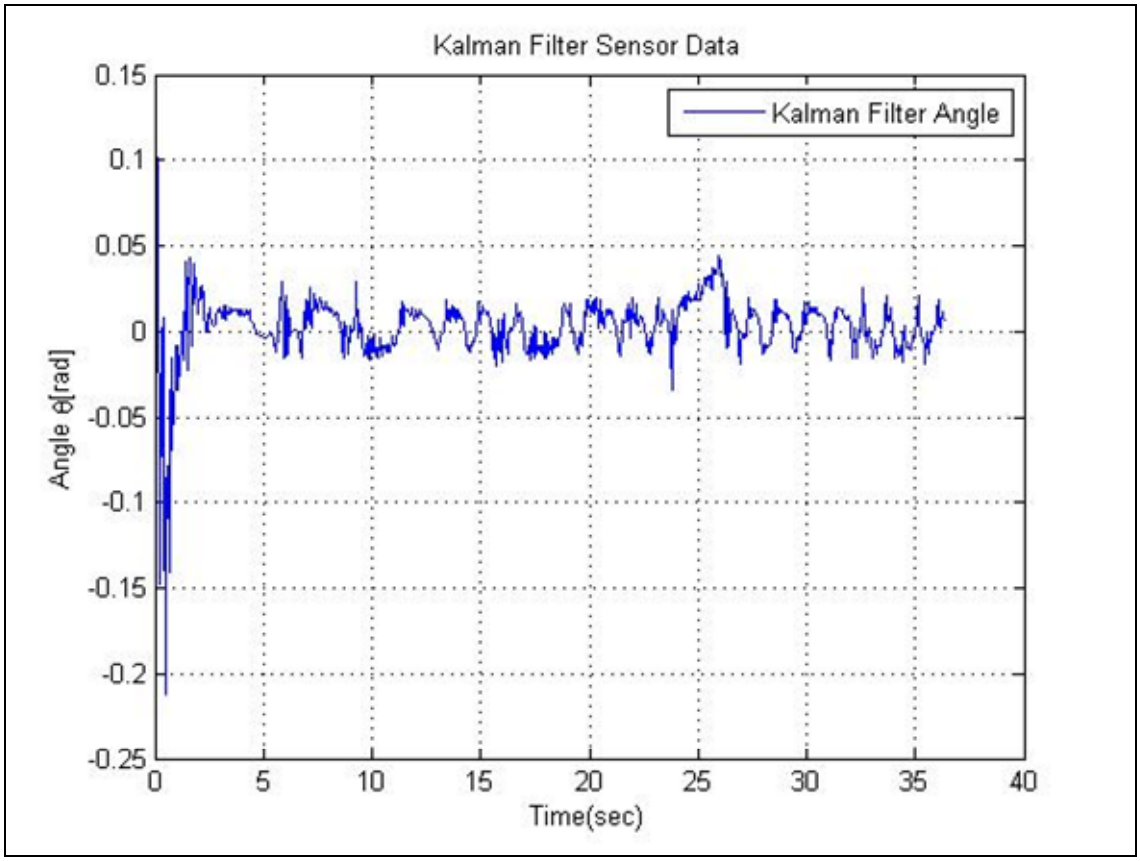

Fig. 32. Measured system output response.

Results from Figure 29 show the estimated Kalman filter angle measurement. This plot also displays the closed loop output response for the robot system. Figure 29 displays how robot tried to maintain a stable state by hovering around the zero radian value. The measured angle hovered around the zero radian value. The figure also shows

that the first few seconds the system response had an overshoot. This initial overshoot is caused by the initial Kalman angle estimate, but the system soon recovers from the overshoot. The output response of the system still suffers from the noise generated by the sensors. All efforts were made to minimize as much of the signal noise as possible.

Results were also captured for both the input and output of the PID controller. The input to the controller is the error. The output of the controller provides the drive signal used to throttle up or down the motor speed. Figure 33 shows the input to the PID controller.

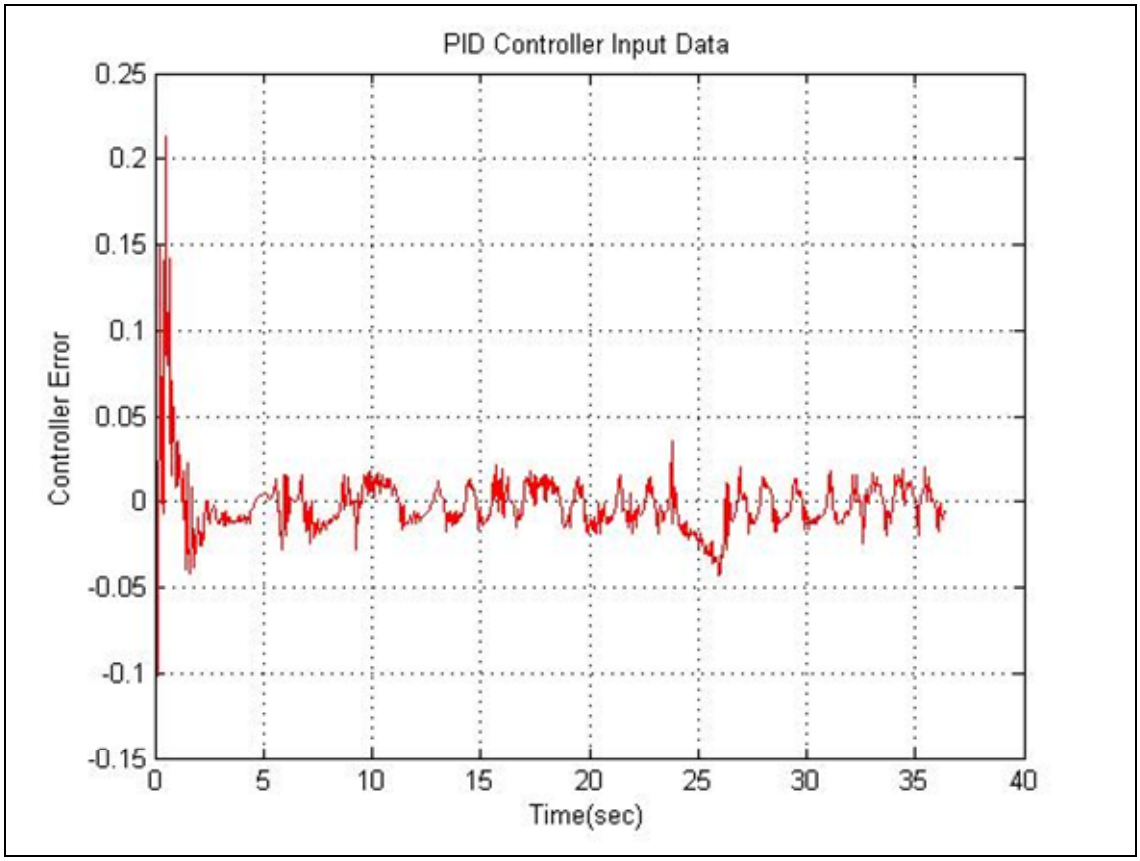

Fig. 33. PID controller input error.

The goal or the ideal case is to drive the error signal input to zero. As mentioned before, error signal from the system also suffers from an initial overshoot. The overshoot quickly dissipated within a few seconds. The output to the PID controller is shown on Figure 34. The PID controller output signal gets fed into VNH2SP3O DC motor driver. The signal determines how fast the motors are driven to prevent the robot chassis from tipping over. The objective is to have the drive signal get as close as possible to zero.

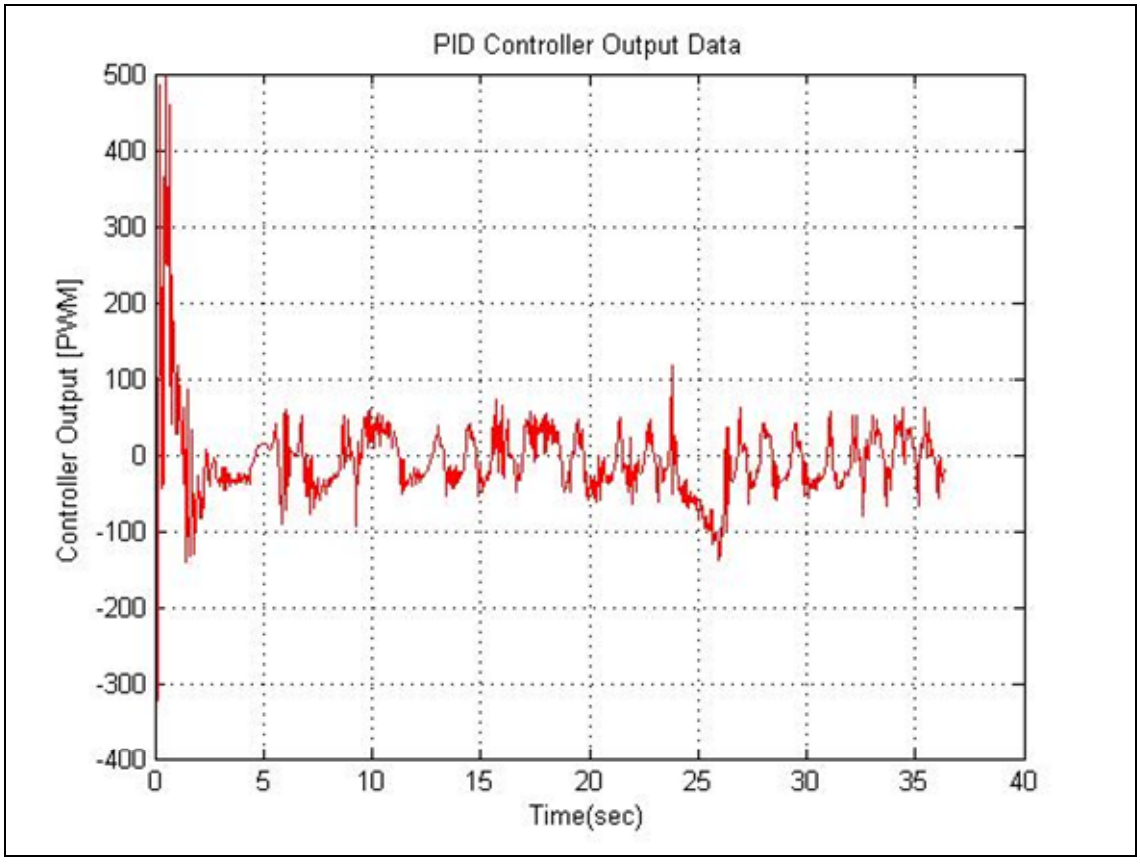

Fig. 34. PID controller output.

The PID gain values that were used for the PID controller are:

- *P= 1800.001*
- *I= 0.10250*
- *D= 95.7501*

CHAPTER VI

SUMMARY, CONCLUSION, AND RECOMMENDATIONS

Summary

The autonomous self-balancing robot project was very successful in achieving its goal. The robot maintained its vertical balance position. The overall robot control system is classified as a single input single output, also known as *SISO*. Two sensors were used to provide tilt angle and angular velocity information. Only the accelerometer tilt angle data was used in the closed loop system. The accelerometer measures the inclination of the robot structure, while the gyroscope provides the angular rate. To attain stability, a digital filter and a control algorithm were implemented as part of the project. The Kalman filter was the filter of choice to code in software. The Kalman filter fused both of the sensors to provide reliable estimated angle information. The Kalman filter was successful in removing most of the noise from the sensor measurements. The Kalman filter needs tuning to operate properly. The tuning of the filter was done by trial and error. The estimated filter data was then fed onto the PID control algorithm.

For the self-balancing robot project, stability was successfully achieved by implementing the PID controller. The controller works by subtracting the desired set point from the Kalman estimated tilt angle. The subtraction generates an error, which was then fed onto the PID control algorithm. The control algorithm processed the error to

generate a pulse width modulation drive signal. The signal drive is used to drive the DC motor accordingly. Before the PID controller can provide system stability, it needs to be tuned. There many ways to tune the PID controller, to get the desired output response. One of the tuning methods that was used on this project was the Ziegler-Nichols tuning algorithm. The Zeigler-Nichols tuning method makes it very simple to properly tune the PID controller. The tuning method also requires very little process knowledge. The other method for tuning the PID controller was by trial and error. Tuning the controller by trial and error is not the best choice, but it provided the desired output response.

Conclusion

Over the past few years, self-balancing robots have become a popular topic of research. Most of the research and development involves control algorithms and system dynamics. Advanced controllers provide robust and optimal control for self balancing robots. The two wheeled self-balancing robots are excellent examples of the classical inverted pendulum problem. In recent years microcontrollers have become the popular choice to implement control theory at lower costs. Making microcontrollers ideal cheap components to use on self balancing platforms.

One of the goals for the project was to implement a closed loop control system using low cost components. Using lost cost components in this type of project can sometimes lead to drawbacks, in this case from both the accelerometer and the gyroscope. Both sensors experience initial problems that made them difficult to use in this type of project. The accelerometer data was highly corrupted with unwanted noise. The gyroscope suffered from a problem that made its bias value drift away with time. A

solution to the problems associated with these sensors outputs can be found in the use of a Kalman filter. This filter is used to minimize and totally eliminate the unwanted output results from the sensors. The PID controller relies on the input signal to be almost free of noise.

The project demonstrated that a fully functional self balancing robot can be built with off the shelf components. Being based on low cost components, the project is an ideal demonstration of control theory for a classroom setting. The project can also serve as a test platform to write the necessary software control algorithms to implement for commercial products.

Recommendations

This master's project provides the basis for future research on Kalman filter applications and implementations. Also to research more advanced control systems. As stated by Rich Ooi, "more research should be conducted to exploit the Kalman filter technology and its application for other projects that require sensor fusion technology" [7, pg. 54]. Such sensor fusion technology can be used in personal navigation systems. The tuning of the Kalman filter needs to be studied more in depth. Tuning the Kalman filter by trial and error is time consuming. With more research a better method to tune the Kalman filter covariance matrices Q_w and R_v may be discovered.

The linear PID control system developed in this project was able to establish that the robot was able to balance under some minimal disturbances. The robustness of the system is not fully tested and more testing is needed to assess the robustness of the system. Further fine tuning of the PID control algorithm is required for to achieve better

output performance. Further research can be conducted to implement intelligent controllers on balancing robot system. Such intelligent controllers might be a fuzzy PID controller or a self tuning PID controller.

Another future improvement to the self-balancing two wheel robot might be to add encoders. The encoders may be placed along side the motors to provide motor rotation information. Having the motor rotation information from the encoders, the robot will have better locomotion control. Improved locomotion control will allow the robot to move precise distances.

REFERENCES

REFERENCES

[1] D. Ibrahim, D *Microcontroller Based Applied Digital Control.* West Sussex, England: John Wiley & Sons Ltd, 2006.

[2] R. Hollis, "BallBots," *Scientific American*, October 2006. Retrieved February 4, 2009 from the World Wide Web: http://www.sciam.com/article.cfm?id=ballbots

[3] T. Atwood, "VECNA's Battlefield Extraction-Assist Robot," *Robot Magazine*, April 25, 2007. Retrieved February 6, 2009 from the World Wide Web: http://www.botmag.com/articles/04-25-07_vecna_bear.shtml

[4] G. Welch and G. Bishop, "Kalman Filter." *An Introduction to the Kalman Filter*, 2007. Retrieved February 16, 2009 from the World Wide Web: http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html

[5] D. Simon, "Kalman Filtering." *Embedded Systems Programming*, 2001. Retrieved January 18, 2009 from the World Wide Web: http://www.embedded.com/9900168?_requestid=8285

[6] V.J. VanDoren, "PID: Still the One," *Control Engineering*, October 2003. Retrieved January 19, 2009 from the World Wide Web: http://www.controleng.com/article/CA325983.html

[7] R.C Ooi, "Balancing a Two-Wheeled Autonomous Robot," 2003. Retrieved January 18, 2009 from the World Wide Web:http://robotics.ee.uwa.edu.au/theses/2003-Balance-Ooi.pdf>.

APPENDIX A

APPLICATION SOURCE CODE

main_pid.c

```
//====================================================================
/*
  Programmer: Jose L. Corona
  Project Name: PID Test Software for PIC32
  Date: 2/19/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//====================================================================
//#include <proc/p32mx360f512l.h>
#include <p32xxxx.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <plib.h>
#include "config_adc.c"
#include "config_pwm.c"
#include "config_timer_1.c"
#include "config_timer_2.c"
#include "kalman.c"
#include "USART_32.c"
#include "pid.c"
#include "accel.c"
#include "gyro.c"
#include "knight_rider.c"

//----------------------- Config. settings -------------------------
// POSCMOD = HS, FNOSC = PRIPLL, FWDTEN = OFF
// PLLIDIV = DIV_2, PLLMUL = MUL_20
// PBDIV = 8 (default)
// Main clock = (8MHz/2) * 20/ 1 = 80MHz
// Peripheral clock = 80MHz /8 = 10MHz

// Configuration Bit settings
// SYSCLK = 80 MHz->(8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz->SYSCLK/PBDIV
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care

/************************ -Description- ***************************
 *
 * Oscillator Configuration Bit Settings:
 *           - Oscillator Selection Bits = Primary Osc w/PLL (XT+HS+EC+PLL)
 *           - Primary Oscillator Config = HS osc mode
 *           - PLL Input Divider = 2x Divider
 *           - PLL Multiplier = 20x Multiplier
 *
```

```
* Notes:
 *      - Set PBCLK divisor = 1:8 to achieve a 10 MHz PBCLK.
 *      - To cal. baudrate (SYS_FREQ/PB_DIV)/(16*baudrate_2)-1
 ************************************************************/
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1, FWDTEN =
OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_8

//---- Let compile time pre-processor calculate the PR1 (period) ----
#define SYS_FREQ                80000000L
#define PB_DIV                  8
#define PRESCALE                256
#define SAMPLES_PER_SEC         20
#define T1_TICK_RATE            (SYS_FREQ/PB_DIV/PRESCALE/SAMPLES_PER_SEC)


//---- Kalman Filter Parameters ----
/*
#define R_matrix                0.690
#define Q_Gyro_matrix           0.002
#define Q_Accel_matrix          0.001
*/
#define R_matrix                0.590
#define Q_Gyro_matrix           0.002
#define Q_Accel_matrix          0.001

// LED's
#define LED_1 PORTAbits.RA0
#define LED_2 PORTAbits.RA1
#define LED_3 PORTAbits.RA2
#define LED_4 PORTAbits.RA3
#define LED_5 PORTAbits.RA4
#define LED_6 PORTAbits.RA5
#define LED_7 PORTAbits.RA6
#define LED_8 PORTAbits.RA7

//---- Constants ----
#define pi                 3.1415926535898
#define degrees            (180.0/pi)

// Boolean values used on Timer 1 Interrupt
//#define FALSE           0
//#define TRUE            1

//---- Variables Declarations ----
char *message_test= "\r\n ---->>> USART 2 Up and WORKING <<<----- \r\n";
char *carr_return= "\r";

// buffer offset to point to the base of the idle buffer
unsigned int offset= 0;

//---- Variable Declarations ----
double angle_est= 0.00;
//double gyro_rate= 0.0;

double accel_x,rate_y= 0.00;
int adc_accel= 0;
int adc_gyro= 0;

double drive_pwm= 0.00;
double reference= 0.0010;
```

```
double system_error= 0.00;

double timer_counter= 0.00;
//unsigned char flag;

//---- Kalman State Structures ----
kalman filter_pitch;


//----PID Structures ----
pid pid_access;

void drive_system(double system_input);

/*****************************************************************
 P, I and D parameter values
 The K_P, K_I and K_D values (P, I and D gains)
 need to be modified to adapt to the application at hand
*****************************************************************/
#define T_S  1.0/(SAMPLES_PER_SEC)

// PID Gains at 50ms sample rate
double K_P= 1800.001;
double K_I= 0.10250;
double K_D= 95.7501;

int main()
{

 // Configure the device for maximum performance but do not change the PBDIV
 // Given the options, this function will change the flash wait states, RAM
 // wait state and enable prefetch cache but will not change the PBDIV.
 // The PBDIV value is already set via the pragma FPBDIV option above.
 SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

 // Set PORTA to all outputs
 TRISA= 0x0000;

 // Clear PORTA
 PORTA= 0x0000;

 // Used on Motor Control pins
 TRISG= 0x0000;
 PORTG= 0x0000;

 // Used on PWM pins on Motor Driver
 TRISD= 0x0000;

 //--- Disable JTAG port ---
 DDPCONbits.JTAGEN = 0;
 //--- Disable Trace port ---
 DDPCONbits.TROEN = 0;

 //--- Initilize On-board Devices ---
 setup_adc();
 setup_pwm();
 setup_timer_1();
 setup_timer_2();

 setup_usart(USART_1, 65);        // (32) 19200 bps @ PBCLK = 10 MHz
 setup_usart(USART_2, 65);        // (65) 9600 bps @ PBCLK = 10 MHz
```

```
                                    // (10) 57600 bps @ PBCLK = 10 MHz

 //--- Setup Timer 1 Interrupts ---
 IEC0bits.T1IE = 0;     // Disable Timer 1 interrupts
 IFS0bits.T1IF = 0;     // Clear the Timer 1 interrupt flag
 IPC1bits.T1IP = 7;     // Set Timer 1 interrupt to priority level 7 (Highest)
 IPC1bits.T1IS = 0;     // Set Timer 1 interrupt to sub-priority level 0
(Lowest)

 PR1 = T1_TICK_RATE;    // Load Period (Timer 1) Register to produce 0.050 sec.


 //--- Initilize PWM Module ---
 PR2 = 499;             // Load Period (Timer 2)Register with Freq = 20 kHz for
PWM.
 OC1R = 250;            // Initialize Primary Compare Resigter
 OC1RS = 250;           // Initialize Secondary Compare Resigter (PWM Duty
Cycle)

 T2CONbits.ON = 1;      // Enable Timer2
 OC1CONbits.ON = 1;     // Enable Output Compare Module

/*********************************************************************
Initialize the Kalman Filter for pitch state.  The measurement noise
covariance (R_angle) is the noise in the accelerometer measurements.
The process noise covariance are (Q_angle, Q_gyro).The proper settings for
these values needs to be explored further, but Q_angle is how much we
trust the accelerometer and Q_gyro is how much we trust the gyro.
(R_angle) - can be calculated from off-line samples of mesurements.
(Q_angle, Q_gyro) - Are tuned by trial and error.
*********************************************************************/
 // Parameters ( R_angle, Q_gyro, Q_angle )
 kalman_init(&filter_pitch, T_S, R_matrix, Q_Gyro_matrix, Q_Accel_matrix);

 //-------- Start PID --------
 pid_setup(K_P, K_I, K_D, &pid_access);

 //---- Knight Rider LED Effect ----
 knight_rider();

 //---- Turn ON the ADC module ----
 AD1CON1bits.ON = 1;          // ADC On bit (1:ADC Enable /0:ADC Disable)
bit<15>
 AD1CON1bits.ASAM = 1;        // ADC Sample Auto-Start bit
                              // (1:ADC auto sampling /0:ADC manual sampling)
bit<2>

 //---- Enable Timer Interrupts ----
 T1CONbits.ON = 1;           // Enable Timer1
 IEC0bits.T1IE = 1;          // Enable Timer 1 interrupts

 //TX_string(USART_2,message_test);

 //--- Enable Multi-vector Interrupts ---
 INTEnableSystemMultiVectoredInt();

//--- Start Kalman Estimates ---
// Start the pitch angle estimate
while(!IFS1bits.AD1IF);      // Poll for ADC interrupt
IFS1bits.AD1IF = 0;          // Clear ADC interrupt status flag
                             // (1:ADC interrupt occurred /0: NO interrupt)
```

```
// Determine which buffer is idle and create an offset
offset= 8 * ((~AD1CON2bits.BUFS & 0x01));

// Read the result of AN0(accel.) conversion from the idle buffer
adc_accel= (ReadADC10(offset));
angle_est= read_accel(adc_accel,'r');

       //Loop forever
       while(1)
       {
              while(!IFS1bits.AD1IF);


              IFS1bits.AD1IF = 0; // Clear ADC interrupt status flag


       }// End of while()

return 0;
}
/*****************************************************************
*    Function Name:  drive_system                                *
*    Return Value:   None                                        *
*    Parameters:     system input                                *
*    Description:    Sends the PID controller output to the process *
*                    input. The signal is PWM.                   *
*                                                                *
*          _________________          ___________               *
* error |                 |   PWM    |           |   Yout        *
* ------| PID Controller |--------->| DC Motors |--------->      *
*       |_________________|          |___________|                *
*                                                                *
*****************************************************************/
void drive_system(double system_input)
{

   signed short int process_in;

   process_in= (signed short int)system_input;

   if(process_in==0)
   {
      PORTG= 0x0000;
      LED_6= 0;
        LED_7= 0;
      OC1RS= 0;
   }
       else if(process_in < -0)
       {
              LED_7= 0;
              LED_6= 1;

              //--- Go Forward ---
              //PWM 1 Duty Cycle (RD0)
              PORTG=0x0009;
       }
              else if(process_in > 0)
              {
                     LED_6= 0;
                     LED_7= 1;
```

```
                    //--- Go Backward ---
                    //PWM 1 Duty Cycle (RD0)
                    PORTG=0x0006;
               }
     // Send PID Signal to DC Motors
     OC1RS= abs(process_in);

}
/*****************************************************
Timer 1 Interrupt Handler

Note: Timer 1 interrupt every 50ms.
*****************************************************/
void __ISR(_TIMER_1_VECTOR, ipl7) Timer1Handler(void)
{

  // Timer 1 interrupt flag set
  if(IFS0bits.T1IF == 1)
  {
       mPORTAToggleBits(BIT_0);
       OC1RS=0;
        timer_counter= timer_counter + T_S;

       //---- Get ADC Data from Buffers ----
       // Determine which buffer is idle and create an offset
       offset= 8 * ((~AD1CON2bits.BUFS & 0x01));

       // Read the result of AN0(accel.) conversion from the idle buffer
       adc_accel= ReadADC10(offset);

       // Read the result of AN1(gyro.) conversion from the idle buffer
       adc_gyro= ReadADC10(offset + 1);

       //-------- Read Gyroscope ADC --------
       rate_y= read_gyro(adc_gyro, 'r');

       //------ Read Accelerometer ADC ------
       accel_x= read_accel(adc_accel, 'r');

  //------------ Kalman Filter Pitch Estimate ---------------
  /*******************************************************
   Pass the measured pitch values (accel.) and rates (gyro.)
   through the Kalman filter to determine the estimated pitch
   and unbias gyro. values in radians.
   *******************************************************/
  //---- Execute Kalman Filter Algorithm ----
  kalman_predict(&filter_pitch, rate_y);       // Kalman Predict State
  kalman_update(&filter_pitch, accel_x);       // Kalman Update State

  //------ Get Process Measurement ------
  angle_est= kalman_get_angle(&filter_pitch);  // Kalman Estimate Angle Result


  //------ Calculate system error ------
  system_error= reference - angle_est;

  //-------- Execute PID Controller --------
  drive_pwm= pid_controller(&pid_access, system_error, angle_est);

  //------------- Drive Motors -------------
```

```
  drive_system(drive_pwm);

  //---- Save Kalman Data on Mirco-SD Card ----
  float_to_text(USART_1, timer_counter, 3);

  TX_string(USART_1, "  ");
  float_to_text(USART_1, accel_x, 3);

   TX_string(USART_1, "  ");
   float_to_text(USART_1, angle_est, 3);

   TX_string(USART_1, "  ");
   float_to_text(USART_1, system_error, 3);

   TX_string(USART_1, "  ");
   float_to_text(USART_1, drive_pwm, 3);

   TX_string(USART_1,carr_return);        */
}

      // Clear the Timer 1 interrupt flag
      IFS0bits.T1IF = 0;

}//End of IF

}// End of Timer 1 Interrupt Handler
```

config_adc.c

```
//============================================================================
/*

  Programmer: Jose L. Corona
  Project Name: ADC Setup Software for PIC32
  File: config_adc.c
  Date: 1/19/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32

*/
//============================================================================
void setup_adc(void);

/*****************************************************************
*    Function Name:  Setup ADC                                   *
*    Return Value:   NONE                                        *
*    Parameters:     NONE                                        *
*    Description:    This routine setup the ADC module for RB0(AN0). *
*                                                                *
*****************************************************************/
void setup_adc()
{

/*----- Setup ADC Module -----*/

  // Make All PORTB digital, but RB0(AN0)/RB1(AN1) = Analog Mode
  AD1PCFGbits.PCFG0 = 0;
  AD1PCFGbits.PCFG1 = 0;
```

```
  // Connect RB0(AN0) to MUX A (CH0 as positive input is AN0)
  AD1CHSbits.CH0SA=0x0;
  // Connect RB1(AN1) to MUX B (CH0 as positive input is AN1)
  AD1CHSbits.CH0SB=0x1;

  // NEG input select for MUX A (1:NEG input is AN1/0:NEG input is VR-)
  AD1CHSbits.CH0NA=0x0;
  // NEG input select for MUX B (1:NEG input is AN1/0:NEG input is VR-)
  AD1CHSbits.CH0NB=0x0;
  // ADC On bit (1:ADC Enable /0:ADC Disable) bit<15>
  AD1CON1bits.ON = 0;
  // Freeze in Debuge mode bit (1:Freeze op / 0:Continue op) bit<14>
  AD1CON1bits.FRZ = 0;
  // Stop in Idle mode bit (1:Stop ADC op / 0:Continue ADC op) bit<13>
  AD1CON1bits.SIDL = 0;
  // ADC Converison Trigger Source set to auto-convert bits<10:8>
  AD1CON1bits.SSRC = 7;
  // ADC result format as integer (16-bit) bits<7:5>
  AD1CON1bits.FORM = 0;
  AD1CON1bits.CLRASAM = 0;

  // Voltage Ref. Config. bits set to Vr+(Vdd) & Vr-(Vss) bits<15:13>
  AD1CON2bits.VCFG = 0;
  // Scan input selections for MUX A (1:Scan inputs / 0:Disable Scan inputs)
bit<10>
  AD1CON2bits.CSCNA = 0;
  // Sample/Convert Seq. Per Interrupt selection bits<5:2>
  AD1CON2bits.SMPI = 1;
  // ADC result buffer mode (1:two 8 word buffers / 0:one 16 word buffer)
bit<1>
  AD1CON2bits.BUFM = 1;
  // Alternate Input Sample Mode(1:ALT. between MUX A/B / 0:Disable ALT.)
bit<0>
  AD1CON2bits.ALTS = 1;
  AD1CON2bits.OFFCAL = 0;

  // ADC conversion CLK source (1:Internal RC CLK / 0:PBCLK)   bit<15>
  AD1CON3bits.ADRC = 0;
  // Auto-Sample Time bits<12:8> (31-Tad)
  AD1CON3bits.SAMC = 16;
  // ADC conv. CLK select bits<7:0> Tad=Tpb(ADCS+1)*2 (Tad=1us)
(Tpb=(80/8)=10MHz)
  AD1CON3bits.ADCS = 2;
  // Scan All (AN0-AN15) Analog inputs
  AD1CSSL= 0xFFFF;

}//End of ADC setup.
```

config_pwm.c

```
//============================================================================
/*

  Programmer: Jose L. Corona
  Project Name: PWM Setup Software for PIC32
  Date: 1/14/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
```

```
*/
//============================================================================
void setup_pwm(void);

/*****************************************************************
*    Function Name:  Setup PWM                                   *
*    Return Value:   NONE                                        *
*    Parameters:     NONE                                        *
*    Description:    This routine setup the PWM modules 1 & 2 for   *
*                    DC Motors.                                  *
*****************************************************************/
void setup_pwm()
{


/*---- Setup output compare 1 module to PWM ----*/
  // Output Compare Peripheral On bit (1:OC module ON/0:OC module OFF) bit<15>
  OC1CONbits.ON = 0;
  // Freeze in Debuge mode bit (1:Freeze op/0:Continue op) bit<14>
  OC1CONbits.FRZ = 0;
  // Stop in Idle mode bit (1:Stop USART op/0:Continue op) bit<13>
  OC1CONbits.SIDL = 0;
  // 32-bit Compare Mode bit (1:32-bit timer/0:16-bit timer) bit<5>
  OC1CONbits.OC32 = 0;
  // PWM Fault Condition Status bit (1:PWM fault cond./0:NO PWM fault cond.)
bit<4>
  OC1CONbits.OCFLT = 0;
  // Output Compare Timer Sel. bit (1:Timer3 is CLK src./0:Timer2 is CLK src.)
bit<3>
  OC1CONbits.OCTSEL = 0;
  // Output Comapre Mode Select bit (110:PWM mode on OCx, Fault pin disable)
bits<2:0>
  OC1CONbits.OCM = 6;

}//End of PWM setup.
```

config_timer_1.c

```
//============================================================================
/*

  Programmer: Jose L. Corona
  Project Name: Timer 1 Setup Software for PIC32
  Date: 1/15/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32

*/
//============================================================================
void setup_timer_1(void);

/*****************************************************************
*    Function Name:  Setup PWM                                   *
*    Return Value:   NONE                                        *
*    Parameters:     NONE                                        *
*    Description:    This routine setups up the Timer 1.         *
*                                                                *
*****************************************************************/
```

```
void setup_timer_1()
{

/*----- Setup output Timer1 module -----*/
  // Timer On bit (1:Timer2 Enable /0:Timer2 Disable) bit<15>
  T1CONbits.ON = 0;
  // Freeze in Debuge mode bit (1:Freeze op/0:Continue op) bit<14>
  T1CONbits.FRZ = 0;
  // Stop in Idle mode bit (1:Stop USART op/0:Continue op) bit<13>
  T1CONbits.SIDL = 0;
  // Asynchronous Timer Write Disable bit<12>
  T1CONbits.TWDIS = 0;
  // Asynchronous Timer Write in Progress bit<11>
  T1CONbits.TWIP = 0;
  // Gate Time Accul. En. bit (1:Gate Time Acc. En./0:Gate Time Acc. Dis.)
bit<7>
  T1CONbits.TGATE = 0;
  // Timer Input Clock Prescaler Select bits (00: 1:1 prescaler value)
bits<5:4>
  T1CONbits.TCKPS = 3;
  // Timer External Clock Input Synchronization Selection bit<2>
  T1CONbits.TSYNC = 0;

  // Timer Clock Source Select bit (1:External CLK/0:Internal CLK) bit<1>
  T1CONbits.TCS = 0;

  // Clear Timer2 Register
  TMR1 = 0;

}//End of Timer 1 setup.
```

config_timer_2.c

```
//============================================================================
/*

  Programmer: Jose L. Corona
  Project Name: Setup Timer 2 Software for PIC32
  Date: 2/19/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32

*/
//============================================================================
void setup_timer_2(void);

/*********************************************************************
*    Function Name:  Setup Timer 2                                   *
*    Return Value:   NONE                                            *
*    Parameters:     NONE                                             *
*    Description:    This routine setup Timer 2, used for PWM        *
*                    modules for DC Motors.                          *
```

```
*****************************************************************/
void setup_timer_2()
{

/*----- Setup output Timer2 module -----*/
  // Timer On bit (1:Timer2 Enable /0:Timer2 Disable) bit<15>
  T2CONbits.ON = 0;
  // Freeze in Debuge mode bit (1:Freeze op/0:Continue op) bit<14>
  T2CONbits.FRZ = 0;
  // Stop in Idle mode bit (1:Stop USART op/0:Continue op) bit<13>
  T2CONbits.SIDL = 0;
  // Gate Time Accl. En. bit (1:Gate Time Acc. En./0:Gate Time Acc. Dis.)
bit<7>
  T2CONbits.TGATE = 0;
  // Timer Input Clock Prescaler Select bits (000: 1:1 prescaler value)
bits<6:4>
  T2CONbits.TCKPS = 0;
  // 32-bit Timer Mode Se. bits (1: 32-bit timer/0: 16-bit timer) bit<3>
  T2CONbits.T32 = 0;
  // Timer Clock Source Select bit (1:External CLK/0:Internal CLK) bit<1>
  T2CONbits.TCS = 0;
  // Clear Timer2 Register
  TMR2 = 0x0000;

}//End of Timer 2 setup.
```

USART_32.h

```
//=============================================================================
/*
   Programmer: Jose L. Corona
   Project Name: PIC32 USART setup code
   Date: 2/20/09
   MCU: PIC32MX360F512L
   Complier: MPLAB C32
*/
//=============================================================================
//-----------------------------------------------------------------------------
#ifndef _USART_32_H
#define _USART_32_H

  //USART Selection
  #define USART_1          1      // Used For Logomatic Data Logger
  #define USART_2          2      // Used as a general purpose USART

  // MAX number of chars for float mantissa
  #define MAX_MANTISA 6
  // Number of chars
  #define MAX_STRING 10

  //PIC32 USART fxn's
  void TX_usart(char tx_sel, char data);
  void TX_string(char tx_string_sel, char *data);

  char RX_usart(char rx_sel);
  double string_RX(char rx_string_sel);
```

```
  void serial_digit(char digit_sel, short int data);
  void setup_usart(char usart_sel,int baud_rate);

  void enable_PLL(void);
  void disable_PLL(void);

  void float_to_text(char tx_string_sel, double number, char precision);
  int ftoa(double x, char *str, char  prec, char format);

  char rx_buffer[15];
  char x=0;

#endif
```

USART_32.c

```
//=============================================================================
/*
   Programmer: Jose L. Corona
   Project Name: PIC32 USART setup code
   Date: 1/06/09
   MCU: PIC32MX360F512L
   Complier: MPLAB C32
*/
//=============================================================================
//-----------------------------------------------------------------------------

#include "USART_32.h"

//=======>> The following Fxn's are used by the PIC32 USART <<=======

/*********************************************************************
*    Function Name:  setup_usart                                     *
*    Return Value:   none                                            *
*    Parameters:     none                                            *
*    Description:    This routine configures the USART               *
*                    1. USART is set as 8N1                          *
*                    2. Asynchronous Mode                            *
*                    3. Both RX ant TX interrupt disable             *
*                    4. RX set for Continuous Receive                *
*                    5. High Baud Rate set                           *
*********************************************************************/
//setup_usart(25);
void setup_usart(char usart_sel,int baud_rate)       // <<====
{

// Configure USART 2
if(usart_sel == 2)
{
  U2BRG = baud_rate;

  /*------ USARTx Mode Resigter ------*/

  // USARTx Enable bit (1:USART Enable/0:USART Disable) bit<15>
  U2MODEbits.ON = 1;
  // Freeze in Debuge mode bit (1:Freeze op/0:Continue op) bit<14>
  U2MODEbits.FRZ = 0;
  // Stop in Idle mode bit (1:Stop USART op/0:Continue op) bit<13>
```

```
  U2MODEbits.SIDL = 0;
  // IrDA Encoder and Decoder bit (1:Enable/0:Disable) bit<12>
  U2MODEbits.IREN = 0;
  // Mode select for UxRTS pin bit (1:Simplex mode/0:Flow control mode) bit<11>
  U2MODEbits.RTSMD = 0;
  // USARTx Enable bits<9:8>
  U2MODEbits.UEN = 0;
  // Enable Wake up on start bit detect (1:Wake-up enable/0:Wake-up disable)
bit<7>
  U2MODEbits.WAKE = 0;
  // USARTx Loopback mode sel. bit<6>
  U2MODEbits.LPBACK = 0;
  // Auto-Baud enable bit<5>
  U2MODEbits.ABAUD = 0;
  // RX polarity inversion bit (1:UxRX idle state '0'/0:UxRX idle state '1' )
bit<4>
  U2MODEbits.RXINV = 0;
  // High Baud rate enable bit (1: High spd. mode 4x/0:Std. spd. mode 16x)
bit<3>
  U2MODEbits.BRGH = 0;
  // Parity and Data selection bits <2:1>
  U2MODEbits.PDSEL = 0;
  // Stop selection bit (1:2 stop bits /0:1 stop bits) bit<0>
  U2MODEbits.STSEL = 0;
  /*---------------------------------*/

  /*----- USARTx Status Resigter -----*/
  // Automatic address detect mode enable bit (1:Enable/0:Disable) bit<24>
  U2STAbits.ADM_EN = 0;
  // Automatic address mask bits<23:16>
  U2STAbits.ADDR = 0;
  // TX Interrupt mode selection bits<15:14>
  U2STAbits.UTXISEL = 2;
  // TX polarity inv. bit (1:UxTX idle state '0'/0:UxTX idle state '1' )
bit<13>
  U2STAbits.UTXINV = 0;

  // Receiver enable bit (1:RX enable/0:RX disable) bit<12>
  U2STAbits.URXEN = 1;
  // Transmit break (1:Send Break / 0:Don't send break) bit<11>
  U2STAbits.UTXBRK = 0;
  // Transmit enable bit (1:TX enable / 0:TX disable) bit<10>
  U2STAbits.UTXEN = 1;
  // RX interrupt mode selection bits<7:6>
  U2STAbits.URXISEL = 0;
  // Address character detect bit (1: Enable / 0: Disable) bit<5>
  U2STAbits.ADDEN = 0;

  /*---------------------------------*/
}
  else   // Configure USART 1
  {
      U1BRG = baud_rate;

      /*------ USARTx Mode Resigter ------*/
      // USARTx Enable bit (1:USART Enable/0:USART Disable) bit<15>
      U1MODEbits.ON = 1;
      // Freeze in Debuge mode bit (1:Freeze op/0:Continue op) bit<14>
      U1MODEbits.FRZ = 0;
      // Stop in Idle mode bit (1:Stop USART op/0:Continue op) bit<13>
      U1MODEbits.SIDL = 0;
```

```
      // IrDA Encoder and Decoder bit (1:Enable/0:Disable) bit<12>
      U1MODEbits.IREN = 0;
      // Mode select for UxRTS pin bit (1:Simplex mode/0:Flow control mode)
bit<11>
      U1MODEbits.RTSMD = 0;
      // USARTx Enable bits<9:8>
      U1MODEbits.UEN = 0;
      // Enable Wake up on start bit detect (1:Wake-up en./0:Wake-up dis.)
bit<7>
      U1MODEbits.WAKE = 0;
      // USARTx Loopback mode sel. bit bit<6>
      U1MODEbits.LPBACK = 0;
      // Auto-Baud enable bit<5>
      U1MODEbits.ABAUD = 0;
      // RX polarity inversion bit<4>
      U1MODEbits.RXINV = 0;
      // High Baud rate enable bit (1: High spd. mode 4x/0:Std. spd. mode 16x)
bit<3>
      U1MODEbits.BRGH = 0;
      // Parity and Data selection bits <2:1>
      U1MODEbits.PDSEL = 0;
      // Stop selection bit (1:2 stop bits /0:1 stop bits) bit<0>
      U1MODEbits.STSEL = 0;
      /*---------------------------------*/

      /*----- USARTx Status Resigter -----*/
      // Automatic address detect mode enable bit (1:Enable/0:Disable) bit<24>
      U1STAbits.ADM_EN = 0;
      // Automatic address mask bits<23:16>
      U1STAbits.ADDR = 0;
      // TX Interrupt mode selection bits<15:14>
      U1STAbits.UTXISEL = 2;
      // TX polarity inv. bit (1:UxTX idle state '0'/0:UxTX idle state '1' )
bit<13>
      U1STAbits.UTXINV = 0;

      // Receiver enable bit (1:RX enable/0:RX disable) bit<12>
      U1STAbits.URXEN = 1;
      // Transmit break (1:Send Break / 0:Don't send break) bit<11>
      U1STAbits.UTXBRK = 0;
      // Transmit enable bit (1:TX enable / 0:TX disable) bit<10>
      U1STAbits.UTXEN = 1;
      // RX interrupt mode selection bits<7:6>
      U1STAbits.URXISEL = 0;
      // Address character detect bit (1: Enable / 0: Disable) bit<5>
      U1STAbits.ADDEN = 0;

  }

}
```

```
/*****************************************************************
*    Function Name:  string_RX                                   *
*    Return Value:   NONE                                        *
*    Parameters:     NONE                                        *
```

```
*    Description:    This routine Receives a string from the USART  *
*****************************************************************/
double string_RX(char rx_string_sel)
{
   unsigned char temp_buffer;
//---- Place the received byte into the rx_buffer[] ----
   for(temp_buffer=0; temp_buffer < 16; temp_buffer++)
   {
     rx_buffer[temp_buffer] = RX_usart(rx_string_sel);

     if(rx_buffer[temp_buffer] == '\r')
     {
          break;
       }

   }
//---- Now transmit(echo back) the rx_buffer[] that holds the string ----
/*   for(x=0; x < 10; x++)
   {
       if(rx_buffer[x] == '\r')
       {
           break;
         }

       TX_usart(rx_string_sel, rx_buffer[x]);
   }
*/
//---- Use the atof() ----

return(atof(rx_buffer));

}
/*****************************************************************
*    Function Name:  RX_usart                                    *
*    Return Value:   Value that is on the RCREG                  *
*    Parameters:     rx_sel: RX from either USART1 or USART2     *
*    Description:    This routine Receives a byte from the USART *
*                    Poll for received character.                *
*****************************************************************/
char RX_usart(char rx_sel)       // <<====
{
// RX from USART 2
if(rx_sel==2)
{
    // Overrun error occurred
    if (U2STAbits.OERR==1)
    {
          // ERROR cleared Previously OERR to '0'
         U2STAbits.OERR=0;
    }

     // while U2RXREG is empty, keep polling
     while(!U2STAbits.URXDA);

     // return what ever is in the U2RXREG register
     return U2RXREG;
}
else   // RX from USART 1
{
    // Overrun error occurred
    if (U1STAbits.OERR==1)
```

```
    {
       // ERROR cleared Previously OERR to '0'
        U1STAbits.OERR=0;
    }

     // while U1RXREG is empty, keep polling
     while(!U1STAbits.URXDA);

     // return what ever is in the U1RXREG register
     return U1RXREG;
}

}
/*********************************************************************
*    Function Name:  TX_string                                       *
*    Return Value:   void                                            *
*    Parameters:     data: pointer to string of data                 *
*    Description:    This routine transmits a string of characters   *
*                    to the USART including the null.                *
*********************************************************************/
void TX_string(char tx_string_sel, char *data)       // <<====
{
  // Transmit a Byte, stop after null is transmitted
  while(*data)
  {
     // Point(increment) to next character to be transmitted
     TX_usart(tx_string_sel,*data++);
  }

}
/*********************************************************************
*    Function Name:  TX_usart                                       *
*    Return Value:   none                                           *
*    Parameters:     data to transmit                               *
*    Description:    This routine transmits a byte out the USART    *
*                    after the transmit ready bit goes high         *
*                    Transmit byte and wait for ready.              *
*********************************************************************/
void TX_usart(char tx_sel, char data)   // <<====
{
// TX out the USART 2
if(tx_sel==2)
{
    //Load data to the TXREG
    U2TXREG = data;

    //while TXREG is FULL, keep sending data
    while(!U2STAbits.TRMT);

}
   else      // TX out the USART 1
   {

       //Load data to the TXREG
       U1TXREG = data;

       //while TXREG is FULL, keep sending data
       while(!U1STAbits.TRMT);

    }
}
```

```
/*********************************************************************
*    Function Name:  ser_digit                                       *
*    Return Value:   void                                            *
*    Parameters:     data                                            *
*    Description:    This routine transmits a character out the      *
*                    serial port as a 4 digit number.                *
*********************************************************************/
void serial_digit(char digit_sel, short int data)    // <<====
{
    TX_usart(digit_sel, '0' + data/10000 );          //extract/send first digit
    TX_usart(digit_sel, '0' + (data%10000)/1000 );   //extract/send second
digit
    TX_usart(digit_sel, '0' + (data%1000)/100 );     //extract/send third digit
    TX_usart(digit_sel, '0' + (data%100)/10 );       //extract/send fourth
digit
    TX_usart(digit_sel, '0' + data%10 );             //extract/send fifth digit
}
/*********************************************************************
*    Function Name:  putsf()                                         *
*    Return Value:   string legnth                                   *
*    Parameters:     x, string, precision                            *
*    Description:    prints float out the USART                      *
*********************************************************************/
void float_to_text(char tx_string_sel, double number, char precision)
{
      char string[MAX_STRING];

      ftoa(number, string, precision, 'f');

      // Print(Send) data out of USART
      TX_string(tx_string_sel, string);
}
/*********************************************************************
*    Function Name:  ftoa()                                          *
*    Return Value:   string legnth                                   *
*    Parameters:     x, string, precision, format                    *
*    Description:    Converts float to ascii                         *
*                                                                    *
*    Note: Code was adpated from Trampas Stern.                      *
*********************************************************************/
int ftoa (double x, char *str, char  prec, char format)
{

      int ie, i, k, ndig, fstyle;
      double y;
      char *start;

      start=str;

      // Based on precession set number digits
      ndig=prec+1;
      if(prec<0)
      {
          ndig=7;
      }

      if(prec>22)
      {
         ndig=23;
      }
      // Exponent 'e'
```

```
fstyle = 0;

if(format == 'f' || format == 'F')
{
    //normal 'f'
    fstyle = 1;
}
if(format=='g' || format=='G')
{
    fstyle=2;
}
ie = 0;

// If x negative, write minus and reverse
if( x < 0)
{
  *str++ = '-';
  x = -x;
}

// If (x<0.0) then increment by 10 till betwen 1.0 and 10.0
if(x!=0.0)
{
    while (x < 1.0)
    {
       x= x* 10.0;
       ie--;
    }
}

// If x>10 then let's shift it down
while(x >= 10.0)
{
       x = x*(1.0/10.0);
       ie++;
}

if(abs(ie)>MAX_MANTISA)
{
    if(fstyle==1)
    {
       fstyle=0;
       format='e';
    }
}

// In f format, number of digits is related to size
if(fstyle)
{
    ndig= ndig + ie;
}

if(prec==0 && ie>ndig && fstyle)
{
   ndig=ie;
}


// Round x is between 1 and 10 and ndig will be printed to
// right of decimal point
y=1;
```

```
       // Find lest significant digit
       for (i = 1; i < ndig; i++)
         y = y *(1.0/10.0);             // Multiply by 1/10 is faster than
divides

         x = x+ y *(1.0/2.0);           // Add rounding

       // Repair rounding disasters
       if(x >= 10.0)
       {
          x = 1.0;
          ie++;
          ndig++;
       }

       // Check and see if the number is less than 1.0
       if(fstyle && ie<0)
       {
           *str++ = '0';
           if(prec!=0)
           {
              *str++ = '.';
           }
            if(ndig < 0)
            {
               ie = ie-ndig;   // Limit zeros if underflow
            }
             for (i = -1; i > ie; i--)
             {
                 *str++ = '0';
             }
        }

       // For each digit
       for (i=0; i < ndig; i++)
       {
              float b;
              k = x;                      // k = most significant digit
              *str++ = k + '0';           // Output the char representation
              if((((!fstyle && i==0) || (fstyle && i==ie)) && prec!=0)
              {
                // Output a decimal point
                 *str++ = '.';
              }
               b=(float)k;
              // Multiply by 10 before subtraction to remove
              // errors from limited number of bits in float.
              b=b*10.0;
              x=x*10.0;
             // Subtract k from x
              x =x - b;

       }

       // Now, in estyle, put out exponent if not zero
       if(!fstyle && ie != 0)
       {
              *str++ = format;
              if(ie < 0)                     // If number has negative exponent
              {
```

```
                ie = -ie;
                *str++ = '-';
        }

        // Now we need to convert the exponent to string
        for (k=1000; k>ie; k=k/10);    // Find the decade of exponent

        for (; k > 0; k=k/10)
        {
                char t;
                *str++ = t + '0';
                ie = ie -(t*k);
        }

    }
    *str++ = '\0';
     //return string length
    return (str-start);
}
```

accel.c

```
//================================================================================
/*
  Programmer: Jose L. Corona
  Project Name: Read (get) the accelerometer ADC reading.
  Date: 2/21/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//================================================================================

#define zero_g_offset      (2.520117*1024)/3.3

// (3.3V)/1024= 0.00322265625 mV/bit
#define adc_res_value      0.00322265625

// (1g/1000mV)*(0.00322265625mV/bit)= 0.00322265625 mg/bit
#define accel_mg_res       0.00322265625

// Constants
#define pi                 3.1415926535898
#define degrees            (180.0/pi)

// Fxn declaration
double read_accel(int accel_value, char mode);

/*****************************************************************
*    Function Name:  read_accel                                  *
*    Return Value:   tilt angle(degrees / radians)               *
*    Parameters:     accelerometer value, mode                   *
*    Description:    accelerometer ADC reading                   *
*****************************************************************/
double read_accel(int accel_value, char mode)
{
  double adc_accel_result= 0.0;
  double accel_reading= 0.0;
  //double adc_accel_mV= 0.0;
```

```
  double accel_deg= 0.0;
  double mg= 0.0;

  //---- Accel. Data Conversion ----
   adc_accel_result= (double)accel_value;

   // Convert ADC value to mV units
   //adc_accel_mV= adc_res_value*(adc_accel_result-2.0);

   // Calculate ADC mV --> rad (mG)
   mg=((adc_accel_result-2.0)-zero_g_offset)*accel_mg_res;

   //---- Check for out +/- 1G range ----
   if( mg>= 1.0 )
   {
      mg= 1.0;
   }
     else if( mg<= -1.0 )
     {
        mg= -1.0;
     }

   //---- Choose between Degrees/Radians ----
   if(mode == 'd')
   {
        // Result in degrees
        accel_deg= asin(mg);
        accel_reading= accel_deg*degrees;
   }
   else if(mode == 'r')
   {
        // Result in radians
        accel_reading= asin(mg);
   }

       return accel_reading;
}
```

gyro.c

```
//============================================================================
/*
  Programmer: Jose L. Corona
  Project Name: Read (get) the gyroscope ADC reading.
  Date: 2/21/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//============================================================================

#define zero_gyro_offset    (1.5000*1024)/3.3

// (3.3V)/1024= 0.00322265625 mV/bit
#define adc_res_value       0.00322265625

//(0.00322265625 mV)/(2mV/degree/sec)= 1.6113281 degree/sec (0.0281230 rad/sec)
//#define gyro_res_deg      1.6113281
#define gyro_res_rad        0.0281230
```

```
// Constants
#define pi                  3.1415926535898
#define degrees             (180.0/pi)

// Fxn declaration
double read_gyro(int gyro_value, char mode);
```

```
/*****************************************************************
*    Function Name:  read_accel                                  *
*    Return Value:   tilt angle(degrees / radians)               *
*    Parameters:     accelerometer value, mode                   *
*    Description:    accelerometer ADC reading                   *
*****************************************************************/
double read_gyro(int gyro_value, char mode)
{
   double adc_gyro_result= 0.0;
   double gyro_reading= 0.0;
   //double adc_gyro_mV= 0.0;
   double angle_rate= 0.0;
   double gyro_deg= 0.0;

   //---- Gyro. Data Conversion ----
   adc_gyro_result= (double)gyro_value;

   // Convert ADC value to mV units
   //adc_gyro_mV= adc_res_value*(adc_gyro_result-2.0);

   // Calculate ADC mV --> rad/s
   angle_rate= ((adc_gyro_result-2.0)-zero_gyro_offset)*gyro_res_rad;

   //---- Choose between Degrees/Radians ----
   if(mode == 'd')
   {
        // Result in degrees
        gyro_reading= angle_rate*degrees;
   }
   else if(mode == 'r')
   {
        // Result in radians
        gyro_reading= angle_rate;
   }

   return gyro_reading;

}
```

knight_rider.c

```
//===========================================================================
/*
  Programmer: Jose L. Corona
  Project Name: Knight Rider LED Effect
  Date: 2/19/09
```

```
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//============================================================================
#define LED_DELAY   250000

void knight_rider(void);

void knight_rider(void)
{

   //Set array declaration
   const char num[]={1,2,4,8,16,32,64,128,64,32,16,8,4,2,1};
   char count;
   int i, j;

// Scan the array structure 10 times.
for(i=0; i<10; i++)
{
    // Clear PORTA
    PORTA=0x0000;

    // Small delay
    for(j=0; j< LED_DELAY; j++);

       // Scan the array structure elements.
       for(count= 0;count <= 15; count++)
       {
           // Small delay.
           for(j=0; j< LED_DELAY; j++);
                  // Display array element on LED's an increment to next element
array.
           PORTA= num[count];

       }

}//End of main for loop
    // Clear PORTA
    PORTA=0x0000;
}
```

kalman.h

```
//============================================================================
/*
  Programmer: Jose L. Corona
  Project Name: ADC Test Software using Kalman Filter for PIC32
  Date: 2/15/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//============================================================================
#ifndef KALMAN_H
#define KALMAN_H

typedef struct
{
    // Two states, angle and gyro bias. Unbiased angular rate is a byproduct.
    double x_bias;
```

```
    double x_rate;
    double x_angle;

    // Covariance of estimation error matrix.
    double P_00;
    double P_01;
    double P_10;
    double P_11;

    // State constants.
    double dt;
    double R_angle;
    double Q_gyro;
    double Q_angle;

} kalman;



void kalman_init(kalman *filter, double dt, double R_angle, double Q_gyro,
double Q_angle);
void kalman_predict(kalman *filter, double gyro_rate);
void kalman_update(kalman *filter, double angle_measured);

// Get the bias.
double kalman_get_bias(kalman *filter)
{
    return filter->x_bias;
}
// Get the rate.
double kalman_get_rate(kalman *filter)
{
    return filter->x_rate;
}
// Get the angle.
double kalman_get_angle(kalman *filter)
{
    return filter->x_angle;
}

#endif
```

kalman.c

```
//===========================================================================
/*
  Programmer: Jose L. Corona
  Project Name: ADC Test Software using Kalman Filter for PIC32
  Date: 2/15/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//===========================================================================
#include "kalman.h"

/*****************************************************************
 The implememted Kalman filter estimates the angle that will be used
 on the PID controller alogrithm. The filter consists of two states
 [angle, gyro_bais].
```

```
*********************************************************************/

/*********************************************************************
*    Function Name:  kalman_init                                     *
*    Return Value:   none                                            *
*    Parameters:     struct filter, dt, R_angle, Q_gyro, Q_angle     *
*    Description:    Initialize the Kalman Filter parameters.        *
*********************************************************************/
void kalman_init(kalman *filter, double dt, double R_angle, double Q_gyro,
double Q_angle)
{
    // Initialize the two states, the angle and the gyro bias. As a
    // byproduct of computing the angle, we also have an unbiased
    // angular rate available.
    filter->x_bias = 0.0;
    filter->x_rate = 0.0;
    filter->x_angle = 0.0;



    // Initialize the delta in seconds between gyro samples.
    filter->dt = dt;

    // Initialize the measurement noise covariance matrix values.
    // In this case, R is a 1x1 matrix tha represents expected
    // jitter from the accelerometer.
    filter->R_angle = R_angle;

    // Initialize the process noise covariance matrix values.
    // In this case, Q indicates how much we trust the acceleromter
    // relative to the gyros.
    // Q_gyro set to 0.003 and Q_angle set to 0.001.
    filter->Q_gyro = Q_gyro;
    filter->Q_angle = Q_angle;

    // Initialize covariance of estimate state.  This is updated
    // at every time step to determine how well the sensors are
    // tracking the actual state.
    filter->P_00 = 1.0;
    filter->P_01 = 0.0;
    filter->P_10 = 0.0;
    filter->P_11 = 1.0;
}
/*********************************************************************
*    Function Name:  kalman_predict                                  *
*    Return Value:   none                                            *
*    Parameters:     struct filter, measured gyroscope value         *
*    Description:    Called every dt(Timer 1 overflow with a biased  *
*                    gyro. Also updates the current rate and angle   *
*                    estimate).                                      *
*********************************************************************/
void kalman_predict(kalman *filter, double dot_angle)
{
    // Static so these are kept off the stack.
    static double gyro_rate_unbiased;
    static double Pdot_00;
    static double Pdot_01;
    static double Pdot_10;
    static double Pdot_11;
```

```
    // Unbias our gyro.
    gyro_rate_unbiased= dot_angle - filter->x_bias;

    // Store our unbiased gyro estimate.
    filter->x_rate= gyro_rate_unbiased;

    // Update the angle estimate.
    filter->x_angle= filter->x_angle + (dot_angle - filter->x_bias)*filter->dt;

    // Compute the derivative of the covariance matrix
    // Pdot = A*P + P*A' + Q
    Pdot_00 = filter->Q_angle - filter->P_01 - filter->P_10;
    Pdot_01 = -filter->P_11;
    Pdot_10 = -filter->P_11;
    Pdot_11 = filter->Q_gyro;

    // Update the covariance matrix.
    filter->P_00 += Pdot_00 * filter->dt;
    filter->P_01 += Pdot_01 * filter->dt;
    filter->P_10 += Pdot_10 * filter->dt;
    filter->P_11 += Pdot_11 * filter->dt;

}
/*****************************************************************
*    Function Name:  kalman_update                                *
*    Return Value:   none                                         *
*    Parameters:     struct filter, measured angle value          *
*    Description:    Called when a new accelerometer angle        *
*                    measurement is available. Updates the estimated*
*                    angle that will be used.                     *
*****************************************************************/
void kalman_update(kalman *filter, double angle_measured)
{
    // Static so these are kept off the stack.
    static double y;
    static double S;
    static double K_0;
    static double K_1;

    // Compute the error in the estimate.
    // Innovation or Measurement Residual
    // y = z - Hx
    y= angle_measured - filter->x_angle;

    // Compute the error estimate.
    // S = C P C' + R
    S = filter->R_angle + filter->P_00;

    // Compute the Kalman filter gains.
    // K = P C' inv(S)
     K_0 = filter->P_00 / S;
     K_1 = filter->P_10 / S;

    // Update covariance matrix.
    // P = P - K C P
    filter->P_00 -= K_0 * filter->P_00;
    filter->P_01 -= K_0 * filter->P_01;
    filter->P_10 -= K_1 * filter->P_00;
    filter->P_11 -= K_1 * filter->P_01;

    // Update the state (new)estimates (Correct the prediction of the state).
```

```
    // Also adjust the bias on the gyro at every iteration.
    // x = x + K * y
    filter->x_angle= filter->x_angle + K_0 * y;
    filter->x_bias= filter->x_bias + K_1 * y;

}
```

pid.h

```
//================================================================================
/*
  Programmer: Jose L. Corona
  Project Name: PID algorithm Software for PIC32
  Date: 2/19/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//================================================================================
#ifndef PID_H
#define PID_H



// Data structure for setpoints and data used by
// the PID control algorithm
// Maximum value of variables (32767.00)
// Set limits to a overflow of sign problems
#define MAX_INT        16000.00
#define MAX_I_TERM     250.00

typedef struct
{
  // Maximum allowed error, avoid overflow
  double max_error;

  // Maximum allowed sum error, avoid overflow
  double max_sum_error;

  // Summation of errors, used for integrate calculations
  double sum_error;
```

```
  // Last measured value, used to find derivative of process value.
  double prev_measured_value;

  // The Proportional tuning constant, multiplied with SCALING_FACTOR
  double P_Factor;

  // The Integral tuning constant, multiplied with SCALING_FACTOR
  double I_Factor;

  // The Derivative tuning constant, multiplied with SCALING_FACTOR
  double D_Factor;

} pid;

// Initialize the PID controller
void pid_setup(double p, double i, double d, pid *set_pid)

// PID controller
double pid_controller(pid *pid_get, double sys_error, double y);

#endif
```

pid.c

```
//=============================================================================
/*
  Programmer: Jose L. Corona
  Project Name: PID Algorithm Software for PIC32
  Date: 2/19/09
  MCU: PIC32MX360F512L
  Complier: MPLAB C32
*/
//=============================================================================

#include "pid.h"

#define LED_1 PORTAbits.RA0
#define LED_2 PORTAbits.RA1
#define LED_3 PORTAbits.RA2
#define LED_4 PORTAbits.RA3
#define LED_5 PORTAbits.RA4
#define LED_6 PORTAbits.RA5
#define LED_7 PORTAbits.RA6
#define LED_8 PORTAbits.RA7

#define PID_Limit   375.0

//=======>> The following Fxn's are used for PID Controller <<=======

/*******************************************************************
*    Function Name:  pid_setup                                     *
*    Return Value:   none                                          *
*    Parameters:     P,I,D factors, struct pid                     *
*    Description:    Set up PID controller parameters              *
*                    1. p_factor  Proportional term                *
```

```
*                   2. i_factor  Integral term                    *
*                   3. d_factor  Derivate term                    *
*                   4. struct pid                                 *
*****************************************************************/
void pid_setup(double p, double i, double d, pid *set_pid)
{
    // Clear values for PID controller
    set_pid->sum_error= 0.00;
    set_pid->prev_measured_value= 0.00;

    // Tuning constants for PID loop
    set_pid->P_Factor = p_factor;
    set_pid->I_Factor = i_factor;
    set_pid->D_Factor = d_factor;

    // Limits to avoid integral overflow or windup
    set_pid->max_error = 8000.0;
    set_pid->max_sum_error = 250.0;









}
/*********************************************************************
*    Function Name:  pid_controller                                  *
*    Return Value:   the (u) which is fed to the process             *
*    Parameters:     struct pid, reference point, process output (y) *
*    Description:    PID control algorithm. Calculates output from   *
*                    setpoint, process value and PID status.         *
*                    1. ref --> Desired value                         *
*                    2. y --> Measured process value.                 *
*                    3. struct pid                                    *
*                                                                     *
*                     _________________          ___________         *
*  ref   ____  error |                 |    u   |           |   y    *
* ----->|    |------>| PID Controller |------->| DC Motors |------>   *
*       |____|       |_________________|        |___________|  |      *
*        ^                                                    |      *
             *        |                                                |
*
*         |                     _____________                  |      *
*         |                    |             |                 |      *
*         |--------------------| IMU Sensors |-----------------|      *
*                              |_____________|                        *
*                                                                     *
*********************************************************************/
double pid_controller(pid *pid_get, double sys_error, double y)
{
  LED_2= 0;
  LED_3= 0;
  LED_4= 0;
  LED_5= 0;

  double p_term;
  double i_term;
```

```
  double d_term;
  double u;

//------ Calculate Proportional term and limit error overflow ------
if(sys_error > pid_get->max_error)
{
      LED_2= 1;
      p_term = MAX_INT;
}
 else if(sys_error < -pid_get->max_error)
 {
      LED_3= 1;
      p_term = -MAX_INT;
 }
  else
  {
       p_term = pid_get->P_Factor * sys_error;
  }

//------ Calculate Integral term and limit integral windup/runaway ------
  pid_get->sum_error += sys_error;

  if(pid_get->sum_error > pid_get->max_sum_error)
  {
      LED_4= 1;
      i_term = MAX_I_TERM;
      pid_get->sum_error = pid_get->max_sum_error;
  }
  else if(pid_get->sum_error < -pid_get->max_sum_error)
  {
      LED_5= 1;
      i_term = -MAX_I_TERM;
      pid_get->sum_error = -pid_get->max_sum_error;
  }
   else
  {
    i_term = pid_get->I_Factor * pid_get->sum_error;
  }

//------ Calculate Derivate term ------
  d_term = pid_get->D_Factor * (pid_get->prev_measured_value - y);
  pid_get->prev_measured_value= y;

//------ Sum the P,I,D factors to get a controller output ------
  u= p_term + i_term + d_term;

//------ Limit the Controller Output ------
  if(u > PID_Limit)
  {
     u= PID_Limit;
  }
  else if(u < -PID_Limit)
  {
    u= -PID_Limit;
  }

  return u;
}
```