FEEDBACK SPEED CONTROL OF A SMALL TWO-STROKE INTERNAL COMBUSTION ENGINE THAT PROPELS AN UNMANNED AERIAL VEHICLE

by

Paul D. Fjare

Bachelor of Science in Mechanical Engineering
University of Nevada, Las Vegas

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Engineering - Mechanical Engineering

Department of Mechanical Engineering
College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2014

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Paul D. Fjare

entitled

Feedback Speed Control of a Small Two-Stroke Internal Combustion Engine that Propels an Unmanned Aerial Vehicle

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Engineering - Mechanical Engineering
Department of Mechanical Engineering

William Culbreth, Ph.D., Committee Chair

Robert Boehm, Ph.D., Committee Member

Woosoon Yim, Ph.D., Committee Member

Sahjendra Singh, Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

August 2014

Abstract

Unmanned aerial vehicles (UAV) require intelligent control of their power source. Small UAV are typically powered by electric motors or small two-stroke internal combustion (IC) engines. Small IC engines allow for longer flight times but are more difficult to control and cause significant ground noise. A hybrid operation that uses the engine at high altitudes and the electric motors at low altitudes is desired. This would allow for extended flight with acceptable ground noise levels. Since the engine can not be restarted in the air it must be able to remain at idle for an extended time without stalling. A feedback controller is created for an OS160FX carbureted two-stroke engine. The controller implements a Proportional-Integral-Derivative (PID) algorithm to regulate the rotational speed of the engine shaft. The controller also monitors the temperature of the engine and is capable of monitoring the altitude of the aircraft. It is constructed with commercially available components and is based on an open-source micro-controller. The engine and the controller were ground tested to determine the engine's performance characteristics and the appropriate tuning parameters of the PID algorithm. The controller allows the engine to idle at 1800 rpm without stalling. The controller is able to quickly respond to changes in the commanded speed and settle on this speed within 10 seconds. The speed is regulated through the engine's full range of speeds. The performance of the controller was found to be negatively affected by sub-optimal carburetor fuel-valve settings.

Acknowledgments

First, I would like to thank my parents for encouraging my education and supporting me in all my endeavors. Next, I would like to thank my research advisor, Dr. Culbreth, for giving me the opportunity to work on this project. Also, I would like to thank Dr. Boehm, Dr. Singh, and Dr. Yim for serving on my committee.

A special thank you to my father for assisting me. I would not have completed this research without his help.

Table of Contents

List of Tables

List of Figures

Nomenclature

AFR	Air-to-fuel ratio.
HSN	High-speed needle valve.
IC	Internal combustion.
IRE	Infrared reflectance encoder.
ISR	Interrupt service routine.
LSN	Low-speed needle valve.
MAX6675	MAX6675 K-Thermocouple-to-Digital Converter.
MC1	Arduino Duemilanove micro-controller.
MC2	Arduino Mega 2560 micro-controller.
MPL115A1	MPL115A1 barometric pressure sensor.
OS160FX	O.S. Engines 160FX glow engine.
PID	Proportional-Integral-Derivative.
UAV	Unmanned aerial vehicle.

1 Introduction

1.1 Background

In recent years there have been efforts to develop small unmanned aerial vehicles (UAV) capable of extended flight. UAV have piloting systems with embedded intelligence that determines the trajectory of the aircraft [1]. The piloting system or auto-pilot sends the appropriate commands to the aircraft's power source required to maintain the desired flight speed. Small aircraft typically rely on propeller propulsion that is powered by electric motors or small internal combustion (IC) engines.

Electric motors have the advantage of simple control, quiet operation, and low maintenance. Small IC engines allow for longer flight times than electric motors but are more difficult to control. Their performance is dependent on ambient temperature, altitude, and fuel flow. They also generate significant ground noise and require regular maintenance.

A UAV was developed at the University of Nevada, Las Vegas through a 2007 research grant from the Air Force Research Laboratory [1]. The UAV can be powered by a single small two-stroke diesel engine or by two small electric motors. The current piloting system on the UAV sends operating commands to the engine in the form of throttle position. The next step in this research is to design and build a digital feedback controller that can bring the engine to a rotational speed commanded by the auto-pilot and maintain that speed.

At high altitudes the ground noise generated by an engine is less than at low altitudes.

A hybrid operation that uses an IC engine at high altitudes and switches to the electric motors at low altitudes would potentially allow for acceptable ground noise levels and extended flight. The IC engines on small aircraft typically are started by hand and can not be re-started in flight. This means that the engine must run at idle while the electric motors are operating. The engine is at idle when it is rotating at its lowest running speed. At this speed the engine does not produce significant thrust. The noise created by the engine is significantly lower when idling then when running at high speeds so this is an acceptable operation. This hybrid operation requires an engine that will run steadily at low speeds without stalling. The new engine controller should allow for this hybrid operation.

1.2 Research Objectives

The main objective of this research was to build and implement an inexpensive digital feedback controller that regulates the speed of a small two-stroke IC engine that propels a small UAV. The representative engine was the O.S. Engines 160FX (OS160FX) carbureted two-stroke engine that is popular in the recreational aircraft community. It was desired that the controller hardware be based on commercially available parts that can be adapted for use on engines comparable to the OS160FX. The controller software and the implemented control algorithm were to be written so that they could easily be adapted by future operators. The requirements for the engine controller were that it allow the engine to quickly and reliably change to a desired speed and maintain that speed within a defined tolerance. Another requirement for the engine controller was that it allow the engine to idle for an extend period without stalling. The controller also had to monitor the temperature of the engine and the altitude of the aircraft.

A secondary objective was to determine the sensitivity of the controller to carburetor settings. The carburetor on the OS160FX and other small two-stroke engines is typically adjusted or ‘tuned’ manually in the field. These tunings are crude and require an experienced operator to perform. Minor adjustments can cause significant variations of the engine’s performance characteristics.

1.3 Research Methodology

The first step to achieving the research goals was to set up a facility to test the engine on the ground. For this research the engine was not tested on an aircraft in flight. The facility chosen was a garage area that had adequate ventilation. A test stand was built that supports the engine, fuel tank, and controller hardware.

The next step was to set up a control system that could adjust the engine’s throttle and monitor the engine’s performance. The control system consisted of the engine controller and a ground station that communicated wirelessly. The engine controller was based on an Arduino micro-controller which read the sensors, implemented the control algorithm, and sent a control signal to a small DC servo motor that served as the throttle actuator. The throttle position was set manually by entering commands in a laptop computer or was determined by the control algorithm. The sensor values were printed to a small LCD screen so the engine could be monitored in real-time. The sensor values and other data related to the control algorithm were logged for analysis.

After the control system was in place, the engine was put through a ‘break in’ period that is required of new engines and the carburetor fuel valves were adjusted or ‘tuned’ to provide optimal performance. The break-in period involved running the engine at low-speeds for short periods of time. The speed and run-time were gradually increased as the engine ran through several tanks of fuel. The purpose of the break-in period was to allow the engine parts to settle together. Once the break-in period was completed the carburetor could be tuned for optimal engine performance. The fuel valves were set so the engine could reach high-speeds without overheating, have smooth acceleration throughout the speed range, and idle reliably.

After the carburetor was tuned, the engine’s performance characteristics were determined. The performance characteristics were determined by recording and analyzing the response of the engine speed to changes in the throttle position. The test results with different carburetor tunings were used to determine the variability in engine performance that results from imprecise carburetor adjustments.

Next, the control algorithm was written, implemented, and tested. A Proportional-Integral-Derivative (PID) algorithm was chosen because it can be implemented with inexpensive micro-controllers and it is a widely adopted and proven control method. There are three tuning parameters of the PID algorithm that must be carefully chosen for the controller to provide acceptable control of a dynamic process. Tests were run with different combinations of these parameters to determine a suitable setting for each parameter.

2 Literature Review

2.1 Engine Theory

An internal combustion (IC) engine converts chemically bound fuel energy into mechanical energy [2]. In an IC engine the combustion of an air-fuel mixture in a combustion chamber and the resulting expansion of high-pressure gases drives a piston that oscillates in a cylinder. The piston is attached to a connection rod and crankshaft linkage that translates the reciprocating motion of the piston to the angular rotation of an output shaft [3]. On an aircraft engine the output shaft is connected to a propeller to generate thrust [4].

The first type of IC engine is a four-stroke engine. In a four-stroke engine there are four strokes of the piston required to complete a cycle. Each stroke performs a different function of the cycle. The functions of an engine cycle are intake, compression, power, and exhaust. The second type of IC engine is two-stroke engine which completes the cycle functions in just two strokes of the piston. The two-stroke engine is smaller and simpler in operation than the four-stroke engine and produces an equivalent amount of power [5].

2.1.1 Two-Stroke Engines

The sequence of events that occur in a typical two-stroke cycle are shown in Figure 2.1. The cycle starts with the intake of air and fuel that have been mixed with the carburetor. As the piston is traveling upward there is a drop in crankcase pressure. The air-fuel mixture from the carburetor, driven by the difference between atmospheric and crankcase pressures, enters the crankcase. This air-fuel mixture remains in the crankcase until the piston opens

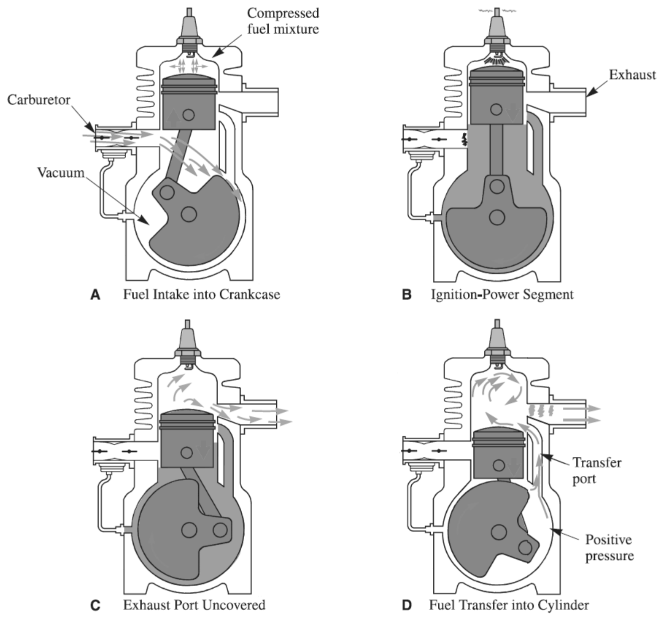


Figure 2.1: Sequence of events in a two-stroke engine cycle from [5].

up the transfer port on the down-stroke. Once the transfer port is open the air-fuel mixture in the crankcase, which has been compressed by the piston, rushes into the combustion chamber. This mixture is further compressed during the upstroke of the piston until it ignites due to the heat of compression or spark ignition. The combustion of the air-fuel mixture produces high pressure gases that expand rapidly and force the piston downward. As the piston moves downward it uncovers the exhaust port which allows the products of combustion to exit the chamber. The compression and intake functions both occur on the up stroke while the power and exhaust function both occur on the down stroke or power stroke [5].

2.1.2 Scavenging Theory

The process of pushing out exhaust gases and drawing in a fresh air-fuel mixture is called scavenging. In a properly scavenged engine all exhaust gases are pushed out of the combustion chamber and a fresh air-fuel mixture is drawn in from the crankcase. The two main types of scavenging configurations are cross-scavenging and loop-scavenging. On a cross-scavenged engine the transfer port is positioned directly across from the exhaust port. A baffle on the piston head deflects the air-fuel charge upward to prevent it from flowing directly out the exhaust port [5]. There is significant mixing between the exhaust gases and the fresh air-fuel mixture in cross-scavenged engines and some exhaust gases remain in the combustion chamber. A loop-scavenged engine directs the incoming air-fuel mixture in a looping path that pushes the exhaust gases out of the combustion chamber with less mixing than a cross-scavenged engine. One type of loop-scavenging is Schnuerle scavenging. A Schnuerle scavenged (ported) engine has two transfer ports angled away from the exhaust port that direct the air flow upward in a looping path. Some Schnuerle scavenged engines have an additional boost-port opposite the exhaust port [6].

2.1.3 Carburetor Theory

Small two-stroke engines typically use a carburetor to prepare the air-fuel mixture that enters the engine crankcase. A diagram of a basic carburetor is shown in Figure 2.2. There are many carburetor designs of varying complexity but the general concept is as follows. A throttle valve controls the mass flow rate of air through the carburetor inlet. The intake air flows through a converging-diverging nozzle called a venturi which increases the air velocity. The higher air velocity corresponds with a drop in pressure. The pressure difference created between the air inlet and the throat of the venturi draws in fuel through a discharge port near the throat distributing the fuel across the air flow. The pressure difference, which depends on the air flow rate, is used to regulate the fuel flow. A higher air flow rate will

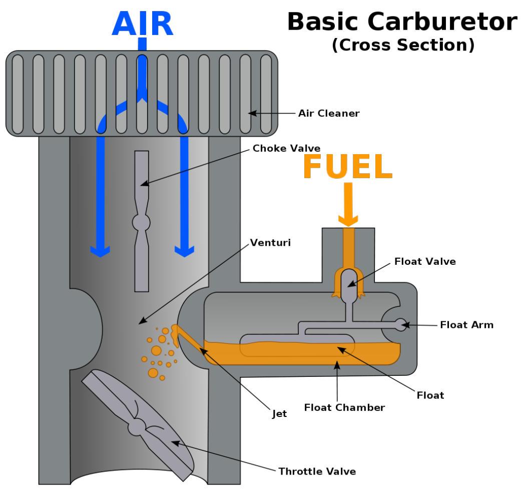


Figure 2.2: Basic carburetor from [7].

cause a rise in the pressure difference. Thus a carburetor provides a different air-to-fuel ratio (AFR) at different speeds and operating conditions. The stochiometric ratio is the AFR that provides just enough air in the mixture for complete combustion of the fuel. An AFR below the stochiometric ratio is considered 'rich' and an AFR above the stochiometric ratio is considered 'lean'. A basic carburetor can not provide an optimal AFR over the whole engine load range [3]. Since most two-stroke engines use the fuel for lubrication and coolant the AFR is kept rich. The typical carburetor on a small aircraft engine has fuel valves that must be tuned correctly to provide acceptable performance through the entire speed range.

2.1.4 Ignition Theory

There are three main types of engines used in small aircraft: spark-ignition, compression ignition, and glow ignition. Spark ignition engines trigger combustion by activating an electric discharge in a spark plug . A gasoline engines rely on spark ignition. Compression ignition engines compress the air so intensely that the temperature generated is greater than the ignition temperature of the fuel. The diesel engine cycle utilizes compression ignition [2].

Glow-ignition engines run on a nitromethane-methanol blend of fuel with an ignition temperature that cannot be reached through compression alone. A glow plug located at the top of the combustion chamber is utilized to help ignite the air-fuel mixture. When an electric current is applied to a glow plug, a platinum coiled-wire element incandesces and produces temperatures that exceed 1,500 degrees Fahrenheit. During the engine operation there is an exothermic reaction that takes place between the platinum in the coil and methanol vapor that further increases the temperature of the coil. The engine is started by turning the engine by hand or with an electric starting motor. The combination of the glow plug and the compression of the mixture causes ignition. Once the engine has been started the electric current can be removed and the engine will continue to fire due to the heat from the exothermic reaction [8].

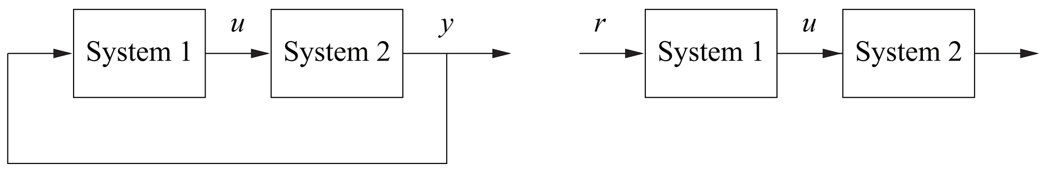
The fuel requirement of an engine depends on its ignition method. Compression engines require a fuel that is highly ignitable. Spark ignition engines require a fuel that is ignition resistant so that auto-ignition does not cause uncontrolled combustion [2].

2.1.5 Glow Engines

The test engine used in this study is a carbureted, glow-ignition, Schnuerle ported, two-stroke engine. This type of engine will be referred to as a 'glow engine'. Glow engines are commonly used on small aircraft because they are small, lightweight, simple in operation, and provide satisfactory performance.

2.2 Control Theory

The engine controller designed and implemented in this research implements Proportional-Integral-Derivative (PID) control which is a form of feedback control. A feedback control system makes corrective actions based on the difference between a desired quantity of a dynamic process and the actual measured quantity. A feedback control system is a closed loop system. A closed-loop system consists of two (or more) systems that are interconnected in a cycle. If there is no interconnection the system is an open-loop system. Figure 2.3 shows the idea of an open-loop system and a closed-loop system. In the closed-loop system the output of system one is the input of system two and the output of system 2 is the input of system one. In an open-loop system the first system is not affected by the second [9].



(a) Closed loop (b) Open loop

Figure 2.3: Open loop system vs. a closed loop system from [9].

In a typical feedback control system the two systems are the controller and the process. Feedback control is necessary to account for disturbances and changes in the process dynamics [10]. A classic example of feedback control is the cruise control system found in modern cars. The purpose of a cruise control system is to maintain a speed set by the driver. It maintains this speed by varying the throttle. The correct throttle position is determined by considering the difference between the reference speed and the actual measured value of speed [9]. Manual control of the throttle (gas pedal) would be an example of open-loop control.

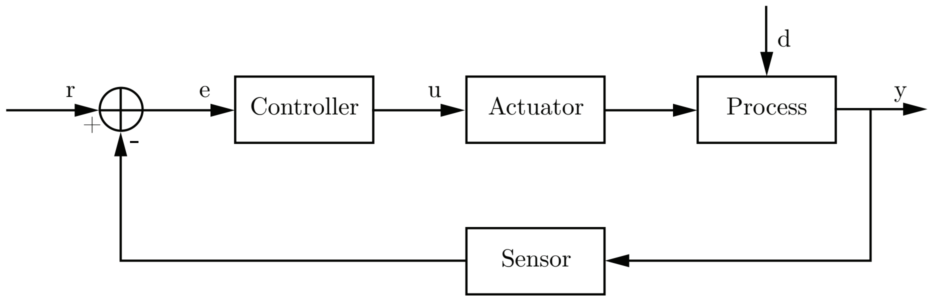


Figure 2.4: Typical feedback control block diagram adapted from [10].

Figure 2.4 shows the block diagram of a typical feedback controller where y is the measured process variable (process output), r is the reference variable, $e = r - y$ is the control error, u is the control variable, and d is a disturbance signal. The reference variable is often called the setpoint. The controller determines the value of the control variable that will bring the process variable to the value of the reference variable and maintain that value despite unplanned or unmeasured disturbances.

2.2.1 PID Control Algorithm

A PID controller is a three-term controller that has become the standard feedback controller in commercial industries. PID controllers have been widely adopted because they provide satisfactory performance while being intuitive and relatively simple. PID controllers are often the fundamental component of more advanced control schemes [10].

The PID algorithm consists of three terms: the proportional term (P-term), the integral term (I-term), and the derivative term (D-term). The terms are summed to determine the value of the control variable. The proportional, integral, and derivative terms can be thought of as acting on the current, past, and future error respectively. Each term of this algorithm will be discussed in detail. The full continuous-time form of the PID algorithm is:

$$\begin{aligned} u(t) &= K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \\ &= K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right) \end{aligned} \tag{2.1}$$

The parameters of the PID algorithm are the proportional gain K_p, the integral gain K_i, and the derivative gain K_d. Sometimes the parameters of integral time T_i and derivative time T_d are used instead of integral and derivative gain [9].

2.2.2 Proportional Term

$$K_p e(t) \tag{2.2}$$

The proportional term shown in Equation 2.2 is proportional to the current control error. It acts to increase or decrease the control variable in attempt to limit the future error. The proportional term is able to provide a small control variable change when error is small to avoid excessive effort from the actuator. The proportional gain determines how aggressively the proportional term responds to an error. Figure 2.5 shows a simulated response of a system with proportional control. It shows that the proportional term by

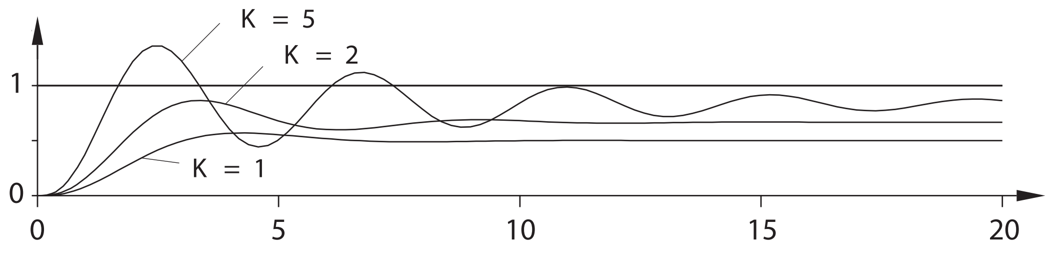


Figure 2.5: P-term response from [9].

itself will always produce a steady-state error. A high proportional gain will give a result in a quick response to a reference change but if the proportional gain is too high the controller will become unstable. The steady state error is eliminated by adding the integral term [10].

2.2.3 Integral Term

$$K_i \int_0^t e(\tau)d\tau \tag{2.3}$$

The integral term is shown in Equation 2.3. It is proportional to the integral, or continual summation, of the control error over time. It allows the controller to account for any unexpected or unmeasured disturbances and eliminates the steady-state error. If there is a positive error the integral term will increase until the error is zero. A negative error causes the integral term to decrease. A large integral gain will allow for effective attenuation of disturbances but too large of an integral gain will cause oscillations in the process and control variables [10].

2.2.4 Derivative Term

$$K_d \frac{de(t)}{dt} \tag{2.4}$$

The derivative term is shown in Equation 2.4. It is proportional to the derivative or rate of change of the control error. The rate of change of the error can offer a prediction of error. The derivative term has the potential to anticipate an incorrect trend in control error and counter it. The derivative term is used to dampen an oscillatory system. The greater the derivative gain the more damped the system becomes but performance will deteriorate if it is too high [10].

2.2.5 Modifications of the PID Algorithm

A controller that implements the PID Algorithm as shown in Equation 2.1 will likely not perform well. There are several issues with this standard from that must be addressed to ensure a well performing controller [10].

The first issue that is commonly faced is a phenomenon known as integral wind-up. Recall that the integral term sums the error history. Integral wind-up occurs when the control variable attains the actuator limit (saturation) during a transient response. When the controller is saturated the control error does not decrease as fast as it would otherwise.

This causes the integral term to become large or wind up. This large integral term causes the controller to stay saturated even if the process variable has attained the set-point value leading to large overshoots and settling times. There are several strategies used to deal with integral wind-up. One of which is to disable integration when the control variable saturates. A modification of this strategy is to only stop integration when the control variable and control error have the same sign. This allows the integral term to help bring the control variable out of saturation. The issue of integral wind-up is not an issue when the incremental form of the algorithm is used [10]. The incremental algorithm is explained in Section 2.2.6

Another issue is that proportional gains required for fast disturbance rejection cause an oscillatory response following a step change in the reference variable. A strategy for dealing with this is to define the proportional term as follows:

$$u(t) = K_p(\beta r(t) - y(t)) \tag{2.5}$$

where β is a weighting factor between 0 and 1. The weighting factor is applied to the reference variable so that the proportional gain acts on only part of the control error during the step change. Setpoint weighting acts to smooth the step change of the reference variable to damp the response of the process variable[10].

While the derivative term can improve the controller performance there are critical issues that cause it to not be frequently used. More than 80% of today's PID controllers are actually PI controllers [10]. Since the derivative term responds to the rate of change of error an abrupt reference variable change will cause the derivative to term to be large. This causes an undesired spike of the control output during a reference variable step. This spike is known as *derivative kick*. Derivative kick is solved by applying the derivative term to the process variable instead of control error. This is referred to as *derivative on measurement*. It is shown below that when the reference variable is constant that derivative of the error is equal to the negative of the derivative of the process variable.

$$\begin{aligned} \frac{de(t)}{dt} &= \frac{d(r(t)-y(t))}{dt} \\ \frac{de(t)}{dt} &= \frac{d(0-y(t))}{dt} = -\frac{dy(t)}{dt} \end{aligned} \tag{2.6}$$

The derivative term also amplifies measurement noise in the manipulated variable. This noise can result in rapid changes of the control variable that can damage the actuator [10].

The PID algorithm with derivative on measurement and setpoint weighting is

$$u(t_k) = K_p \left(\beta r(t_k) - y(t_k) + \frac{1}{T_i} \int_0^t e(\tau) d\tau - T_d \frac{dy(t)}{dt} \right) \tag{2.7}$$

2.2.6 Algorithm Discretization

The algorithm must be transformed to a discrete form so that it can be implemented with a digital controller. The discrete form of Equation 2.7 can be found by applying the backwards finite difference method to the derivative and integral terms [10]. The backwards finite difference method is a common discretization technique. The proportional term is discretized by replacing the continuous variables with their sampled versions. The integral is expressed as a sum of the error history.

$$\int_0^t e(\tau)d\tau = \sum_{i=1}^{k} e(t_i)\Delta t \tag{2.8}$$

where $e(t_i)$ is the error at the current iteration and Δt is a predefined sample time. The derivative term becomes:

$$\frac{de(t)}{dt} = \frac{e(t_k) - e(t_{k-1})}{\Delta t} \tag{2.9}$$

where t_k is the the current sampling instant, and t_{k-1} is the previous sampling instant. The discrete form of equation 2.7 is

$$u(t_k) = K_p \left(\beta r(t_k) - y(t_k) + \frac{\Delta t}{T_i} \sum_{i=1}^{k} e(t_i) - \frac{\Delta T_d}{\Delta t}(y(t_k) - y(t_{k-1})) \right) \tag{2.10}$$

As an alternative to equation 2.10 the control variable at time instant $u(t_k)$ can be calculated using its value at the previous time instant $u(t_{k-1})$ [10]. Subtracting $u(t_{k-1})$ from $u(t_k)$:

$$\begin{aligned} u(t_k) - u(t_{k-1}) = {} & K_p \left(\beta r(t_k) - y(t_k) + \frac{\Delta t}{T_i} \sum_{i=1}^{k} e(t_i) - \frac{\Delta T_d}{\Delta t}(y(t_k) - y(t_{k-1})) \right) \\ & - K_p \left(\beta r(t_{k-1}) - y(t_{k-1}) + \frac{\Delta t}{T_i} \sum_{i=1}^{k-1} e(t_i) - \frac{\Delta T_d}{\Delta t}(y(t_{k-1}) - y(t_{k-2})) \right) \end{aligned} \tag{2.11}$$

This can be re-written as:

$$\begin{aligned} u(t_k) = u(t_{k-1}) + K_p\left(\beta r(t_k) - \beta r(t_{k-1}) + y(t_{k-1}) - y(t_k)\right) \\ + K_i \Delta t e(t_k) + \frac{K_d}{\Delta t}(2y(t_{k-1}) - y(t_k) - y(t_{k-2})) \end{aligned} \tag{2.12}$$

Equation 2.12 is the incremental form of the PID algorithm. It is also known as the velocity form. The incremental form avoids the issue of integral windup and allows for bumpless transfer [15].

3 Experimental Setup

The engine used in this research is a 160FX glow engine manufactured by O.S. Engines (OS160FX). The OS160FX has a 1.6 cubic inch displacement with a power output of 3.7 horsepower at 9,000 rpm. The engine weight with muffler is 42.3 ounces. The engine has a listed practical speed range of 1,800 rpm to 10,000 rpm [11]. The actual speed range is dependent on fuel and the pitch and diameter of the propeller. The engine was fitted with a 17 inch plastic propeller with a 5 inch pitch.

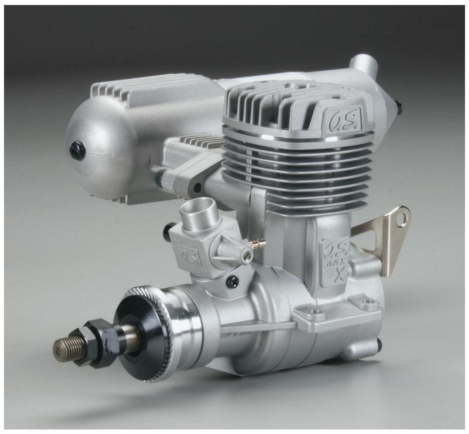


Figure 3.1: OS160FX Engine from [11].

A temporary facility was prepared for testing the engine. The chosen test facility was a well ventilated garage area. A test stand was constructed and secured to the floor of the test facility. The test stand was constructed from wood and can easily be disassembled for transport. The top surface was covered in aluminum sheeting for easy oil cleanup. The assembled engine, fuel tank, and controller hardware were mounted to the test stand. A 12 volt battery was placed under the test stand to provide power for the controller hardware. The test stand with installed equipment is shown in Figure 3.2.

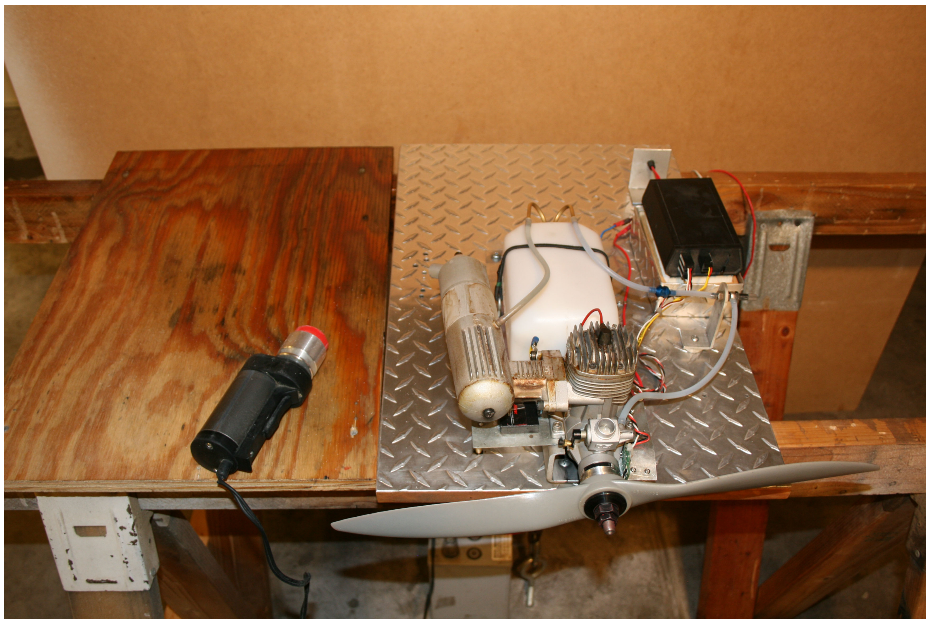


Figure 3.2: Engine test stand.

3.1 Control System Setup

A control system was designed and built to control the engine and monitor its performance. The control system consists of the engine controller and a ground station that communicates with the engine controller and records data. The engine controller measures the rotational speed of the engine shaft, the engine head temperature, and the barometric pressure. It uses a servo motor to adjust the engine throttle. The components of the control system are listed below.

- Arduino Duemilanove micro-controller (MC1)
- Arduino Mega 2560 micro-controller (MC2)
- Infrared reflectance encoder (IRE)
- K-type thermocouple wire
- MAX6675 K-Thermocouple-to-Digital Converter (MAX6675)
- MPL115A1 barometric pressure sensor (MPL115A1)
- Hitec HS-5125MG digital servo
- XBee radio modules
- 20x4 Character LCD Screen
- Laptop computer

The Arduino Duemilanove micro-controller (MC1) takes readings from the sensors and implements the control algorithm. The MC1 sends sensor values and control algorithm data to the Arduino Mega 2560 micro-controller (MC2). The two micro-controllers communicate wirelessly with a pair of XBee radio modules. The MC2 is connected to a laptop computer and relays commands that are entered in the computer's serial monitor to the MC1. The MC2 also prints data received from the MC1 to a file and displays select data on the LCD screen. A diagram of this control system is shown in Figure 3.3. A schematic of the sensor connections to the MC1 is in Appendix B.

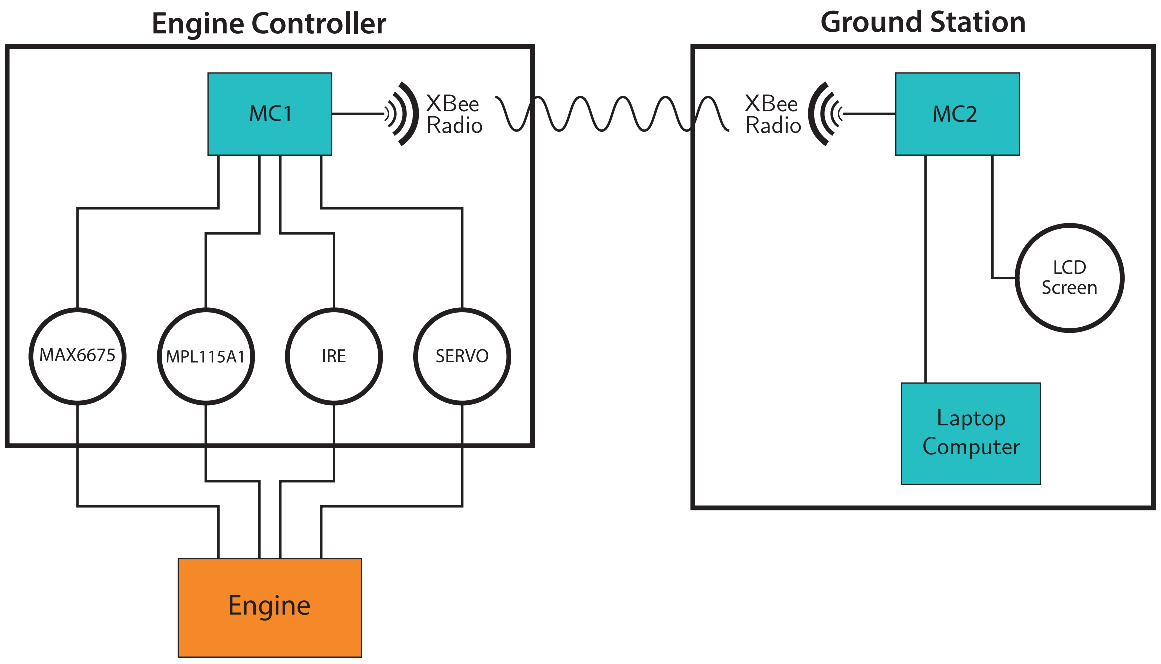


Figure 3.3: Diagram of the control system.

The engine controller is comprised of the MC1, the sensors, and the servo motor. The ground station is comprised of the MC2, the LCD screen, and the laptop computer. The MC1 has 14 digital input/output pins, 6 analog input pins, and a 16 MHz crystal oscillator [12]. Figure 3.4 shows the wiring in the engine controller housing.

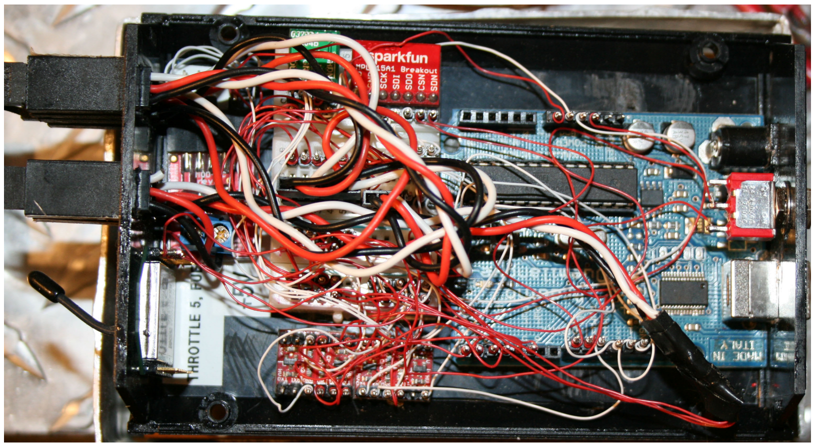


Figure 3.4: Engine controller housing.

The throttle position was adjusted with the Hitec HS-5125MG digital servo. Servos are small DC motors with internal circuitry that provides fine control of angular position. The position is set by sending a digital square wave signal with a specific pulse-width to the servo with a micro-controller. The pulse-width can be specified in microsecond increments. The Hitec HS-5125MG has a 90° total range. This angle range corresponds to a pulse-width range of 900 to 2100 microseconds [13]. The angular precision of the servo is 0.0750 degrees. The engine throttle does not require the entire range of the servo. The throttle is fully closed at 1810 microseconds and fully open at 1300 microseconds. This gives a throttle precision of 0.2 percent. Figure 3.5 shows the servo and throttle arm linkage.

The rotational speed of the engine shaft was determined with an infrared reflectance encoder (IRE). The IRE carries an infrared LED and photo-transistor pair that allows it to sense the reflectance of a surface. The IRE has a voltage output of 5 volts when it detects a surface with a reflectance below a threshold value and outputs 0 volts otherwise [14]. A mark was placed on the shaft hub with black tape that passes the IRE once per shaft

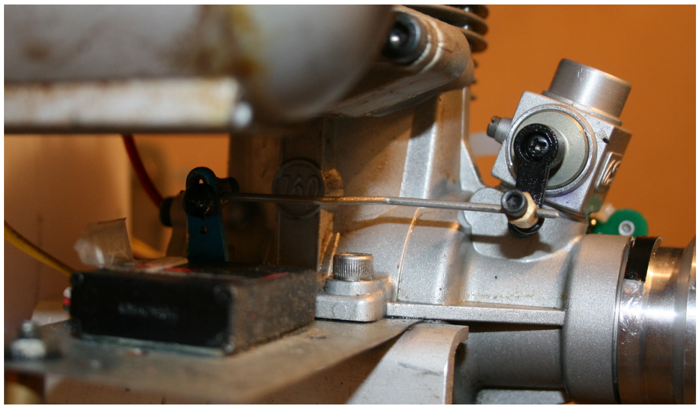


Figure 3.5: Servo and throttle arm linkage.

revolution. As a result the MC1 receives one pulse from the IRE for each shaft revolution. The rotational speed was calculated using the elapsed time between pulses from the IRE or the period. The calculation of the period requires the use of hardware interrupts. The MC1 has two pins that support hardware interrupts. Each time a pulse is received from the IRE the program is halted while an interrupt service routine (ISR) is executed. The ISR is simply a function that performs some action and occurs during every interrupt. During an interrupt all other operations are paused including those responsible for timing. For this reason ISR functions must be kept as short as possible. The ISR triggered by pulses from the IRE calculates the time that has elapsed since the last pulse. The relationship for calculating speed is shown in Equation 3.1.

$$Speed\ [rev/min] = \frac{60,000,000\ [\mu s/min]}{Period\ [\mu s/rev]} \tag{3.1}$$

The temperature of the engine head was monitored using a K-type thermocouple and the MAX6675 K-Thermocouple-to-Digital Converter (MAX6675). The thermocouple was attached to a bolt near the exhaust port. It was determined with an infrared thermal camera that the temperature at the exhaust bolt is consistently 30 degrees lower than the surface of the engine head. The MAX6675 digitizes the voltage signal from the thermocouple and performs cold junction compensation. It outputs the temperature with a resolution of 0.25 degrees Celsius. The barometric pressure was measured with the MPL115A1 barometric pressure sensor (MPL115A1). The barometric pressure can be related to the altitude of the aircraft.

3.2 Controller Software and PID Algorithm Implementation

3.2.1 Serial Communication

A custom program was written in the C language for each of the micro-controllers. The full code for both programs is listed in Appendix D. Function libraries were written to read the MAX6675 and MPL115A1 sensors via a serial protocol interface (SPI). The programs have a serial communication system that allows the MC1 and MC2 to communicate with each other and any computer with a serial monitor. The program on the MC1, aside from reading sensors and implementing the control algorithm, sends sensor values and other data to the MC2. Text commands entered in the laptop's serial monitor are used to interface with the program on the MC1. The purpose of the MC2 is to relay commands entered in the laptop's serial monitor to the MC1. It also has to the receive the data from the MC1, print this data to the laptop's serial monitor, and display select data to a LCD screen. The serial communication system uses ASCII characters to control the flow of data. For example, when the MC1 sends the measured speed value of 6500 rpm to the MC2 it is in the form ‘6500V’. The ‘V’ character tells the MC2 that the incoming numerical values represent the measured speed.

3.2.2 Automatic Control Loop

The controller implements the incremental form of the discrete PID algorithm. The control variable is the value sent to the servo that corresponds to a throttle position. A change in the throttle position results in the change of the process variable, the rotational speed of the engine shaft. The control algorithm supports setpoint weighting, derivative on measurement, low-pass speed measurement filtering, on-line tuning parameter updating, and throttle servo overdrive protection. The program repeats the following control sequence if a predefined sample time has passed.

1. Read sensors and update measurement values.
2. Execute PID algorithm if in automatic mode.
3. Report sensor measurements and other data.

The program function that executes the PID algorithm is shown below.

```
void velPID(){ // PID algorithm, velocity (incremental) form

   /* Apply 1st order low-pass filter to measured speed. Alpha is a weighting
      factor between 0 and 1 that that determines the aggressiveness of the
      filter. Setting alpha to zero disables filtering.
    */
   lowpassSpeed = alpha*lastLowpassSpeed + (1-alpha)*measuredSpeed;

   // Calculate terms of PID velocity algorithm
   // Proportional Term
   K1 = kp*setpointWeight*(setpointSpeed - lastSetpointSpeed)
      + kp*(lastMeasuredSpeed - lowpassSpeed);
   // Integral Term
   K2 = ki*(setpointSpeed - lowpassSpeed);
   // Derivative Tem
   K3 = kd*(2*lastMeasuredSpeed - lowpassSpeed - lastLastMeasuredSpeed);

   // Calculate control variable
   output = lastOutput - K1 - K2 - K3;

   // Convert output to closest integer
   throttlePos = floor(output + 0.5);

   /* Prevent overdriving of throttle servo and stalling of
      engine in the event of control variable saturation.
    */
   if(throttlePos < throttle_open){
      output = (double) throttle_open;
      throttlePos = throttle_open;
   }
   if(throttlePos > throttle_safe){
       output = (double) throttle_safe;
      throttlePos = throttle_safe;
   }

   // Remember variables for next iteration
   lastLowpassSpeed = lowpassSpeed;
   lastLastMeasuredSpeed = lastMeasuredSpeed;
   lastMeasuredSpeed = lowpassSpeed;
   lastSetpointSpeed = setpointSpeed;
   lastOutput = output;

   // Write the throttle position to servo.
   throttle.writeMicroseconds(throttlePos);
}
```

The control variable unit is the pulse-width of the signal to the servo. The way the throttle is installed an increasing throttle opening corresponds to a decreasing pulse-width. This requires that the terms of the PID be subtracted from the previous value of the control variable.

The throttle takes approximately 94 milliseconds to move from idle position to open throttle. So there is an inherent delay that is less than 94 milliseconds. The controller must allow the servo to move to the desired position before giving it a new position. The sample time of the control algorithm is 100 milliseconds to allow for servo movement.

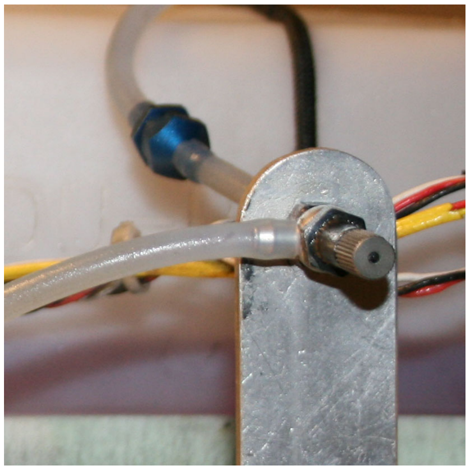
An important requirement for a feedback controller is bumpless transfer. A switch from manual to automatic control that does not disrupt the controlled process is a bumpless transfer. To ensure an uneventful transfer to automatic control the program performs the following actions immediately after receiving a command to enter automatic control.

- Set reference speed $r(t_k)$ to current measured speed $y(t_k)$.
- Set $r(t_{k-1})$ to $r(t_k)$.
- Set $y(t_{k-1})$ and $y(t_{k-2})$ to $y(t_k)$.
- Set control output $u(t_k)$ (throttle position) to current manually set value.
- Set all terms of PID algorithm to zero to remove any residual value created by previous periods of automatic operation.

3.3 Break-in Engine and Tune Carburetor

3.3.1 Carburetor and Fuel System

The OS160FX has a rotary barrel carburetor with a low-speed needle (LSN) valve for adjusting fuel flow at low speeds and an external high-speed needle (HSN) valve for adjusting fuel flow at high speeds. Figure 3.6 shows the HSN which is the primary way of controlling the flow of fuel into the engine. At large throttle openings the HSN is the only fuel valve that effects the flow. The LSN, shown in Figure 3.7, allows more precise fuel flow control at low speeds. The LSN consists of a tapered needle that enters a fuel spray-bar as the


(a) Installation

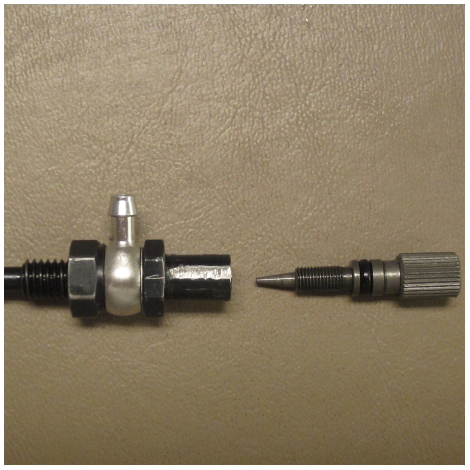
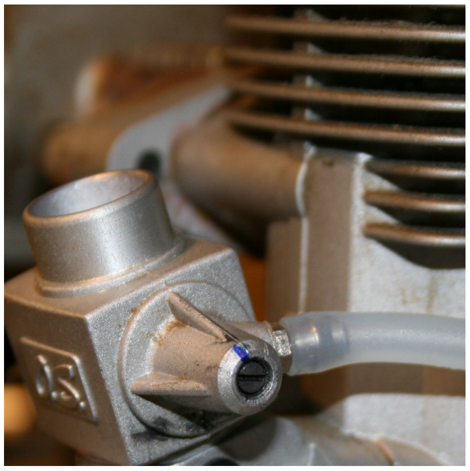

(b) Disassembled view

Figure 3.6: High-speed needle valve (HSN).

throttle opening closes. This is shown in Figure 3.7b. The needle restricts the flow of fuel at low-speeds while allowing full flow at high-speeds. An adjustment screw, seen in Figure 3.7a, controls how far the spray-bar extends into the carburetor. This adjustment determines the range of throttle positions that are affected by the LSN.


(a) Adjustment screw

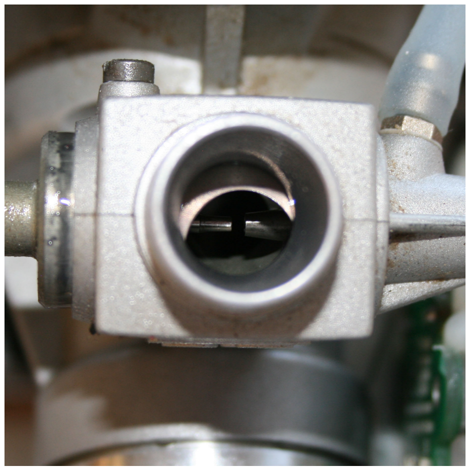

(b) Tapered needle entering spraybar

Figure 3.7: Low-speed needle valve (LSN).

The position of the knob on the HSN and screw on the LSN must be set or 'tuned' correctly for good engine performance. The correct position depends on many factors including ambient temperature, humidity, altitude, engine wear, and desired performance characteristics. The HSN and LSN must be tuned together because they effect one another.

The fuel system is shown in Figure 3.8. A 1200 cubic centimeter fuel tank was used with 15 percent nitromethane fuel. The fuel line runs from the exhaust chamber to the tank so that the tank can be pressurized with exhaust gases. The fuel line runs from the tank to the HSN with a fuel filter in-line and from the HSN to the carburetor.

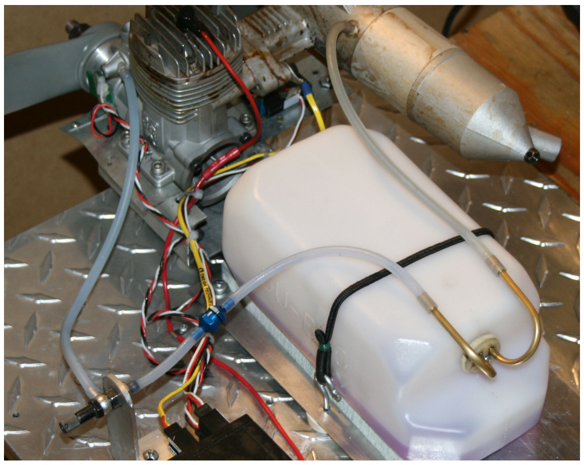


Figure 3.8: Engine fuel system.

3.3.2 Start-up Sequence

The engine was started manually. The start-up sequence of the engine was as follows. First the engine was primed to introduce fuel into the carburetor. This was done by sealing the top of the carburetor with one hand while rotating the propeller with the other. The throttle was fully open during this process. Next the glow plug was ignited and the throttle was set to idle position. The engine was then started by spinning the propeller at approximately 2000 rpm with an electric starter motor. After the engine was started the power source was removed from the glow plug. Figure 3.9 shows the use of the starting motor and Figure 3.10 shows the location of the glow plug.

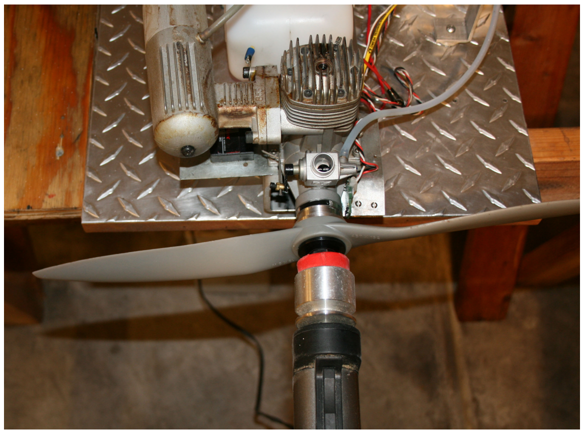


Figure 3.9: Demonstrated use of electric starting motor.

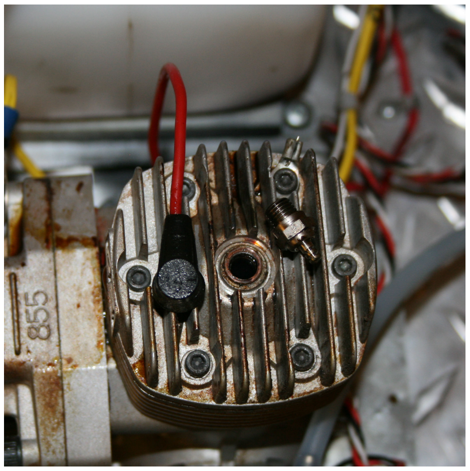


Figure 3.10: Glow plug location.

3.3.3 Break-in Procedure

A new engine is required to go through a break-in procedure. During a break-in procedure an engine is run at low speeds with a rich air-fuel mixture. The purpose of the break-in procedure is to allow the moving parts of the engine to 'mate' or settle together. This limits the stresses on the components. The rich air-fuel mixture ensures a well lubricated engine.

The break-in procedure was conducted as follows. The engine was started and allowed to run for several minutes at a time, allowing it to cool between runs. Each run was progressively longer. The engine was operated near idle for the first half-tank of fuel. Next, the throttle was gradually cycled between quarter and half throttle for a second half-tank of fuel. Then, the engine was run at half to three-quarter throttle while occasionally bringing it to full throttle for a tank of fuel. After 2 tanks of fuel the engine was considered broken in. The tank was not allowed to be completely emptied because an engine that runs out of fuel has a momentary jump in speed before it dies. This acceleration can damage a new engine.

3.3.4 Tuning Procedure

The tuning procedure requires making small adjustments (1/8 turns) of the HSN and LSN and observing how the adjustments effect performance. The speed is monitored but the judgement of a good tuning was largely based on the sound and temperature of the engine. It is a process that requires an experienced operator to achieve acceptable performance throughout the entire speed range.

There were many techniques used to determine the correct carburetor settings but the following is a procedure that was used often to make sure the engine would run well and be controllable. The first step was to run the engine at moderate to high speeds to warm up the engine. Next, the engine was brought to full throttle and the HSN was set to the position that gave the maximum speed. Typically this setting was too lean and caused the engine temperature to be too high. The HSN was richened gradually until the temperature dropped to an acceptable value. With the HSN set, the LSN was then set to a position that allowed for smooth acceleration through the speed range. This was an iterative process because adjustments to the LSN effect the HSN. An acceptable temperature as measured by the thermocouple was around 160 degrees Celsius.

3.4 Throttle and Speed Relationship

Understanding the effect of the control output variable on the process variable is critical to controlling a system. In this case the process variable is the rotational speed of the engine and the control variable is the throttle position. In order to see the relationship between speed and throttle position, throttle step tests were performed. During a throttle step test the throttle position was stepped in equal increments and the response of the speed to these steps was recorded. The results of the throttle step tests are shown in Figures 3.11 and 3.12 . Throttle Step Test A has steps of 20 percent throttle taken every 80 seconds. Throttle Step Test B has steps of 9.8 percent taken every 40 seconds.

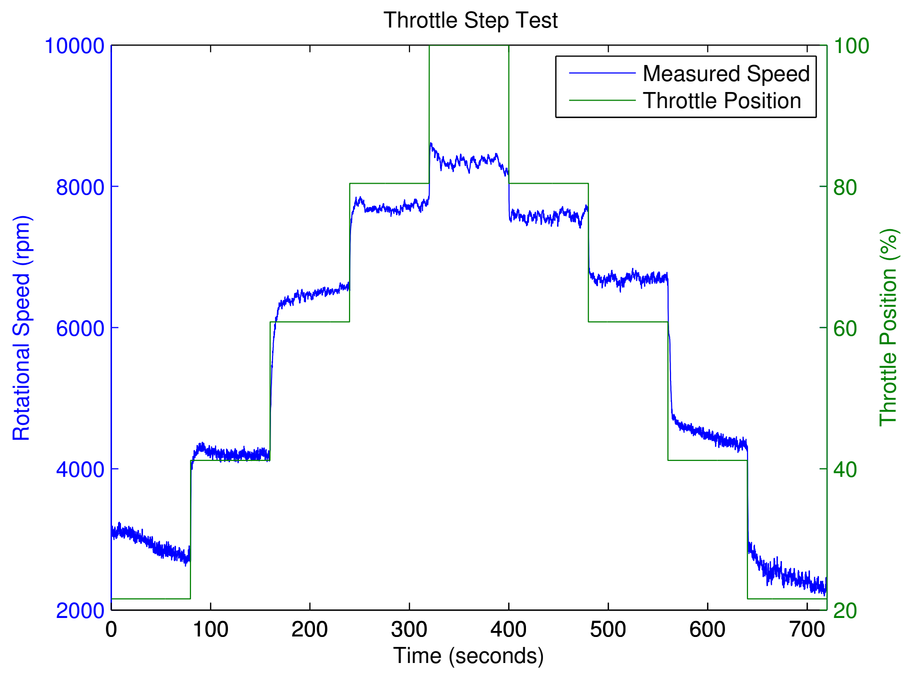


Figure 3.11: Throttle Step Test A.

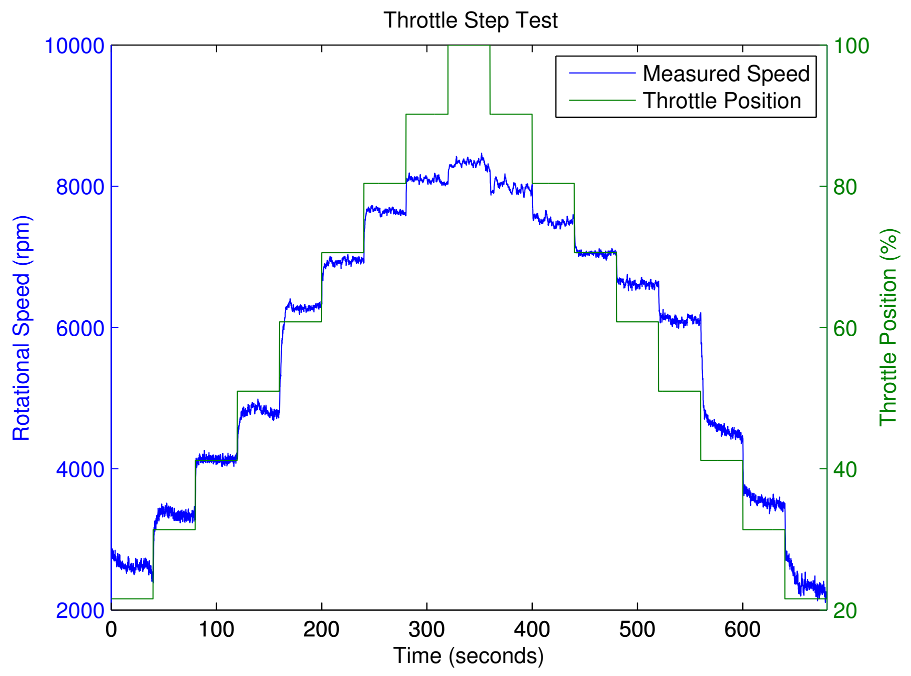


Figure 3.12: Throttle Step Test B.

The relationship between throttle position and speed is clear. As expected a larger opening of the throttle results in higher speeds. The speed is not steady at a given throttle position though. Figure 3.11 shows that the speed has tendency to drift or gradually move away from a value. This drifting of speed is seen most following negative throttle steps when the engine is slowing down. Figure 3.13 shows that at 40 percent throttle the speed gradually drops from 4723 rpm to 4278 rpm in a span of 75 seconds. This tendency to drift is not seen as much during the higher speeds. The speed is particularly unstable at low speeds near idle.

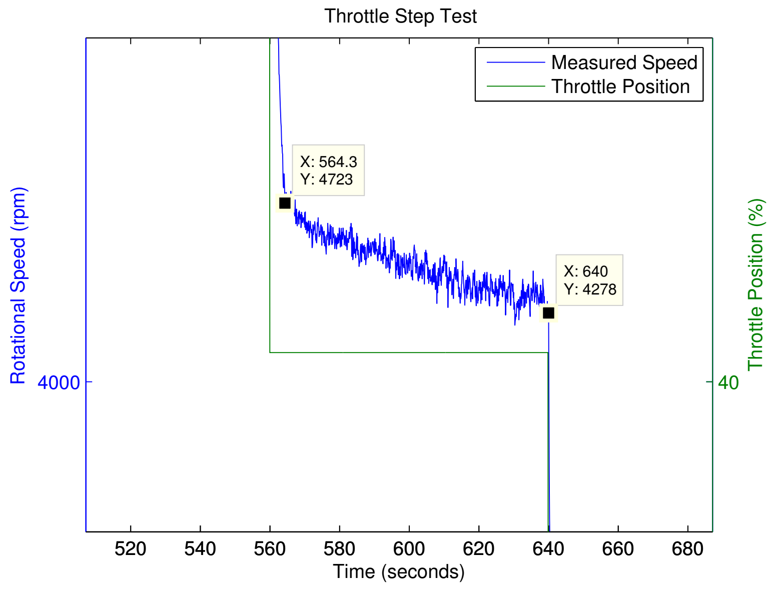


Figure 3.13: Gradual speed drop following a negative throttle step.

The results of Throttle Step Test B, Figure 3.12, shows that the the engine performs differently on the upward speed trajectory then it does on the downward speed trajectory. More steps must be taken to bring the engine from high speeds to low speeds. This could be due to the momentum of the shaft and propeller that was produced at higher speeds. Another potential source is excess fuel in the crankcase that is introduced during the wider openings of the throttle. The automatic controller had to counter this phenomenon when adjusting speed. The average speed at each throttle position was calculated in order to determine the relationship between speed and throttle position. This relationship is shown in Figure 3.14.

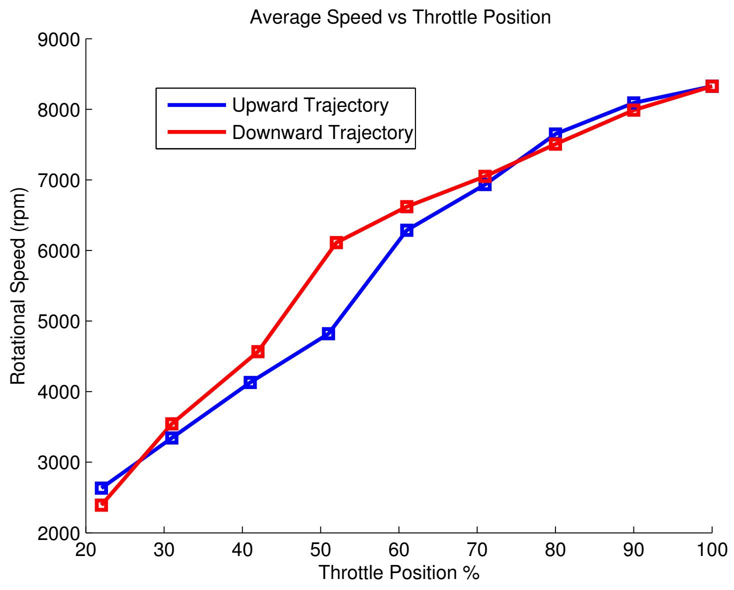


Figure 3.14: Relationship between throttle and speed.

A common measure for describing the dynamic behavior of a process is the process gain. The process gain defines the sensitivity of the output variable to changes in the input variable and is calculated as shown in Equation 3.2 [15].

$$K = \frac{\Delta Output \ [rpm]}{\Delta Input \ [\%]} \tag{3.2}$$

The output and input units depend on the process and measurement methods. In this case the output units are in rpm and the input units are throttle position expressed as a percentage. The process gain is often dependent on the load or operating point. Figure

3.15 shows the process gain calculated at each positive step of Throttle Step Test B. The process gain is $150\frac{rpm}{\%}$ at step 4 which is more than twice the value seen at steps 3 and 5. This suggests that the speed is most sensitive to changes in throttle when the engine is transitioning from low to high speeds.

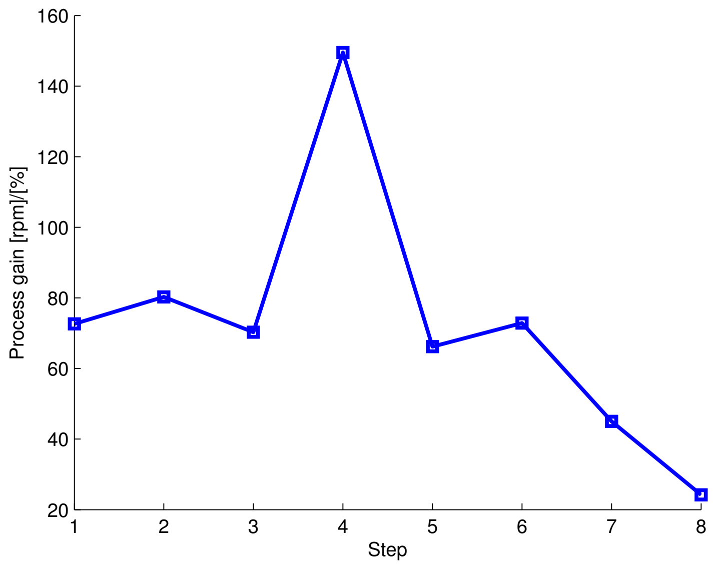


Figure 3.15: Process gain at different operating points.

3.4.1 Sensitivity to Carburetor Tuning

Specific tests were not completed to investigate the effects of different fuel valve positions on the engine position. Figure 3.16 shows throttle step test results with three different carburetor tuning settings. Tuning 1 and 2 were considered good tunings since they provided a consistent speed response to the throttle steps. The variation seen between Tunings 1 and 2 was expected between any two given tests. Tuning 3 was considered a poor tuning due to the elevated speeds at small throttle openings and small response to a change in throttle seen near 5000 rpm.

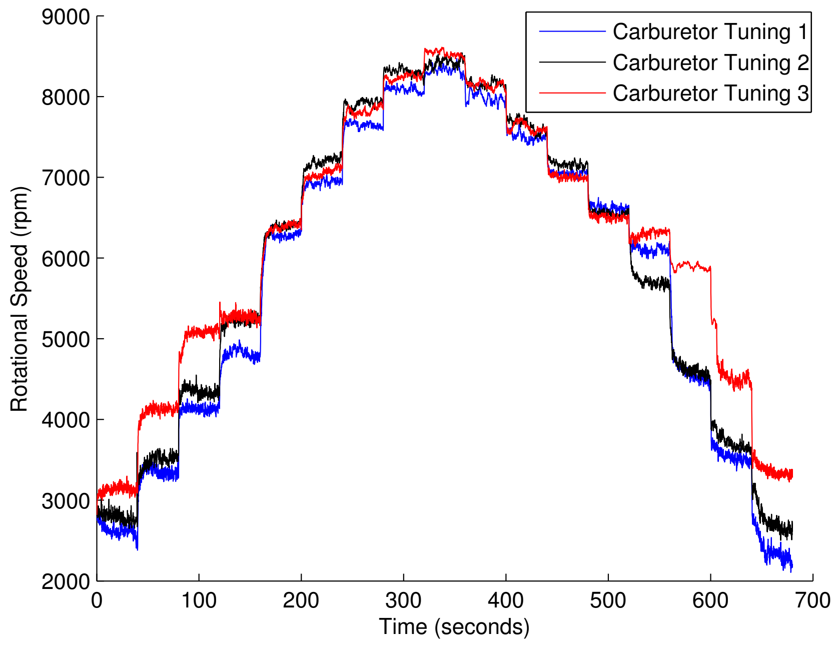


Figure 3.16: Performance variations due to different carburetor settings.

3.5 Performance Measures and Goals

With the throttle and speed relationship determined the next step was to test the performance of the PID algorithm. This involved running the engine and observing the responses of the measured speed to step changes in the reference speed. The measured speed and throttle positions were recorded for analysis.

The controller code has a subroutine that allows for programmed reference speed changes. This allowed for consistent tests of the controller performance. The programmed reference speed paths are shown in Figure 3.17. Each reference speed path has a 40 second period between each step. Reference Path A was used most when comparing tuning parameter combinations because it has large and moderate speed steps. The symmetric Reference Path B was useful because it showed if the controller corrected for the engines performance difference between upward and downward trajectories.

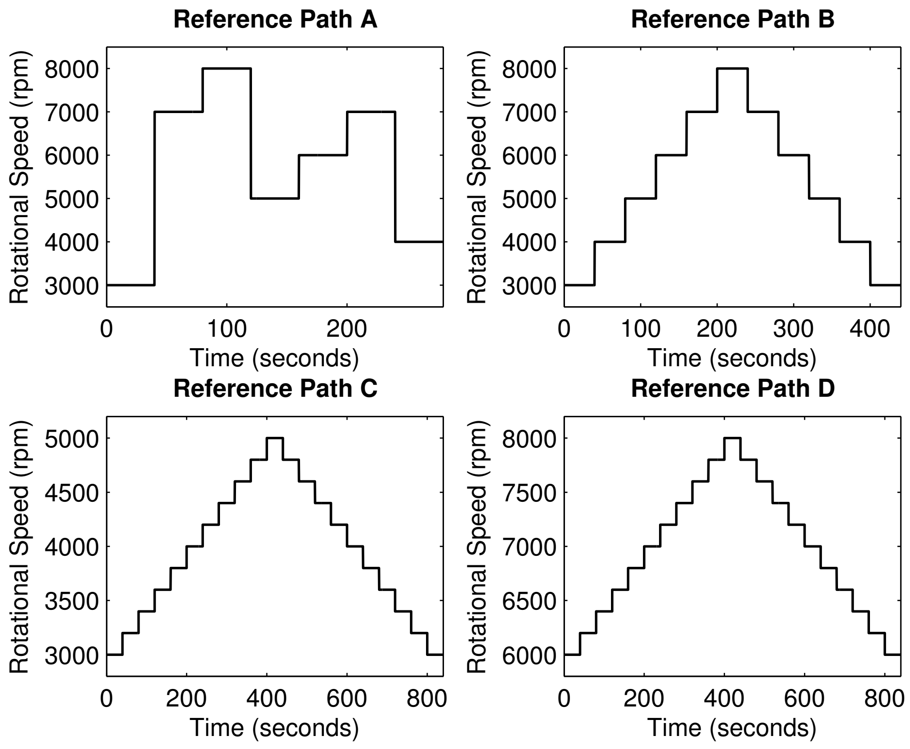


Figure 3.17: Reference speed test paths.

The performance measures that were used when judging the responses were steady-state error, settling time, overshoot, steady-state standard error, oscillation amplitude, and maximum steady-state deviation. A steady-state error of zero was required for all responses to a reference change. The settling time is the time it takes for the measured speed to reach its steady-state after a reference command is received. The overshoot is the amount the speed overshoots the reference speed during the initial response. Only overshoot greater than 100 rpm was considered overshoot. The steady-state standard error is a measure of the variation of the measured speed $y(t)$ around a reference speed $r(t)$ during a steady-state and is given by the Equation 3.3.

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(y_i - r)^2} \tag{3.3}$$

where N is the number of samples available during the steady-state period.

Variation from the reference value during steady-state can be caused by a disturbance, measurement noise, or persistent oscillations due to process or controller dynamics. The maximum steady-state deviation is the maximum deviation from the reference value during steady-state. The maximum oscillation amplitude is the maximum amplitude of any sinusoidal type oscillations seen in a response. The acceptable values for the performance measures are shown in Table 3.1.

Table 3.1: Performance Measures

Measure	**Acceptable Value(s)**
SS error	0 rpm
Maximum SS deviation	-100 rpm to 100 rpm
Settling time	$<$ 10 seconds
Maximum overshoot	200 rpm
Maximum oscillation amplitude	100 rpm

3.6 Selection of PID Tuning Parameters

The most challenging part of implementing the PID algorithm was selecting the appropriate tuning parameters. The parameters were chosen to minimize settling time, overshoot, and oscillations. It was known that the derivative term adds complications to controller design, often with limited benefit. For this reason a PI controller was designed first. After an acceptable PI controller was designed it was determined whether the derivative term provided any clear benefit.

The first tests of the PI algorithm required trial-and-error to determine a realistic range for the tuning parameters which were the proportional gain K_p and the integral gain K_i. The trial and error tests pointed to a range of 0.10 to 0.01 for K_p and 0.00015 to 0.00001 for K_i with a sample time of 100 milliseconds. The next tests systematically varied the parameters to determine their effects on the response to Reference Path A. The knowledge gained from these tests was used to select candidates for the final parameter combination. These candidates were tested and a best choice was selected based on a compromise between responsiveness and stability.

Table 3.2: Steps of Reference Path A.

Step 1	3000 rpm to 7000 rpm
Step 2	7000 rpm to 8000 rpm
Step 3	8000 rpm to 5000 rpm
Step 4	5000 rpm to 6000 rpm
Step 5	6000 rpm to 7000 rpm
Step 6	7000 rpm to 4000 rpm

Tests with a moderate K_i and no K_p helped to illustrate the effects of the parameters. Figure 3.18 shows the speed response with a K_i of 0.000045 and no K_p. The steps of Reference Path A are referred to as shown in Table 3.2. There is excessive overshoot seen in the response at step 1, step 3, and step 6. The response also exhibits steady-state oscillations at 3000, 5000, and 6000 rpm. The oscillations are largest at 6000 rpm. Figure 3.19 shows a change in the response with a lower K_i of 0.000030. The overshoot at step 1 has been nearly eliminated but still exists at step 3 and step 6. There are still oscillations but they are smaller. The high magnitude of the overshoots and oscillations were not seen with moderate levels of K_p during the trial-and-error tests.

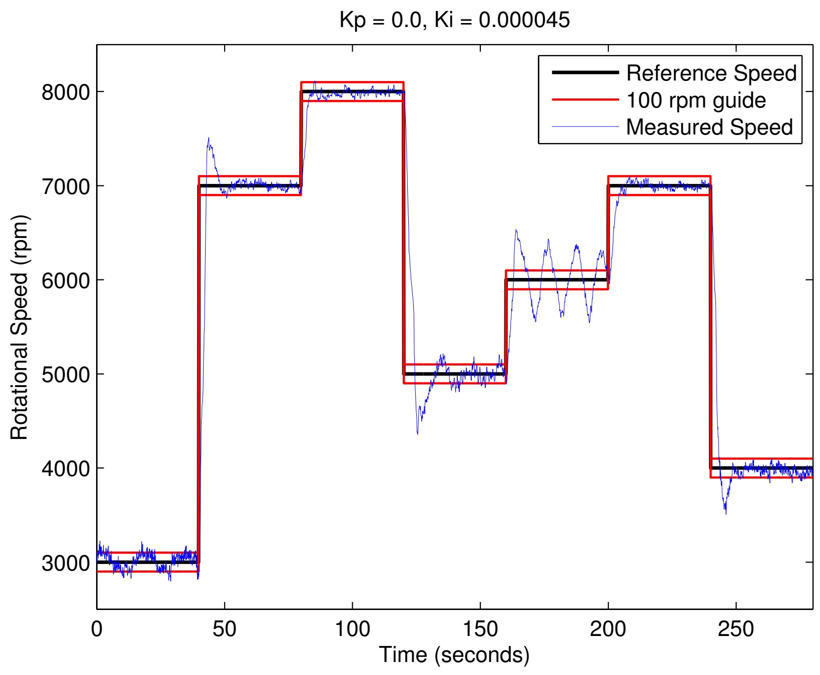


Figure 3.18: Response with $K_i = 0.000045$ and no K_p.

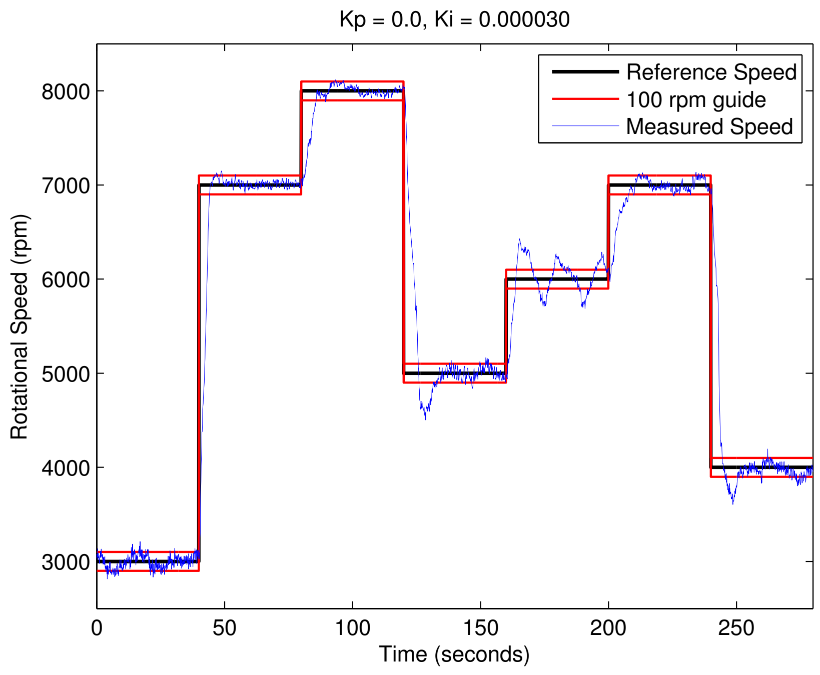


Figure 3.19: Response with $K_i = 0.000030$ and no K_p.

More tests were run to determine the effect of adding proportional gain. The tables below summarize how the response changed with the addition and increase of the proportional gain. The addition of the proportional gain dampened the oscillations and diminished overshoot but it had a drawback. An increase in proportional gain was accompanied with a greater settling time that was most evident in positive steps. Note that in Table 3.4 the large setting times seen with a K_p of 0.00 and 0.02 were due to overshoot that occured.

Table 3.3: Response performance with $K_i = 0.000030$.

Ki	Kp	Overshoot at 5000 rpm (rpm)	Oscillation Amplitude at 6000 rpm (rpm)	Settling Time Step 1 (seconds)
0.00003	0.00	497	358.5	9.3
0.00003	0.01	428	355	5.6
0.00003	0.02	326	363	8.1
0.00003	0.03	298	251	13.7
0.00003	0.04	241	271	15.5
0.00003	0.05	153	187	15.0
0.00003	0.06	189	179	14.9
0.00003	0.07	144	153	14
0.00003	0.08	0	131	15.8

Table 3.4: Response performance with $K_i = 0.000045$.

Ki	Kp	Overshoot at 5000 rpm (rpm)	Oscillation Amplitude at 6000 rpm (rpm)	Settling Time Step 1 (seconds)
0.000045	0.00	645	491	11.32
0.000045	0.01	533	406	15.1
0.000045	0.02	457	291	7.9
0.000045	0.03	413	304	8.2
0.000045	0.04	270	231	10.4
0.000045	0.05	191	280	7.24
0.000045	0.06	183	220	11.3
0.000045	0.07	0	183	11.2
0.000045	0.08	0	100	11.6

The best response was obtained with a K_i of 0.000030 was with a K_p of 0.080. This response is shown in Figure 3.20. The oscillation amplitude at 6000 rpm was nearing the acceptable range with a value of 131 rpm but there was a long settling time of 15.8 seconds. The best response with a K_i of 0.000045 was also with a K_p of 0.080. It is shown in Figure 3.21. The oscillation amplitude was within the acceptable range, and the overshoot was eliminated, but the settling time at 7000 rpm was still greater than 10 seconds. The rest of the plots of the responses listed in Tables 3.3 and 3.4 can be found in Appendix A.

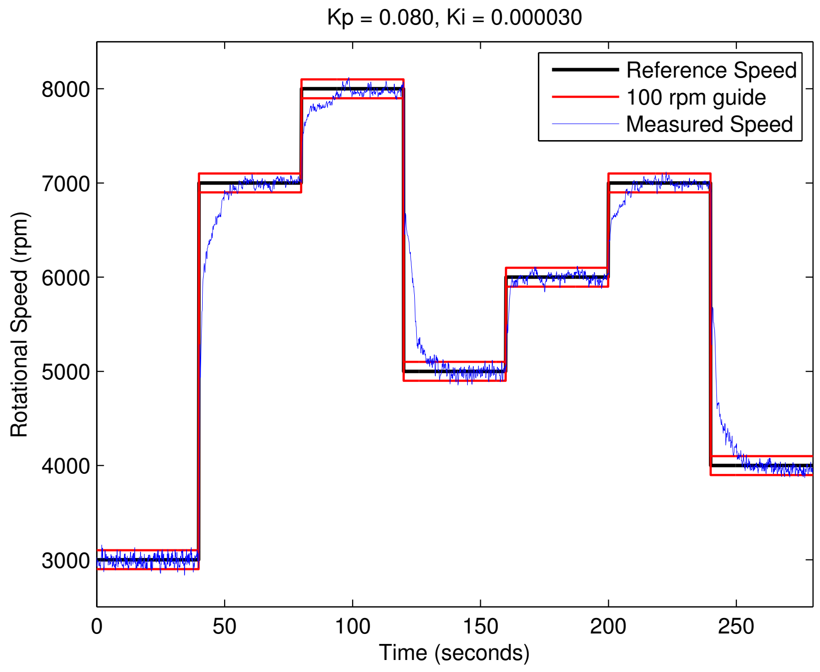


Figure 3.20: Response with $K_p = 0.080$ and $K_i = 0.000030$.

The results of these tests indicated that a larger K_i provides a quicker response but there must be an appropriate K_p to dampen the integral term without increasing the settling time to an unacceptable level. Since a faster response was desired, tests were run with a integral gains of 0.000050 and 0.000055. Table 3.5 outlines the results.

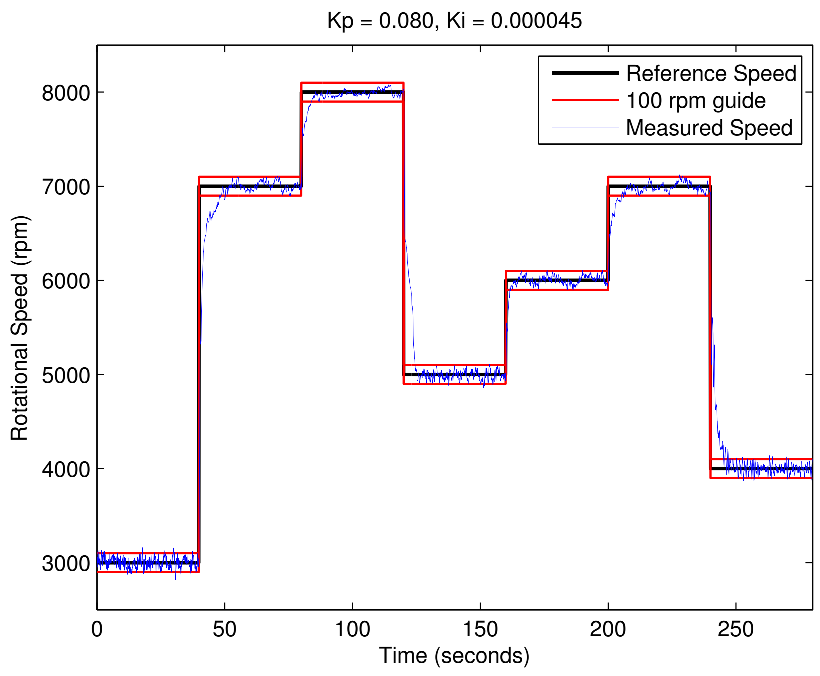


Figure 3.21: Response with $K_p = 0.080$ and $K_i = 0.000045$.

Table 3.5: Response performance with $K_i = 0.000050$ and 0.000055.

Ki	Kp	Overshoot Step1 (rpm)	Maximum Deviation 6000 rpm (rpm)	ST* Step 1 (seconds)	ST* Step 2 (seconds)	ST* Step 5 (seconds)	ST* Step 6 (seconds)
0.00005	0.060	284	100	5.6	8	5.3	4.8
0.00005	0.065	197	150	8	8.4	8.8	7.4
0.00005	0.070	192	179	10.4	10	8	6.2
0.00005	0.080	127	160	9.44	11.2	10.8	10.3
0.000055	0.065	326	183	9.75	9.5	10.7	5.7
0.000055	0.080	235	143	9.33	5.5	9.8	6.7
*ST=settling time							

The quickest response was with a K_i of 0.000050 and a K_p of 0.06. The settling time at step 1 was only 5.6 seconds but there was a large overshoot of 284 rpm at step 3. A K_p of 0.065 reduced this overshoot to 197 rpm and still provided a quick response. Increasing the K_p to 0.07 and 0.08 decreased this overshoot further but increased settling times to an unacceptable level. The responses with a K_i of 0.000050 are shown in the figures below.

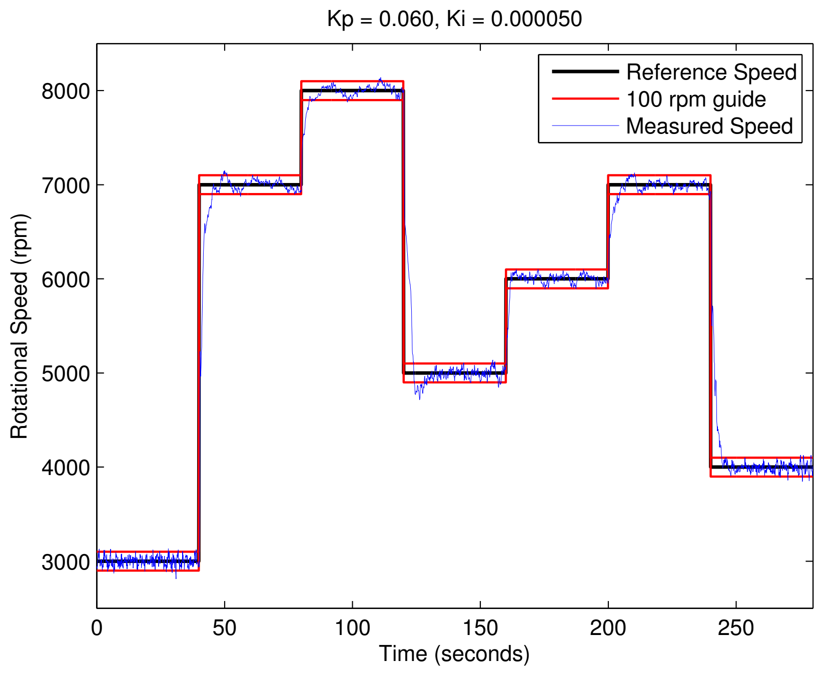


Figure 3.22: Response with $K_p = 0.060$ and $K_i = 0.000050$.

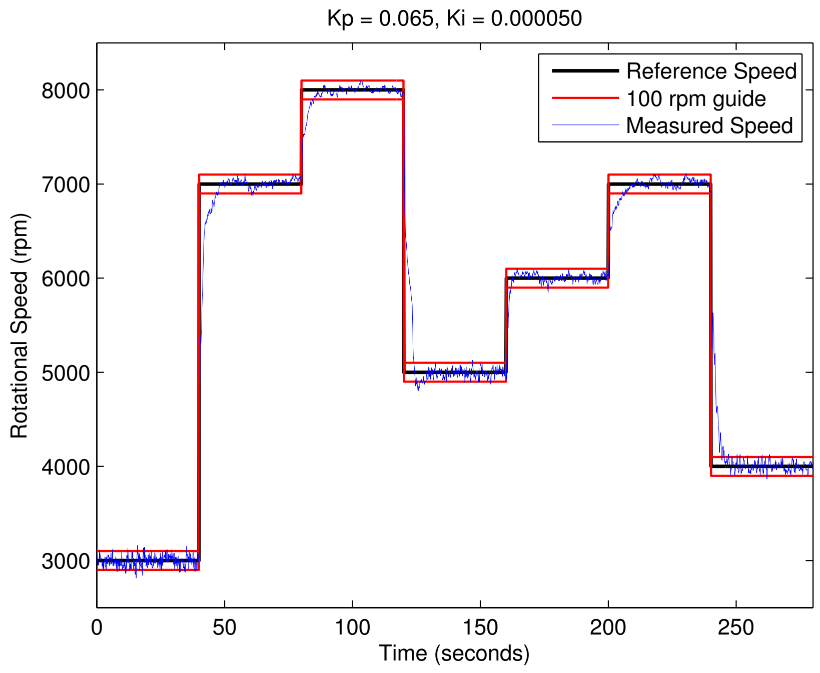


Figure 3.23: Response with $K_p = 0.065$ and $K_i = 0.000050$.

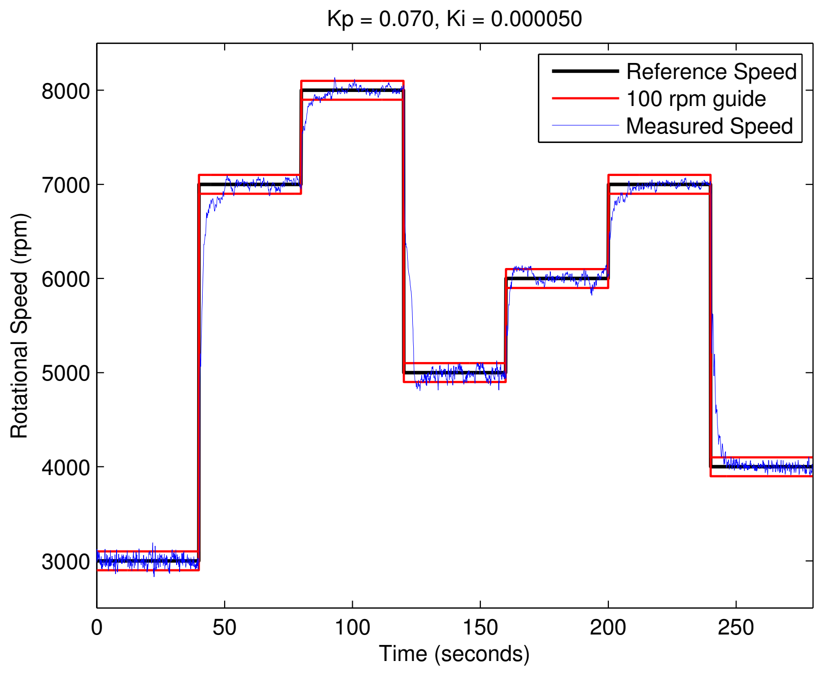


Figure 3.24: Response with $K_p = 0.070$ and $K_i = 0.000050$.

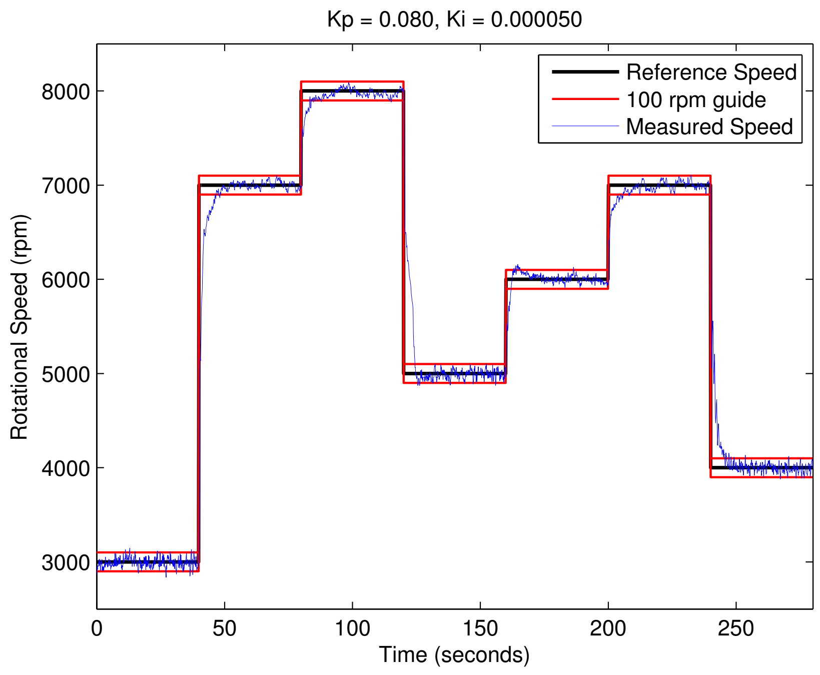


Figure 3.25: Response with $K_p = 0.080$ and $K_i = 0.000050$.

It was expected that a K_i of 0.000055 would provide a quicker response than a K_i of 0.000050. Figure 3.26 shows that while the immediate response is quicker with a K_i of 0.000055 the acceleration decreases just before reaches the reference speed. The responses with a K_i of 0.000055 also had too much overshoot at step 3 and had rapid oscillations at 4000 rpm.

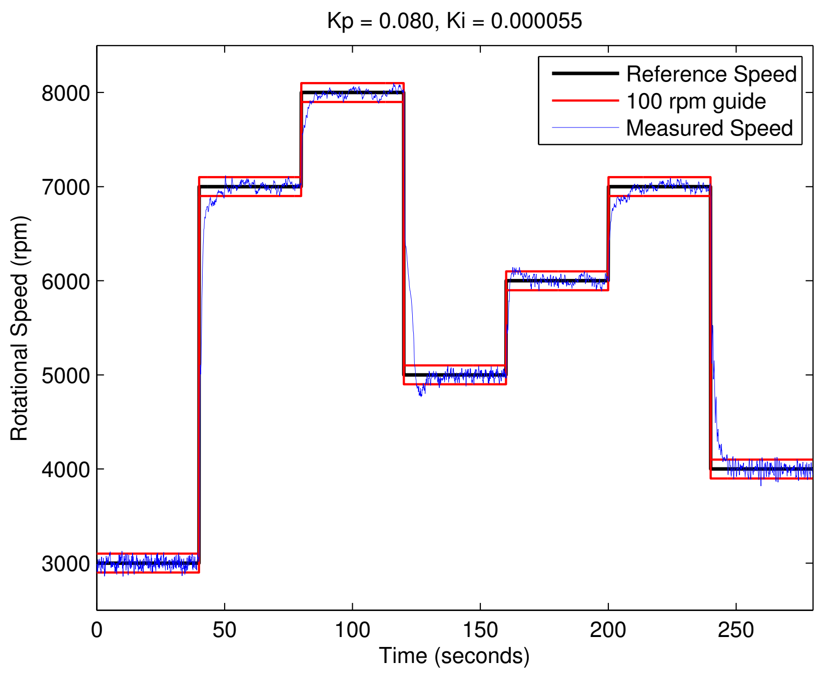


Figure 3.26: Response with $K_p = 0.080$ and $K_i = 0.000055$.

Tests with the derivative term did not provide conclusive results but they were limited. It is likely that the value of the derivative gain K_d was too low to provide a clear effect. A process that changes direction within a few samples is typically not benefited by derivative term. The derivative term is most useful in a slow moving control loop where overshoot is not acceptable. Furthermore a well tuned PI control is likely to provide better performance than a moderately tuned PID controller [16]. A PI controller with a K_p of 0.065 and a K_i

of 0.000050 was chosen as the final controller.

Through further research it was found that the incremental algorithm has a unique property that can cause issues. The integral term and proportional do not necessarily have the same sign at a given moment. The signs are the same when the process variable is moving away from the reference but are opposite when approaching the reference [15]. This explains why an increasing proportional term caused longer settling times. This unique property can cause oscillatory behavior if there is not a strong integral term. Modifications can be made to the algorithm so that the signs of the integral and proportional term are always the same. This would allow for a quicker response. The property can also be advantageous because it can effectively dampen the integral term as the reference is approached. If the modifications were only applied when the process variable was outside a conditional band around the reference, the algorithm would theoretically allow for quick response while taking advantage of the integral dampening near the reference [15]. It is believed this would have given better results.

4 Results

The final controller met the performance requirements described in Section 3.5 with a few exceptions. The results the reference path tests are shown in the figures below with a K_p of 0.065 and a K_i of 0.000050. The largest settling time in the Reference Path A test, shown in Figure 4.1, was 8.8 seconds and the largest overshoot was 197 rpm. During the largest step response the speed reached 90 percent of the reference in less than 3 seconds. There was a single steady-state deviation of 150 rpm from the 6000 rpm reference which was greater than the acceptable value of 100 rpm. This single deviation was not indicative of the overall performance and was considered acceptable.

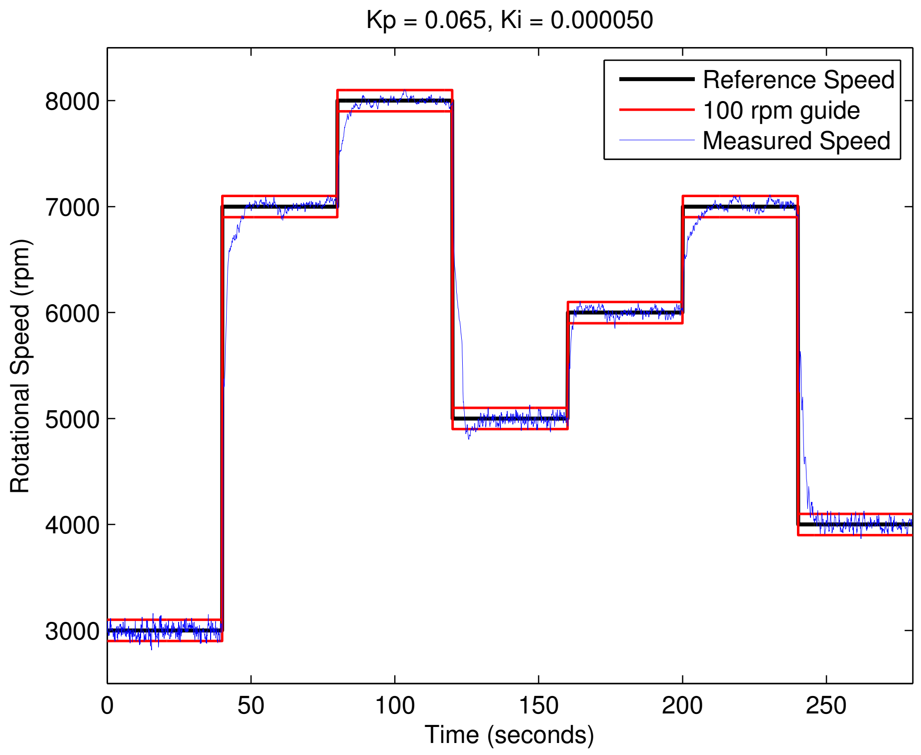


Figure 4.1: Response to Reference Path A with $K_p = 0.065$ and $K_i = 0.000050$.

The Reference Path B test in Figure 4.2 shows that the responses to negative and positive steps are comparable. This indicates the the controller can correct for the difference in engine performance seen between the upward and downward trajectories of the throttle step tests. There is an overshoot greater than 200 rpm during the second negative step. In this region of engine operation the LSN starts to effect the fuel flow. This sudden restriction of the fuel flow was believed to cause this instability. This instability source is unavoidable with the given carburetor and fuel system.

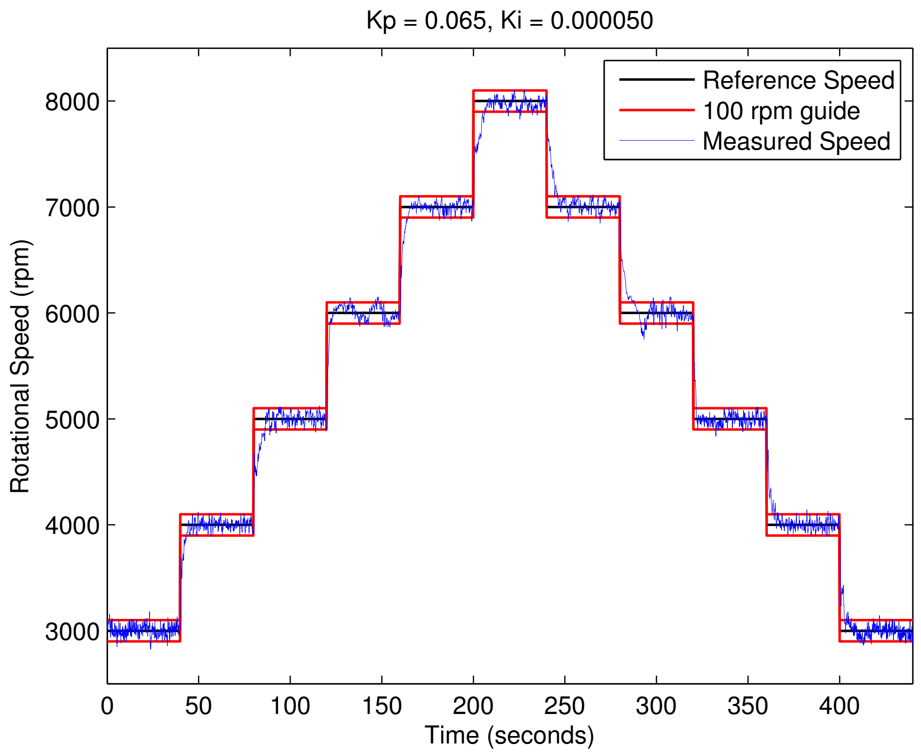


Figure 4.2: Response to Reference Path B with $K_p = 0.065$ and $K_i = 0.000050$.

The responses to Reference Path C and D, Figure 4.3 and 4.4, show that the controller performs well when small reference speed commands are given. The high frequency oscillations in Figure C is measurement noise created by engine vibrations that occurred at low speeds.

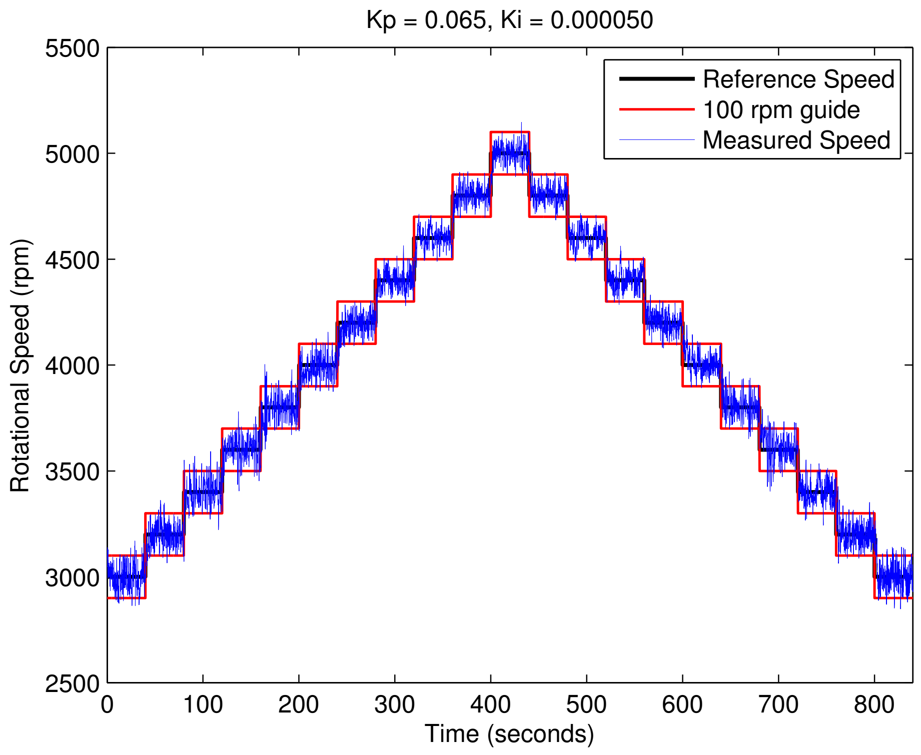


Figure 4.3: Response to Reference Path C with $K_p = 0.065$ and $K_i = 0.000050$.

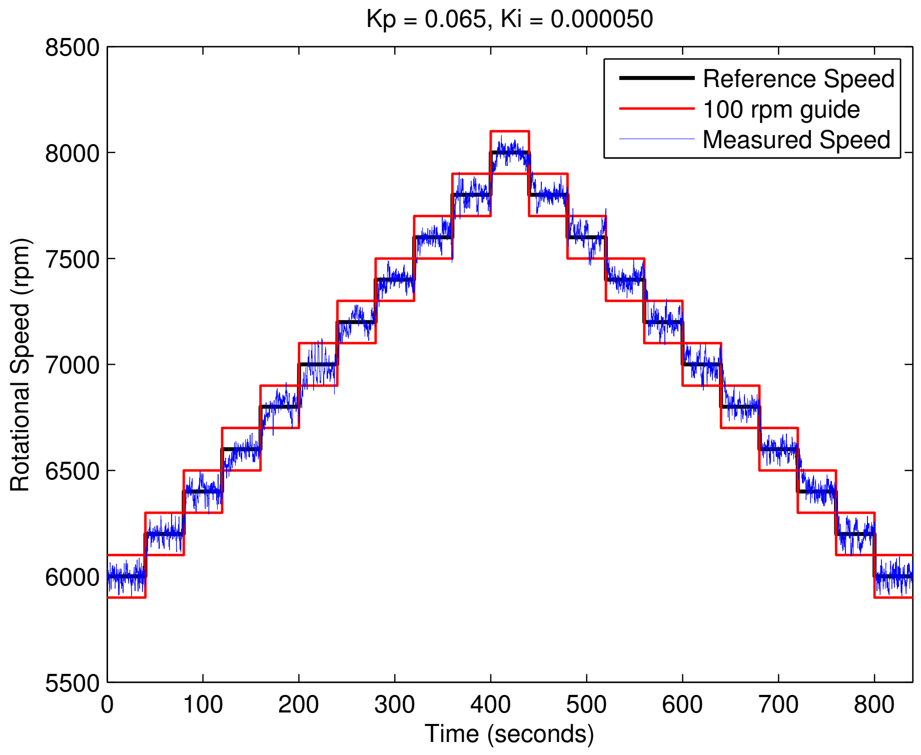


Figure 4.4: Response to Reference Path D with $K_p = 0.065$ and $K_i = 0.000050$.

4.1 Long Term Reference Following

Figure 4.5 shows at the controller can maintain a reference speed over a long term period.

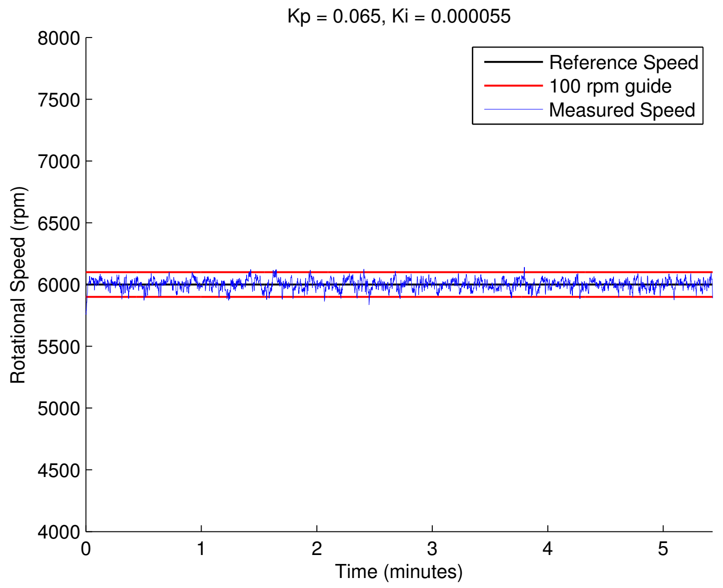


Figure 4.5: Long term reference speed tracking.

4.2 Setpoint Weighting

Figure 4.6 shows the response to Reference Path A if setpoint weighting is enabled. When compared to Figure 4.1 there is a smoother response to steps 1 and 6. The step 1 response reaches 90% of the reference value in 4.8 seconds with setpoint weighting compared to 2.9 seconds without. The step 1 settling time increases from 8 seconds to 10.2 seconds. The oscillations caused by the rapid drop in speed during step 6 are removed with setpoint weighting. With poor carburetor settings a sudden opening of the throttle would cause the engine to die. These sudden throttle openings resulting from a large reference step changes were eliminated with setpoint weighting without a large drop in responsiveness.

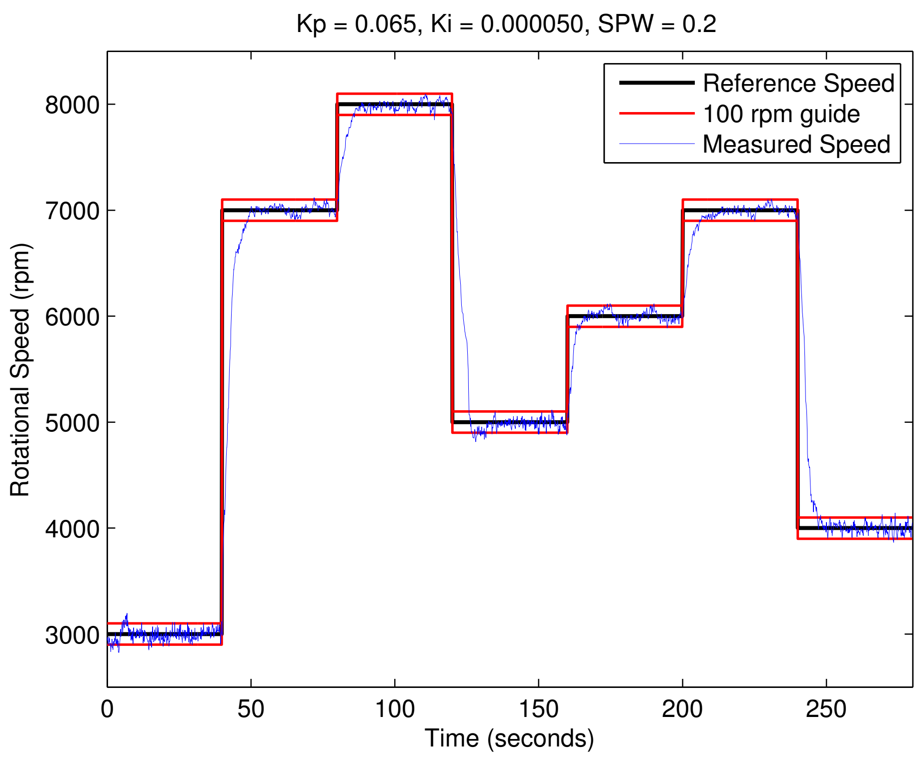


Figure 4.6: Response with setpoint weight of 0.20.

4.3 Bumpless Transfer

Figure 4.7 shows the transfer from manual to automatic control. The red guide lines indicate the start of automatic control. It is seen that this transfer is a bumpless transfer since there was no significant disruption of the controlled process. When the command is received to enter automatic control the reference speed is set to be the last measured speed. The algorithm keeps the speed near this last measured speed. The speed under manual control appears to have less variation around its average value than under automatic control. The standard error of the speed under manual manual control was 45.08 while the steady-state standard error under automatic control was 46.73. This increase in variance around the reference speed (average speed in the case of manual control) was considered acceptable.

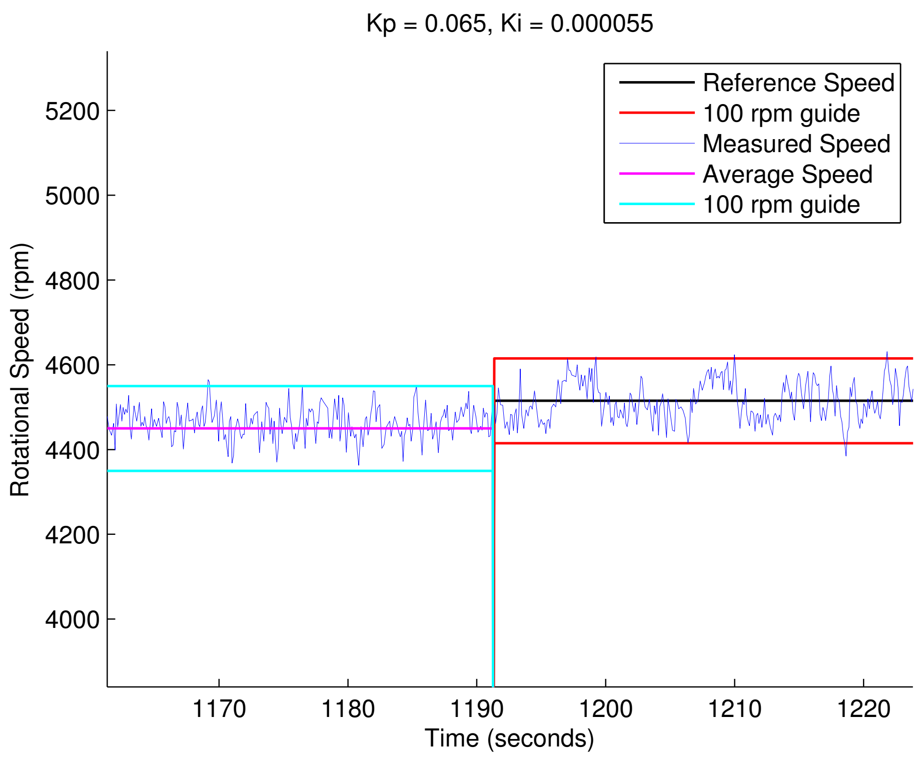


Figure 4.7: Successful bumpless transfer from manual to automatic control.

4.4 Idle performance

The engine was able to idle safely at 1800 rpm without issue under PI control. The manufacture lists 1800 rpm as the lowest practical speed that can be achieved with this engine [11]. Figure 4.8 shows the speed during idle with a reference speed of 1800 rpm. There are oscillations at this low speed that are not seen at speeds above 3000 rpm. A precise control of speed is not required at idle. The controller was required to keep the engine from stalling at low speeds. The controller met this requirement.

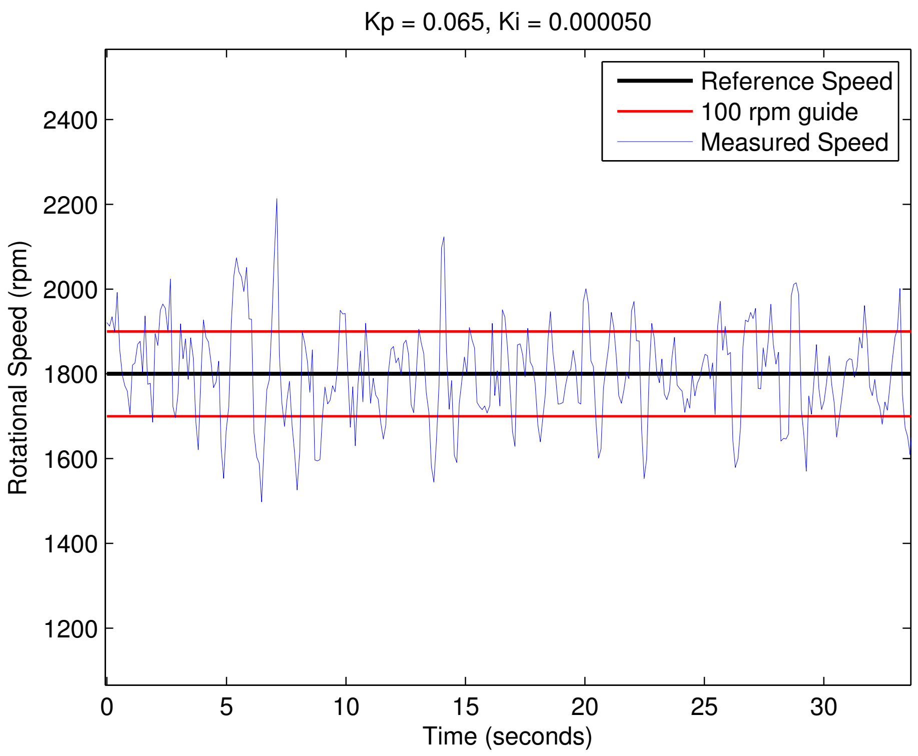


Figure 4.8: Controller performance with reference speed of 1800 rpm.

4.5 Disturbance Rejection

The disturbances on the engine that effect speed can come in the form of head or cross winds during flight and internal disturbances such as poor fuel flow and other irregularities. Figure 4.9 shows the rejection to a disturbance in the form of an emptying fuel tank. While this particular disturbance will be avoided in practice it helps to illustrate how the algorithm handles disturbances. As the fuel line nears the bottom of the tank there are sudden drops in speed. The control algorithm sees this and counters by increasing the value of the control variable and the speed returns to the reference value with some overshoot. The larger the drop in speed the greater the overshoot. A final surge of fuel allows the engine to maintain speed for about 20 seconds before the engine dies.

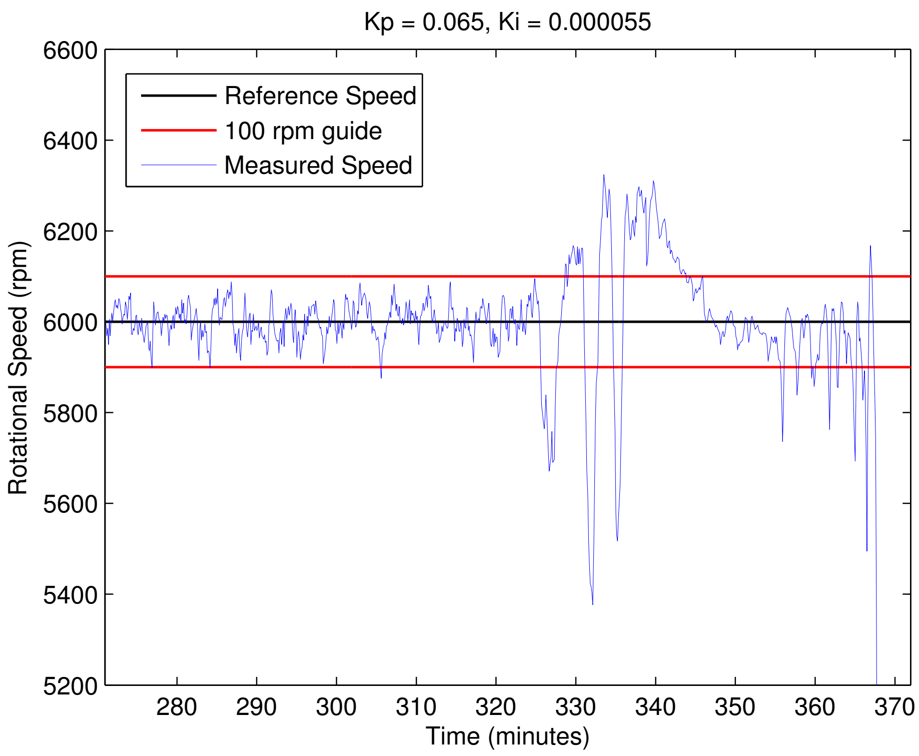


Figure 4.9: Disturbance in the form of an emptying fuel tank.

4.6 Control Variable Saturation

Figure 4.10 shows the speed response during control variable saturation. When the reference speed is increased to above the engine speed range the controller tries to bring the speed to this value but reaches the throttle limit. The throttle stays fully open until the reference speed is set to an achievable value. At this point the control algorithm resumes normal operation without a large disruption in the control process.

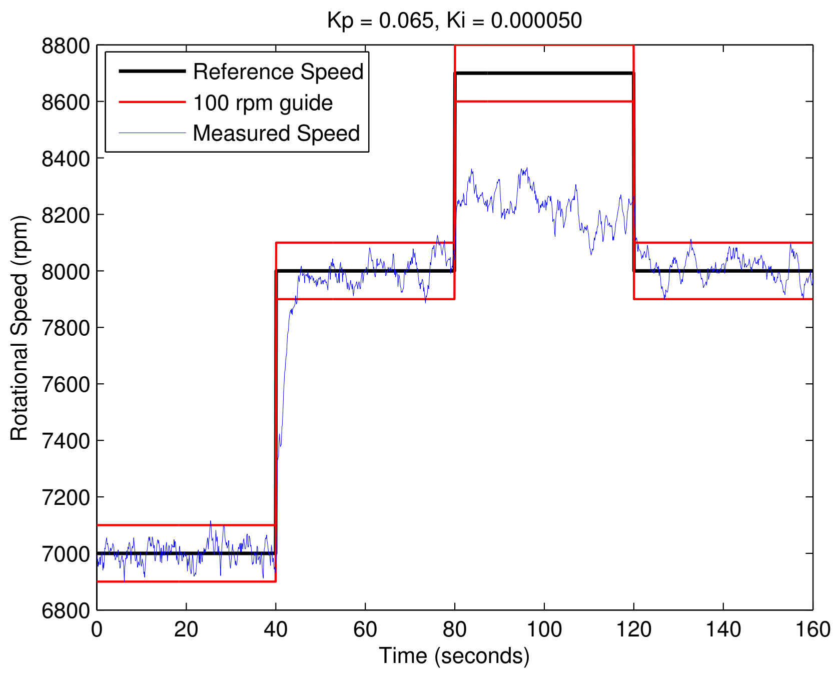


Figure 4.10: Control variable saturation.

4.7 Engine Temperature Monitoring

A requirement of the engine controller was that it monitor the temperature during flight. Figure 4.11 shows the measured values of engine temperature during a Reference Path A test. Higher speeds result in higher temperatures and a drop in speed allows the engine to cool. The initial drop in temperature is a result of the engine cooling after a warm up period. After the first increase in speed the temperature decreases for about 2 seconds before starting to rise. This is due to a sudden increase in the flow rate of air across the engine head and a rich air-fuel mixture that results from a wide throttle opening. The momentary increase in temperature after the first drop in speed is due to a suddenly lean air-fuel mixture and low air-speed across the engine head.

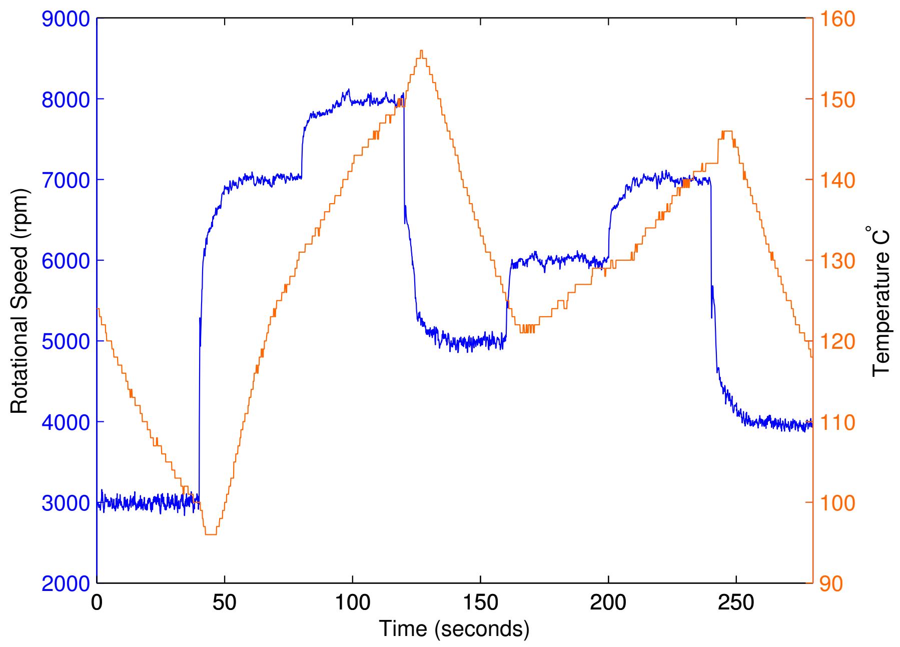


Figure 4.11: Engine temperature during a Reference Path A test

4.8 Controller Sensitivity to Carburetor Tuning

It was found that the controller is negatively affected by sub-optimal carburetor tunings. The main reason for the carburetor to be out of tune was a change in ambient temperature. In Section 3.4, it was found that the engine is most sensitive to changes in throttle position when transitioning from low to high speeds. This sensitivity was magnified by poor carburetor tunings, causing the response to be unstable in this region. The carburetor was tuned often to limit this sensitivity and ensure a controllable engine.

5 Conclusions

An engine controller was built to regulate the speed of a small two-stroke engine intended for use on a small UAV. The controller was developed for the OS160FX engine but it can be adapted to other comparable engines. The controller monitors the temperature of the engine and will alert the ground station if there is a unsafe operating temperature. It can also monitor altitude but the effects of altitude on the engine performance were not accounted for in the control algorithm. The engine controller was able to reliably regulate the speed of the OS160FX engine through the entire speed range of the engine. Responses to reference speed steps smaller than 4000 rpm have settling times less than 10 seconds with less than 200 rpm overshoot. Deviations from the reference speed at steady-state are kept under 100 rpm with occasional exceptions. The engine was able to safely idle at speeds as low as 1800 rpm. The engine controller can prevent the engine from stalling by dropping too far below this speed. The controller provides good rejection of disturbances caused by moderate drops and surges in fuel flow. The controller performance was found to be sensitive to the carburetor's fuel valve settings. A poorly tuned carburetor caused instability in the transition region between low and high speeds.

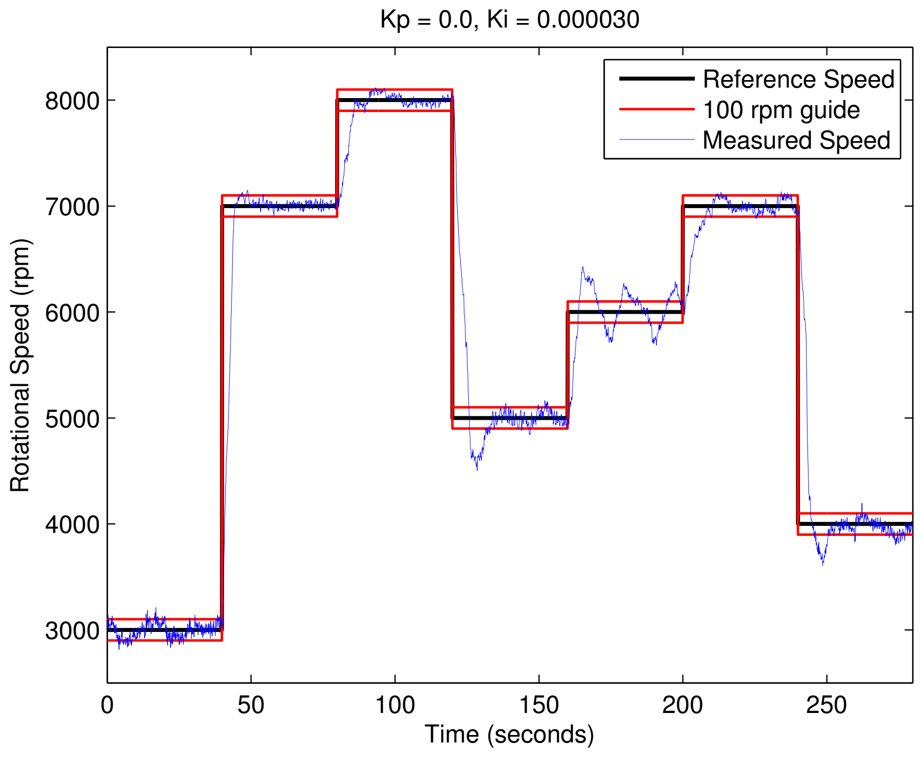
The incremental form of the PID algorithm was suited to the task of small two-stroke engine feedback control. It was found that only the proportional and integral term of the PID algorithm were necessary to achieve the performance goals. The derivative term was switched off by setting the derivative gain to zero. The integral term was the dominant factor in determining how quickly the engine reached the reference speed. The proportional term was necessary to dampen the integral term. An integral gain K_i of 0.000050 and a proportional gain K_p of 0.065 were chosen as the final tuning parameters. The incremental form of the PID algorithm allowed the controller to avoid issues related to control variable saturation and enabled a bumpless transfer from manual to automatic mode.

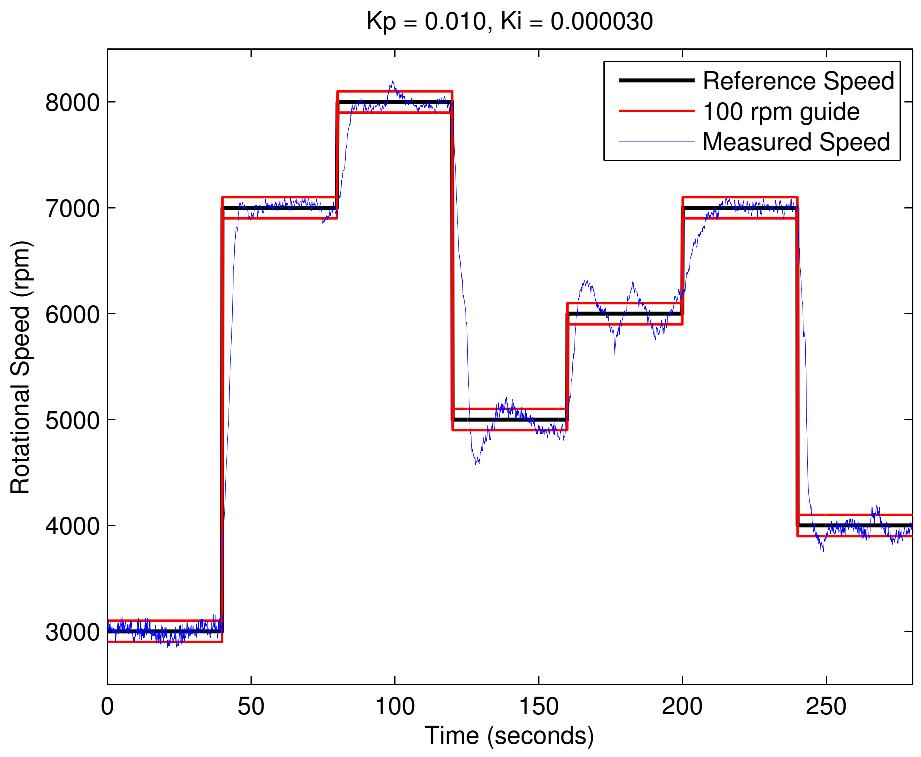
6 Future Work and Recommendations

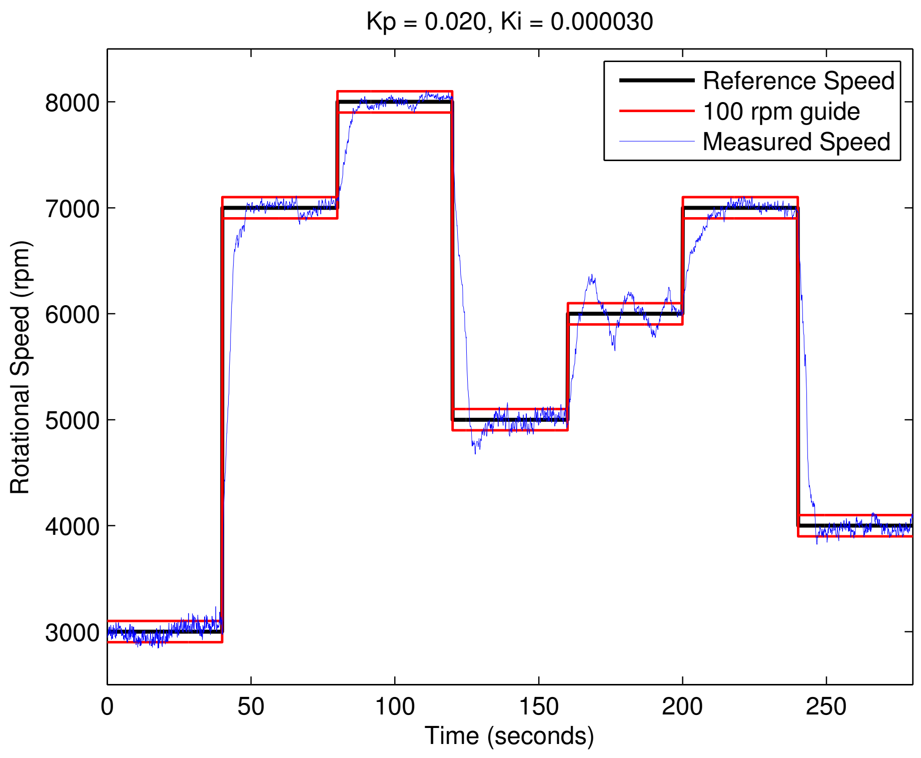
The next step in the development of the engine controller is to test it in flight. This requires interfacing the engine controller with the UAV's auto-pilot. The auto-pilot must send reference speed commands to the engine controller in the form of serial data. The auto-pilot must also be able to receive serial data from the engine controller so that it can relay sensor data to the ground station. The components of engine controller should integrated together in a smaller form than the current controller housing. This would require the sensors and micro-controller chip to be wired together using a printed PCB circuit board. Further steps should be taken to limit the effects of engine vibrations on the sensors to provide long term reliability. The performance of the IRE used to measure speed was sensitive to dust and drops of fuel. It should be replaced with a more durable and consistent encoder.

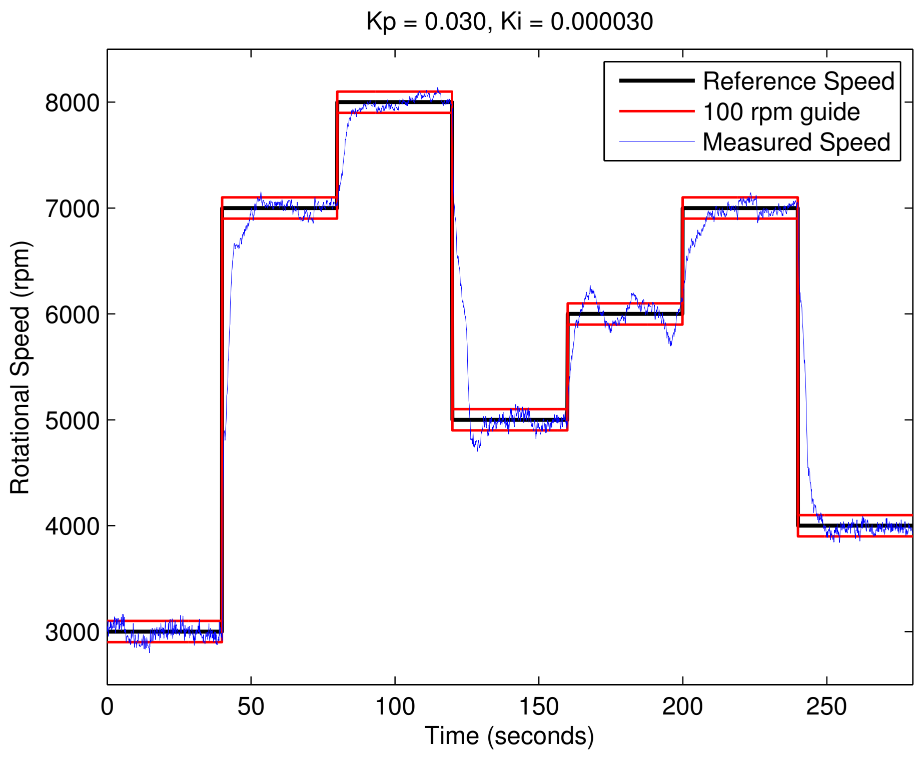
A possible avenue for future research would be to use the simplex method, or other numerical search method, to automatically tune the PID parameters. Automatic tuning would allow the controller to adjust for changing conditions and optimize the algorithm for the current operating point of the engine. The simplex method would perform a pattern search to find the PID tuning parameter combination that minimizes a performance measure or a function that incorporates multiple measures.

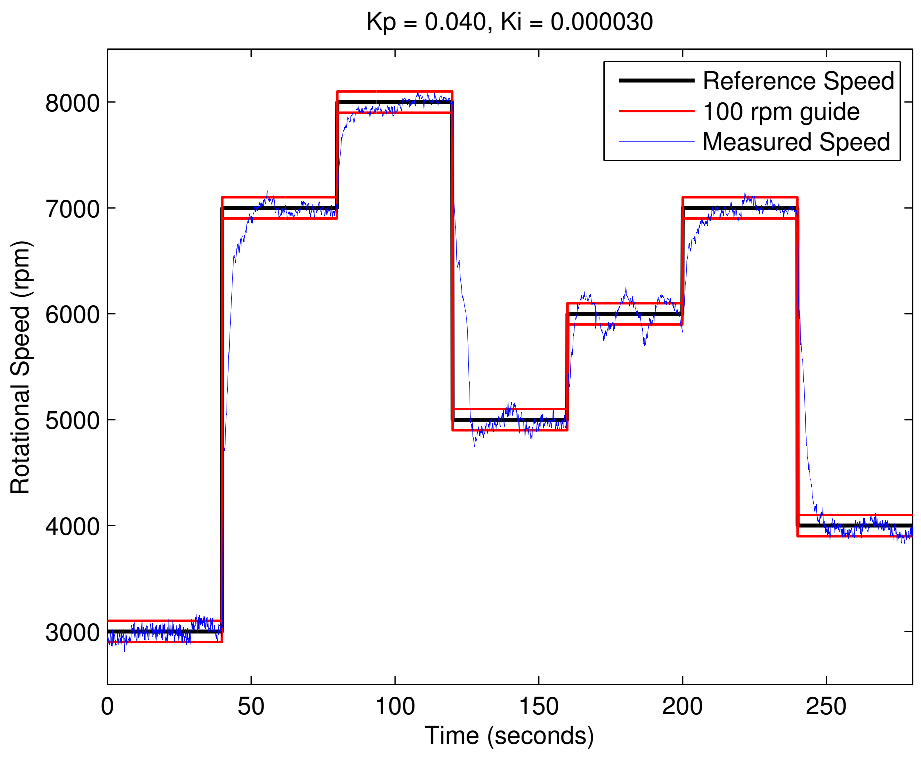
Appendix A Additional Test Response Plots

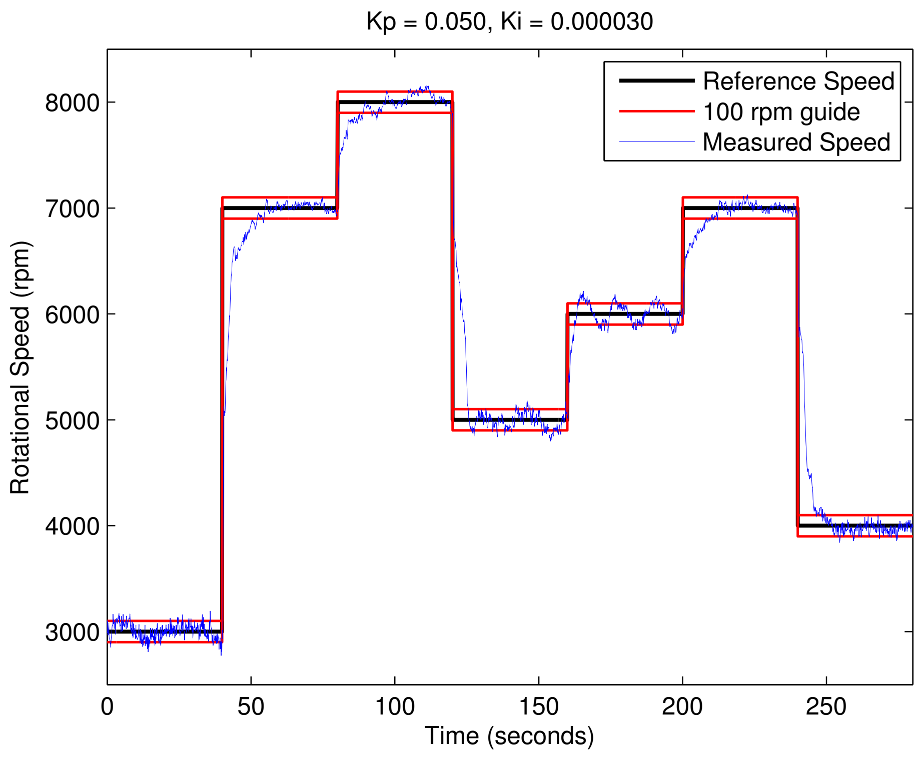



Kp = 0.010, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

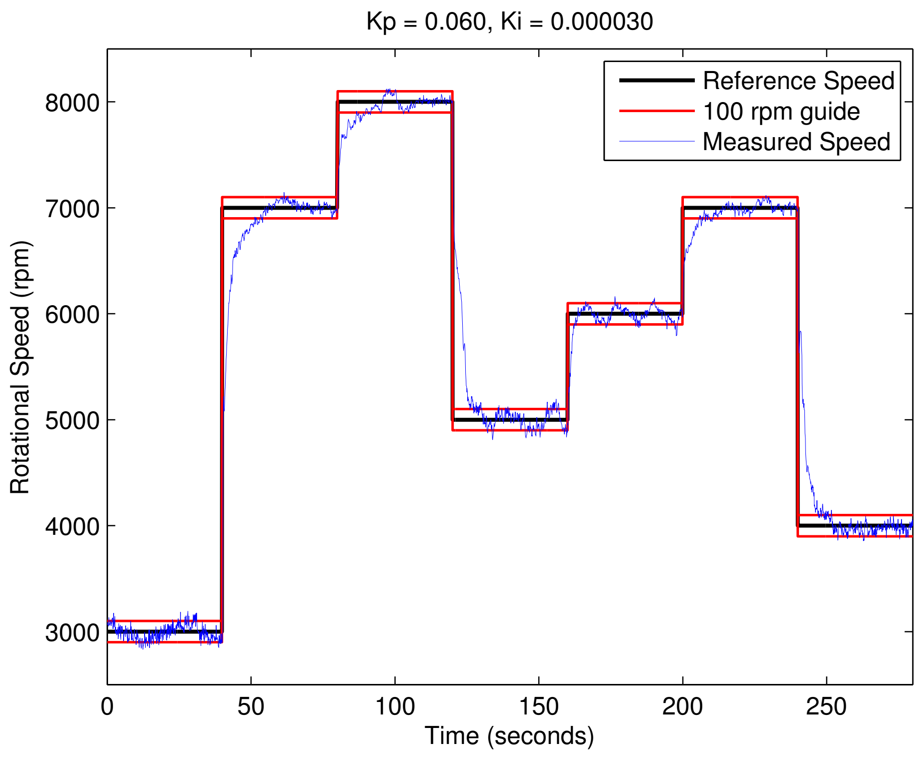

Kp = 0.020, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250


Kp = 0.030, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

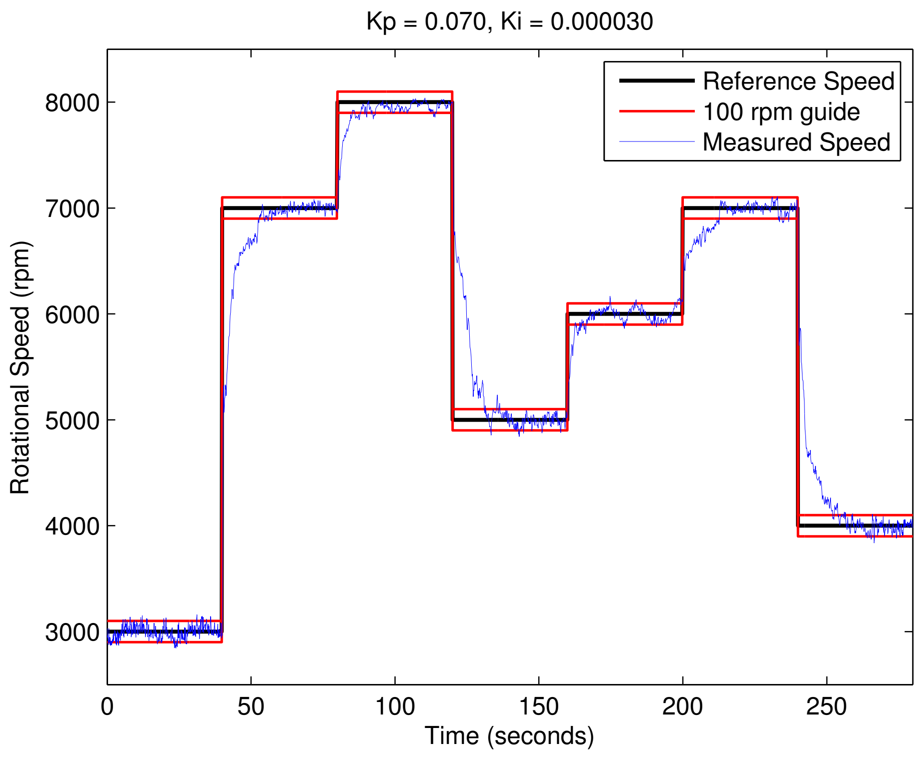

Kp = 0.040, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

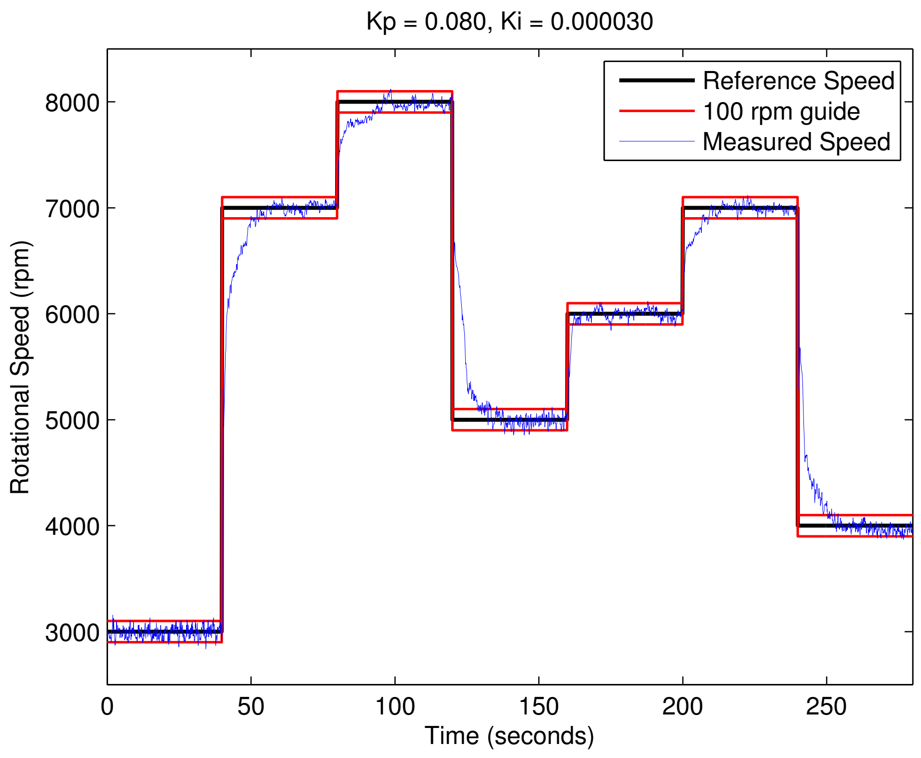


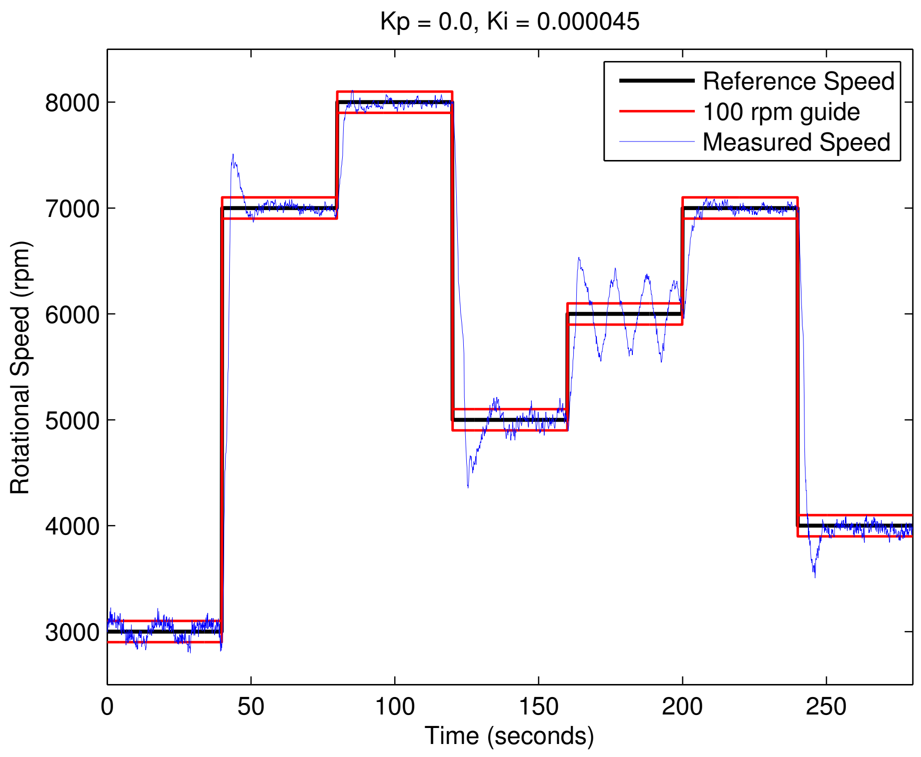
Kp = 0.050, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)



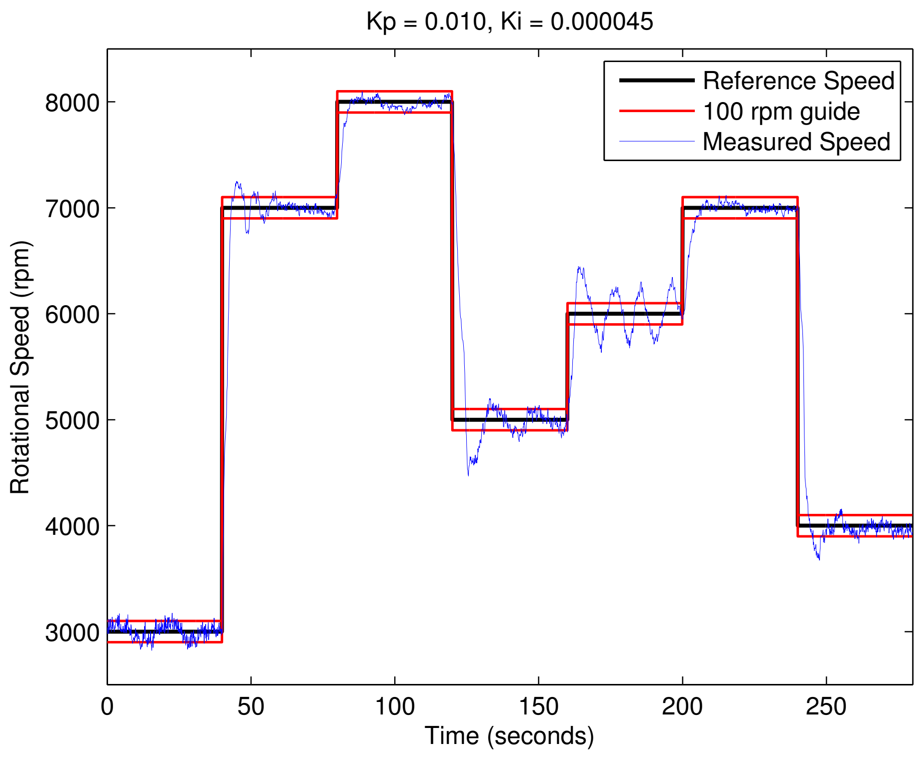
Kp = 0.060, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)


Kp = 0.070, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

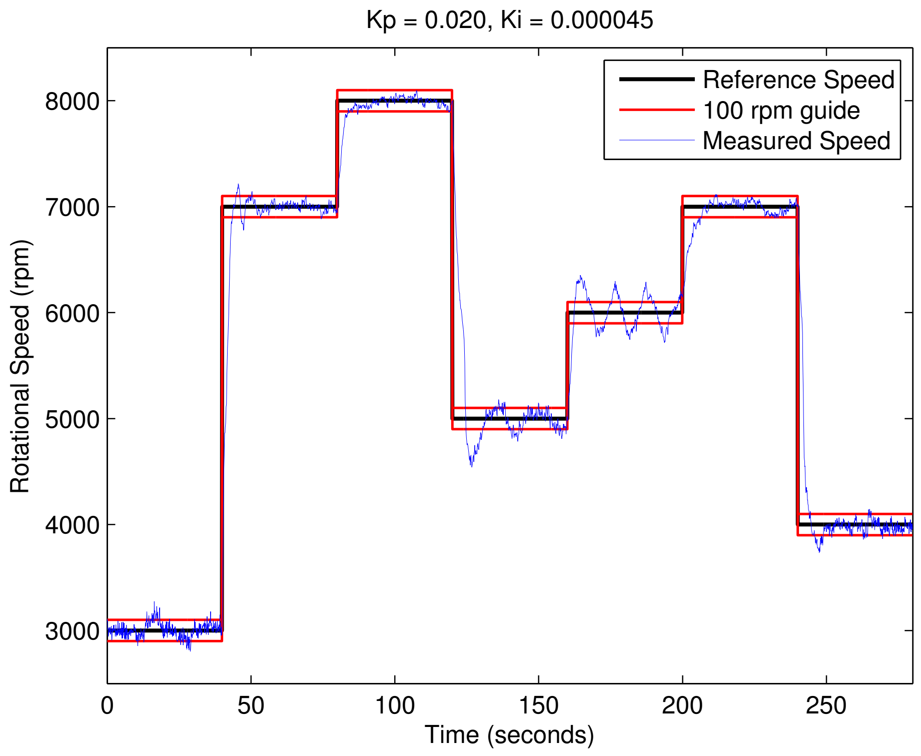

Kp = 0.080, Ki = 0.000030
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

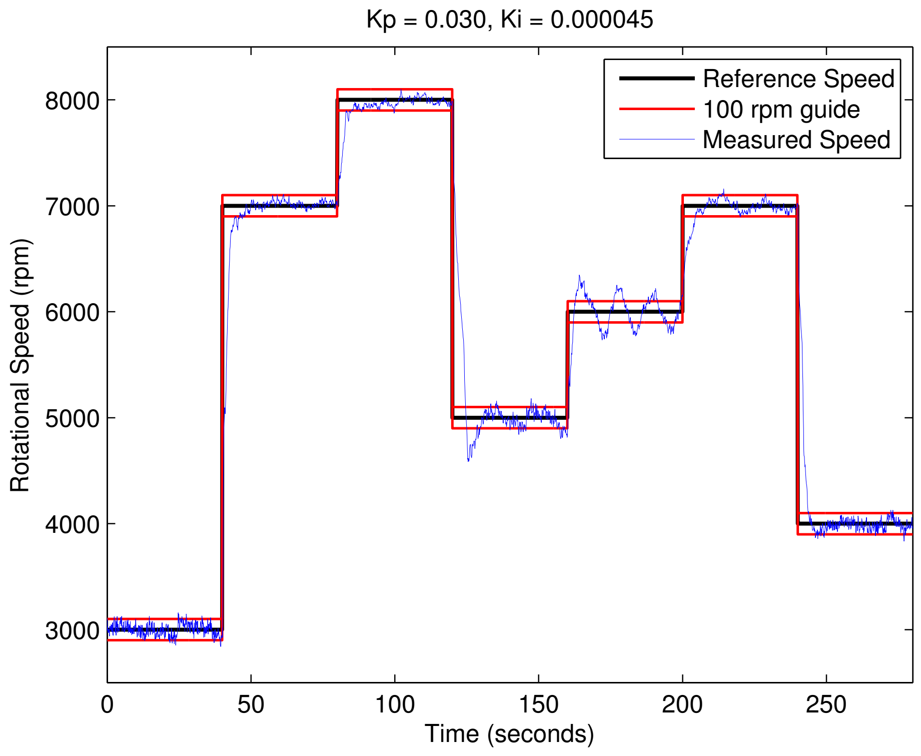


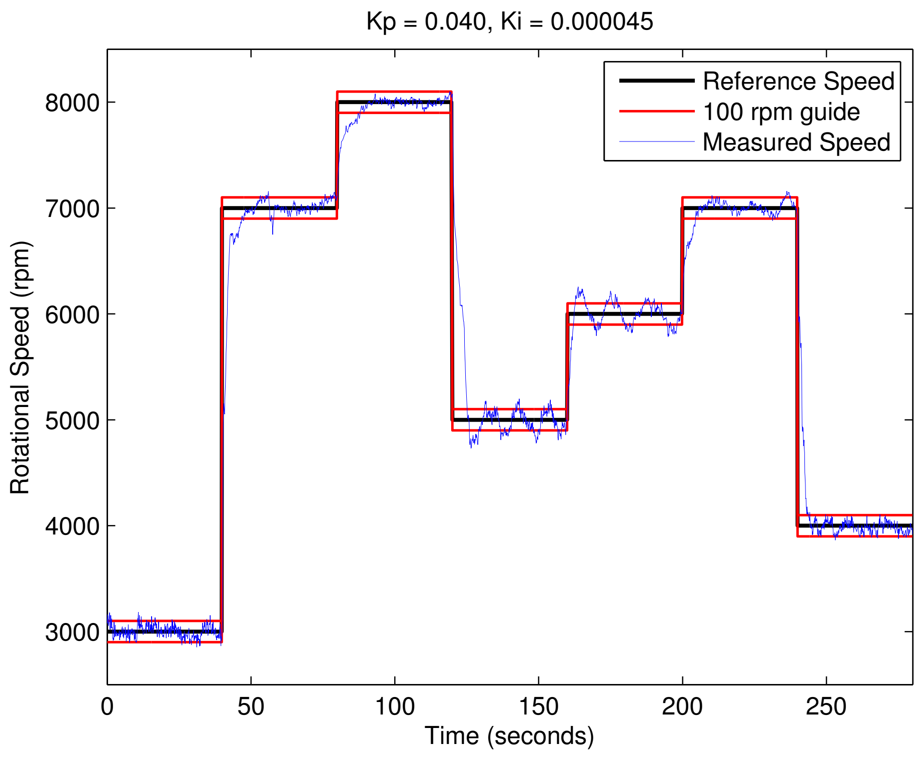
Kp = 0.0, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)

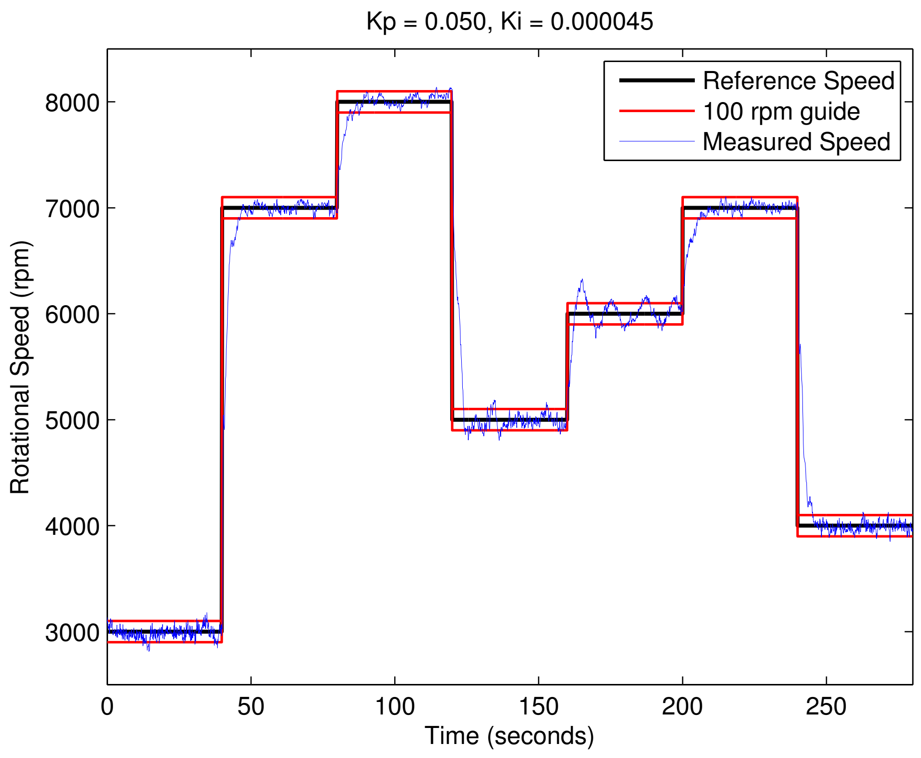


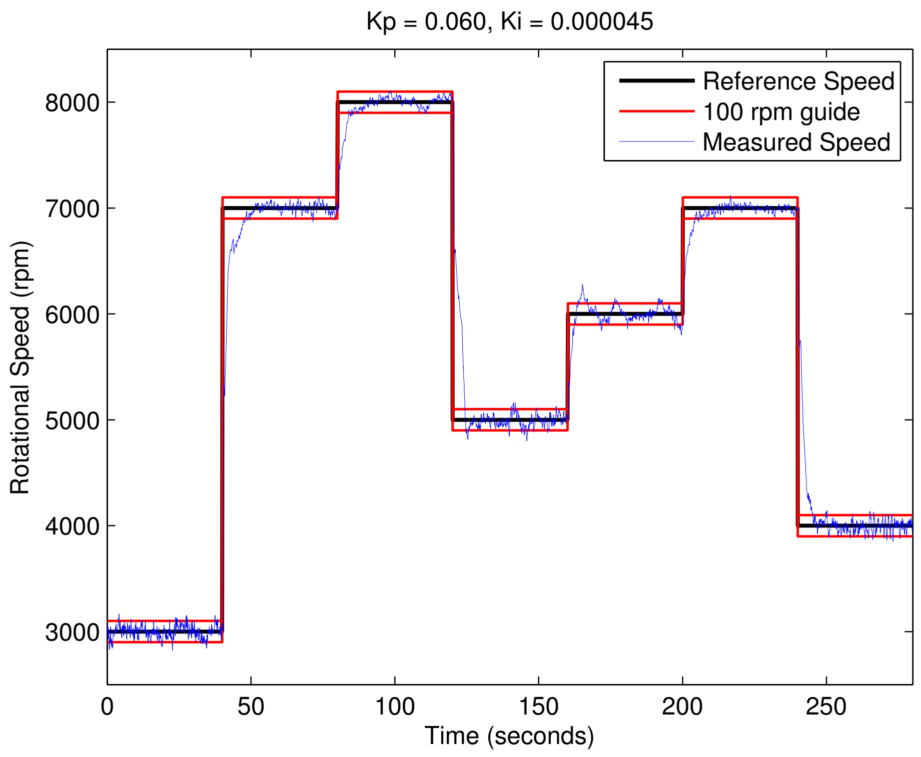
Kp = 0.010, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)

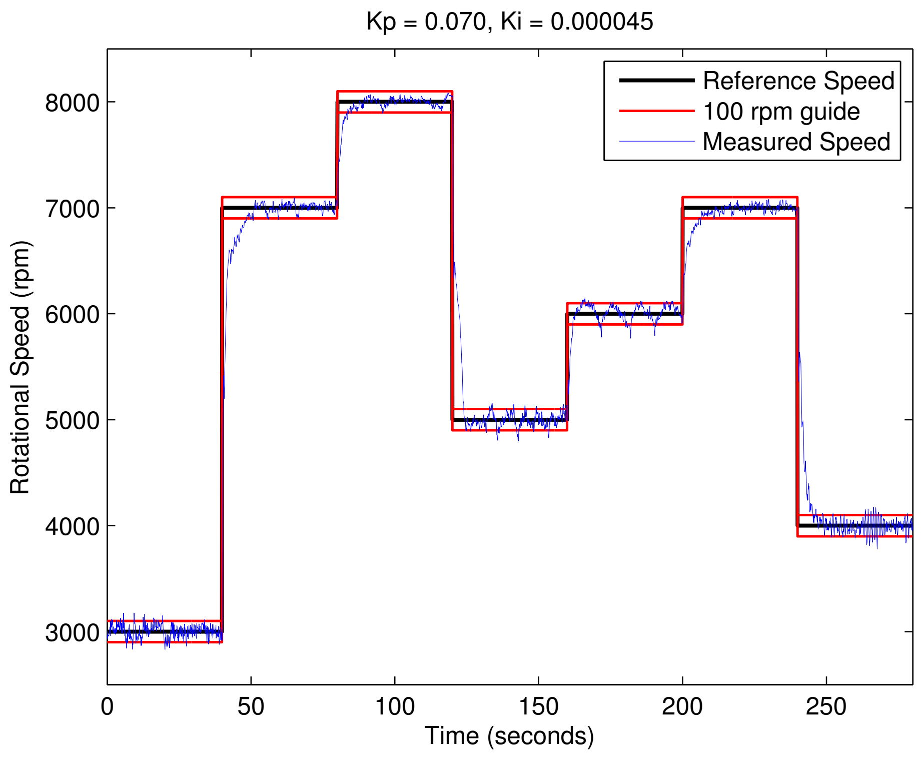

Kp = 0.020, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)

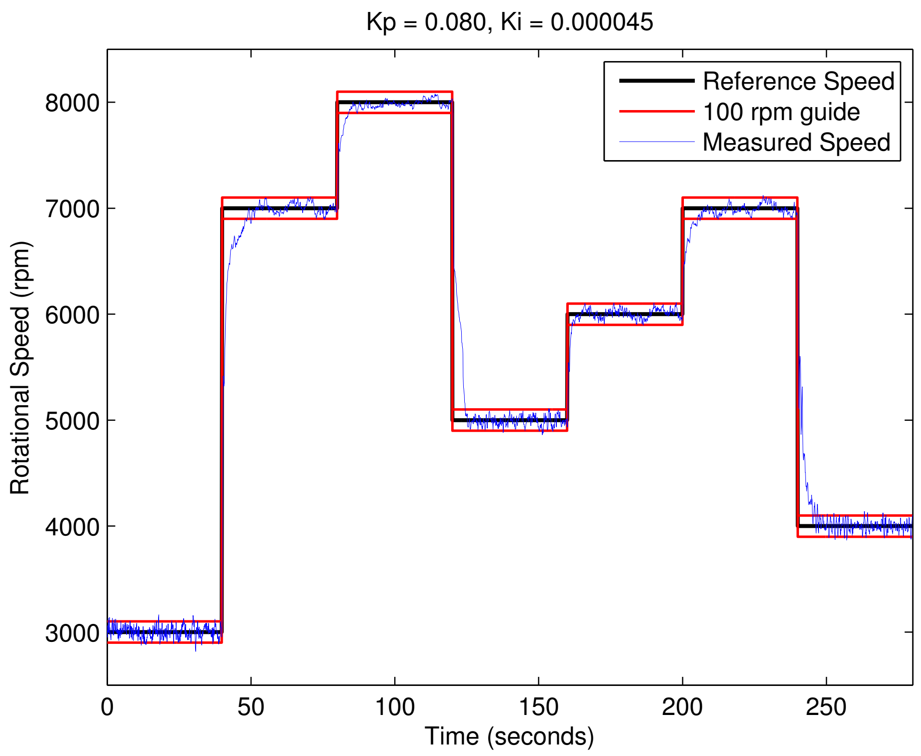

Kp = 0.030, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)


Kp = 0.040, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

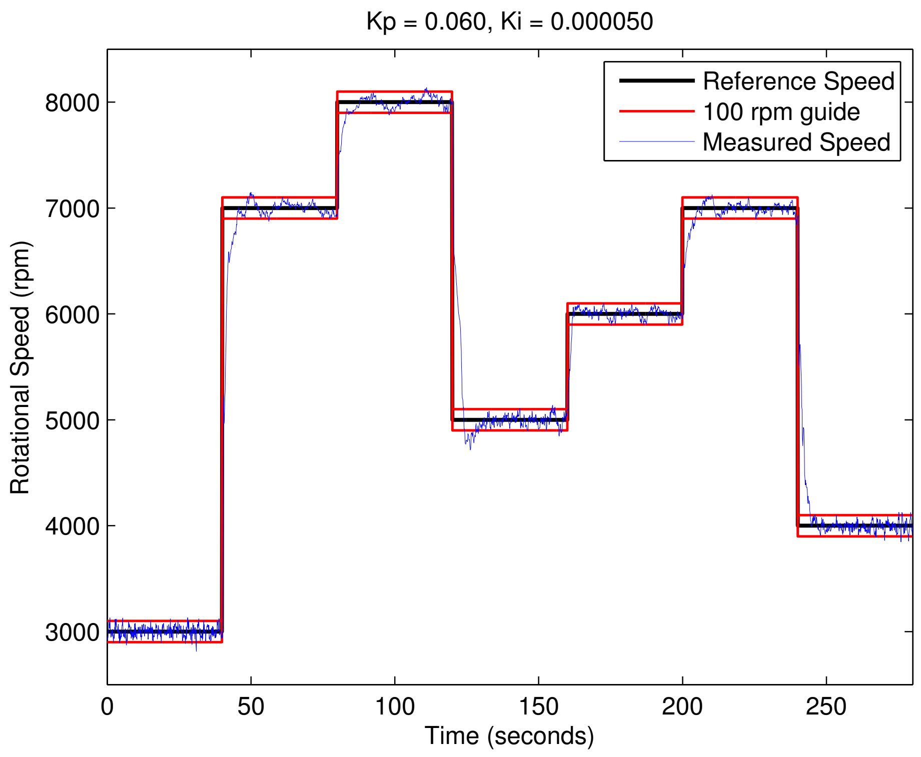

Kp = 0.050, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250

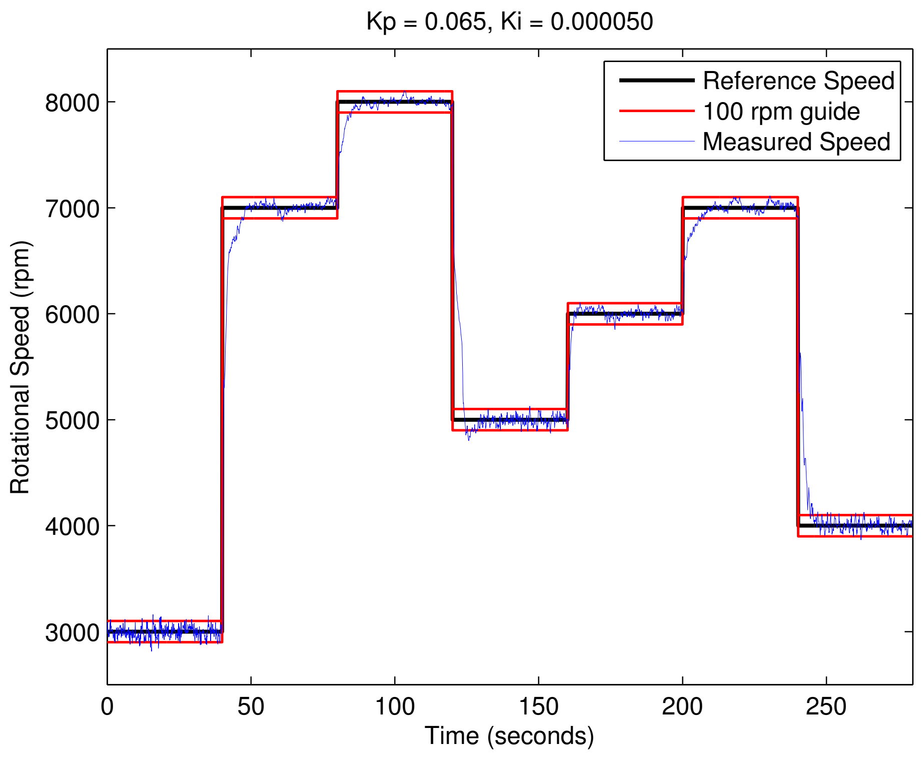

Kp = 0.060, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)

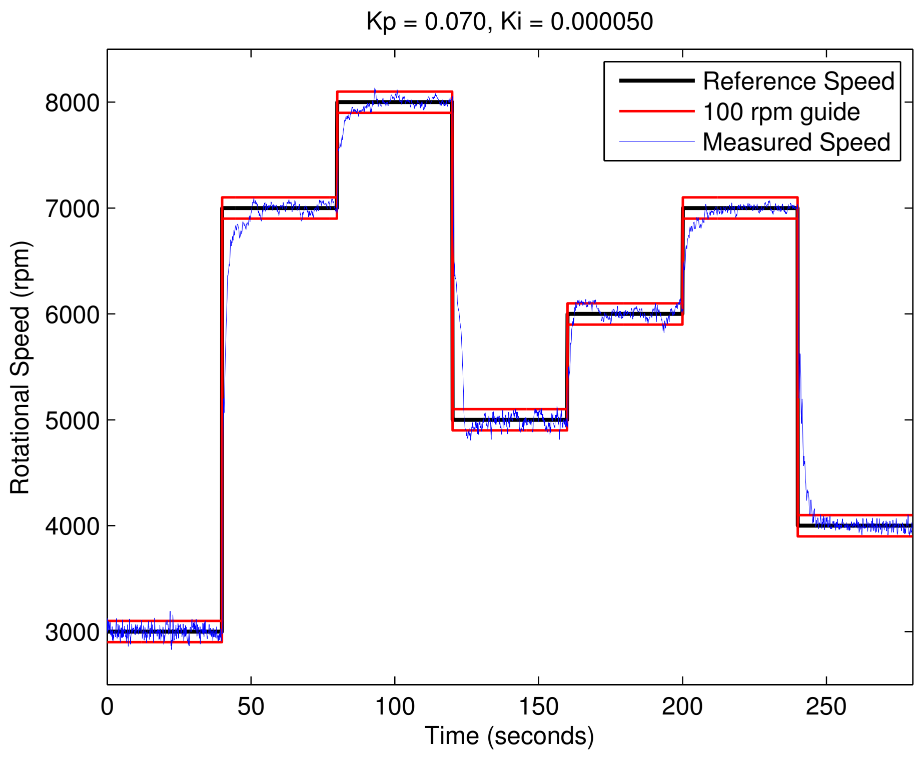

Kp = 0.070, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)

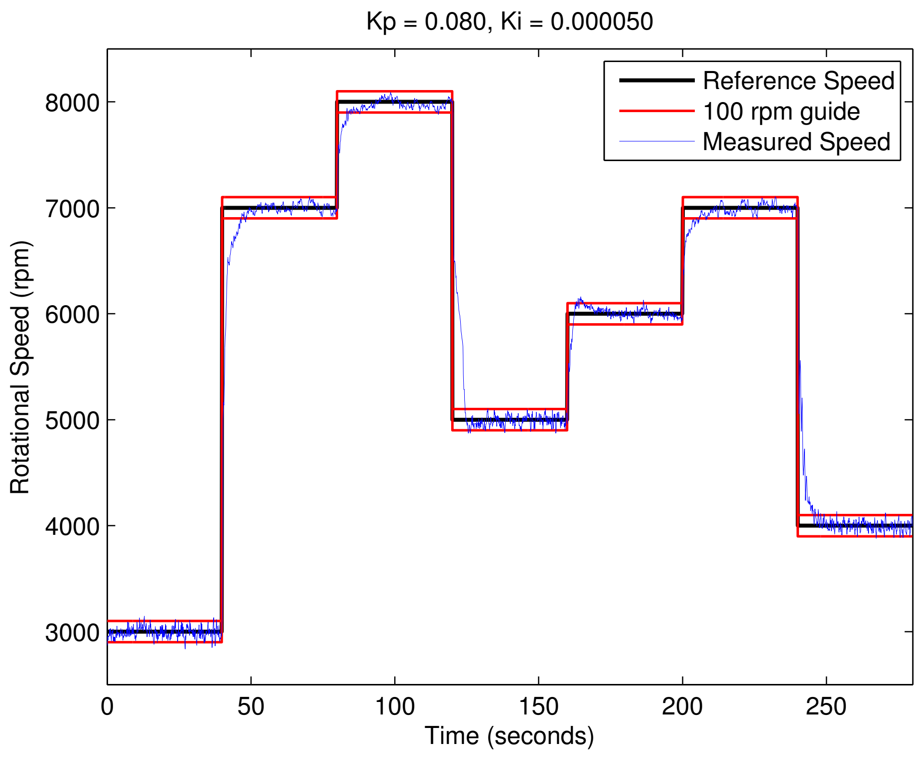


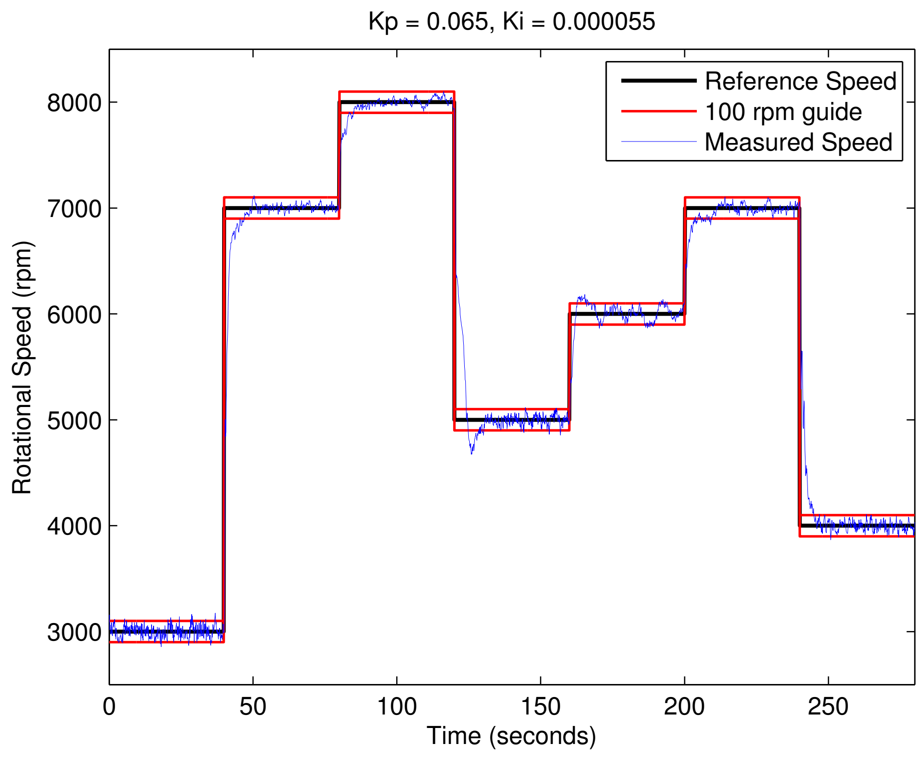
Kp = 0.080, Ki = 0.000045
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
3000
4000
5000
6000
7000
8000
0
50
100
150
200
250

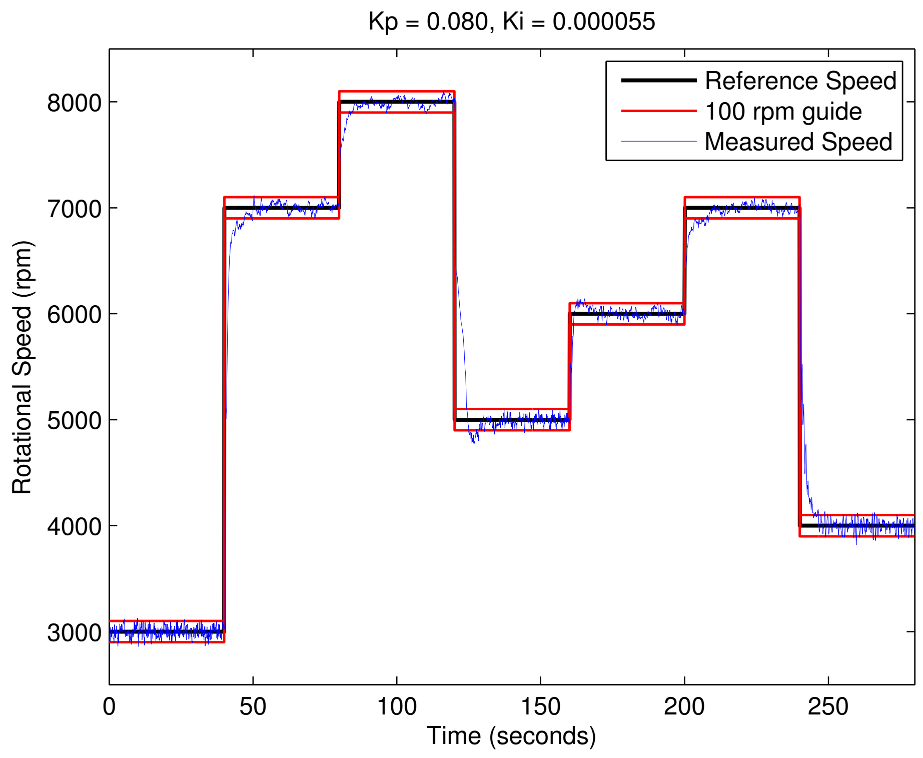

Kp = 0.060, Ki = 0.000050
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
0
50
100
150
200
250
3000
4000
5000
6000
7000
8000


Kp = 0.065, Ki = 0.000050
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
0
50
100
150
200
250
3000
4000
5000
6000
7000
8000

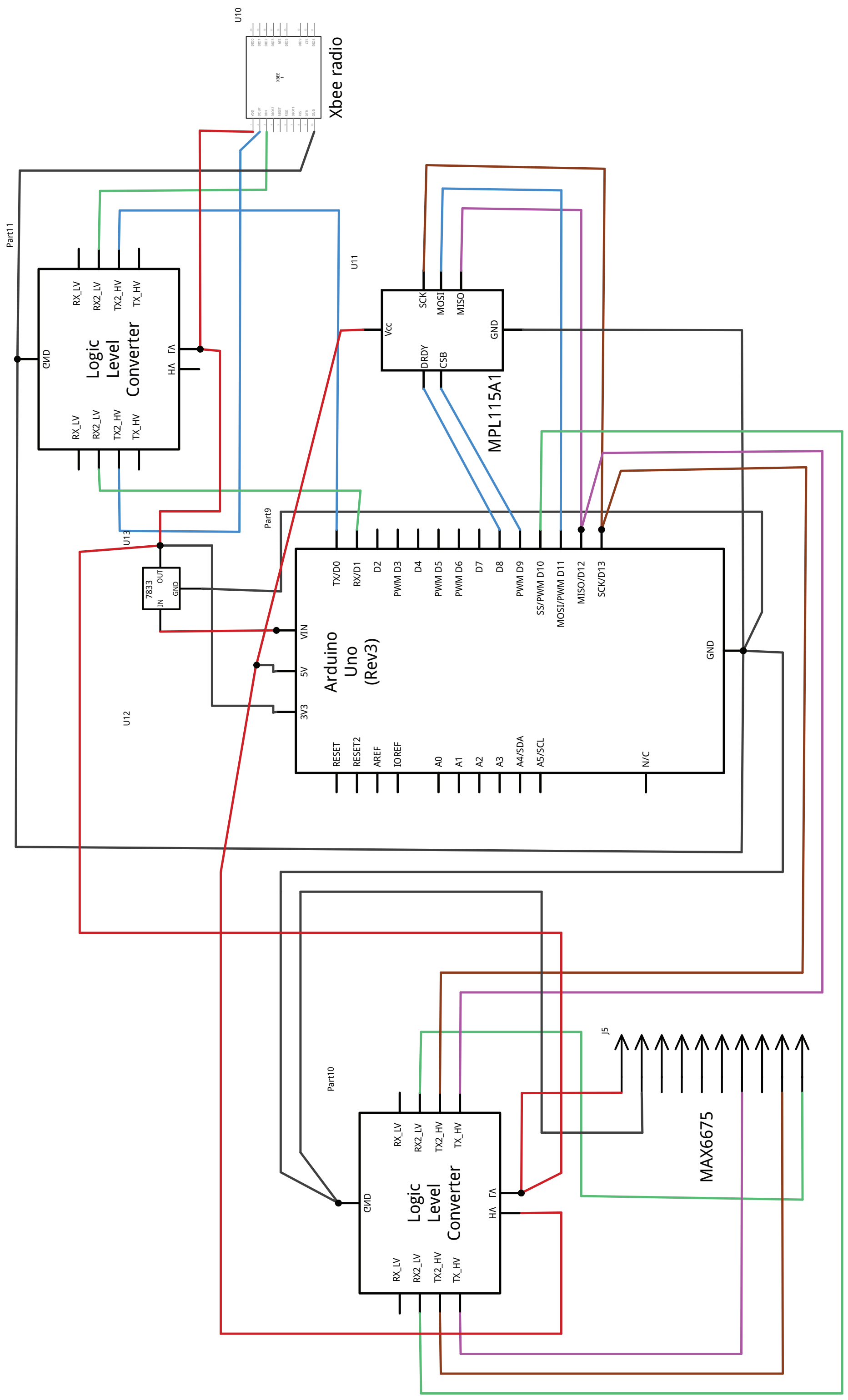

Kp = 0.070, Ki = 0.000050
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)


Kp = 0.080, Ki = 0.000050
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
8000
7000
6000
5000
4000
3000
0
50
100
150
200
250
Time (seconds)


Kp = 0.065, Ki = 0.000055
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
3000
4000
5000
6000
7000
8000
0
50
100
150
200
250


Kp = 0.080, Ki = 0.000055
Reference Speed
100 rpm guide
Measured Speed
Rotational Speed (rpm)
Time (seconds)
3000
4000
5000
6000
7000
8000
0
50
100
150
200
250

Appendix B Controller Schematic

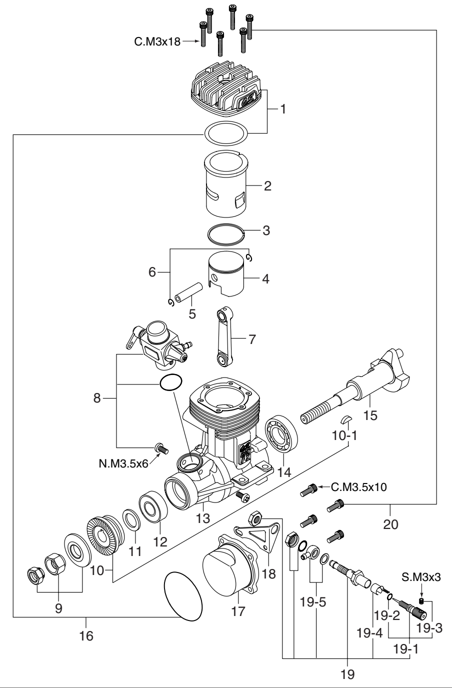


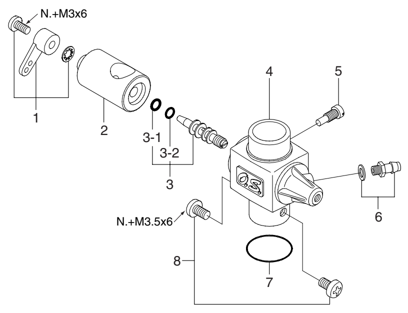
Appendix C Exploded View of Engine

O.S. ENGINE

MAX-160FXRING
(19260)

TYPE 60F
(29581000)





*Type of screw
C...Cap Screw M...Oval Fillister-Head Screw
F...Flat Head Screw N...Round Head Screw S...Set Screw

No.	Code No.	Description
1	29604000	Cylinder Head
2	29603100	Cylinder Liner
3	29603400	Piston Ring
4	29603200	Piston
5	29606000	Piston Pin
6	28117000	Piston Pin Retainers (2pcs.)
7	29405000	Connecting Rod
8	29581000	Carburetor Complete (Type 60F)
9	29610100	Propeller Lock Nut Set
10	29608000	Drive Hub
10-1	45508200	Woodruff Key
11	29320000	Thrust Washer
12	46231000	Crankshaft Ball Bearing (F)
13	29601000	Crankcase
14	29630000	Crankshaft Ball Bearing (R)
15	29602000	Crankshaft
16	29614000	Gasket Set
17	29607000	Cover Plate
18	71704240	Remote Needle Valve Bracket
19	28282000	Remote Needle Valve Assembly
19-1	28281970	Needle Assembly
19-2	24981837	"O" Ring (2pcs.)
19-3	26381501	Set Screw
19-4	26711305	Ratchet Spring
19-5	46181950	Fuel Outlet
20	29613000	Screw Set
	71608001	Glow Plug No.8
	29325000	E-5010 Silencer Assembly
	29325300	Assembly Screw
	29325400	Retaining Screw (C.M5x20)
	29326000	Exhaust Adaptor

No.	Code No.	Description
1	27881400	Throttle Lever Assembly
2	29581200	Carburetor Rotor
3	45582300	Mixture Control Valve Assembly
3-1	46066319	"O" Ring (L)
3-2	24881824	"O" Ring (S)
4	29581100	Carburetor Body
5	45581820	Rotor Guide Screw
6	22681953	Fuel Inlet (No.1)
7	29015019	Carburetor Gasket
8	25081700	Carburetor Fixing Screw

O.S. GENUINE PARTS & ACCESSORIES

Code No.	Description
71920000	Radial Motor Mount
72200080	Needle Valve Extension Cable Set
72200130	Booster Terminal Kit
29310110	3/8" -M5(S) Propeller Locknut Set For Truturn Spinner
71531000	Non-Bubble Weight
72403050	Super Filter (L)
79870050	M5 Blind Nut (10pcs.)
55500004	M5 Lock Washer (10sets)
79871070	M3.5x10 Cap Screw Set (10pcs.)
71521000	Long Socket With Plug Grip

The specifications are subject to alteration for improvement without notice.

Appendix D Micro-controller Code

```
///////////////////////////////////////////////
///////       HOWARD HUGHES      ////////////////
////// / / / / | / / /| |  / / /////////////////
///// / / / /  |/ / / | | / / //////////////////
//// / /-/ / /|  / /--| |/ / ///////////////////
///  \----/-/ |-/-----/---/ //// PAUL FJARE ////
//  COLLEGE OF ENGINEERING ///// MC1 CODE //////
////////////////////////////////////////////////

////////////////
// LIBRARIES  //
////////////////
#include <stdlib.h> // C Standard General Utilities Library
#include <util/delay.h> // AVR delay functions
#include <Servo.h>  // Servo library
#include <MPL115A1.h> // MPL115A1 barometric pressure sensor
    library
#include <max6675.h> // MAX6675 K-type thermocouple sensor
    library
#include <SPI.h> // SPI library (required for MPL115A1 and
    MAX6675)

//////////////////////////////
// ENGINE UNIQUE PARAMETERS //
//////////////////////////////

// Maximum realistic RPM
const long max_RPM = 9000;
// Minimum realistic RPM
const long min_RPM = 1;

//////// SERVO POSITION CONSTANTS ////////
// Position of throttle servo at closed throttle
const int throttle_closed = 1810;
// Position of throttle servo at open throttle
const int throttle_open = 1300;
// Position of throttle servo at idle
const int throttle_idle= 1726;
// Position of throttle servo at minimum throttle
// where engine will not stall.
const int throttle_safe = 1810;
// Throttle servo position range
const int throttle_range = throttle_closed - throttle_open;

//////////
```

```
// PINS //
//////////
const int throttlePin = 5; // throttle servo
const int fuelPin = 3; // fuel valve servo
const int relayPin1 = 4; // glow plug relay
const int relayPin2 = 6; // heater relay
const int tachometerPin = 2; // tachometer

/////////////
// SENSORS //
/////////////
// Initialize thermocouple sensor object
const int thermoCS = 10; // chip select pin
MAX6675 thermocouple(thermoCS);
double minimumTemp = 100.0;
double maxTemp = 200.0;
// Initialize barometric pressure sensor object
const int baroCS = 9; // chip select pin
const int baroSDN = 8; // shutdown pin
MPL115A1 barometer(baroCS, baroSDN);

//SSS
////////////
// SERVOS //
////////////
// Initialize throttle servo
Servo throttle;
// Initialize fuel valve servo
Servo fuel;
// Servo positions
int throttlePos, fuelPos;

////////////
// TIMING //
////////////
long Time; // milliseconds
long now, lastNow, thermoNow, thermoLastNow;
long reportNow,reportLastNow; // milliseconds
long before, after, duration; // milliseconds
long nowTest, lastNowTest; // milliseconds
long testInterval = 40000; // milliseconds

////////////////
// TACHOMETER //
////////////////
// Time between pulses from tachometer (microseconds)
volatile long period;
volatile long microseconds;
```

```
//////////////////////
// MEASURED VALUES //
//////////////////////

// Temporary storage for incoming data
long value;
double temperature; // Celcius
double pressure; // inHg
double measuredSpeed, setpointSpeed;
double smoothMeasuredSpeed, lastSetpointSpeed;
double lastMeasuredSpeed, lastLastMeasuredSpeed;
double lowpassSpeed, lastLowpassSpeed;

/////////////////////////////////////
// CONTROL ALGORITHM PARAMETERS //
/////////////////////////////////////

// Sample time of algorithm (milliseconds)
long SampleTime = 100;
// Parameters for velocity form of algorithm
double K1, K2, K3;
// Output parameters
double output, lastOutput;

// Proportional gain
double Kp = 0.045;
// Integral gain
double Ki = 0.000045;
// Derivative gain
double Kd = 0.0;
double kp, ki, kd;


double setpointWeight = 1.0; // set to 1.0 to disable effect of
    setpoint weighting
double alpha = 0.0; // set to 0.0 to disable lowpass filter on
    measured speed signal

///////////////
// BOOLEANS //
///////////////

// Manual of Automatic (PI)
boolean manual = true;
// Start or stop test routine
boolean testRoutine = false;
boolean heating = false;
```

```
int testIteration = 0;

void setup(){

   // Initialize serial communication
   Serial.begin(9600);

   // Set modes for tachometer hardware interrupt pin
   // and relay pin.
   pinMode(tachometerPin, INPUT);
   pinMode(relayPin1, OUTPUT);
   pinMode(relayPin2, OUTPUT);

    // Declare hardware interrupt pin and
    // interrupt service routine (ISR).
   attachInterrupt(0, tachometer, RISING);

   // Set throttle pin
   throttle.attach(throttlePin); // attach servo objects to pins

   // Set initial throttle position
   throttlePos = throttle_idle;
   throttle.writeMicroseconds(throttle_idle);

   // Set initial tuning values
   SetTunings(Kp,Ki,Kd);

}

///////////////
// MAIN LOOP //
///////////////

void loop(){ // Main program loop

   // Get serial commands
   GetCommands();

   auxillaryHeating();

   manageTestRoutine();

   // Determine current time
   now = millis();

   if(now-lastNow >= 80){

      // Take sensor readings to update values
```

```
        // of pressure, temperature, and speed.
        updateMeasurements();

        // Run control algorithm if in auto mode
         if(!manual)
           velPID();

        Time = millis();

        // Report sensor and control algorithm data
        ReportData();

         lastNow = millis();

    }
}

//////////////////////
// CONTROL ALGORITHM //
//////////////////////

void velPID(){ // PID algorithm, velocity form

    /* Apply 1st order lowpass filter to measured speed. Alpha is
        a weighting
        factor between 0 and 1 that that determines the
            aggressiveness of the
        filter. Setting alpha to zero disables filtering.
     */
    lowpassSpeed = alpha*lastLowpassSpeed + (1-alpha)*
        measuredSpeed;

    // Calculate terms of PID velocity algorithm
    // Proportional Term
    K1 = kp*setpointWeight*(setpointSpeed - lastSetpointSpeed)
        + kp*(lastMeasuredSpeed - lowpassSpeed);
    // Integral Term
    K2 = ki*(setpointSpeed - lowpassSpeed);
    // Derivative Tem
    K3 = kd*(2*lastMeasuredSpeed - lowpassSpeed -
        lastLastMeasuredSpeed);

    // Calculate control variable
    output = lastOutput - K1 - K2 - K3;

    // Convert output to closest integer
    throttlePos = floor(output + 0.5);
    /* Prevent overdriving of throttle servo and stalling of
       engine in the event of control variable saturation.
     */
    if(throttlePos < throttle_open){
       output = (double) throttle_open;
       throttlePos = throttle_open;
    }
    if(throttlePos > throttle_safe){
        output = (double) throttle_safe;
       throttlePos = throttle_safe;
    }

    // Remember variables for next iteration
    lastLowpassSpeed = lowpassSpeed;
    lastLastMeasuredSpeed = lastMeasuredSpeed;
    lastMeasuredSpeed = lowpassSpeed;
    lastSetpointSpeed = setpointSpeed;
    lastOutput = output;

    // Write the throttle position to servo.
    throttle.writeMicroseconds(throttlePos);
}

void SetTunings(double propgain, double intgain, double dergain){
    kp = propgain;
    ki = intgain * (double) SampleTime;
    kd = dergain / (double) SampleTime;
}

/////////////////////////
// UPDATE MEASUREMENTS //
/////////////////////////
void updateMeasurements(){
    /**
     * Update measured values of engine speed, engine head
     * temperature, and barometric pressure.
     */
    measuredSpeed = calculateSpeed(period, max_RPM, min_RPM,
       measuredSpeed);
    //Serial.println(millis());
    detachInterrupt(0);
    pressure = barometer.baropPressure();
    thermoNow = millis();
    if(thermoNow - thermoLastNow > 300){
       temperature = thermocouple.readCelsius();
       thermoLastNow = millis();

    }
    attachInterrupt(0, tachometer, RISING);
    //Serial.println(millis());
    //duration of this function is about 6 ms
}

///////////////////////////////
// ENGINE SPEED MEASUREMENT //
///////////////////////////////

double calculateSpeed(long PERIOD_uS, long MAX_RPM, long MIN_RPM,
     double DEFAULT_RPM){

    /**
     * This function calculates the speed of an engine shaft with
        a
     * 1 PPR (pulses per revolution) tachometer. The function
        returns
     * speed in RPM [rev/min].
     *
     * ///////// PARAMETERS /////////
     *
     * PERIOD: time between pulses of the tachometer in
        microseconds.
     * MAX_RPM: maximum realistic value of engine speed in RPM.
     * MIN_RPM: minimum realistic value of engine speed in RPM.
     * DEFAULT_RPM: value of speed to return if the value of the
     * period is not within limits.
     *
     * ///////// EQUATIONS /////////
     *
     * speed [rev/min] = 60,000,000 [microsec/min] / period [
        microsec/rev]
     *
     * period [microsec/rev] = 60,000,000 [microsec/min] / rpm [
        rev/min]
     */

    double rpm;
    long period_us = PERIOD_uS;
    long max_rpm = MAX_RPM;
    long min_rpm = MIN_RPM;
    double default_rpm = DEFAULT_RPM;

    // Maximum allowable period
    static long period_max = 60000000L/min_rpm;
    // Minimum allowable period
    static long period_min = 60000000L/max_rpm;
    // Check that period is within limits
    if(period_us > period_min && period_us < period_max)
       // Awesome! Calculate speed
       rpm = 60000000.0 / (double) period_us;
    else
       // Use default value of speed
       rpm = default_rpm;

    return(rpm);
}

void tachometer() {
    /**
     * ISR (Interrupt Service Routine) that is triggered when the
        microcontroller
     * recieves a pulse from the tachometer. It calculates the
        time between the
     * current pulse and the previous pulse, the period.
     *
     * Variables shared between ISR functions and normal functions
      * should be declared "volatile". This tells the compiler
         that such
      * variables might change at any time, and thus the compiler
         must
      * reload the variable whenever you reference it, rather than
      * relying upon a copy it might have in a processor register.
     */

    period = micros() - microseconds;
    microseconds = micros();
}

void bumplessTransfer(){
    /**
     * This function is called when the program switches
     * to automatic mode (PID algorithm is active). It sets
     * all terms of the PID algorithm to zero to remove any
     * residual values created during previous periods of
     * automatic operation. The setpoint speed is set to
     * the most recent measured speed to prevent an abrupt
     * jump in speed. The algorithm remembers the control
     * output, setpoint speed, and measuremed speed from
     * previous iterations. Since there has been no previous
     * iteration these values must be set to the current values.
     */
    K1 = 0;
    K2 = 0;
    K3 = 0;
    filteredK3 = 0;

    output = (double) throttlePos;
    lastOutput = output;
    setpointSpeed = measuredSpeed;
    lastSetpointSpeed = measuredSpeed;
    lowpassSpeed = measuredSpeed;
    lastLowpassSpeed = lowpassSpeed;
    lastLastMeasuredSpeed = measuredSpeed;
    lastMeasuredSpeed = measuredSpeed;

}

////////////////////////
// AUXILLARY HEATING //
////////////////////////

void auxillaryHeating(){
    if(temperature > maxTemp){
         Serial.println("e");
    }
    if(heating && temperature >= minimumTemp){
        digitalWrite(relayPin1, LOW);
        digitalWrite(relayPin2, LOW);
        heating = false;
    }
}
///////////////////
// TEST ROUTINE //
///////////////////

void manageTestRoutine(){

static long steps = 22;
long testSetpointSpeed[22] =
    {6000,6200,6400,6600,6800,7000,7200,7400,7600,7800,
    8000,7800,7600,7400,7200,7000,6800,6600,6400,6200,6000,2999};

    if(testRoutine){
        nowTest = millis();
        if(nowTest-lastNowTest > testInterval){
            // throttlePos = testThrottle[testIteration];
            // throttle.writeMicroseconds(throttlePos);
            setpointSpeed = testSetpointSpeed[testIteration];
            testIteration += 1;

            if(testIteration > steps-1 ){
                testRoutine = false;
            testIteration = 0;
          }
          lastNowTest = millis();
        }
      }
    }

//////////////////
// GET COMMANDS //
//////////////////

void GetCommands(){
  if(Serial.available() > 0){
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9'){
      // accumulate the value and put accumlated values in
         array
      value = (value*10) + (ch - '0');
    }
    else{
      switch (ch) {

        //// THROTTLE ////
        case 't': // set throttle position
          if(value >= throttle_open && value <=
             throttle_closed){
             throttlePos = value;
             throttle.writeMicroseconds(throttlePos);
          }
             break;
        case 'z': // throttle position to closed (kill engine
          )
          throttle.writeMicroseconds(throttle_closed);
          throttlePos = throttle_closed;
          break;
        case 'x': // throttle position to fully open
          throttle.writeMicroseconds(throttle_open);
          throttlePos = throttle_open;
          break;
        case 'a': // throttle to idle position
          throttle.writeMicroseconds(throttle_idle);
          throttlePos = throttle_idle;
          break;
         case 'q': // open throttle by 10
          if(throttlePos >= throttle_open + 10){
             throttlePos -= 10;
             throttle.writeMicroseconds(throttlePos);
          }
          break;
        case 'w': // close throttle by 10
          if(throttlePos <= throttle_closed - 10){
             throttlePos += 10;
             throttle.writeMicroseconds(throttlePos);
          }
          break;
         case 'Q': // open throttle by 100
          if(throttlePos >= throttle_open + 100){
             throttlePos -= 100;
             throttle.writeMicroseconds(throttlePos);
          }
          break;
        case 'W': // close throttle by 100
          if(throttlePos <= throttle_closed - 100){
             throttlePos += 100;
             throttle.writeMicroseconds(throttlePos);
          }
          break;


        //// SETPOINT SPEED ////
        case 's': // set setpoint speed value
          if(value >= 0 && value <= max_RPM){
              setpointSpeed = (float) value;
          }
          break;
        case 'b': // increase setpoint speed by 100
          if(setpointSpeed <= max_RPM - 100.0)
             setpointSpeed += 100.0;
          break;
        case 'n': // decrease setpoint speed by 100
          if(setpointSpeed >= 100.0)
             setpointSpeed -= 100.0;
          break;
        case 'B': // increase setpoint speed by 1000
          if(setpointSpeed <= max_RPM - 1000.0)
             setpointSpeed += 1000.0;
          break;
        case 'N': // decrease setpoint speed by 1000
          if(setpointSpeed >= 1000.0)
             setpointSpeed -= 1000.0;
          break;

        //// TUNING PARAMS ////
        case 'j': // increase prop. gain by 0.001
          Kp += 0.001;
          SetTunings(Kp, Ki, Kd);
```

```
        break;
    case 'k': // decrease prop. gain by 0.001
        if(Kp >= 0.001){
            Kp -= 0.001;
            SetTunings(Kp, Ki, Kd);
        }
        break;
    case 'J': // increase prop. gain by 0.01
        Kp += 0.01;
        SetTunings(Kp, Ki, Kd);
        break;
    case 'K': // decrease prop. gain by 0.01
        if(Kp >= 0.01){
            Kp -= 0.01;
            SetTunings(Kp, Ki, Kd);
        }
        break;
    case 'y': // increase integral gain by 0.0000001
        Ki += 0.000001;
        SetTunings(Kp, Ki, Kd);
        break;
    case 'u': // decrease integral gain by 0.0000001
        if(Ki >= 0.000001){
            Ki -= 0.000001;
            SetTunings(Kp, Ki, Kd);
        }
        break;
    case 'Y': // increase integral gain by 0.00001
        Ki += 0.00001;
        SetTunings(Kp, Ki, Kd);
        value = 0;
        break;
    case 'U': // decrease integral gain by 0.00001
        if(Ki >= 0.00001){
        Ki -= 0.00001;
            SetTunings(Kp, Ki, Kd);
        }
        break;
    case 'g': // increase derivative gain by 0.01
        Kd += 0.01;
        SetTunings(Kp, Ki, Kd);
        break;
    case 'h': // decrease derivative gain by 0.01
        if(Kd >= 0.01){
            Kd -= 0.01;
            SetTunings(Kp, Ki, Kd);
        }
        break;
```

```
                case 'G': // increase derivative gain by 0.1
                    Kd += 0.1;
                    SetTunings(Kp, Ki, Kd);
                    break;
                case 'H': // decrease derivative gain by 0.1
                    if(Kd >= 0.1){
                        Kd -= 0.1;
                        SetTunings(Kp, Ki, Kd);
                    }
                    break;
                case 'd': // increase setpoint weight
                    setpointWeight += 0.05;
                    break;
                case 'D': // decrease derivative gain by 0.0000001
                    if(setpointWeight >= 0.05)
                        setpointWeight -= 0.05;
                break;
                case '(': // increase alpha
                    alpha += 0.05;
                    break;
                case ')': // decrease alpha
                    if(alpha >= 0.05)
                        alpha -= 0.05;
                break;

                //// BOOLEANS ////
                case 'm': // enable or disable auto mode
                    manual = !manual;
                    if(!manual){
                        bumplessTransfer();

                    }
                    break;
                case 'o': // initiate or terminate test routine
                    testRoutine = !testRoutine;
                    break;
                case 'O': // reset testIteration
                    testIteration = 0;
                    break;
                case 'r': // toggle auxillary heating on/off
                    heating = !heating;
                    break;
            }

            value = 0;
        }
    }
}
```

```
/////////////////
// REPORT DATA //
/////////////////

void ReportData(){

    // send control mode state manual/auto
    Serial.print(manual);
    Serial.println("!");
    _delay_us(100);

    // send test routine state on/off
    Serial.print(testRoutine);
    Serial.println("&");
    _delay_us(100);

     // send heating state on/off
     Serial.print(heating);
     Serial.println("R");
     _delay_us(100);

    // send current time
    Serial.print(Time);
    Serial.println("T");
    _delay_us(100);

    // send throttle position
    Serial.print(throttlePos);
    Serial.println("F");
    _delay_us(100);

    // send pressure
    Serial.print(pressure*10.0,0);
    Serial.println("P");
    _delay_us(100);

    // send temperature
    Serial.print(temperature,0);
    Serial.println("C");
    _delay_us(100);

    // send Kp
    Serial.print(Kp*1000.0,0);
    Serial.println("{");
    _delay_us(100);

    // send Ki
    Serial.print(Ki*1000000.0,0);
    Serial.println("}");
    _delay_us(100);

    // send Kd
    Serial.print(Kd*1000.0,0);
    Serial.println("[");
    _delay_us(100);

    // send setpoint speed
    Serial.print(setpointSpeed,0);
    Serial.println("S");
    _delay_us(100);

    // send lowpass speed
    Serial.print(lowpassSpeed*100.0,0);
    Serial.println("#");
    _delay_us(100);

    // send measure speed
    // always send last because it tells
    // MC2 when it is time to print values
    // to serial monitor.
    Serial.print(measuredSpeed*100.0,0);
    Serial.println("V");
    _delay_us(100);
    _delay_ms(5);
}

////////////////////////////////////////////////
///////      HOWARD HUGHES     ////////////////
////// / / / / | / / /| |  / / /////////////////
///// / / / /  |/ / / | | / / //////////////////
//// / /-/ / /|  / /--| |/ / ///////////////////
///  \----/-/ |-/-----/---/ //// PAUL FJARE ////
//  COLLEGE OF ENGINEERING ///// MC2 CODE /////
////////////////////////////////////////////////

////////////////
// LIBRARIES  //
////////////////
#include <stdlib.h> // C Standard General Utilities Library
#include <util/delay.h> // AVR delay functions
#include <LiquidCrystal.h> // LCD Library
// Create LCD library object
LiquidCrystal lcd(13, 12, 8, 9, 10, 11); // lcd(rs, e, D4, D5, D6
    , D7)
```

```
///////////////////////////////
// ENGINE UNIQUE PARAMETERS //
///////////////////////////////

// Maximum realistic RPM
const long max_RPM = 9000;
// Minimum realistic RPM
const long min_RPM = 1;

//////// SERVO POSITION CONSTANTS ////////
// Position of throttle servo at closed throttle
const int throttle_closed = 1810;
// Position of throttle servo at open throttle
const int throttle_open = 1300;
// Position of throttle servo at idle
const int throttle_idle= 1746;
// Position of throttle servo at minimum throttle
// where engine will not stall.
const int throttle_safe = 1756;

//////////
// PINS //
//////////
// Potentiometer knob controlling throttle
int throttleKnobPin = 8;
// Potentiometer knob controlling throttle
int fuelKnobPin = 9;

////////////
// TIMING //
////////////
long now;
long lastNow;
long Time;

////////////
// VALUES //
////////////
// Temporary storage for incoming data
long value;

double temperature; // Celcius
double pressure; // inHg
double measuredSpeed, lastMeasuredSpeed, lowpassSpeed;  // RPM
double setpointSpeed; // RPM
int fuelPos, lastFuelPos;

// Control Algorithm Parameters
```

```
double Kp, Ki, Kd;
double K1, K2, K3, filteredK3;
double error, lastError, lastLastError;
double output;

//////////////
// BOOLEANS //
//////////////

// Manual of Automatic (PI)
boolean manual;
// Start or stop test routine
boolean testRoutine;
boolean heating;

///////////
// SETUP //
///////////

void setup(){
    Serial.begin(9600); // PC serial
    Serial1.begin(9600); // Wireless serial
    lcd.begin(20,4);
    lcd.display();
}

///////////////
// MAIN LOOP //
///////////////

void loop(){
    getCommands();
    getReportedData();

    now = millis();
    if((now-lastNow) >= 100){

        if(KnobControl){
            setThrottle();

        }
        updateLCD();

        lastNow = now;
    }
}

/////////////////
```

```
// GET COMMANDS //
//////////////////

void getCommands(){
   /*Get commands from serial monitor and relay these commands
   to other microcontroller.*/
   if(Serial.available() > 0){
      char ch = Serial.read();
      if(ch >= '0' && ch <= '9'){
         // accumulate the value and put accumlated values in
            array
         value = (value*10) + (ch - '0');
      }
      else{

         switch (ch) {

            case 't': // set throttle position  ex. 1600t
               if(value >= throttle_open && value <=
                  throttle_closed){
                  Serial1.print(value);
               }
               value = 0;
               break;

            case 's': // set setpoint speed value
               if(value >= 0 && value <= max_RPM){
                  Serial1.print(value);
               }
               value = 0;
               break;
         }

         Serial1.println(ch);
         value = 0;
      }


   }
}

//////////////
// GET DATA //
//////////////

void getReportedData(){
   if(Serial1.available() > 0){
      char ch = Serial1.read();
```

```
if(ch >= '0' && ch <= '9'){
    // Accumulate the value
    value = (value*10) + (ch - '0');
}
else{
    switch (ch) {
        case '!': // get current time
            manual = value;
            break;
        case '&': // get test routine state
            testRoutine = value;
            break;
        case 'R': // get heating state on/off
            heating = value;
            break;
        case 'T': // get Time
            Time = value;
            break;
        case 'F': // get throttle position
            throttlePos = value;
            break;

         case '#': // get lowpass speed
             lowpassSpeed = (double) value/100.0;
             break;
         case 'S': // get setpoint speed
             setpointSpeed = (double) value;
             break;
        case 'P': // get pressure
            pressure = (double) value/10.0;
            break;
        case 'C': // get temperature
            temperature = (double) value;
            break;
        case 'I':
            K1 = (double) value/100.0;
            break;
        case 'E':
            K2 = (double) value/100.0;
            break;
        case 'v':
            K3 = (double) value/100.0;
            break;
        case '{': // get Kp
            Kp = (double) value/1000.0;
            break;
        case '}': // get Ki
            Ki = (double) value/1000000.0;
```

```
                break;
            case '[': // get Kd
                Kd = (double) value/1000.0;
                break;
            case 'X': // get alpha
                alpha = (double) value/100.0;
            case 'M': // get setpoint weight
                setpointWeight = (double) value/100.0;
            case 'V': // get measured speed
                measuredSpeed = (double) value/100.0;
                if(measuredSpeed != lastMeasuredSpeed)
                  logData();
                break;
        }
        value = 0;
      }
    }
}


//////////////////////////
// DISPLAY DATA ON LCD //
//////////////////////////

void updateLCD(){

    /**
        ////////////////////// LCD MAP //////////////////
        0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
        -----------------------------------------------
      0| K i 0 . 0 0 0 0 0 0 - - - K p 0 . 0 0 0
      1| K d 0 . 0 0 0 - - - M A N - A U X - T R
      2| y 0 0 0 0 0 - r 0 0 0 0 0 - c o 0 0 0 0
      3| 0 0 0 * C - 0 0 . 0 i n H g - -
     */

    lcd.clear(); // clear all characters on screen

    /// LINE 0 ///

    lcd.setCursor(0,0); // lcd.setCursor(col,row)
    lcd.print("Ki");
    lcd.print(Ki,6);

    lcd.setCursor(13,0);
    lcd.print("Kp");
    lcd.print(Kp,3);
    /// LINE 1 ///
    lcd.setCursor(0,1);
    lcd.print("Kd");
    lcd.print(Kd,3);

    lcd.setCursor(10,1);
    if(!manual)
        lcd.print("AUTO");

    lcd.setCursor(14,1);
    if(heating)
        lcd.print("AUX");

    lcd.setCursor(18,1);
    if(testRoutine)
        lcd.print("TR");

    /// LINE 2 ///

    lcd.setCursor(0,2);
    lcd.print("PV");
    lcd.print(measuredSpeed,0);

    lcd.setCursor(8,2);
    lcd.print("SP");
    lcd.print(setpointSpeed,0);

    lcd.setCursor(15, 2);
    lcd.print("T");
    lcd.print(throttlePos);

    /// LINE 3 ///

    lcd.setCursor(0,3);
    lcd.print(temperature,0);
    lcd.print((char)223);
    lcd.print("C");

    lcd.setCursor(6,3);
    lcd.print(pressure,1);
    lcd.print("inHg");
}


//////////////
// LOG DATA //
//////////////
```

```
void logData(){
    Serial.print(Time);
    Serial.print(",");
    _delay_us(100);

    Serial.print(Kp,3);
    Serial.print(",");
    _delay_us(100);

    Serial.print(Ki,6);
    Serial.print(",");
    _delay_us(100);

    Serial.print(Kd,3);
    Serial.print(",");
    _delay_us(100);

    Serial.print(throttlePos);
    Serial.print(",");
    _delay_us(100);

    Serial.print(setpointSpeed);
    Serial.print(",");
    _delay_us(100);

    Serial.print(temperature);
    Serial.print(",");
    _delay_us(100);

    Serial.print(lowpassSpeed);
    Serial.print(",");
    _delay_us(100);

    Serial.print(measuredSpeed);
    Serial.print(",");
    _delay_us(100);

    Serial.println();
    lastMeasuredSpeed = measuredSpeed;
    _delay_ms(10);
}
```

References

[1] B. Sodoma, "Taking flight," UNLV Mag., vol. 22, pp. 22-24, Jun. 2014.

[2] K. Mollenhauer and H. Tschoeke, "History and Fundamental Principles of the Diesel Engine," in *Handbook of Diesel Engines*. Heidelberg, Berlin: Springer-Verlag, 2010.

[3] J. B. Heywood, "Engine fuel metering and manifold phenomena", in *IC Engines Fundamentals*, McGraw-Hill, 1998.

[4] (2014). *Propeller Propulsion* [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/propeller.html

[5] A. C. Roth, "Two-Cycle and Four-Cycle Engines," in *Small Gas Engines*. Goodheart-Willcox Co. , 2009.

[6] K. W. Stinson, "The Diesel Engine," in *Diesel Eng. Handbook*, 10th ed. Stamford, CT: Diesel Publications, Inc. , 1959.

[7] (2009). *Carburetor: mixing the gas vapor with air to make combustion* [Online]. Available: http://hdabob.com/the-vehicle/fuel-injection/carburetor/

[8] C. D. Gierke, "Glow plugs exposed," *Model Airplane News*, pp. 170-178, Aug. 2004.

[9] K. Astrom, R. Murray, "PID Control," in *Feedback Systems*, Princeton, NJ: Princeton Univ. Press, 2008.

[10] A. Visioli, *Practical PID Control*. London, UK: Springer-Verlag, 2006.

[11] *160FX* [Online]. Available: http://www.osengines.com/engines-airplane/osmg0660/index.html

[12] *Arduino Duemilanove* [Online]. Available: http://arduino.cc/en/Main/arduinoBoardDuemilanove

[13] *HS-125MG Slim-Wing* [Online]. Available: http://www.servocity.com/html/hs-125mg_slim-wing.html

[14] *Encoder for Pololu wheel* [Online]. http://www.pololu.com/product/1217

[15] B. G. Liptak, "Control Theory," in *Instrument Engineers' Handbook*, 4th ed., vol. 2, Boca Raton: CRC Press, 2003, ch. 2.

[16] P. Welender, "Understanding Derivative in PID Control," *Control Eng.*, Feb. 2010. [Online]. Available: http://www.controleng.com/search/search-single-display/understanding-derivative-in-pid-control/4ea87c406e.html

PAUL FJARE

(702) 373-1188
pfjare@gmail.com
8221 Green Clover Ave. Las Vegas, NV 89149
linkedin.com/in/paulfjare
Nevada EI #0T6809

EDUCATION

Master of Science, Mechanical Engineering *Expected 2014*
University of Nevada, Las Vegas
GPA - 3.68

Bachelor of Science, Mechanical Engineering *2012*
University of Nevada, Las Vegas
Major GPA - 3.36

SKILLS

• Revit MEP • Solidworks • Excel/Word/Powerpoint
• Matlab • Comsol • Python • Hypermesh • Engineering Equation Solver (EES)

TECHNICAL PROJECTS

Solar Decathlon *2012-2013*

- Worked with a talented team of UNLV students, professors, and industry professionals to build an ultra-efficient solar powered home that was one of 20 entries around the world in the United States Department of Energy Solar Decathlon 2013.
- Designed the domestic water supply and sanitary plumbing systems.
- Modeled the plumbing systems in Revit MEP.
- Contributed to the creation of graphics for the team's website.
- Team Las Vegas placed 2nd overall and 1st in the nation.

Senior Design *2011*

- Worked in a team with three other students to design and build a prototype of a window cleaner for office buildings that utilized a dual winch positioning system.
- Designed the control system for the prototype's DC motors.
- Managed the team's progress.
- Awarded 2nd Place Mechanical Engineering in the annual Senior Design competition at the UNLV College of Engineering.

ORGANIZATIONS

American Society of Mechanical Engineers - Member
Society of Asian Scientists and Engineers UNLV Chapter - Director of Marketing *2013*