

ONLINE LEARNING OF A NEURAL FUEL CONTROL
SYSTEM FOR GASEOUS FUELED SI ENGINES

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Mechanical Engineering
University of Saskatchewan
Saskatoon

By
Travis Kent Wiens

©Travis Kent Wiens, September 2008.

PERMISSION TO USE

In presenting this thesis/dissertation in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis/dissertation in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis/dissertation work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis/dissertation or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis/dissertation.

Reference in this thesis/dissertation to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favouring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Mechanical Engineering
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
Canada
S7N 5A9

In addition to the above terms, permission to reproduce this work is also granted by the author under the Creative Commons Attribution-Noncommercial-No Derivative Works 2.5 Canada License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.5/ca/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

ABSTRACT

This dissertation presents a new type of fuel control algorithm for gaseous fuelled vehicles. Gaseous fuels such as hydrogen and natural gas have been shown to be less polluting than liquid fuels such as gasoline, both at the tailpipe and on a total cycle basis. Unfortunately, it can be expensive to convert vehicles to gaseous fuels, partially due to small production runs for these vehicles. One of major development costs for a new vehicle is the development and calibration of the fuel controller. The research presented here includes a fuel controller which does not require an expensive calibration phase.

The controller is based upon a two-part model, separating steady state and dynamic effects. This model is then used to estimate the optimum fuelling for the measured operating condition. The steady state model is calculated using an artificial neural network with an online learning scheme, allowing the model to continually update to improve the controller's performance. This is important during both the initial learning of the characteristics of a new engine, as well as tracking changes due to wear or damage.

The dynamic model of the system is concerned with the significant transport delay between the time the fuel is injected and when the exhaust gas oxygen sensor makes the reading. One significant result of this research is the realization that a previous commonly used model for this delay has become significantly less accurate due to the shift from carburetors or central point injection to port injection.

In addition to a description of the control scheme used, this dissertation includes a new method of algebraically inverting a neural network, avoiding computationally expensive iterative methods of optimizing the model. This can greatly speed up the control loop (or allow for less expensive, slower hardware).

An important feature of a fuel control scheme is that it produces a small, stable limit cycle between rich and lean fuel-air mixtures. This dissertation expands the currently available models for the limit cycle characteristics of a system with a linear controller as well as developing a similar model for the neural network controller by linearizing the learning scheme.

One of the most important aspects of this research is an experimental test, in which the controller was installed on a truck fuelled by natural gas. The tailpipe emissions of the truck with the new controller showed better results than the OEM controller on both carbon monoxide and nitrogen oxides, and the controller required no calibration and very little information about the properties of the engine.

The significant original contributions resulting from this research include:

- collection and summarization of previous work,
- development of a method of automatically determining the pure time delay between the fuel injection event and the feedback measurement,
- development of a more accurate model for the variability of the transport delay in modern port injection engines,

- developing a fuel-air controller requiring minimal knowledge of the engine's parameters,
- development of a method of algebraically inverting a neural network which is much faster than previous iterative methods,
- demonstrating how to initialize the neural model by taking advantage of some important characteristics of the system,
- expansion of the models available for the limit cycle produced by a system with a binary sensor and delay to include integral controllers with asymmetrical gains,
- development of a limit cycle model for the new neural controller, and
- experimental verification of the controller's tailpipe emissions performance, which compares favourably to the OEM controller.

ACKNOWLEDGEMENTS

This dissertation would not be possible without the support of a large number of people.

Certain people at the U of S provided invaluable support, including Professors Schoenau and Burton and my supervisory committee, as well as Sherri Haberman, Janai Simonson, April Wettig, Kelley Neale and Doug Bitner.

Thanks are due to the Saskatchewan Research Council for providing equipment and expertise, including Mike Sulatisky, Sheldon Hill, Bryan Lung, Joe Lychak, Tyler Leepart, Joe Jones, Marcy Green, Kim Young, Nathan Peter, Ken Babich, and Brett Smith. Thanks are also given to Natural Resources Canada and Precarn Inc. who funded SRC's initial work in this area.

The author would like to acknowledge the financial support of NSERC through a PGS D scholarship and also via Professors Burton and Schoenau's NSERC Discovery Grants. Financial support was also provided by the Department of Mechanical Engineering.

General Motors Alternative Fuels also made this research possible through the loan of the test vehicle as well as providing technical expertise.

Finally, this research was dependant upon the support of my family and friends, in particular Reuben Wiens, Sharon Wiens and Amy Templeman.

CONTENTS

Permission to Use	i
Abstract	ii
Acknowledgements	iv
Contents	v
List of Tables	vii
List of Figures	viii
List of Abbreviations	x
1 Introduction and Objectives	1
1.1 Introduction	1
1.2 Objectives	2
1.3 Layout of Dissertation	2
1.4 Contributions	3
2 Background	4
2.1 The Spark Ignition (SI) engine	4
2.1.1 Stoichiometry	6
2.1.2 Sensors	9
2.1.3 Current Control Methods	11
2.2 Artificial Neural Networks	12
3 A Survey of Literature Applicable to Online Learning for Transient Control of Fuel-Air Ratio in SI Engines Powered by Gaseous Fuels.	15
3.1 Objectives	15
3.2 Approaches	15
3.3 Results	15
3.4 Contributions	16
4 Experimental Determination of Transport Delay in a Spark Ignition Engine	32
4.1 Objectives	32
4.2 Approaches	32
4.3 Results	33
4.4 Contributions	34
5 Intelligent Fuel Air Ratio Control of Gaseous Fuel SI Engines	42
5.1 Objectives	42
5.2 Approaches	42
5.3 Results	45
5.4 Contributions	45
5.5 Erratum	46

6 Algebraic Inversion of an Artificial Neural Network Classifier	54
6.1 Objectives	54
6.2 Approaches	54
6.3 Results	55
6.4 Contributions	55
7 Limit Cycle Behaviour of a Neural Controller with Delayed Bang-Bang Feedback	63
7.1 Objectives	63
7.2 Approaches	63
7.3 Results	63
7.4 Contributions	64
8 Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller	71
8.1 Objectives	71
8.2 Approaches	71
8.3 Results	72
8.4 Contributions	73
9 Conclusions, Contributions and Recommendations	81
References	83
A Nomenclature	84
B Benchmarking the Performance of Activation Functions in Generalized Neural Networks	86
C Computer Code	96
C.1 Simulation Code	96
C.1.1 main.c	96
C.1.2 model.h	99
C.1.3 model.c	100
C.1.4 controller.h	101
C.1.5 controller.c	102
C.1.6 training.h	106
C.1.7 training.c	107
C.2 Microcontroller Code	109
D Copyright Information	141
D.1 Limit Cycle Behaviour of a Neural Controller with Delayed Bang-Bang Feedback, Chapter 7	141
D.2 Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller, Chapter 8	141
D.3 Computer Code in Appendix C	142

LIST OF TABLES

8.1	Post-catalyst emissions concentrations over Hot 505 driving cycle.	73
-----	--	----

LIST OF FIGURES

2.1	Typical engine and control system, showing accelerator pedal (41), throttle (23), intake manifold (3), pressure transducer (46), fuel injector (21), intake port (4), intake valves (15), combustion cylinder (5), spark plug (14), piston (6), crank position sensor (33), exhaust valves (16), exhaust manifold (8), exhaust gas oxygen sensor (47) and catalytic converter (9). The engine shown is also fitted with an exhaust gas recirculation (EGR) valve (26), although it was disabled for this study.	5
2.2	Equilibrium mole fraction for NO and CO after combustion of iso-octane and air at 1750 K and 30 atmospheres, as a function of relative air-fuel ratio, λ . This approximates the engine-out concentrations of these pollutants, showing that CO is only produced when the mixture is rich ($\lambda < 1$), while NO has its highest production when the mixture is lean.	7
2.3	Graph of catalyst efficiency with respect to relative fuel-air ratio. The efficiency for unburned hydrocarbons generally follows that of CO.	8
2.4	Example exhaust gas oxygen (EGO) response curve, showing the strongly nonlinear behaviour around the stoichiometric point ($\lambda = 1$).	10
2.5	Fuel controller block diagram of a typical ECU (Engine Control Unit). Before the EGO reaches operating temperature, the “fuel map” controls the fuelling in open loop mode, calculating an injector pulse width, t_i , based on an operating point composed of engine speed, N_e , intake manifold pressure, P_m , and various other measurements such as temperatures.	11
2.6	A single neuronal unit calculates a weighted sum of its inputs (using weights W_1 through W_3) and applies a nonlinear activation function to the result.	12
2.7	A generalized neural network (GNN) for the application of interest. This network has four inputs (P_m , N_e , t_{inj} and a constant 1), three hidden neurons (labelled x_5 , x_6 and x_7) and one output neuron (y). Each line represents a weight (W_{ij}) connecting the inputs to the neurons as well as connecting the neurons to each other. A multi-layer perceptron may be converted to this form by setting certain weights to zero.	13
4.1	The truck used for experimental studies in this dissertation was a General Motors 2001 GM2500HD chassis with a 2003 Vortec 6.0L V8 bi-fuel (compressed natural gas or gasoline) engine.	33
4.2	The delay in units of injection events is shown here. This figure demonstrates that the near-constant delay predicted by Kaidantzis’ model does not occur in a modern port-injected engine.	34
5.1	The network used for the model, showing inputs of scaled engine speed, N_e , manifold pressure, P_m , injection pulse width, t_i and the output, which is a model of the steady state binary sensor measurement. The dashed weights are constrained to zero to allow for inversion of the network (see Chapter 6).	43
5.2	Compensating for delay may be achieved even if the exact delay isn’t known. For the example above, the delay is estimated to be some value between four and eight samples. The input point A may then correspond to any of outputs A_1 through A_5 and B may correspond to any of points B_1 through B_5 . Since the possible values to match to A are all the same (A_1 through A_5) it doesn’t matter which value is the true value and training can proceed. However since B_1 through B_5 are not constant, point B cannot be used for training, as is not possible to determine which value to use.	44
5.3	The engine used for experimental and simulation studies in this dissertation was a 2003 Vortec 6.0L V8 bi-fuel (compressed natural gas or gasoline) engine. This image shows the 2006 version of the gasoline version; the engine used is very similar. Used by permission of General Motors.	46

5.4	Simulated lambda for the entire test (top). The initial lambda has an error of approximately 0.3. However, after an hour the error is 0.02, which eventually converges to a steady state error of 3×10^{-4} . The limit cycle of the last 100 points (bottom) is desired as it is required for proper three-way catalytic converter operation.	47
8.1	Experimentally measured λ over the “hot 505” portion of the Federal Test Procedure driving cycle using natural gas.	72

LIST OF ABBREVIATIONS

CMAC	Cerebellar Model Articulation Controller
ECU	Engine Control Unit
EGO	Exhaust Gas Oxygen sensor
FA	Fuel-air Ratio
FAR	Fuel-air Ratio
GNN	Generalized Neural Network
HO2S	Hot Oxygen Sensor (aka EGO)
MLP	Multi-layer Perceptron
NOx	Nitrogen oxides
OEM	Original Equipment Manufacturer
RBF	Radial Basis Functions
RMSE	Root Mean Squared Error

CHAPTER 1

INTRODUCTION AND OBJECTIVES

1.1 Introduction

In recent years, atmospheric air quality has become a particularly important topic. This is due to two main concerns: ground level pollution and smog (particularly in urban areas), and greenhouse gas effects on global climate change. Tailpipe emissions from vehicles are significant contributors to both aspects, and there has been considerable interest in reducing vehicle engine tailpipe emissions, driven both by consumer and regulatory pressure.

One solution to these problems that has shown considerable potential is the use of gaseous fuels, such as natural gas and hydrogen. These fuels have proven to be less polluting, both at the tailpipe and on a total-cycle basis [1]. However, due to the low number of gaseous fuelled vehicles being offered for sale by the major automakers, the conversion of vehicles to gaseous fuels can be particularly cost prohibitive. One significant cost involved in converting a vehicle is the development of the fuel controller.

The development of the fuelling controller is a time consuming and expensive process, typically requiring highly skilled engineering labour and expensive equipment. The goal of the research presented here was to eliminate this step, by creating a general control scheme which would work with any gaseous fuelled spark ignition (SI) engine requiring only the most basic of information for initialization.

A controller was developed based on a two-part model of the engine: a pure time delay and an artificial neural network model of the static components of the engine and sensor. A method of generating an initial model adequate to start the engine was developed. This model is then updated, using online neural network learning, to improve controller performance while the vehicle is driven. The controller does not require any specialized training to initialize and does not require a dynamometer or gas analyzer to calibrate. Furthermore, it uses the sensors available on any relatively modern vehicle, further reducing the cost of conversion.

A number of theoretical simulation tests was performed and the controller's performance was verified on a real-world test vehicle: a General Motors 2500HD truck. The drivability and emissions performance of the controller compared favourably to the truck's factory controller.

In addition to the development of this specific controller, a number of other more general notable outcomes resulted from the course of this work. It was discovered that the classic model for the transport delay

between fuel injection and the oxygen sensor has a number of assumptions that are no longer applicable to modern vehicles and a more detailed model must be used. The author also discovered an analytical method of inverting a neural network classifier, eliminating the computationally expensive iterative methods. Finally, by linearizing the network, it was shown that it is possible to analytically predict the dynamic limit-cycle characteristics of the controller, which previously required numerical simulation.

1.2 Objectives

Beginning in the 1950s, in response to increasing haze and smog in urban areas (particularly Los Angeles and London), atmospheric and ground level air pollution became important to citizens, governments and industry, particularly in the case of automotive emissions. The tailpipe emissions of on-road vehicles are a considerable contributor to both noxious chemicals (carbon monoxide and oxides of nitrogen) and greenhouse gases (carbon dioxide). Considerable advances have been made in the area of light vehicle emission technology, such as the introduction of unleaded gasoline, electronic fuel control systems and general fuel economy improvements. However, due to the continuing increase in the number and size of vehicles on the roads of the world, automobiles must continue to become cleaner.

One potential source of considerable improvement would be the use of gaseous fuels, such as natural gas and hydrogen. These fuels have been shown to be less polluting than liquid petroleum fuels [1] and, in the case of hydrogen, may be generated from renewable energy sources. However, in order for the use of gaseous vehicle fuel to become widespread, the increased cost of buying or converting a gaseous fuelled vehicle must be low enough to be attractive to consumers. Therefore the general motivation for this research was to help improve atmospheric air quality by reducing the cost to convert vehicles to gaseous fuels.

One of the sponsors of this work, the Saskatchewan Research Council, had identified the cost of generating and, particularly, calibrating fuel control algorithms as a potential source of cost reduction. Generating the data required to fuel a modern engine is time consuming and expensive, as it requires a large amount of time (on the order of days) with expensive equipment, such as a dynamometer and gas analyzer. It was suggested that a method of automatically generating and analyzing this information online would significantly reduce the cost of vehicle conversions.

With this in mind, the specific goal of this research was to create a method of automatically calibrating a gaseous fuel controller which would work on a wide variety of SI engines with only minimal initial information required.

1.3 Layout of Dissertation

Since much of the information to be presented in this dissertation has already been published in papers and technical reports, the main body will be brief, relying on the included papers for the bulk of the information.

Since this research programme includes the two generally unrelated topics of automotive engine technology and artificial neural networks, this dissertation begins with a short introduction to both topics to bring the reader up to speed. Next, the papers are presented. Each paper is prefaced by a short summary including objectives, approaches to the problem, results and the significant contributions contained in the paper.

The first paper is a literature review of the state of the art of neural networks and engine control technology. Since this paper was prepared in 2006 it should not be viewed as complete and the remaining papers include research discovered later. The next report describes a method of determining the elapsed time between injection of fuel and the registration of those data by the exhaust sensor. This report also shows that a model previously used for this delay should be used with caution. Following this is a report describing the proposed controller. The next two papers include some further analysis of the controller, including the neural network inversion scheme and a derivation of a model of the limit cycle of the system. The final paper verifies the controller performance with an experimental emissions study.

The appendix includes a list of mathematical symbols in Appendix A, a study of the computational speed of various activation functions in Appendix B, a listing of the computer code used in Appendix C and an additional copyright section covering the reproduction of the papers, in Appendix D

One note on terminology is important. Since this dissertation deals with both gasoline and gaseous substances, to avoid confusion, the word “gas” will not be used to refer to gasoline.

1.4 Contributions

All papers contained in this manuscript have co-authors; however, it is the mutual understanding of the authors that Travis Wiens, as first author, is the primary investigator of this research. The contribution of the other co-authors has been limited to an advisory and editorial capacity.

CHAPTER 2

BACKGROUND

Before presenting the papers and reports that make up the bulk of the content of this dissertation, some background information must be presented. The papers assume an audience which is familiar with the subject matter and begin at quite a high level. Since there are currently few readers of this dissertation who can be expected to be knowledgeable in both neural networks and engine technology, this background will start at a relatively basic level and quickly accelerate to more advanced concepts.

2.1 The Spark Ignition (SI) engine

The internal combustion engine (ICE) has been one of the most important shapers of the twentieth century. Since the late 1800s, when the first practical engines were invented by familiar names like Nikolaus Otto, Gottlieb Daimler, Wilhelm Maybach, Karl Benz, and Rudolf Diesel, the internal combustion engine has revolutionized transportation, leading to fast and inexpensive travel of passengers and freight over the ground, sea and air.

In general terms, a spark ignition (SI) engine is a device for converting chemical energy to rotational mechanical energy. It achieves this by combusting a fuel with oxygen to apply a force to a piston. This piston is connected to a crank, which converts the linear motion of the piston to rotational motion of an output shaft.

Typical modern ground vehicle engines are four-stroke spark ignition engines or four-stroke compression ignition engines. Compression ignition engines burn diesel fuel, which is ignited by the elevated pressures in high compression ratio engines. These engines are outside the scope of this thesis which is concerned with SI engines.

In a modern four-stroke, fuel injected SI engine, the fuel is injected by the fuel injector (labelled 21 in Figure 2.1) into the intake manifold (3) or intake port (4). Fuel injectors are solenoid operated valves which, when energized, allow pressurized fuel to flow into the intake air stream. The mass airflow into the manifold is regulated by the throttle plate (23), which is a variable restriction in the intake air stream. The fuel and air mix in the intake manifold to form what is known as the charge mixture. During the intake stroke of the piston (6), the intake valves (15) open and the piston moves downward, drawing the charge mixture into the cylinder. Near the bottom of the piston's stroke, the intake valves close. As the piston moves upward in the

sealed cylinder, the charge mixture is compressed.

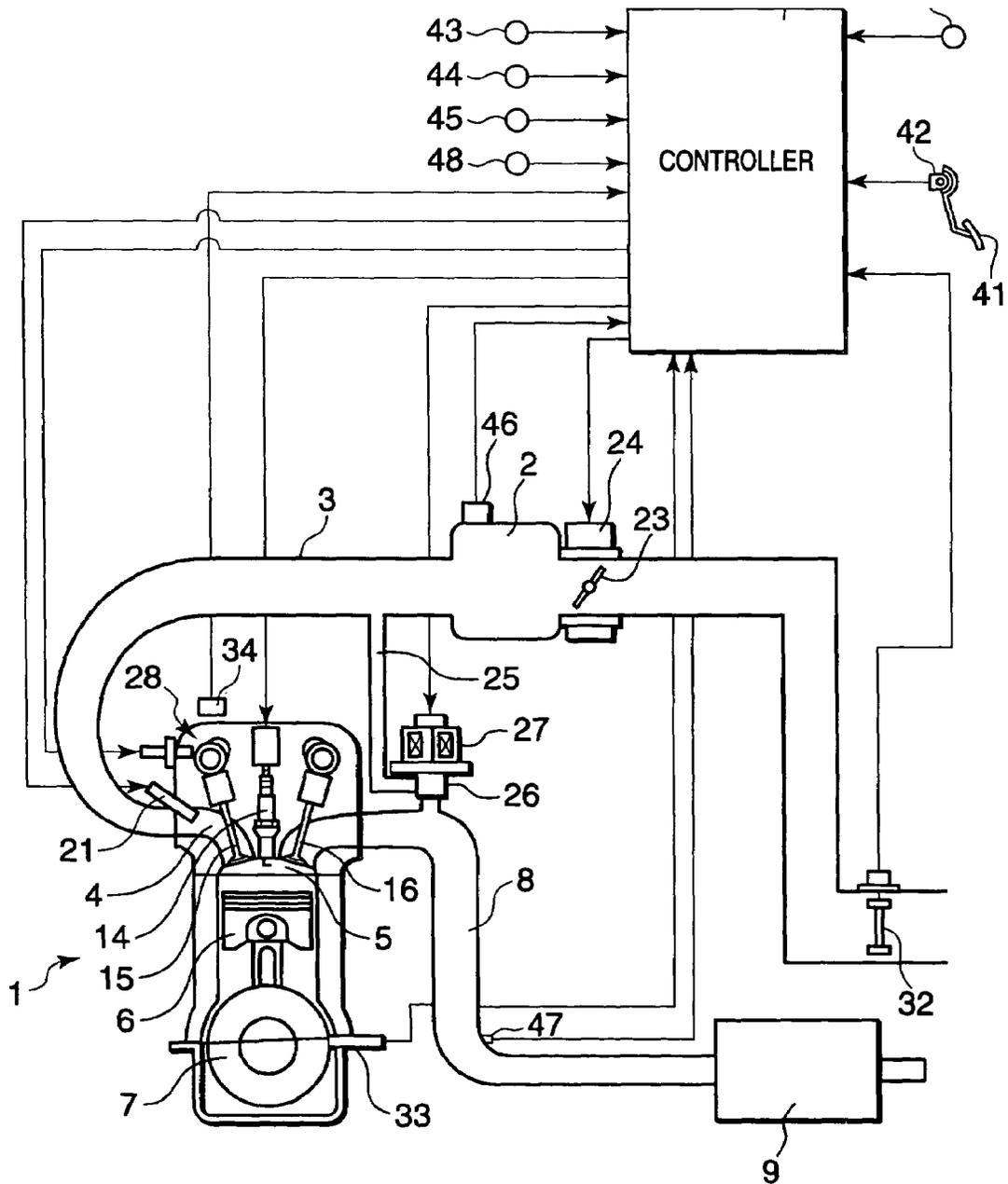


Figure 2.1: Typical engine and control system, showing accelerator pedal (41), throttle (23), intake manifold (3), pressure transducer (46), fuel injector (21), intake port (4), intake valves (15), combustion cylinder (15), spark plug (14), piston (6), crank position sensor (33), exhaust valves (16), exhaust manifold (8), exhaust gas oxygen sensor (47) and catalytic converter (9). The engine shown is also fitted with an exhaust gas recirculation (EGR) valve (26), although it was disabled for this study. (Image source: US Pat 699068 [2].)

Near the top of the compression stroke of the piston, the spark plug (14) fires, igniting the mixture, provided that the ratio of fuel to air is within the explosive range. The rapidly burning mixture expands, forcing the piston back down again. This stroke is known as the expansion or power stroke. Near the bottom

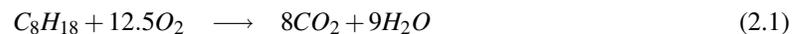
of the piston's movement, the exhaust valves (16) open. As the piston moves back up again, the exhaust gases are forced out of the cylinder through the exhaust system (8).

These four strokes: intake, compression, expansion and exhaust, make up the four-stroke cycle, also known as the Otto cycle, which power most gasoline-fuelled light vehicles in North America. Note that the four-stroke engine repeats the cycle every two rotations of the crankshaft, so the frequency of injections is half the speed of the engine. Also, it is important to notice that power is only produced during the expansion stroke, meaning that for approximately three quarters of the time, the cylinder is consuming energy that must either be provided by the power strokes of other cylinders, or be stored in a flywheel.

Although this dissertation is only concerned with the four-stroke Otto cycle, there are other schemes used for reciprocating engines. A modification to the four-stroke cycle is the two-stroke spark ignited engine. This engine uses the last part of the expansion stroke and the first part of the compression stroke for intake and exhaust, allowing for one power stroke per engine revolution. This allows for a higher power to weight ratio, although at a lower efficiency. These engines are commonly used for small high-performance engines such as those in racing motorcycles or model aircraft. Another common engine is the four-stroke compression ignition engine. This engine, commonly called a Diesel engine, does not have spark plugs. Instead, the higher pressure in the high compression ratio engine ignites the charge mixture. This engine type can have a greater thermal efficiency due to its greater compression ratio, although at a greater cost.

2.1.1 Stoichiometry

An important factor that affects the performance of an engine is the ratio of fuel to air in the charge mixture. If there is just enough oxygen in the air to totally combust the fuel, the ratio of fuel to air is called stoichiometric. The stoichiometric chemical equations for combustion of iso-octane (similar to gasoline), methane (an important component of natural gas) and hydrogen gas are

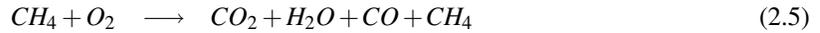
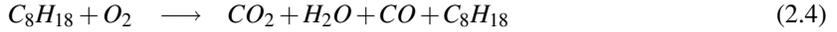


The change in enthalpy for each of these reactions is -5065 MJ/kmol for iso-octane, -802 MJ/kmol for methane, and -241 MJ/kmol for hydrogen [3]. If one calculates the amount of carbon dioxide released for a given energy output, one can see a reason to select gaseous fuels: 1.58 mol/MJ for isooctane, 1.24 mol/MJ for methane, and 0 for hydrogen.

The stoichiometric fuel-air mass ratios are 14.6 for gasoline, 14.5 for natural gas, and 34.3 for hydrogen [3], although the actual value for natural gas may vary due to the composition of the mixture.

If one deviates from stoichiometry, pollutants are formed. If there is too much fuel for the air (defined as a "rich" mixture), the exhaust from the engine will contain carbon monoxide (for fuels that contain carbon)

and unburned fuel. The unbalanced equations are



Notice that hydrogen gas does not produce either the greenhouse gas carbon dioxide, or the toxin carbon monoxide.

Conversely, if there is too little fuel in the mixture (defined as a “lean” mixture), at high temperatures the excess oxygen can combine with the normally inert nitrogen in the air to form nitrogen oxides (NO_x), which are toxic pollutants:

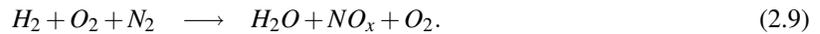
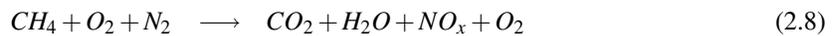
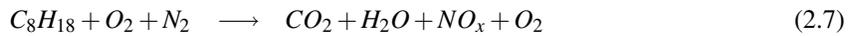


Figure 2.2 shows a graph of the various gases produced at equilibrium by the combustion of iso-octane and air at 1750 K and 30 atmospheres.

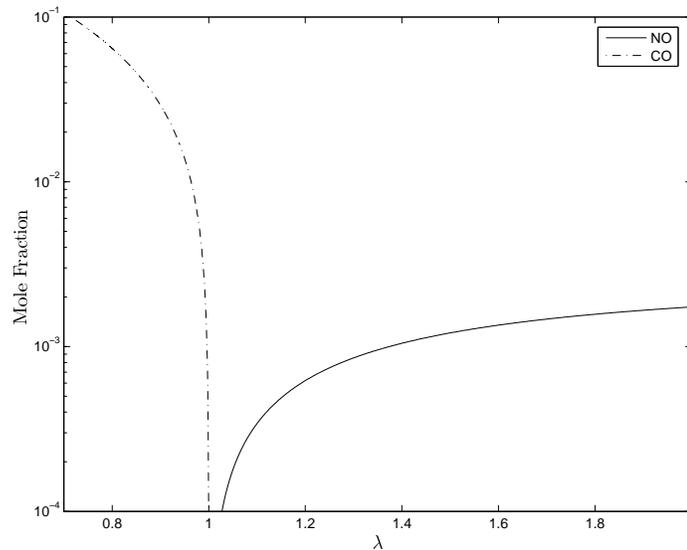


Figure 2.2: Equilibrium mole fraction for NO and CO after combustion of iso-octane and air at 1750K and 30 atmospheres, as a function of relative air-fuel ratio, λ [3]. This approximates the engine-out concentrations of these pollutants, showing that CO is only produced when the mixture is rich ($\lambda < 1$), while NO has its highest production when the mixture is lean.

These engine-out emissions do not, however, necessarily reach the atmosphere. A number of exhaust treatments may be applied to reduce their concentration. By far, the most common is the three-way catalytic

converter. This converter has three functions:

- reduce NO_x into oxygen and nitrogen gas,
- oxidize carbon monoxide into carbon dioxide, and
- oxidize unburned fuel into carbon dioxide and water vapour.

The conversion efficiency of each pollutant is strongly dependent on the relative air-fuel ratio:

$$\lambda = \frac{AF}{AF_s} \quad (2.10)$$

where AF is the air-fuel mass ratio and AF_s is the stoichiometric ratio for the fuel. The effect of λ on conversion efficiency is shown in Figure 2.3. The converter is very efficient at reduction of NO_x when the fuel air ratio is rich ($\lambda < 1$) and is more efficient at oxidation of CO and fuel when the mixture is lean (excess oxygen available in the exhaust steam). However, as mentioned earlier and shown in Figure 2.2, the production of NO_x occurs when the mixture is lean and unburned hydrocarbons and carbon monoxide occur when the mixture is rich. This means that there is no value for λ when there is corresponding production and conversion of a pollutant.

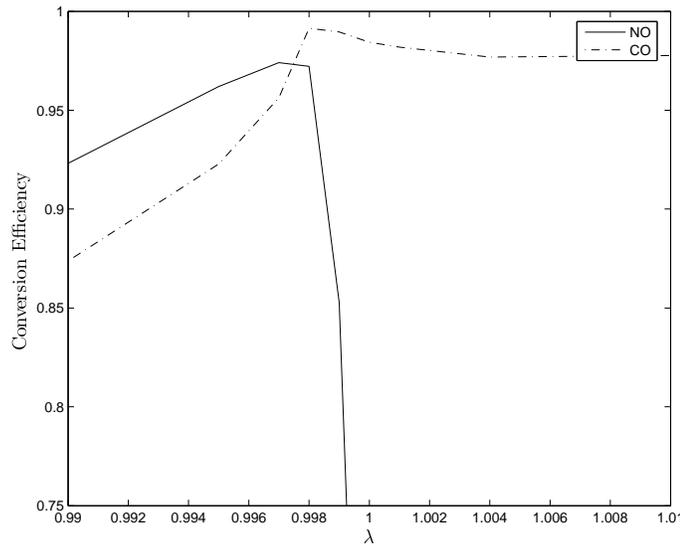


Figure 2.3: Graph of catalyst efficiency with respect to relative fuel-air ratio [4]. The efficiency for unburned hydrocarbons generally follows that of CO.

This problem is overcome by the oxygen storage capacity of the catalytic converter. During times when the mixture is lean and there is oxygen available in the exhaust stream, the catalyst is able to absorb and store oxygen. This oxygen is then made available during rich periods to oxidize the pollutants that occur then. This means that if the fuel air ratio has fast, small oscillations about the stoichiometric ratio, high conversion efficiency of all three pollutants may be achieved.

Due to the requirement of stoichiometric operation for proper catalyst efficiency, this is the most common type of control scheme for SI engines. One common deviation from stoichiometry is often employed when maximum power is requested from a liquid-fuel engine. If excess liquid fuel is injected into the intake manifold, its endothermic evaporation can cool the charge mixture. The cooler air is more dense and more oxygen can be inducted into the cylinder, meaning more fuel can be combusted and more power developed. This cooling effect can also reduce overheating of the engine. Since this effect does not occur with gaseous fuels, it is not considered further in this dissertation.

One tradeoff of stoichiometric operation is reduced fuel economy. It has been shown by Hill et al. [5] that significantly higher fuel economy may be achieved by providing a natural gas engine with a lean mixture (λ of approximately 1.4). This can be attributed to the reduced pumping pressure losses due to less throttle restriction (this is also the motivation for exhaust gas recirculation). However, it is very difficult to control the NOx emissions of a “lean burn” engine. Thus, toxic emissions of NOx are currently reduced at the cost of increased carbon dioxide (a greenhouse gas). Should priorities change in the future in favour of reduced greenhouse gases, lean burn engines may gain a larger following. In any case, in this dissertation only stoichiometric fuelling strategies are considered.

In order to estimate the fuel-air ratio for a given mass of fuel, one must estimate the mass of air being drawn into the engine. This is determined using the volumetric efficiency, which is the mass of fresh charge air inducted into the engine relative to the mass of air that would enter under static conditions at a standard temperature. One common model for the volumetric efficiency, η_v , is given in [3]:

$$\eta_v = \frac{P_m}{P_{atm}} \frac{M_{mix}}{M_a} \frac{T_c}{T_a} \left(1 + \frac{1}{AF_s}\right)^{-1} \left(\frac{r_c}{r_c - 1} - \left(\frac{P_e}{P_m} + \gamma - 1\right) \frac{1}{\gamma(r_c - 1)}\right) \quad (2.11)$$

where P_m is the absolute intake manifold pressure, P_{atm} is the atmospheric pressure, M_{mix} is the fuel-air mixture molecular mass, M_a is the molecular mass of air, T_c is the engine coolant absolute temperature, T_a is the absolute temperature of the intake air, AF_s is the stoichiometric mass air fuel ratio, r_c is the compression ratio, P_e is the exhaust manifold pressure and γ is the specific heat ratio of the fuel. This model is largely linear with intake manifold pressure. Engine speed makes only a small effect via the exhaust back pressure which increases with engine speed. It is important to note that this model does not take into account some important effects such as reflected pressure waves in a tuned intake or exhaust, nor does it model valve effects at high engine speeds.

2.1.2 Sensors

In order to maintain a desired fuel-air ratio in the engine to a high degree of accuracy, feedback is required. Unfortunately, no low-cost sensors are available which can measure the fuel-air ratio at the injection point in the intake. Instead, current sensors are placed in the exhaust stream. There are two commonly used sensors, termed exhaust gas oxygen (EGO) sensors and universal exhaust gas oxygen (UEGO) sensors.

EGO sensors are based upon a Nernst Cell which measures the concentration of oxygen in the exhaust

stream. The sensor is composed of a zirconia ceramic electrolyte which separates the exhaust stream from the atmosphere, along with metal electrodes (typically platinum). Oxygen is required to transport electrons across the cell so the electrical potential is related to the difference in oxygen partial pressures between the exhaust and atmosphere [3].

However, the response is very nonlinear, as shown in Figure 2.4. Since the oxygen partial pressure changes rapidly at the stoichiometric point, the sensor transitions very quickly from a low voltage to a high voltage, making it essentially a binary or “bang-bang” sensor. It provides one voltage for rich mixtures and another for lean mixtures, with very little information available regarding the degree of lean- or richness.

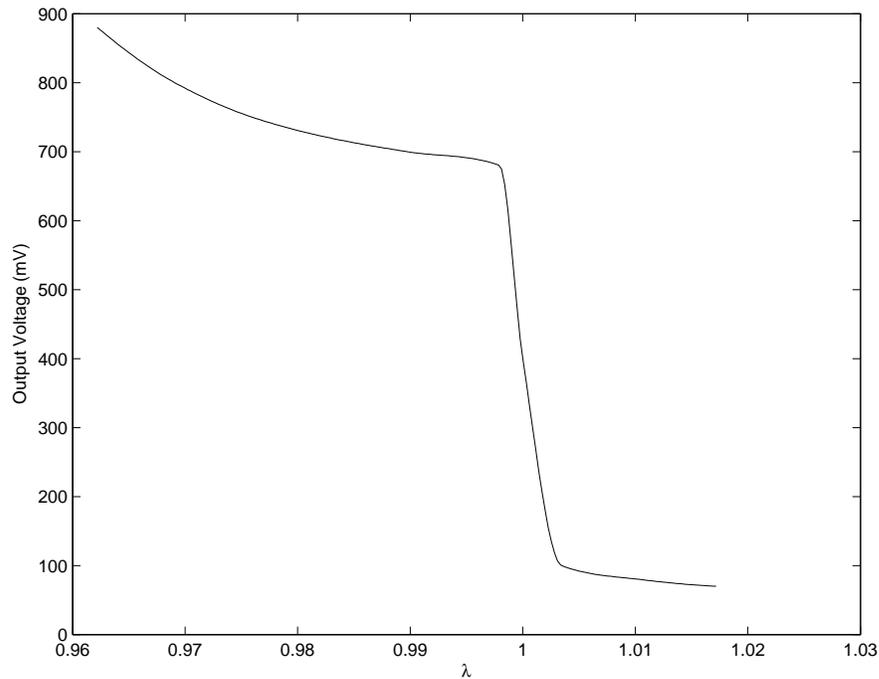


Figure 2.4: Example exhaust gas oxygen (EGO) response curve, showing the strongly nonlinear behaviour around the stoichiometric point ($\lambda = 1$).

The other type of sensor is the universal exhaust gas sensor, also known as a wideband oxygen sensor. This sensor uses a Nernst Cell similar to the EGO sensor, but includes a closed loop electrochemical pump which attempts to maintain a desired voltage across the Nernst Cell. By measuring the current required to maintain this equilibrium, a more linear response is obtained, which can be related to a value for the relative air-fuel ratio for a wide range of fuel-air ratios. These sensors are much more expensive than narrowband EGO sensors and are less commonly used on production vehicles.

In either case, the sensor introduces significant dynamics in a closed-loop system. The most important is due to the fact that the fuel is injected into the intake manifold, whereas the sensor is in the exhaust system. This introduces a pure time delay as the charge mixture travels from the injection point to the engine, resides in the cylinder for up to two engine rotations and then must travel from the engine to the sensor. A model

for this delay, t_d , is

$$t_d = \frac{60 * 2}{N_e} + \frac{\rho_i V_i}{\dot{m}_a} + \frac{\rho_{ex} V_{ex}}{\dot{m}_a} \quad (2.12)$$

where N_e is the engine speed (in RPM), ρ_i is the intake air density, ρ_{ex} is the exhaust density, \dot{m}_a is the mass charge mixture flow, V_i is the effective volume between the injector and the intake valves and V_{ex} is the effective volume between the exhaust valves and the oxygen sensor [6]. Further details may be found in Chapter 4, which shows that while a commonly used model predicts a constant number of injections will occur in this delay, modern port-injected engines have a variable delay.

2.1.3 Current Control Methods

Most modern fuel control schemes are a combination of feedforward and feedback control. During transient conditions, such as during acceleration and during cold starts before the EGO sensor reaches its operating temperature, the controller will be in open-loop mode, as shown in Figure 2.5 (although the shown PID controller is not yet active). In this mode, the controller will measure a number of sensors to determine the operating point. These measurements may include engine speed, intake manifold pressure, and various temperatures. It then consults a look-up table to determine the proper fuelling. Determining the values of this table can be very expensive, due to the need to optimize the fuelling at every operating point, which may take many days on expensive equipment.

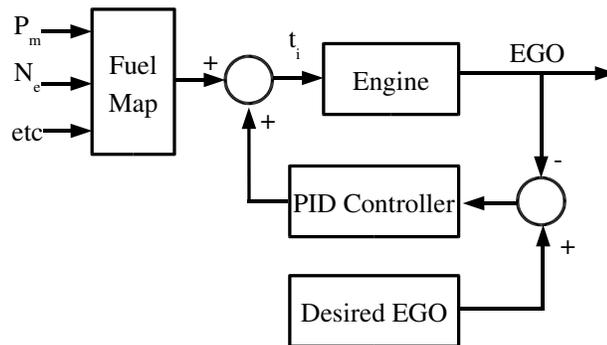


Figure 2.5: Fuel controller block diagram of a typical ECU (Engine Control Unit). Before the EGO reaches opening temperature, the “fuel map” controls the fuelling in open loop mode, calculating an injector pulse width, t_i , based on an operating point composed of engine speed, N_e , intake manifold pressure, P_m , and various other measurements such as temperatures.

During steady state and slow transients such as during idle, cruise or mild acceleration, the controller will be in closed loop mode (i.e. the PID controller is active). This mode will typically use a proportional, integral, and/or derivative (PID) linear compensator [7] to adjust the fuelling according to the measurement

of the EGO sensor, as shown in Figure 2.5. Often a base pulse width is estimated based on the operating point, similar to open loop mode, which is then modified by the PID compensator. Further details on the subject of fuel-control may be found in Chapter 3. In order to solve the problems of using a linear controller on a nonlinear system, one may use an artificial neural network.

2.2 Artificial Neural Networks

Artificial neural networks are parametrized functions which are commonly used for classification or function approximation. They are composed of a large number of very simple units, which can provide very complex ensemble behaviour. This section will introduce some history and background on neural networks. Further details may be found in the included papers, particularly Chapter 3.

Artificial neural networks were originally developed as a model of the brain, which is composed of large numbers of neurons connected in a network [8] [9]. While these models did not provide particularly good models of brain function, mathematicians and engineers quickly realized that they were still useful for practical problems of classification, pattern recognition, and function approximation. A typical artificial neuron, shown in Figure 2.6, takes a weighted sum of the inputs and applies a nonlinear activation function. These neurons are then connected in a network, the most general of which is the generalized neural network (GNN) of Werbos [10], shown in Figure 2.7. This network consists of a string of neurons, with inputs on the left and outputs on the right. Each neuron takes inputs from the outputs of each neuron to its left. It should be noted that the more common multi-layer perceptron (MLP) is a subset of the GNN.

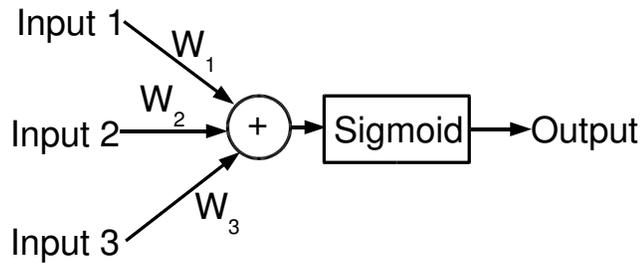


Figure 2.6: A single neuronal unit calculates a weighted sum of its inputs (using weights W_1 through W_3) and applies a nonlinear activation function to the result.

For an N neuron network with m inputs and 1 output, the first m neuron outputs are given by

$$x_i(k) = u_i(k) \quad (2.13)$$

for i equal to 1 to m , where u_i are the network inputs. The outputs of the remaining neurons are given by

$$x_i(k) = \sigma\left(\sum_{j=1}^{i-1} W_{ij}x_j(k)\right) \quad (2.14)$$

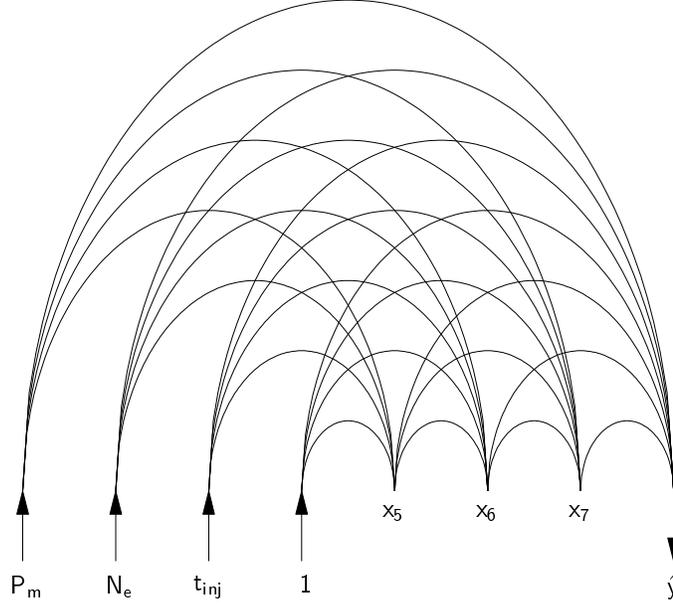


Figure 2.7: A generalized neural network (GNN) for the application of interest. This network has four inputs (P_m , N_e , t_{inj} and a constant 1), three hidden neurons (labelled x_5 , x_6 and x_7) and one output neuron (\hat{y}). Each line represents a weight (W_{ij}) connecting the inputs to the neurons as well as connecting the neurons to each other. A multi-layer perceptron may be converted to this form by setting certain weights to zero.

for i equal to $m + 1$ to N , where the matrix \mathbf{W} is the weight matrix and $\sigma(\cdot)$ is an activation function, typically a sigmoidal function such as the hyperbolic tangent, although better options exist, as shown in Appendix B. The network output is simply the output of the last neuron, x_N or \hat{y} .

The network must then be adjusted to provide the desired response. This is called learning or training and involves optimizing the \mathbf{W} matrix to give the desired response. The optimization is typically gradient descent (called backpropagation [10]) of the error although other schemes may be used such as the Marquardt algorithm [11] or the Nelder-Mead simplex method [12].

Backpropagation involves calculating the gradient of the squared error by propagating the error back through the network. If the activation functions are chosen carefully, the gradient is easier to calculate than the function. For example, if the hyperbolic activation function is used,

$$x = \sigma(z) = \tanh(z), \quad (2.15)$$

it can be shown that the derivative is

$$\frac{dx}{dz} = \text{sech}^2(z) = 1 - x^2, \quad (2.16)$$

which is much quicker to calculate than a hyperbolic function, as shown in Appendix B.

The backpropagation algorithm may be applied in two ways, called batch learning and online learning. In batch learning, a large number of training samples is presented to the network and the mean squared error

of the batch is minimized. In online learning, the weights are updated after each sample is presented (i.e. a batch of 1 sample). The two methods have advantages and disadvantages. Batch learning can converge faster due to the global nature of the algorithm, although certain online algorithms can theoretically match the performance of batch algorithms. The major advantages of online learning are its reduced memory requirements since only one sample must be stored at a time, and the fact that online learning algorithms seem to be less likely to get “stuck” in local optima [13], [14].

With this background in mind, the next chapter will present the first paper, a literature review.

CHAPTER 3

A SURVEY OF LITERATURE APPLICABLE TO ONLINE LEARNING FOR TRANSIENT CONTROL OF FUEL-AIR RATIO IN SI ENGINES POWERED BY GASEOUS FUELS.

Published as :

- Travis Wiens, Greg Schoenau, Rich Burton, “A Survey of Literature Applicable to Online Learning for Transient Control of Fuel-Air Ratio in SI Engines Powered by Gaseous Fuels.” Proceedings of CSME Forum 2006, Kananaskis, 2006.

3.1 Objectives

This paper presents a survey of the literature in the fields of both neural networks and also engine control systems, current to 2006. As the literature review is an ongoing process, this should be viewed as a base of literature to build upon. Sources that were discovered after publication of this paper may be found in the other, later produced, papers.

3.2 Approaches

The literature review presented here concentrates on two broad categories: fuel-air control of spark ignition engines and artificial neural networks. Since a number of recent fuel-air control schemes have utilized neural networks, there is some overlap between the two categories.

3.3 Results

The paper first presents an overview of the spark ignition engine, including the models typically used to simulate the fuel-air system, based on the publications of Kaidantzis et al. [6] and Heywood [3]. This paper notes the assumptions of Kaidantzis’ transport delay model, which are further examined in Chapter 4. The

typical EGO sensor is introduced, along with some alternative sensor schemes which address some of the shortcomings of the EGO sensor.

The paper then introduces artificial neural networks, concentrating on the multilayer perceptron (MLP), which is the commonly used form of neural network. While the other papers will concentrate on the generalized neural network (GNN), the MLP is a subset of GNN, so the same principles apply. The review begins with Hebb's 1949 work [8] and Rosenblatt's 1958 paper [9], which laid the foundation for neural network research.

Next, some of the simple gradient-based learning schemes are introduced, most importantly the back-propagation method of calculating the gradient. A number of other first- and second-order training approaches are also mentioned. Because of the large amount of data available to a fuel-air controller, online learning schemes were identified as a method of dealing with the memory constraints of an embedded controller.

Although recurrent neural networks are not used in this thesis, the literature review presents them as a method of modelling dynamic systems, including an interesting example of using living neurons as a controller of a simulated avionics system [15].

The paper then presents a number of previously studied neural fuel-air control schemes. While there are a number of such control schemes in the literature, they all have features that make them unsuitable for the solution of the problem at hand. The most important is that most use off-line learning, meaning that an extensive "calibration" period is required before the controller can operate; the elimination of this calibration time is one of the goals of this thesis work.

Finally, the review briefly covers some of the research on the use of gaseous fuels for transportation, concentrating on the results produced at the Saskatchewan Research Council which show the environmental and economic benefits of natural gas as a fuel.

3.4 Contributions

While a literature review, by definition, should not include original research, this literature review succeeds in bringing together a large number of references on the topics involved in this research program.

A Survey of Literature Applicable to On-line Learning for Transient Control of Fuel-Air Ratio in SI Engines Powered by Gaseous Fuels

Travis Wiens, Greg Schoenau, Rich Burton
Dept.of Mechanical Engineering
University of Saskatchewan
Saskatoon, Sk

Abstract

This paper presents the current state of research toward a self-learning fuel-air controller for vehicles running on gaseous fuels with spark ignition (SI) engines. The current method of controlling the fuel-air ratio requires the time-consuming and costly step of creating a fuel map by manually optimizing the fuelling over all possible operating conditions on a dynamometer. A solution is proposed which would result in a controller based on a neural network that is trained on-line, with no calibration time or special equipment required. This paper presents a survey of current research toward this goal.

1 Introduction

A major portion of a vehicle's ECU (engine control unit) is devoted to maintaining the proper ratio of fuel to air in the cylinder charge mixture. Deviation from the optimum fuel-air ratio (also known as the FAR or FA ratio) can lead to reduced power and fuel efficiency and increased emissions [Taylor, 1985]. In a typical ECU, transient fuel-air control function is achieved through a look-up table (referred to as a fuel map). The fuel map is a table of fuel injector pulse widths covering all operating points which may include intake manifold pressure, engine speed, mass airflow, intake and coolant temperatures and

throttle position. Generating this fuel map is typically an expensive process, as it takes many hours of calibration on expensive equipment that must be repeated for every different engine model. In some cases, the fuel map has a basic form of adaptation to changes in vehicle properties from vehicle to vehicle and over time. Often, though, it is static in nature.

The objective of this paper is to provide a review of the literature pertaining to those studies applicable to the goal of applying on-line learning which generates the map automatically, and also control the fuel during transient situations. Since artificial neural networks are an integral part of many of the

reviewed papers, an introduction to neural networks and training algorithms is provided.

2 Fuel-Air control

An internal combustion engine works by igniting a mixture of gaseous or atomized fuel and air in the cylinder. The chemical energy stored in the mixture is converted to mechanical energy when the expanding products of combustion exert a force on the piston. The ideal mixture of fuel to air in a spark ignited engine is generally such that there is just enough oxygen in order to combust all of the fuel; this is referred to as a stoichiometric mixture and produces exhaust gasses mainly composed of water vapour and carbon dioxide. For gasoline, the stoichiometric fuel to air ratio is 0.067 by mass; for hydrogen, 0.0292; and for natural gas, 0.058 ([Taylor, 1985]). A lean mixture (too much air) will produce pollutants such as nitrous oxides (NO_x) and may lead to overheating of engine components. An engine burning a rich mixture (too much fuel) is inefficient and produces emissions such as carbon monoxide as well as unburnt hydrocarbons. In both the extreme rich and lean case, the mixture may fail to ignite, resulting in reduced power and unburnt hydrocarbon emissions [Hill et al., 1996].

Modern automotive engines are equipped with catalytic converters which can remove many of these pollutants, but only if the fuel-air ratio makes small oscillations about the stoichiometric ratio¹. Additionally, if the fuel-air ratio deviates to the point of misfire, the unburnt fuel may combust in the catalytic converted, which may be damaged

by overheating. There has also been some research into lean-burn engines [Hill et al., 1996], where the fuel air ratio is controlled at a lean operating point; in any case, the ratio of fuel to air must be controlled to some value to a high degree of accuracy.

There are two main methods of controlling the fuel-air ratio in an SI engine: carburetion and fuel injection. Carburetors were used in most vehicle engines until the 1980's and are still used in many small engines. A carburetor works by using fluid pressures through venturies and orifices to meter the fuel. They are complex components that often include temperature compensation and cold-start adjustments through fluid/mechanical means. In order to avoid this mechanical complexity, most modern vehicle engines use electronic fuel injection. In this fuel-air control scheme, the engine control unit (ECU) measures a number of parameters such as intake air flow and manifold pressure and estimates the required mass of fuel needed. This signal is then used to control the injectors, which meter the flow of fuel into the intake system. This control system can operate in a feed-forward mode and/or with closed loop oxygen or lambda sensor feedback. Electronic fuel injection is generally preferable to carburetion because it is more easily adjustable and less prone to mechanical failure.

A block diagram of a typical ECU fuel controller is shown in Figure 1. The inputs to the controller generally include engine speed, N_e , intake manifold pressure, P_m , ambient air pressure, P_a , intake air temperature, T_a , coolant temperature, T_c , and a feedback signal from the exhaust gas oxygen (EGO) sensor, which is related to the relative fuel-air

¹The optimum centre point for these oscillations is actually slightly rich for gasoline and slightly lean for natural gas [Siewert et al., 1993]

ratio ($\lambda = FA_s/FA$ where FA_s is the stoichiometric fuel-air ratio), amongst other signals. The output is the time that the injector(s) should be open for each fuel injection². It should be noted that the ECU's in production vehicles take control of a number of other functions and are becoming more complex; a typical control unit has 50-120 look-up tables, with approximately 15 inputs and 30 outputs, many of which are calculated by feedforward controllers [Isermann and Muller, 2003].

Due to the nonlinearities in the engine dynamics and the EGO sensor, it can be very difficult to keep the fuel-air ratio at the stoichiometric point, especially during transient operation (for example, aggressive acceleration) [Shayler et al., 2000]. These nonlinearities arise from transport delays and the non-linear character of volumetric efficiency. The transport delay is due to the time it takes the fuel-air mixture to travel from the injection point to the exhaust gas sensor. The time delay, T_d can be modelled as

$$T_d = \frac{120}{N_e} + \frac{\rho_i}{\dot{m}_a}(l_i A_i + 0.5 L_{ex} A_{ex}) \quad (1)$$

where N_e is the engine speed (rpm), ρ_i is the intake air density, \dot{m}_a is the mass air flow, l_i and l_{ex} are the intake and exhaust manifold lengths, and A_i and A_{ex} are the average cross sectional areas of the intake and exhaust manifolds [Kaidantzis et al., 1993]. Note that this makes the relatively major assumption that the exhaust stream density is half that of the intake.

The volumetric efficiency is defined as the mass of fresh fuel-air mixture that enters the cylinder relative to the mass of air that would fill the cylinder displacement at atmospheric

density. This property is integral to the control of fuel-air ratio, since the optimal amount of fuel to inject for stoichiometric operation is proportional to the volumetric efficiency at any operating point. Heywood [1988] gives a relation for the volumetric efficiency, η_v as

$$\eta_v = \left(\frac{M_f}{M_a}\right) \left(\frac{P_m}{P_{atm}}\right) \left(\frac{T_c}{T_{atm}}\right) \left(\frac{1}{[1 + FA_s]}\right) \cdots \times \left(\frac{r_c}{r_c - 1} - \frac{1}{\gamma(r_c - 1)} \left[\frac{P_e}{P_i} + \gamma - 1\right]\right) \quad (2)$$

where

- M_f is the molecular weight of the fuel,
- M_a is the molecular weight of air,
- P_m is the intake manifold pressure,
- P_{atm} is the atmospheric pressure
- T_c is the coolant temperature
- T_{atm} is the intake temperature
- FA_s is the fuel-air mass ratio (assumed to be near stoichiometric)
- r_c is the compression ratio
- γ is the specific heat ratio of the fuel, and

P_e is the exhaust manifold pressure.

It should be noted that this equation does not include a number of effects such as reflected pressure waves, heat transfer, or flow path geometry (such as valves), which tend to increase the non-linear nature of the equation [Taylor, 1985].

While most control feedback schemes rely on exhaust gas oxygen (EGO) or universal (wide range) exhaust gas oxygen sensors (UEGO), they have some significant weaknesses. EGO and UEGO sensors are not operational until they warm up and are therefore not available for start-up feedback. While considerably less expensive than UEGO sensors, EGO sensors are bang-

²Some injection schemes have one injector per cylinder and synchronize their injections to the intake timing of each individual cylinder (sequential injection), some inject in pairs on both intake and power strokes (grouped injection), and some have one injector for all cylinders (throttle body injection).

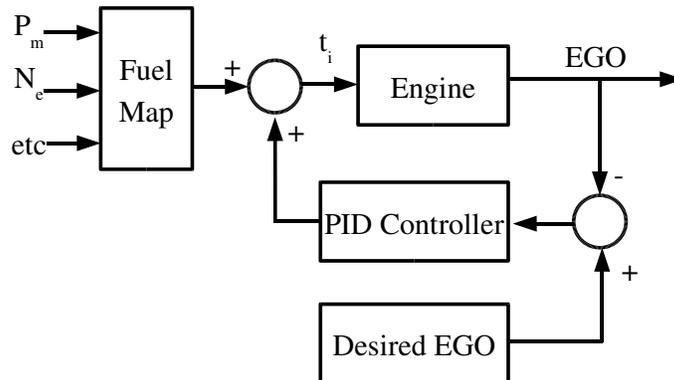


Figure 1: Block diagram of a typical engine control unit fuelling section. The controller computes an appropriate injection time, t_i , given inputs such as engine speed, N_e , intake manifold pressure, P_m , ambient air pressure, P_a , intake air temperature, T_a , coolant temperature, T_c . Any errors in the fuel map are corrected by feedback from the exhaust gas oxygen sensor, EGO .

bang type sensors, only giving information on whether the mixture was rich or lean, but little useful information as to the magnitude of the mixture strength. Also, as mentioned above, since the sensor is mounted in the exhaust, there is a considerable delay between the time of injection and the time when the sensor provides feedback on that injection. There are a number of alternative sensors which can be used for engine control.

As an example of an alternate feedback scheme, the optimum fuelling amount and timing can be extracted from in-cylinder pressure measurement (the higher the mean effective pressure, the more power is produced, generally indicating a better fuelling strategy). Sellnau et al. [2000] presented a method of retrofitting existing engines with non-intrusive pressure sensors mounted in the sparkplug boss. Leonhardt et al. [1999] also proposed using neural networks to control fu-

elling based on cylinder pressure measurement, and successfully applied a static radial basis function to the control of timing and mass of injected fuel for a diesel engine. Another method of measuring cylinder pressure indirectly is via the spark ionization current [Hellring and Holmberg, 2001]. Zhenzhong et al. [2002] developed a control scheme in which only the engine speed was used for feedback; the optimum fuelling results in the greatest engine speed for a given fuelling rate. A similar scheme was presented by Tang et al. [1998], in which the ideal fuel-air ratio could be achieved by biasing individual cylinders very slightly rich and lean, and monitoring the fluctuations in engine speed.

One problem with using a fuel-map based control scheme is that the fuel map can be expensive and time consuming to create, a process that must be repeated for every different engine model, and doesn't take into account

manufacturing differences or wear that occurs over time. One possible solution to this problem is using an adaptive control scheme, such as a neural network controller that can learn the optimum fuelling strategy for any engine. In order to properly discuss the results of the literature in which neural networks are used, a short introduction to neural networks is presented. For detailed discussions on these networks, the reader is referred to the excellent works of Haykin [1999] and Gupta et al. [2003].

3 Neural Networks

Neural networks were originally conceived as a method of modeling the human brain, in the hopes of gaining better insight into its inner workings [Hebb, 1949] Rosenblatt [1958]. A typical artificial neuron, shown in Figure 2, takes a weighted sum of the inputs and applies a non-linear activation function. Engineers and mathematicians quickly realized that a network of these artificial neurons could be used to approximate a wide range of functions (in addition to performing pattern recognition and classification duties). The output of a typical single layer network with n inputs and m neurons is given by

$$\mathbf{Y} = \tanh(\mathbf{W}\mathbf{X}) \quad (3)$$

where

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \quad (4)$$

$$\mathbf{W} = \begin{bmatrix} w_{12} & w_{22} & \dots & w_{1(n+1)} \\ w_{21} & w_{22} & \dots & w_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{m(n+1)} \end{bmatrix} \quad (5)$$

and

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \quad (6)$$

where \mathbf{Y} is the output, \mathbf{W} is a matrix of network weights and \mathbf{X} is the input. Notice that the input matrix, \mathbf{X} , is augmented by a “1”, allowing a one weight to function as a bias, independent of the inputs.

A single layer network cannot approximate a function which is not linearly separable. This limitation can be overcome by the use of the multilayer perceptron. A multilayer network is a network of neurons where the outputs of the neurons of one layer are connected to the inputs of the next. These networks can have a variety of linear or non-linear activation (or output) functions such as the hyperbolic tangent, piecewise linear functions or a simple gain. In a typical feed-forward network, also known as a static network, the outputs of one layer of neurons are connected to the inputs of the next layer, as shown in Figure 3.

From a system control perspective, neural networks are generally used for function approximation. This implies an optimization problem: finding the optimal network weights that minimize the error between the network response y and the desired response y^* . This can be a difficult problem as there are generally a large number of parameters to optimize and the error surface is nonlinear. Gupta and Rao [1998] classify the various methods of error based learning into two categories: stochastic (incorporating a random component) and error-correction learning, which is further subdivided into two subclasses: least mean square and backpropagation. Stochastic methods make pseudo-random changes to

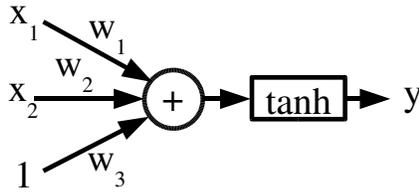


Figure 2: A simple neuron. The output is a weighted sum of the inputs, passed through a non-linear activation function (in this case, a hyperbolic tangent function). The inputs, x_1 and x_2 , are augmented by a “1”, in order to incorporate a bias.

the weights with the objective of reducing the error. Most on-line methods fall under this category due to the semi-random nature of the input sequence, which will be expanded on in the next section. Backpropagation refers to a method of calculating the desired response (or error) for every neuron, such that the gradient

$$\mathbf{g} = \frac{\partial E}{\partial \boldsymbol{\theta}} \quad (7)$$

may be evaluated, where $\boldsymbol{\theta}$ is a vector composed of all the network weights (\mathbf{W} for each neuron) and $E = f(y^* - y)$ is the output error [Haykin, 1999]. For batch training, this function is generally the expectation of the sum squared error, i.e.

$$E = \frac{1}{N} \sum_{k=1}^N (y^*(k) - y(k))^2 \quad (8)$$

for N possible data points.

The gradient is then used to update the network weights. The most simple method is gradient descent

$$\Delta \boldsymbol{\theta}(k) = \eta \mathbf{g} \quad (9)$$

where η is the learning rate.

A slightly more complex method is gradient decent with momentum:

$$\Delta \boldsymbol{\theta}(k) = \eta \mathbf{g} - \alpha \Delta \boldsymbol{\theta}(k-1) \quad (10)$$

where α is a momentum constant [Rumelhart et al., 1986] [Silva and Almeida, 1990] [Gupta et al., 2003]. This algorithm increases the step size when two successive weight changes are in the same direction, and decreases it if the weight change reverses. This allows larger learning rates without instability.

Many other off-line algorithms exist, notably the Marquardt Levenberg algorithm, which approaches second order speed without having to compute the Hessian (second derivative) [Hagan and Menhaj, 1994]; the conjugate gradient method in which weight updates are performed in a direction conjugate to the previous iteration [Fletcher and Reeves, 1964]; the “bold driver”, an adaptive step size method in which the step size is increased slowly until the error increases, at which point the step size is drastically reduced [Battiti and Tecchioli, 1994]; and other first and second order methods [Battiti, 1992], [Bortoletti et al., 2003].

Neural networks can also approximate dynamic systems by changing how the network

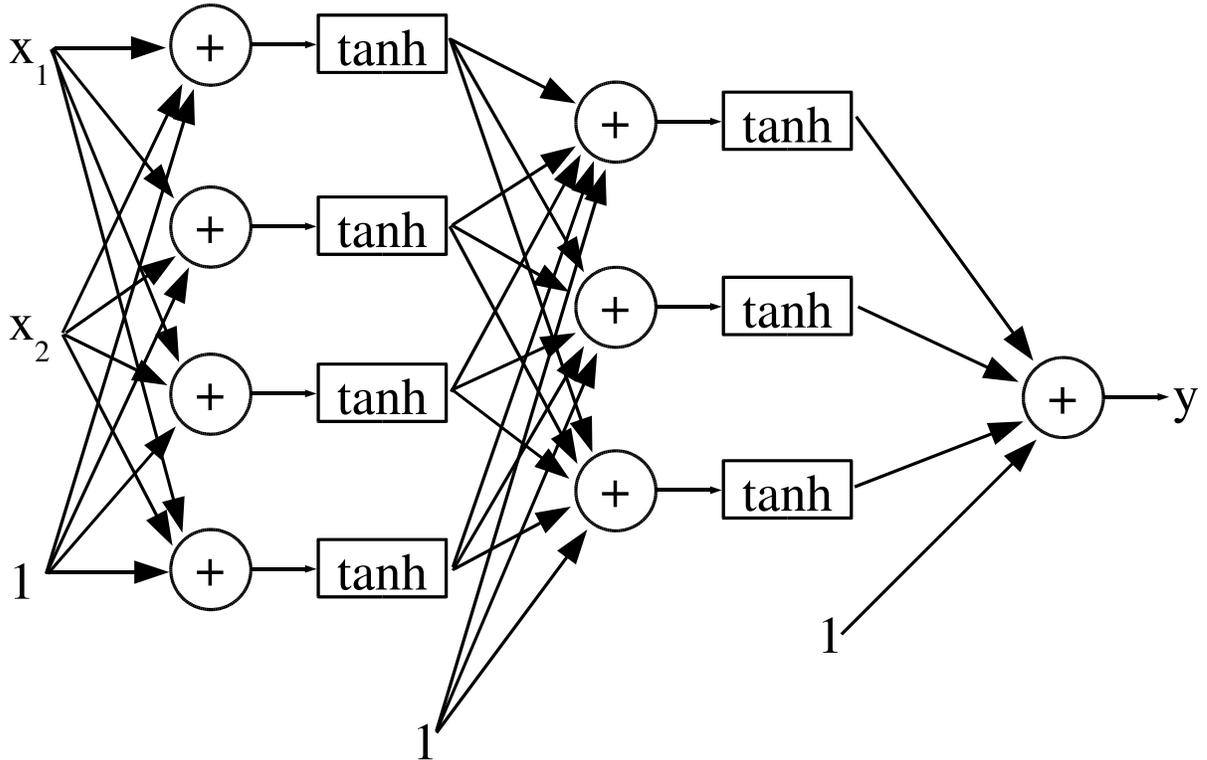


Figure 3: A neural network, with two inputs, two hidden layers with \tanh activation functions and one linear output neuron. Weights are omitted for clarity.

is connected [Elman, 1990]. A locally recurrent network can have connections between any two neurons (or from a neuron's delayed output to its own input). Delayed outputs of a static network may also be connected to the inputs of the network. This type of dynamic neural network is called a feedback network and has the advantage of a simple internal structure, allowing the application of standard training schemes (Ping et al. [1997], Mastorocostas and Theocharis [2002] and Gupta et al. [2003]). The third form, a dynamic neural unit (DNU), incor-

porates self-recurrence within each neuron [Gupta and Rao, 1998].

In addition to being able to identify and model dynamic systems, neural networks can be used to control these systems. There are a number of methods of control, including direct control [Ichikawa and Sawa, 1992], linearized control [Chen and Narendra, 2004], model reference methods and others [Narendra and Mukhopadhyay, 1994].

In an attempt to reintroduce biology to neural network control, Potter [2001] and DeMarse et al. [2001] have applied living neural

cells to mechanical control problems. These controllers use biological neurons to perform computational tasks. One example of DeMarse's work that has recently reached the popular media has been using rat neurons to fly a simulated F-22 combat aircraft [Lovgren, 2004], [DeMarse and Dockendorf, 2005].

4 On-line Training

There are two main methods of supervised training of neural networks: off-line and on-line. In off-line, or batch training, the entire training set is used for every iteration of the network weight updates. For example, a typical algorithm has the following steps:

1. Initialize network
2. Calculate total average error for entire training set using current weights
3. Adjust weights
4. If stopping criteria has been met, end
5. Go to step 2

Conversely, on-line training only uses one input/output pair to calculate the new weights. For example:

1. Initialize network
2. Calculate error for input/output pair k
3. Adjust weights
4. If stopping criteria has been met, end
5. Increment k
6. Go to step 2

The major advantage of on-line training is its reduced memory requirements; batch training requires the entire training set to be held in memory at all times, while the memory for an on-line algorithm only holds the current input/output pair and a small number of calculated variables. This can be a large advantage for very large data sets. Another advantage of the on-line algorithm is that its stochastic properties (due to the semi-random nature of the input sequence) appear to make it more likely to find a global, rather than local, optimum. Thirdly, since on-line algorithms update the network weights in real time at every time step, the model can track changes in plant properties that vary with time [Magoulas et al., 2001]. A disadvantage of on-line training is that if the training algorithm is too aggressive, the network can "forget" previous data, tracking the *short-term output* of the plant rather than learning its *properties*. While batch training algorithms would be expected to converge more quickly, it has been shown by Murata [1998] and others that the performance of certain on-line training algorithms can approach that of off-line methods (albeit under some rather severe assumptions).

Although there is large volume of theoretical work that attempts to analyze the learning dynamics of various on-line learning systems (e.g. Haykin et al. [2004], Schiller and Steil [2005], Venugopal et al. [1995], Saad and Rattray [1998], Rattray and Saad [1998], Biehl et al. [1998], Heskes and Wiergerink [1998], Coolen and Saad [1998]), the design of neural networks is still largely a trial and error process.

5 Neural Networks and Transient Fuel Air Control

A fuel-injected spark ignition engine must control the mass ratio of fuel to air to within tight tolerances in order to achieve the optimum fuel efficiency, power production, and exhaust emissions performance. In most current engine control units, this is achieved with relatively simple control schemes. When the engine is operating near steady state, PI or PID-type feedback control is used with an exhaust gas oxygen sensor to eliminate any error. However, because of nonlinearities and time delays in the engine, this type of linear controller is not accurate during transient conditions. In this case, a fuel map or speed-density algorithm is used estimate the required fuelling.

One potential alternative method of controlling the fuelling is to use a neural network. A number of researchers have applied neural networks to fuel-air control in a number of strategies. One of the first was Shiraishi et al. [1995], who used a Cerebellar model articulation controller (CMAC) neural network to control the air-fuel ratio.

Beltrami et al. [2003] presented a simulation feasibility study of using recurrent neural networks for AFR control. This model included 14 neurons in the hidden layer and was trained off-line, using the Levenberg-Marquardt method.

Nicolau et al. [1996] presented a number of methods of modeling the volumetric efficiency of an IC engine. This model attempted to approximate equation 2 by

$$\eta_v = \eta_v(N_e, P_m, T_m) \quad (11)$$

where N_e is engine speed, P_m is intake mani-

fold pressure and T_m is intake temperature. Static models employing radial basis functions (RBF, a neural network having “fuzzy” characteristics) and multilayer perceptrons (MLP) were developed in simulation. It was found that an acceptable RBF model could be created using 8 neurons and a MLP could model the data with 5 neurons, both with approximately equivalent errors.

There has also been a significant amount of work done at the Saskatchewan Research Council on the application of neural networks to engine control. Sulatisky and Hill [2002] used a neural network to control the fuelling of trucks converted from gasoline to natural gas (this work is further described in the following section). Gnanam [2002] presented a theoretical study (based on simulation), using recurrent neural networks with on-line training. In addition, he applied the Alopex Algorithm [Venugopal and Pandya, 1991], which does not need accurate sensor measurements for modifying the weights; it only requires the sign of the change in error, not the magnitude.

Manzie et al. [2000] applied a radial basis function to model volumetric efficiency and create a model predictive controller for the fuelling process. A model predictive controller requires a model of the plant which it uses to optimize controller performance, projected into the future. The controller presented in this paper was tested via simulation and appeared to work very well, but was actually worse than the factory ECU when tested on an engine in Manzie et al. [2001]. When corrections were made to the algorithm, the modified control scheme outperformed the stock controller, with approximately half the maximum deviation from the desired FA ratio and half the root mean squared error. On-line training took approximately 30 min-

utes, although it is not explained how the engine was prevented from stalling during this time. While these papers present some very favourable results, the tests were performed on a stationary dynamometer-mounted engine and the algorithm seemed to require very repeatable throttle movements and the engine to reach steady state, a requirement not often met by on-road vehicles. For example, the test was only performed on step changes of approximately 5° of throttle position, a situation far removed from a full-throttle tip-in of 90° .

Zhenzhong et al. [2002] presented a simulation of a system in which a fuzzy neural network was used for open and closed loop control of ignition timing, injection timing and injection quantity of a direct-injection hydrogen engine. This control scheme did not require an exhaust gas sensor as training feedback was based on maximizing the engine speed; if a change in weights increased the engine speed, changes were continued in the same direction.

Alippi et al. [2003] developed a neural controller which was trained off-line, using simulated data provided by a neural network system identification of an engine. This process took approximately 15 hours to train 15 neurons. The same paper also predicts that a large user of on-line training will be in small production line cars (e.g. high performance vehicles).

6 Gaseous Fuels and Neural Networks

There has been considerable research into the use of gaseous fuels (such as natural gas and hydrogen) in vehicular applications. Natural gas is readily available (there is an esti-

mated supply of 2200 trillion cubic feet available in North America [Natural Gas Division, 2004]) and is environmentally friendly, producing less greenhouse gases than gasoline [Sulatisky et al., 2004].

Sulatisky et al. [1995] present the initial development of a low-cost, fuel-efficient, environmentally friendly natural gas vehicle with a good driving range, based on modified Geo Metros. They found the environmental emissions to be better than gasoline, especially with a slightly rich bias. Sulatisky et al. [2004] show the final results of over 1.3 million km of testing of natural gas on subcompact vehicles. The measured fuel economy was approximately equal to gasoline, but produced 25.5% less greenhouse gas at the tailpipe and 34.2% less in the full cycle. As a part of the same study, Hill et al. [1996] experimented with lean-burning natural gas and found considerable improvements in fuel efficiency (33% better than gasoline), although the nitrous oxides were much higher, leaving this method as being infeasible unless better NOx removal schemes are developed.

Sulatisky et al. [2002] present two neural network FA controllers for use on trucks fuelled by natural gas. Two controllers were developed, with off-line and on-line training. The controller trained off-line could control the FA ratio over about 90% of the operating range of the vehicle. The on-line controller was able to control the FA ratio at start and idle. Sulatisky and Hill [2002] also note they had success, but that the controller tended to saturation. This network “translated” the fuel injector signal supplied by the OEM gasoline ECU into values useable with natural gas. This was largely based on “decompensating” for a number of dynamic effects which apply to liquid gasoline, but not natural gas. A more in-depth study by Hill

and Lung [2003] resulted in an off-line trained neural controller that was able to successfully control the fuel for a variety of engines and OEM management systems, although they did not successfully develop an on-line solution. Their off-line-trained controller was able to successfully control the vehicle using only two inputs: engine speed and intake manifold pressure or mass air flow. This paper also proposed (but did not implement) an on-line/off-line training scheme, in which representative data is recorded during driving and analysed while the key is off. Gnanam [2002] developed a working simulation of an on-line control scheme, but this was never tested experimentally.

7 Conclusion

While there has been significant research into the use of intelligent fuelling control of vehicles, it appears that the development of a transient fuel-air controller that will work in a variety of vehicles without a lengthy initial calibration period has not yet been successful. Similarly, there does not appear to be a large body of academic work on the application of advanced controls to vehicles powered by gaseous fuels. This presents a significant research challenge that needs to be addressed in order to advance the future of gaseous fuels as a sound replacement for fossil fuels in vehicular applications.

8 Acknowledgements

The authors would like to acknowledge the support of NSERC, in the form of a PGS D Scholarship, as well as the Saskatchewan Research Council for providing equipment and expertise.

References

- C. Alippi, C de Russis, and V Piuri. A neural network based control solution to air-fuel ratio control for automotive fuel-injection systems. *IEEE Transactions on Systems, Man and Cybernetics - Part C: Applications and Reviews*, 33(2):259–268, May 2003.
- R. Battiti and G. Tecchiolli. Learning with first, second, and no derivatives: a case study in high energy physics. *Neurocomputing*, 6:181–206, 1994.
- T. Battiti. First and second-order methods for learning: Between steepest descent and Newton’s method. *Neural Computation*, 4(2):141–166, 1992.
- C. Beltrami, Y. Chamailard, G. Millerioux, P. Higelin, and G. Bloch. AFR control in SI engine with neural prediction of cylinder air mass. In *Proceedings of the American Control Conference*, pages 1404–1409, Denver, June 2003.
- M. Biehl, A. Freking, M Holzer, G. Reents, and Enno Schlosser. *On-line Learning in Neural Networks*, chapter On-line Learning of Prototypes and Principal Components, pages 231–250. Cambridge University Press, Cambridge, 1998.
- A. Bortoletti, C. Di Fiore, S. Fanelli, and P. Zellini. A new class of quasi-newtonian methods for optimal learning in MLP-networks. *IEEE Transactions on Neural Networks*, 14(2):263–273, 2003.
- L. Chen and K.S. Narendra. Identification and control of a nonlinear discrete-time system based on its linearization: A unified framework. *IEEE Transactions*

- on *Neural Networks*, 15(3):663–673, May 2004.
- A. Coolen and D. Saad. *On-line Learning in Neural Networks*, chapter Dynamics of Supervised Learning with Restricted Training Sets, pages 303–344. Cambridge University Press, Cambridge, 1998.
- T. DeMarse and K Dockendorf. Adaptive flight control with living neuronal networks on microelectrode arrays. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1549–1551, Montreal, 2005.
- T. B. DeMarse, D. A. Wagenaar, A.W. Blau, and S. M. Potter. The neurally controlled animat: Biological brains acting with simulated bodies. *Autonomous Robots*, 11:305–310, 2001.
- J.L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.
- R. Fletcher and C.M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2), 1964.
- G. Gnanam. Neural network control of air-to-fuel ratio in a bi-fuel engine. Master’s thesis, University of Saskatchewan, 2002.
- M. Gupta and D. Rao. *Neuro-Control Systems: Theory and Practice*, chapter Neuro-Control Systems: A Tutorial, pages 1–43. IEEE, Cambridge, 1998.
- M.M. Gupta, L. Jin, and N. Homma. *Static and Dynamic Neural Networks; From Fundamentals to Advanced Theory*. Wiley Interscience, New Jersey, 2003.
- M. Hagan and M. Menhaj. Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.
- S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, New Jersey, 2nd edition, 1999.
- S. Haykin, Z. Chen, and S. Becker. Stochastic correlative learning algorithms. *IEEE Transactions on Signal Processing*, 52(8): 2200–2208, 2004.
- D.O. Hebb. *The organization of behavior; a neuropsychological theory*. Wiley-Interscience, New York, 1949.
- M. Hellring and U. Holmberg. An ion current based peak-finding algorithm for pressure peak position estimation. Technical Report 2001-01-1920, Society of Automotive Engineers (SAE), 2001.
- T. Heskes and W. Wiegerinck. *On-line Learning in Neural Networks*, chapter On-line Learning with Time-Correlated Examples, pages 251–278. Cambridge University Press, Cambridge, 1998.
- J.B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
- S. Hill and B. Lung. The application of neural controls to ford and dodge pickup trucks running on natural gas. Technical Report 11305-4C02, Saskatchewan Research Council, Saskatoon, 2003.
- S. Hill, M. Sulatisky, J. Lychak, N. Nakamura, T. Matsui, and G. Rideout. A lean-burn, sub-compact natural gas vehicle. Technical Report 961676, SAE, 1996.

- Y. Ichikawa and T. Sawa. Neural network application for direct feedback controllers. *IEEE Transactions on Neural Networks*, 3(2):224–229, 1992.
- R. Isermann and N. Muller. Design of computer controlled combustion engines. *Mechatronics*, 13:1067–1087, 2003.
- P. Kaidantzis, P. Rasmussen, M. Jensen, T. Vesterholm, and Elbert Hendricks. Robust, self-calibrating lambda feedback for SI engines. Technical Report 930860, SAE, Warrendale, 1993.
- S. Leonhardt, N. Muller, and R. Isermann. Methods for engine supervision and control based on cylinder pressure information. *IEEE Transactions on Mechatronics*, 4(3):235–245, 1999.
- Stefan Lovgren. Brain in dish flies simulated fighter jet. http://news.nationalgeographic.com/news/2004/11/1119_041119_brain_petri_dish.html, November 19 2004.
- G.D. Magoulas, V.P. Plagianakos, and M.N. Vrahatis. Adaptive stepsize algorithms for online training of neural networks. *Nonlinear Analysis*, 47:3425–3430, 2001.
- C. Manzie, M. Palaniswami, and H. Watson. Model predictive control of a fuel injection system with a radial basis function network observer. In *International Joint Conference on Neural Networks*, pages 359–364, Como, Italy, 2000.
- C. Manzie, M. Palaniswami, and H. Watson. Gaussian networks for fuel injection control. In *Proceedings of the Institution of Mechanical Engineers, Part D*, volume 215, 2001.
- P.A. Mastorocostas and J. B. Theocharis. A recurrent fuzzy neural model for dynamic system identification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 32(2):176–190, April 2002.
- N. Murata. *On-line Learning in Neural Networks*, chapter A Statistical Study of On-line Learning, pages 63–92. Cambridge University Press, Cambridge, 1998.
- K.S. Narendra and S. Mukhopadhyay. Adaptive control of nonlinear multivariable systems using neural networks. *Neural Networks*, 7(5):737–752, 1994.
- Natural Gas Division. Canadian natural gas. Technical report, Natural Resources Canada, Ottawa, 2004.
- G. De Nicolau, Scattolini R, and C. Siviero. Modelling the volumetric efficiency of IC engines: Parametric, non-parametric and neural techniques. *Control Engineering Practice*, 4(10):1405–1415, 1996.
- X. Ping, R. Burton, and C. Sargent. Identifying a nonlinear dynamic system with partially recurrent neural networks- feasibility study and issues on error accumulation problem. In *Fluid Power Systems and Technology 1997*, volume FPST-Vol 4, DSC Vol 63, pages 13–19. ASME, New York, 1997.
- S. M. Potter. Distributed processing in cultured neuronal networks. *Progress in Brain Research*, 130:49–62, 2001.
- M. Rattray and David Saad. *On-line Learning in Neural Networks*, chapter Incorporating Curvature Information into On-line Learning, pages 183–208. Cambridge University Press, Cambridge, 1998.

- F Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *In Psychological Review*, 65:386–408, November 1958.
- D.E. Rumelhart, G.E. Hinton, and R.J. Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter Learning Internal Representations By Error Propagation. MIT Press, Cambridge, 1986.
- D. Saad and M. Rattray. *On-line Learning in Neural Networks*, chapter A Statistical Study of On-line Learning, pages 135–164. Cambridge University Press, Cambridge, 1998.
- U.D. Schiller and J.J. Steil. Analyzing the weight dynamics of recurrent learning algorithms. *Neurocomputing*, 63:5–23, 2005.
- M.C. Sellnau, F.A. Matekunas, P.A. Battiston, C.F. Chang, and D. Lancaster. Cylinder-pressure-based engine control using pressure-ratio-management and low-cost non-intrusive cylinder pressure sensors. Technical Report 2000-01-0932, Society of Automotive Engineers (SAE), 2000.
- P. Shayler, M. Goodman, and T. Ma. The exploitation of neural networks in automotive engine management systems. *Engineering Applications of Artificial Intelligence*, 13(2):147–157, April 2000.
- H. Shiraishi, S. Ipri, and D. Cho. CMAC neural network controller for fuel-injection systems. *IEEE Transactions on Control Systems Technology*, 3(2):32–38, 1995.
- R. Siewert, P. Mitchell, and P. Mulawa. Environmental potential of natural gas fuel for light-duty vehicles: An engine-dynamometer study of exhaust emission-control strategies and fuel consumption. Technical Report 932744, SAE, Warrendale, 1993.
- F.M. Silva and L.B. Almeida. *Advances of Neural Computers*, chapter Speeding up Backpropagation. Elsevier Science Publishers, North-Holland, 1990.
- M. Sulatisky and S. Hill. The application of neural control systems to natural gas vehicles. Technical Report SRC Pub No 11120-1E02, Saskatchewan Research Council, Saskatoon, 2002.
- M. Sulatisky, S. Hill, B. Lung, Y. Song, K. Young, C. Murray, J. Jones, J. Lychak, K. Babich, and B. Smith. Intelligent control systems for fuel cell and natural gas vehicles. Technical Report 11305-1E02, Saskatchewan Research Council, Saskatoon, 2002.
- M. Sulatisky, S. Hill, J. Lychak, K. Nakamura, T. Matsui, and G. Rideout. Adapting a geo metro to run on natural gas using fuel-injection technology. Technical Report 951942, SAE, Warrendale, 1995.
- M. Sulatisky, S. Hill, C. Murray, J. Jones, K. Young, J. Lychak, and K. Babich. Natural gas geo metro fleet demonstration. Technical Report 10484-1C03, Saskatchewan Research Council, Saskatoon, 2004.
- X. Tang, J. Asik, G. Meyer, and R. Samson. Optimal A/F ratio estimation model (synthetic UEGO) for SI engine cold transient AFR feedback control. In *Electronic Engine Controls 1998: Diagnostics and Controls (SP-1357)*, pages 177–185, Feb 1998. SAE Technical Paper Series 980798.

- Charles Fayette Taylor. *The Internal-Combustion Engine in Theory and Practice*, volume I. M.I.T. Press, Cambridge, second edition, 1985.
- K.P. Venugopal and A.S. Pandya. Alopex algorithm for training multilayer neural networks. In *International Joint Conference on Neural Networks*, pages 196–201. IEEE, 1991.
- K.P. Venugopal, R. Sudhakar, and A.S. Pandya. An improved scheme for direct adaptive control of dynamical systems using backpropagation neural networks. *Circuits, Systems, and Signal Processing*, 14(2):213–236, 1995.
- Yang Zhenzhong, Wei Jianqin, Fang Zhuoyi, and Li Jinding. An investigation of optimum control of ignition timing and injection system in an in-cylinder injection type hydrogen fueled engine. *International Journal of Hydrogen Energy*, (27):213–217, 2002.

CHAPTER 4

EXPERIMENTAL DETERMINATION OF TRANSPORT DELAY IN A SPARK IGNITION ENGINE

Published as:

- Travis Wiens, Rich Burton, Greg Schoenau, Mike Sulatisky, Sheldon Hill, Bryan Lung, “Experimental Determination of Transport Delay in a Spark Ignition Engine.” Technical Paper MISC-167, Saskatchewan Research Council, Saskatoon, 2006.

4.1 Objectives

This report presents an experimental method of determining the pure time delay in a spark ignition engine. It also shows that a commonly used model for the prediction of this delay has become obsolete due to the replacement of carburetors with port fuel injection.

4.2 Approaches

The previous model, described by Kaidantzis [6], made a “crude approximation” of the exhaust gas density. In carbureted or central point fuel injected engines, this assumption does not have a large effect, since the intake manifold volume dominates the equation, as it is much larger than the exhaust volume. However, in most modern vehicles, the injectors are located at the intake ports, significantly reducing the intake volume. In this situation, the exhaust volume dominates the equation, so a good estimate of the exhaust density is essential to a good model. Thus, the essentially constant delay (in units of injection events) predicted by Kaidantzis’ model is now predicted to vary with operating conditions, which is shown experimentally.

The experiment was performed on a General Motors 2500HD heavy pick-up truck, shown in Figure 4.1. This truck has a bi-fuel (natural gas or gasoline) 2003 Vortec 6.0L V8 engine with port fuel-injection. All experiments were performed using natural gas as a fuel. The engine was mounted in a 2001 chassis with an automatic transmission. A series of relays were installed between the ECU and injector drivers such that the fuel flow could be momentarily interrupted. While relays were used to interrupt the fuel, if one has control of the ECU programming, this function could easily be performed automatically by the ECU in a production

environment. The elapsed time between the interruption of the fuel and the oxygen sensor measurement change was used to determine the transport delay at that operating point. This was repeated at a number of operating points that are typically reached during driving.



Figure 4.1: The truck used for experimental studies in this dissertation was a General Motors 2001 GM2500HD chassis with a 2003 Vortec 6.0L V8 bi-fuel (compressed natural gas or gasoline) engine.

4.3 Results

The calculated data were plotted in two figures: one showing the pure time delay in seconds and another showing the same data converted into the number of injection events during the delay (reproduced in Figure 4.2). If Kaidantzis' model were applicable to this engine, the data in the second figure should fall on a nearly horizontal plane. However, this is clearly not the case. If a linear surface is fit to the data, the delay (in injection event samples), k_d is

$$k_d = (2.33 \times 10^{-3} \text{ samples/RPM})N_e - (0.1404 \text{ samples/kPa})P_m + 16.96 \text{ samples} \quad (4.1)$$

where N_e is the engine speed and P_m is the absolute intake manifold pressure.

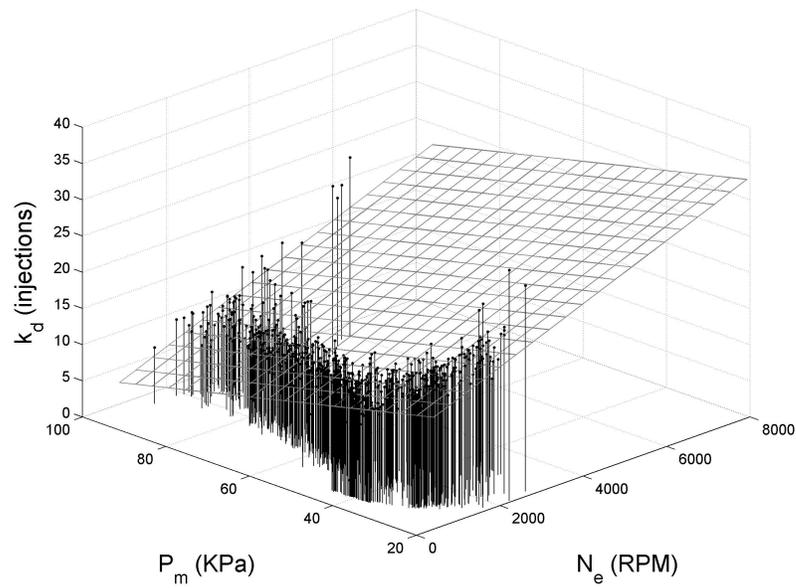


Figure 4.2: The delay in units of injection events is shown here. This figure demonstrates that the near-constant delay predicted by Kaidantzis’ model does not occur in a modern port-injected engine.

4.4 Contributions

This report has two major contributions. The first is a simple automatic method of determining the pure time delay in an engine. The second is the demonstration that the currently used model for the delay is not applicable for the use in port-injection engines and that the assumption of a constant delay should be used with care in design and simulation.

Experimental Determination of Transport Delay in a Spark Ignition Engine

Travis Wiens*, Rich Burton*, Greg Schoenau*, Mike Sulatisky†,
Sheldon Hill†, Bryan Lung†

Abstract

An important part of the feedback loop of the fuel-air control system in a spark-ignition engine is the delay: the time it takes for the charge mixture to move from the injection point, through the engine and exhaust system to the oxygen sensor. Many modern intelligent control schemes are model-based and require a method of automatically determining this delay. This paper presents an automatic, empirical method of determining the delay, requiring no specialized equipment. Sample results from an engine running on natural gas are presented. These results show that caution should be exercised when selecting a theoretical model for the engine delay.

1 Introduction

Since the introduction of electronic fuel injection, the performance of traditional feedback-based fuel-air control schemes has been limited by the delay inherent in the system: the fuel takes a finite time to travel from the injection point to the oxygen sensor in the exhaust. This delay generally causes a limit cycle to occur as the fuel-air mixture alternates from rich to lean. While this limit cycle is necessary for proper catalyst operation, the delay allows large deviations from the optimum fuel-air ratio (FAR or FA ratio) before the feedback can take effect. In order to reduce these fuelling errors, a number of intelligent control schemes have been developed. These control schemes typically gener-

ate an internal model of the engine dynamics, which includes the delay. Therefore, a method of automatically determining the delay was developed and experimentally tested, as presented in this paper

2 Fuel-Air System

A typical engine model is shown in Figure 1. Generally, the fuel-air controller will measure a number of parameters, such as engine speed, intake manifold pressure, etc. and attempt to estimate the optimum fuelling. This optimum fuelling is typically near the stoichiometric point; the fuel-air ratio such that there is just enough air to combust the fuel. The fuel is injected into the intake and en-

*Dept. of Mechanical Engineering, University of Saskatchewan, Saskatoon, Sk

†Alternative Energy Products, Saskatchewan Research Council, Saskatoon, Sk

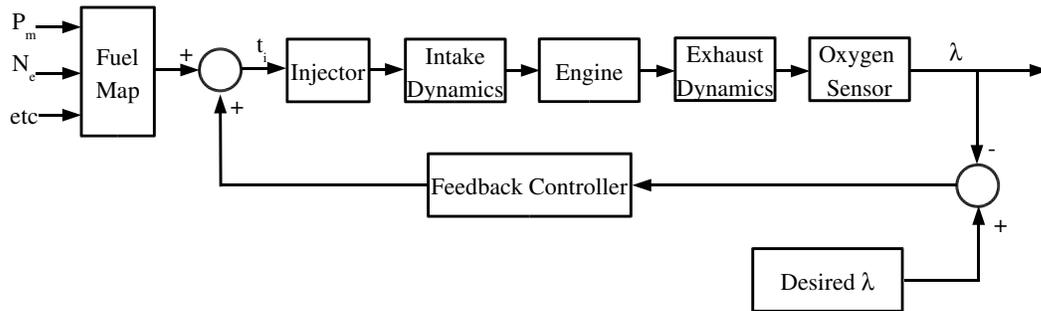


Figure 1: Block diagram of a typical engine control unit fuelling section. The controller computes an appropriate injection time, t_i , given inputs such as engine speed, N_e , intake manifold pressure, P_m , as well as other parameters. Any errors in the fuel map are corrected by feedback from the oxygen sensor, λ . There is a delay in the feedback loop, due to the time required for the charge mixture to travel from the injectors to the exhaust gas oxygen sensor, via the intake, engine and exhaust system.

ters the engine when the intake valve opens. The charge mixture is then compressed, combusts and is exhausted, over the course of two engine revolutions. Finally, the mixture is transported through the exhaust system to the exhaust gas oxygen (EGO) sensor. The oxygen sensor signal is then used to correct any steady state fuelling errors.

One common theoretical model for the delay time t_d is given by Kaidantzis et al. [1993] as

$$t_d = \frac{60 * 2}{N_e} + \frac{\rho_i V_i}{\dot{m}_a} + \frac{\rho_{ex} V_{ex}}{\dot{m}_a} \quad (1)$$

where N_e is the engine speed (in RPM), ρ_i is the intake air density, ρ_{ex} is the exhaust density, \dot{m}_a is the mass air flow, V_i is the effective volume between the injector and the intake valves and V_{ex} is the effective volume between the exhaust valves and the oxygen sensor.

As this model includes the exhaust density, a function of its pressure and temperature, which are not typically measured,

Kaidantzis makes the “crude approximation” that the exhaust manifold pressure is equal to the intake manifold pressure and that the exhaust temperature is double that of the intake, resulting in the form:

$$t_d = \frac{60 * 2}{N_e} + \frac{\rho_i}{\dot{m}_a} (V_i + 0.5V_{ex}) \quad (2)$$

which can be simplified to

$$t_d = \frac{2 * 60}{N_e} \left(1 + \frac{\rho_i}{\rho_{atm}} \frac{V_i + 0.5V_{ex}}{\eta_v N_{cyl} V_{cyl}} \right) \quad (3)$$

if one substitutes the definition for volumetric efficiency, η_v ,

$$\eta_v = \frac{\dot{m}_a * 2 * 60}{N_e \rho_{atm} V_{cyl} N_{cyl}} \quad (4)$$

where V_{cyl} is the cylinder displacement, N_{cyl} is the number of cylinders, and ρ_{atm} is the atmospheric air density [Heywood, 1988].

As engine controllers are typically event based rather than time based, sampling once per injection, it is more useful to express the

delay in terms of samples than time. The sampling period, p_s is

$$p_s = \frac{2 * 60}{N_e N_{cyl}}. \quad (5)$$

The number of samples in the delay is then

$$k_d = N_{cyl} + \frac{\rho_i}{\rho_{atm}} \frac{V_i + 0.5V_{ex}}{\eta_v V_{cyl}}. \quad (6)$$

If one notices that volumetric efficiency is proportional to the ratio of ρ_i/ρ_{atm} , but is only weakly affected by the engine speed [Heywood, 1988]:

$$\eta_v \approx K \frac{\rho_i}{\rho_{atm}} \quad (7)$$

equation 6 is no longer dependent on engine speed or intake manifold pressure and should be only slowly varying with temperatures and fuel properties:

$$k_d = N_{cyl} + \frac{V_i + 0.5V_{ex}}{KV_{cyl}}. \quad (8)$$

When Kaidantzis et al wrote their paper, most vehicles on the road had carburettors or throttle-body injection, both of which have large intake volumes relative to the exhaust volume. Since the delay was dominated by the intake, the exhaust volume had little effect, and therefore Kaidantzis's assumption (of $\rho_{ex} = 0.5\rho_i$) was usually a good one. However, the injectors in most modern vehicles are mounted in the cylinder head, reducing the intake volume from the order of litres to millilitres. This also has the effect of transferring the significant portion of the delay from the intake to the exhaust, invalidating the result of a constant delay (with respect to the number of injection cycles). The complete form of the equation is then

$$k_d = N_{cyl} + \frac{V_i + \frac{\rho_{ex}}{\rho_i} V_{ex}}{KV_{cyl}}. \quad (9)$$

Unfortunately, beyond predicting the expected function form, this equation does not have much practical use for predicting the delay, as exhaust density (from pressure and temperature) is not typically measured in production vehicles.

3 Determination of Delay

The test vehicle used for this research was a 2001 General Motors 2500HD truck with a 2003 6 litre Vortec V-8 engine running on natural gas, coupled to an automatic transmission. In order to experimentally map the delay as a function of engine speed and intake manifold pressure, a set of relays were used to momentarily interrupt the signal driving the injectors as shown in Figure 2. This shut off fuel flow to the engine for a time long enough for the oxygen sensor to register "lean", but not so long that the engine speed decreased significantly. Typically this time was approximately one engine revolution. While this setup requires the installation of a relay, in practical use one would have control of the Engine Control Unit and would simply skip a number of injector pulses.

If the engine is running rich before the injector interruption, the oxygen sensor will transition to lean after the delay time has passed, as shown in Figure 3. A simple algorithm can be used to measure this time difference on-line. In this case, at an engine speed of 2011 RPM and an intake manifold pressure of 50.0 kPa, the delay was 0.112 s or 15.0 injections.

This procedure was repeated periodically while driving in order to cover the range of operating conditions typically encountered. The delay curves in terms of time and injection cycles are shown in Figures 4 and 5, respectively. From Figure 5, notice that the curve is

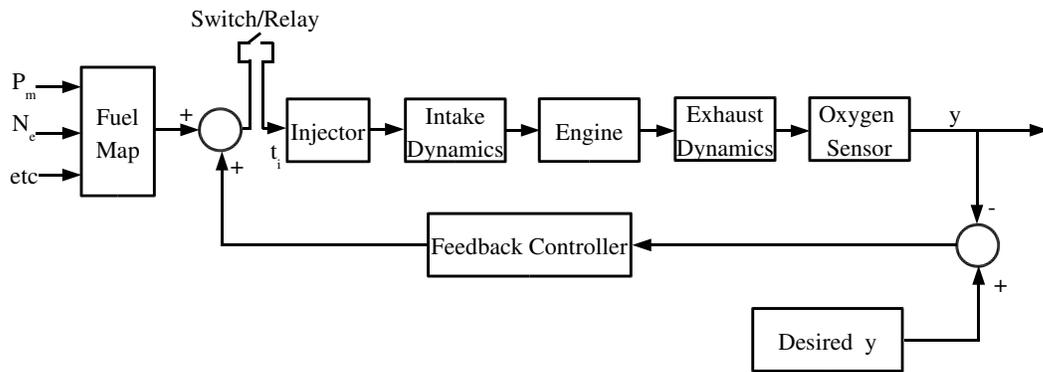


Figure 2: Block diagram showing the minor changes necessary to measure the delay. A switch or relay is used to interrupt the injection signal.

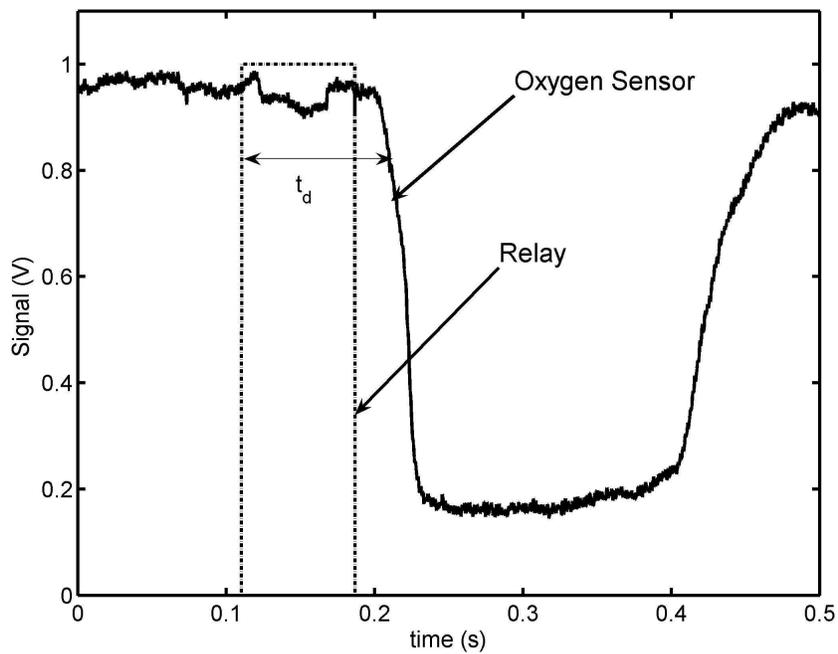


Figure 3: When the injection signal is cut off momentarily, the oxygen sensor transitions from rich to lean after the delay time, t_d has passed. The dashed line shows the relay signal: when the signal is high, the injection signal is interrupted. The oxygen sensor signal is shown by the solid line, signals above approximately 0.5 V signify a rich mixture, while those below signify the mixture has transitioned to lean. This data was recorded from a GM 6L Vortec engine running on natural gas at 2011 RPM and an intake manifold pressure of 50.0 kPa.

not flat, demonstrating the potential error of using the simplified model giving a delay of a constant number of injections. This data set includes 698 data points, recorded from approximately 30 minutes of driving. The data in Figure 5 was fit to a linear curve of

$$k_d = (2.33 \times 10^{-3} \text{ samples/RPM})N_e - \dots \\ (0.1404 \text{ samples/kPa})P_m + \dots \\ 16.96 \text{ samples} \quad (10)$$

This curve was converted to time and is also shown in Figures 4.

4 Conclusion

This paper has demonstrated that the delay between fuel injection and oxygen sensor feedback can be determined automatically in an on-line manner without the need of any specialized equipment. Additionally, it was shown that since the intake volume on modern sequential injection engines is significantly smaller than with throttle body injection or carburetted engines, caution must

exercised when using the standard models which oversimplify the exhaust model.

5 Acknowledgements

The authors would like to acknowledge the support of NSERC, in the form of a PGS D Scholarship, as well as the Saskatchewan Research Council for providing equipment and expertise. Thanks are also given to Natural Resources Canada and Precarn Inc who funded initial work in this area.

References

- J.B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
- P. Kaidantzis, P. Rasmussen, M. Jensen, T. Vesterholm, and Elbert Hendricks. Robust, self-calibrating lambda feedback for SI engines. Technical Report 930860, SAE, Warrendale, 1993.

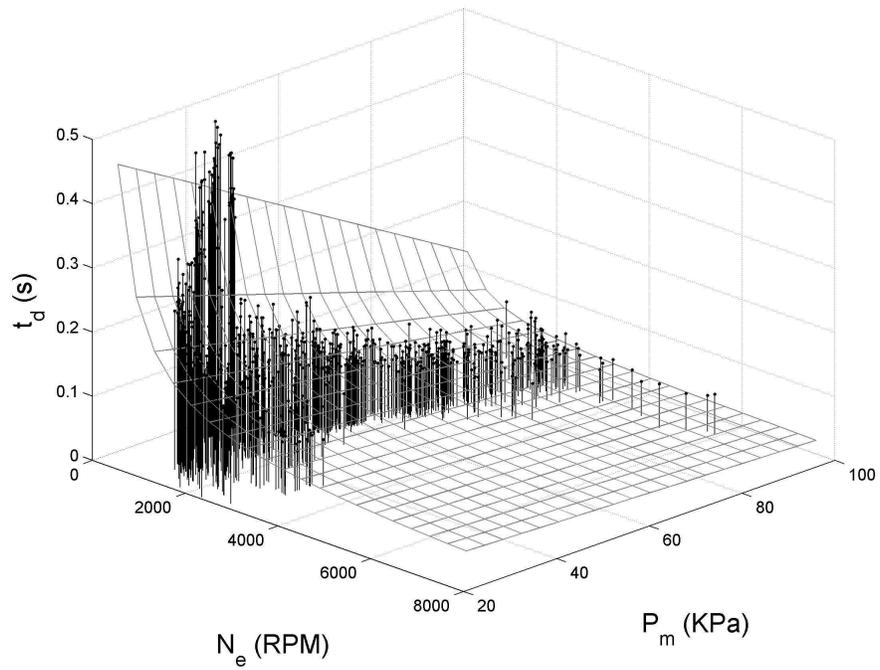


Figure 4: Curves showing the measured delay (in terms of time) over the range of engine speeds and intake manifold pressures typically experienced. The mesh surface shows a linear curve fit to the data in Figure 5

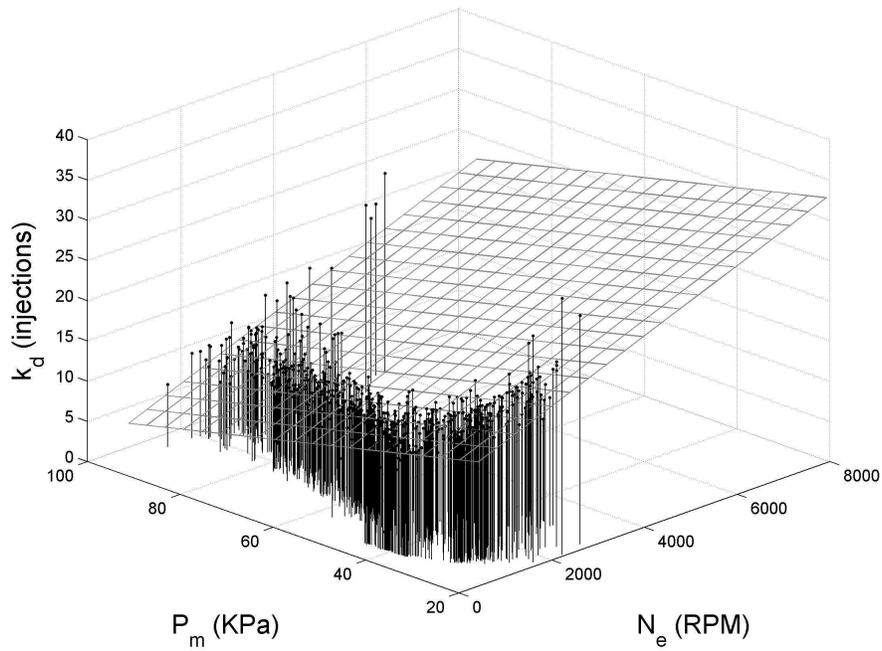


Figure 5: Since engine control units typically sample data once per injection, rather than at a constant sampling frequency, the delay data in Figure 4 can be expressed more usefully in terms of injections, as shown here. The model presented in Kaidantzis et al. [1993] predicts a constant number of injections while varying engine speed and manifold pressure, which is not the case here. Note that the axes have been exchanged for clarity.

CHAPTER 5

INTELLIGENT FUEL AIR RATIO CONTROL OF GASEOUS FUEL SI ENGINES

Published as:

- Travis Wiens, Greg Schoenau, Rich Burton, Mike Sulatisky, “Intelligent Fuel Air Ratio Control of Gaseous Fuel SI Engines.” Technical Paper MISC-168, Saskatchewan Research Council, Saskatoon, 2006.

5.1 Objectives

The objective of this report is to present, in detail, the control scheme that was developed in this research program. The control algorithm is described and the results of a simulation study are presented.

5.2 Approaches

The fuel controller presented in this report has two parts: a system model and an inversion algorithm. The model takes inputs from sensors in order to determine the operating point and predict the system response. The inversion algorithm then optimizes the control action by using the model to determine the best control action to achieve the desired response.

The system model takes inputs of intake manifold pressure, P_m , engine speed, N_e , and injector pulse width, t_i , and attempts to estimate the binary signal of the oxygen sensor in the exhaust manifold, y (the approximately binary EGO signal is rounded to 0 or 1 to make a pure binary signal). While there are other parameters that could be included, such as temperatures, these are typically slowly changing and the model will change to track these slow variations.

The model has two parts, separating dynamic and steady state responses. The dynamic part is modelled as a pure time delay. The value of the delay is not a single value, but a range between a minimum and maximum number of samples. This part of the model is determined by applying a step to the injector pulse width and determining how many samples elapse before the sensor toggles values. This must be repeated at a range of operating points to determine the range of delays. Refer to Chapter 4 for details. It should be noted

that this simple dynamic model is dependant upon the engine using gaseous fuels, as the evaporation and condensation of liquid fuels contribute additional complexity to the dynamic model. This is not to say that the approach introduced in this report can not be used with liquid fuel, but it may require a more complex model, especially during warm-up or cold weather operation.

The second part of the model is the steady state model. This part is a nonlinear model which is expected to classify whether the fuel-air mixture *at the injection point* is rich or lean. This simulates the ideal situation where the oxygen sensor could be mounted in the intake. The model used is a neural network, shown in Figure 5.1, which takes inputs of intake manifold pressure, engine speed, and injector pulse width. These values are scaled to a range of 0 to 1 to make the input vector \mathbf{u} , which also includes a constant value of 1 to allow for a bias. The trained network output, \hat{y} is a binary (0 or 1) value, corresponding to whether the mixture is estimated to be rich or lean.

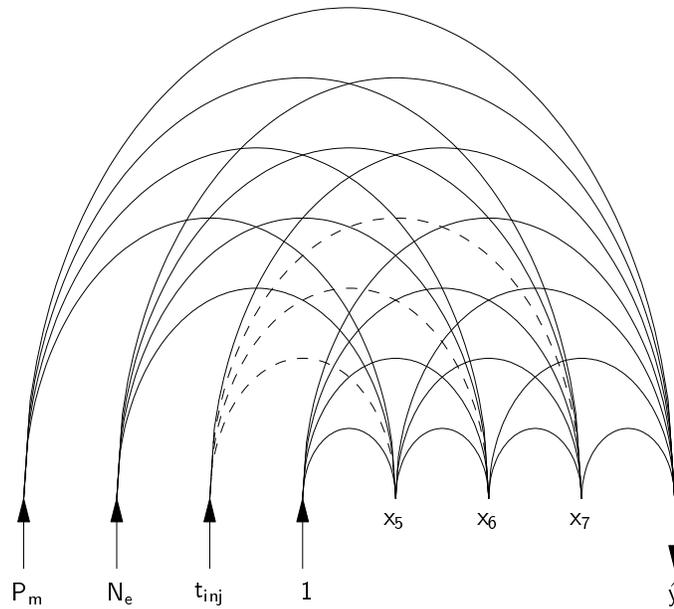


Figure 5.1: The network used for the model, showing inputs of scaled engine speed, N_e , manifold pressure, P_m , injection pulse width, t_i and the output, which is a model of the steady state binary sensor measurement. The dashed weights are constrained to zero to allow for inversion of the network (see Chapter 6).

This part of the model is identified using online backpropagation [10]. The delay model is used to match the measured EGO reading, $y(k)$, with the delayed inputs. However, since the delay is a range of values, it is not known which of the inputs to match the output to. It is still possible to compensate for the delay due to the binary nature of the output. For example, consider Figure 5.2, in which a step change is applied to the input, t_i . After a delay of five samples, the output, y , responds to the change. If the model estimate of the delay is a range between four and eight samples, input point A may correspond to any of the output points A_1 through A_5 (and the same for B and B_1 through B_5). It is unknown exactly which of A_1 to A_5 , but it doesn't

matter, since their values are all equal, so this value may be used for training. However, since the values of B_1 through B_5 vary, it is not possible to know which value to use so point B is skipped during training.

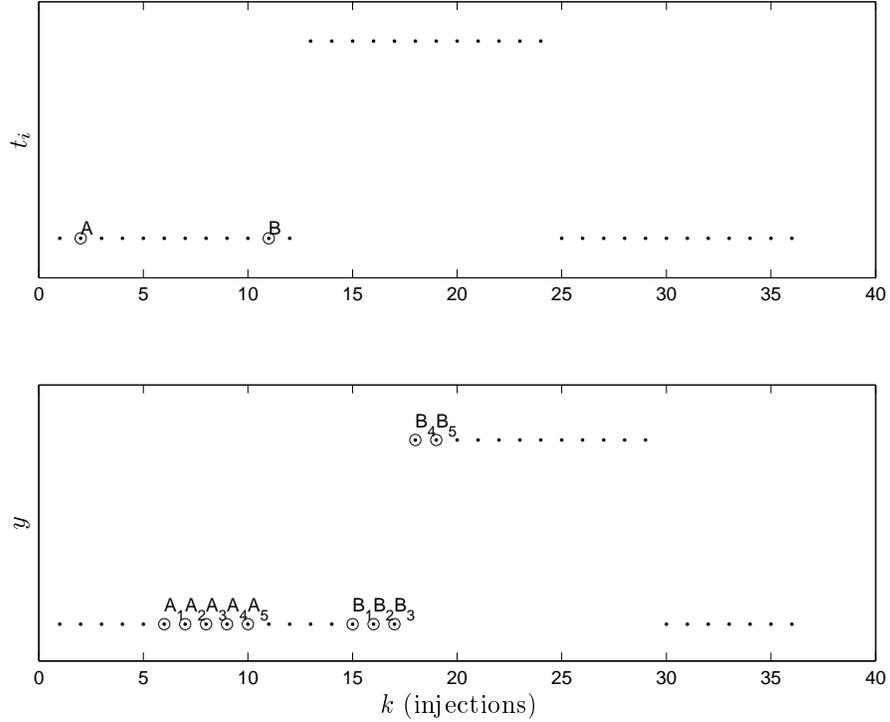


Figure 5.2: Compensating for delay may be achieved even if the exact delay isn't known. For the example above, the delay is estimated to be some value between four and eight samples. The input point A may then correspond to any of outputs A_1 through A_5 and B may correspond to any of points B_1 through B_5 . Since the possible values to match to A are all the same (A_1 through A_5) it doesn't matter which value is the true value and training can proceed. However since B_1 through B_5 are not constant, point B cannot be used for training, as is not possible to determine which value to use.

If a value of $y_i(k)$ is determined, an error is calculated as

$$E = (\hat{y}_i(k) - y_i(k))^2. \quad (5.1)$$

This error is then propagated through the network to determine the proper change to the network weights, as in standard back-propagation. Details of the actual implementation may be found in the code of Appendix C.

Once the model parameters have been identified, the model is then used to determine an estimate of the optimal fuelling for the engine. This is performed by giving the neural network operating point inputs P_m and N_e and determining the injector pulse width, t_i which results in a stoichiometric mixture in the intake. This occurs when the network output is midway between the rich and lean values, when $\hat{y} = 0.5$. Since neural networks are nonlinear and, especially in classification problems such as this, the output can

be flat over large areas separated by a sharp transition, finding this transition can be difficult. Typically, a bisecting binary search algorithm must be performed to find it, requiring a large number of iterations. For example, if a 32-bit integer is used for the value, as many as 32 iterations will be required to find the solution. Unfortunately, this problem is a real-time application since each calculation *must* be calculated before the next injection is to occur. For the 8 cylinder engine used in this study, at 6000 RPM the entire control algorithm must be completed every 2.5 ms. The microcontrollers used in engine controllers are not very fast and cannot complete an iterative solution in this time.

To accommodate the time constraint, it would be preferable to be able to find an algebraic inverse solution to the problem. Unfortunately, for standard neural networks this is impossible. However, by constraining certain weights to zero, the author found that it was possible to find an analytical solution, as shown in Chapter 6.

This report also presents a method of initializing the neural weights, so that the engine can be started with as little prior information as possible. The initial fuel map can be determined, only requiring the maximum engine speed and full scale intake manifold pressure transducer reading.

5.3 Results

A simulation study was performed to determine the feasibility of the control algorithm. A model of the 6.0L V8 Vortec engine was used, as shown in Figure 5.3 (the same engine used for experimental tests in Chapters 4 and 8). A truck with this engine was driven through a typical driving cycle, including both city and freeway driving. The measured engine speed and intake manifold pressure were used as inputs for the simulation.

As shown in Figure 5.4, the simulated relative air-fuel ratio, λ , initially deviates between 0.7 and 1.3. This would correspond to a poorly running engine, but it would likely not stall. However, the learning function reduces the error considerably; within 60 minutes, the root mean squared error in λ is 0.02 which further reduces to a steady state value of 3×10^{-4} . The verification error was also calculated, based on the error in other data. This shows that the controller is learning the characteristics of the plant, not merely tracking the output. One further important result is that the fuel-air ratio exhibits a rich-lean limit cycle (as shown in Figure 5.4), which is required for proper catalyst operation.

5.4 Contributions

This report presents a number of original and significant contributions. The over-arching contribution is the online learning neural fuel-air controller, which, to the author's knowledge, had not been previously achieved. This controller also includes novel features such as an invertible neural network which is further described in Chapter 6, and a method of generating the initial fuel map using a bare minimum of information about the engine.



Figure 5.3: The engine used for experimental and simulation studies in this dissertation was a 2003 Vortec 6.0L V8 bi-fuel (compressed natural gas or gasoline) engine. This image shows the 2006 version of the gasoline version; the engine used is very similar. Used by permission of General Motors.

5.5 Erratum

Equation (8) should read

$$\Phi(P_m, N_e, t_i) = W_5 x_5 + W_6 x_6 + \dots + W_{N-1} x_{N-1}$$

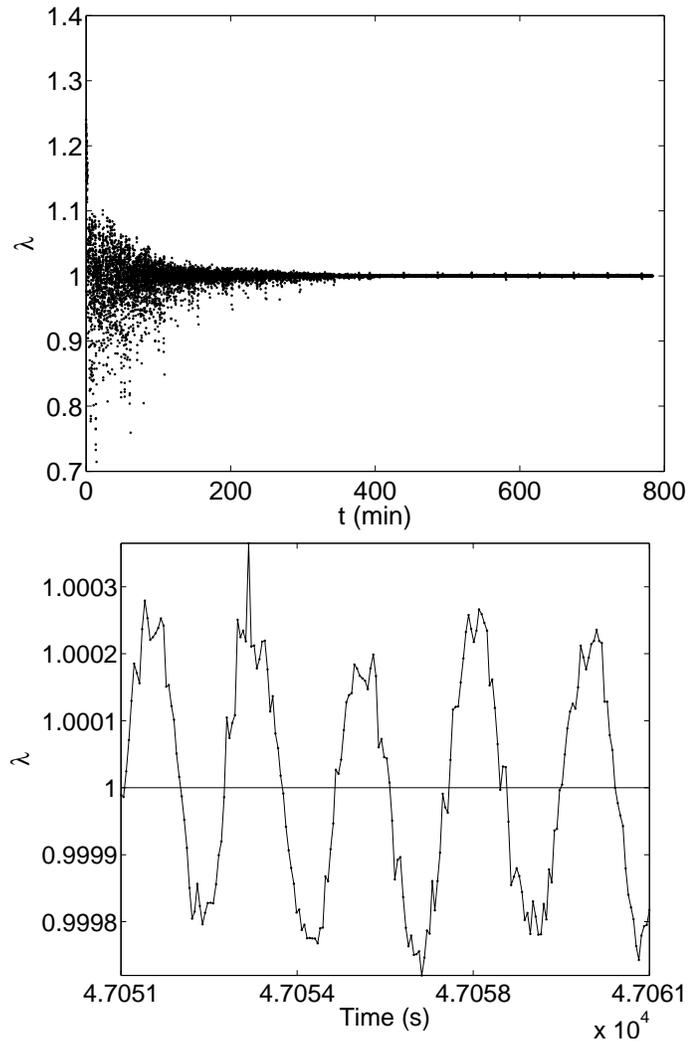


Figure 5.4: Simulated lambda for the entire test (top). The initial lambda has an error of approximately 0.3. However, after an hour the error is 0.02, which eventually converges to a steady state error of 3×10^{-4} . The limit cycle of the last 100 points (bottom) is desired as it is required for proper three-way catalytic converter operation.

INTELLIGENT FUEL AIR RATIO CONTROL OF GASEOUS FUEL SI ENGINES

Travis Wiens, Greg Schoenau, Rich Burton
 Department of Mechanical Engineering
 University of Saskatchewan
 Saskatoon, Sk, Canada
 email: t.wiens@usask.ca

Mike Sulatisky
 Alternative Energy Products
 Saskatchewan Research Council
 Saskatoon, Sk, Canada

ABSTRACT

This paper presents a new architecture for fuel control in spark ignition (SI) engines, particularly those burning gaseous fuels. The controller continually updates an internal model of the engine, which it then optimizes to provide the ideal fuel injection time for any operating point. This paper presents simulation results based on a model of a V8 Vortec engine running on natural gas. The simulated controller is able to improve the control of relative fuel air ratio (λ) from between 0.7 and 1.3 upon initialization to a root mean squared error in λ of 0.02 within 60 minutes of driving, eventually converging to a steady state error of 3×10^{-4} .

KEY WORDS

Neural networks intelligent control automotive

1 Introduction

A major portion of a vehicle's ECU (engine control unit) is devoted to maintaining the proper ratio of fuel to air in the cylinder charge mixture [1]. Deviation from the optimum fuel-air ratio (also known as the FAR or FA ratio) can lead to reduced power and fuel efficiency, and increased emissions [2] [3]. In a typical ECU, transient fuel-air control function is achieved through a look-up table (referred to as a fuel map). Generating this fuel map is typically an expensive process, as it takes many hours of calibration on expensive equipment that must be repeated for every different engine model. In some cases, the fuel map has a basic form of adaptation to changes in properties from vehicle to vehicle and over time; often, though, it is static in nature.

The objective of this paper is to present a new method of continually adaptive fuel control, which is capable of controlling the fuel-air ratio to a high precision, while eliminating the time-consuming and expensive initial calibration period.

2 Problem Definition

The goal of a fuel-air controller is to control the fuel-air mass ratio of the mixture entering the engine, typically at

or near the stoichiometric¹ point for the fuel. It does this by measuring a number of parameters and calculating the length of time the fuel injector should be open for each intake stroke of the engine. Since gaseous fuels do not have evaporation effects, the intake fuel-air mixing transfer function (the plant), shown in Figure 1, has little dynamic effects and can be approximated by a non-linear gain. In a general form, the injection pulse width, $t_i(k)$, and intake relative fuel-air ratio, λ_i , can be expressed in the open loop system form as a nonlinear function of several parameters as

$$t_i(k) = G_c(P_m(k), N_e(k), T_i(k), T_c(k), P_{atm}(k), \dots) \quad (1)$$

$$\lambda_i(k) = G_p(t_i(k), P_m(k), N_e(k), T_c(k), P_{atm}(k), \dots) \quad (2)$$

where $G_c(\cdot)$ is a general control strategy taking into account operating conditions including intake manifold pressure, P_m and engine speed, N_e ; $G_p(\cdot)$ is the plant, a static function of the operating conditions, which may include additional parameters such as intake temperature, T_i , coolant temperature, T_c and atmospheric pressure, P_{atm} . All states and parameters are sampled once per intake stroke, with index k . Note that the sampling time varies with engine speed.

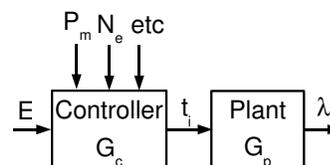


Figure 1. Open loop signal flow diagram of controller and plant including inputs of an error based on the desired air-fuel ratio, manifold pressure, engine speed, and other slowly varying operating point variables, such as temperatures. The goal of a controller is to control the air-fuel ratio in the intake.

If one can determine the inverse of G_p with respect

¹A mixture with a stoichiometric fuel-air ratio has just enough oxygen for complete combustion.

to λ_i and t_i at any operating point, the control problem is solved. That is, use

$$G_c = G_p^{-1} \quad (3)$$

Unfortunately, this is not easy to do, particularly because it is not possible to measure λ_i directly. Rather than measuring the air-fuel ratio in the intake, it is measured at the exhaust. This imposes a number of dynamic effects on the system: the transport delay as the mixture passes from intake to exhaust, and the dynamics of the sensor. Additionally, the exhaust gas oxygen (EGO) or heated oxygen sensors (HO2S) that are typically used on production vehicles are not at all linear, exhibiting a strong ‘‘bang-bang’’ characteristic. Therefore, the system has significant nonlinear dynamics, as shown in Figure 2, where

$$y(k) = H(\lambda_i(k, k-1, k-2, \dots), \lambda(k-1, k-2, \dots)) \quad (4)$$

with $H(\cdot)$ representing a nonlinear function of the relative fuel-air ratio at the intake, λ_i , and exhaust, λ . This is a considerably more difficult control problem, and generally requires a good deal of a priori knowledge of the plant and sensors in order to develop a functional traditional controller.

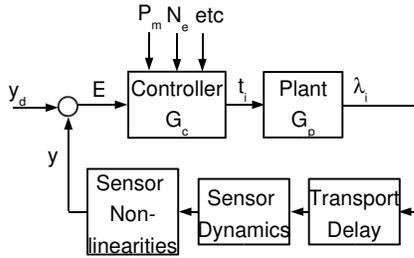


Figure 2. Signal flow diagram of system including sensor dynamics.

3 Proposed Solution

The solution proposed in this paper uses a type of inverse model control to tackle the problem. An inverse model controller first generates a model of the plant (G_p) using system identification techniques. An optimization scheme is then used on the model in order to determine the input (control action) which will give the desired output. In the specific case of a fuel-air controller, an outline of the control sequence is as follows

1. Measure intake manifold pressure, $P_m(k)$, engine speed, $N_e(k)$, injection pulse width, $t_i(k)$, and oxygen sensor output, $y(k)$ for the last injection period, k .

2. Update internal engine model, which predicts the EGO output, $\hat{y}(k)$ for the history of operating points and inputs (P_m , N_e , and t_i).
3. Measure $P_m(k+1)$, $N_e(k+1)$ for the next injection cycle.
4. Find $t_i(k+1)$, such that the predicted internal model output \hat{y} is at the desired point (typically the stoichiometric point).
5. Open the injector for a period of $t_i(k+1)$.
6. Go to step 1.

This outline allows one to divide the problem into a number of subproblems:

1. What form of internal model to use.
2. How to update the model as new data is measured.
3. How to perform the optimization of the model.

A solution to each of these problems will be presented in the following subsections.

3.1 Model Form

This section deals with the problem of predicting the oxygen sensor output, y for any operating point of intake manifold pressure, P_m , engine speed, N_e , and injector pulse width, t_i . Since the oxygen sensor is a bang-bang sensor, this can be viewed as a classification problem: identify the rich and lean operating points.

Since the plant is nonlinear, a form of neural network was selected to model the steady state output. A neural network is a function approximation or classification tool which uses a large number of very simple processing elements (neurons) which can have very complex global characteristics. In this case, each neuron is simply the weighted sum of the inputs, passed through the sigmoidal function $(1 - e^{-x})^{-1}$.

The neurons are then connected together to form a network. The type of network presented in this paper is a general neural network, originally presented by Werbos [4]. This network has a string of neurons, each of which has inputs of the network inputs (P_m , N_e , and t_i , scaled to the range (0,1)) and the outputs of each neuron before it, as shown in Figure 3. The last neuron output is the network output. This form of network has the advantage of simplicity; the architecture of the entire network is defined by one parameter: the number of neurons. Also, since the output neuron is connected to the inputs, it is easy to explicitly initialize the network to a linear function (this will be discussed in the next section).

The neural network is used to model the static nonlinear gain of the system. The transport delay must also be identified. This can be achieved by shutting off the injectors and timing how long it takes for the oxygen sensor to transition from rich to lean [5].

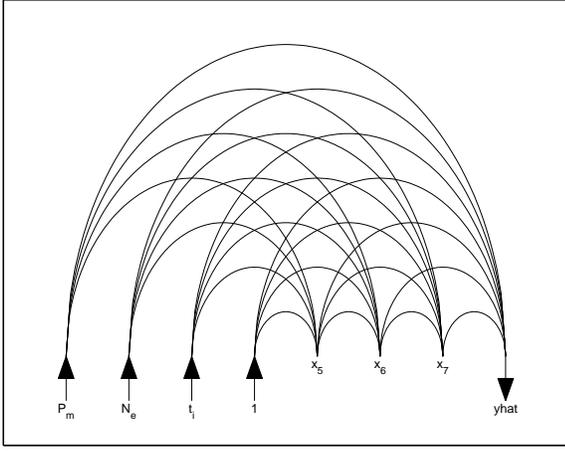


Figure 3. A Generalized Neural Network (GNN) is a string of neurons, shown here with inputs on the left and outputs on the right. Each neuron is connected to every neuron to left of it. This architecture eliminates the need to determine how many layers to use and how many neurons should be in each layer.

3.2 Model Weight Selection

The network presented in the previous section is capable of classifying a wide range of functions. The problem is selecting the weights to give the desired output. In this case, the goal is to minimize the error between the model output and the steady state output of the plant (the engine). Before one can do this, however, the transport delay must be dealt with. The procedure presented in a previous paper[5] to identify the delay does not often return a constant delay for all cylinders and operating points. Rather, a range of delays is identified, due to differing exhaust header lengths and exhaust densities. While it may be possible to construct a model to identify the delay for each cylinder and operating point, being able to identify the range of delays is sufficient to train the network.

Once the range of delays has been identified, one must match the input to the corresponding output. Since the delay is a range of values, each input may correspond to a number of outputs, as shown in Figure 4. This is acceptable because of the bang-bang characteristic of the sensor. The vast majority of measurements will be the rich or lean measurements, with very few falling in between. One then just needs to look for points where the output is constant over the range of delays, and ignore the points where the output changes. For example, the point *A* in Figure 4 may correspond to any of A_1, A_2, \dots, A_5 , but it doesn't matter which, since they are all equal. This point may then be used for training, while point *B* must be discarded since B_1, B_2, \dots, B_5 are not all equal.

Once the input has been matched to an output (or discarded), regular on-line backpropagation with momentum is used to update the weights [4].

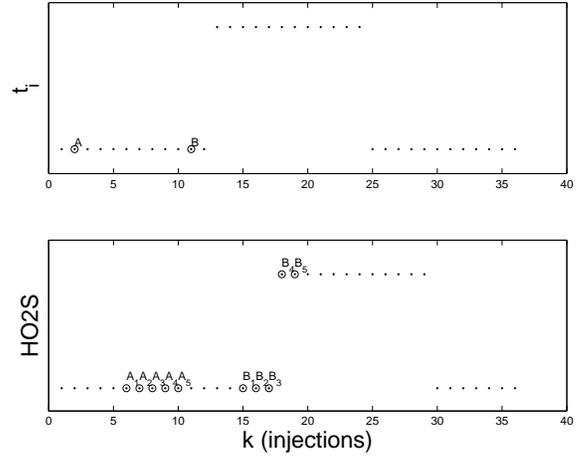


Figure 4. Due to uncertainty in the delay, point *A* in the injection pulse width may correspond to any of oxygen sensor outputs A_1, A_2, \dots, A_5 . Point *A* may still be used for training, since A_1, A_2, \dots, A_5 are all equal, but *B* must be discarded since B_1, B_2, \dots, B_5 are not all equal

3.3 Model Inversion

The purpose of creating a model is so that it can be used to choose the proper injection time for the engine. This is an optimization problem of determining what input to the network (t_i) can produce the desired output (\hat{y}) given the operating condition (P_m and N_e). There are a number of methods of approaching this problem. One is to use an iterative numerical solution, such as Newton's method. Unfortunately, since the function that is being modeled has two nearly level plateaus separated by a sharp transition, most numerical methods require many iterations, which is not acceptable for a realtime application with a fast sampling rate such as an engine. The other approach is to attempt to find a closed form inverse of the neural network. This is generally not possible due to the non-linearities involved, but in this particular case a number of special conditions can be exploited to make this possible.

First, examine the equation for the output neuron in the network

$$\hat{y} = \text{sig}(W_1x_1 + W_2x_2 + W_3x_3 + W_4x_4 \dots + W_5x_5 + \dots + W_{N-1}x_{N-1}) \quad (5)$$

where W_n is the weight on the n^{th} input x_n , in a network with N neurons, and $\text{sig}(x) = (1 - e^{-x})^{-1}$. If one sets up the network as shown in Figure 3, $x_1 = P_m$, $x_2 = N_e$, $x_3 = t_i$, (all scaled to the range (0,1)), and $x_4 = 1$ (for a bias) and the output is scaled such that $y = 0$ is lean and $y = 1$ is rich. Equation 5 now becomes

$$\hat{y} = \text{sig}(W_1P_m + W_2N_e + W_3t_i + W_4 \dots + W_5x_5 + \dots + W_{N-1}x_{N-1}) \quad (6)$$

If the desired output is the stoichiometric point, one is attempting to solve equation 6 for t_i , given $y = 0.5$ and

the measured P_m and N_e . If one applies the inverse sigmoid function to both sides and performs some algebraic manipulation, equation 6 becomes

$$t_i = \frac{-1}{W_3}(W_1 P_m + W_2 N_e + W_4 \dots + W_5 x_5 + \dots + W_{N-1} x_{N-1}) \quad (7)$$

and simplifying the nonlinear hidden neuron outputs as the sum

$$\Phi(P_m, N_e, t_i) = W_5 x_5 + W_6 x_7 + \dots + W_{N-1} x_{N-1} \quad (8)$$

this equation can be written as

$$t_i = \frac{-1}{W_3}(W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e, t_i)) \quad (9)$$

This form cannot be simplified further because Φ is a nonlinear function of t_i . However, it is possible to force each hidden neuron's weight for t_i to zero, as shown in Figure 5. While this means that the output neuron will be linear with respect to t_i , the classification boundary is still a nonlinear function of P_m and N_e . Thus, the closed form solution is

$$t_i = \frac{-1}{W_3}(W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e)) \quad (10)$$

which can be viewed as a linear solution, augmented by a nonlinear neural network with weights selected from the model network.

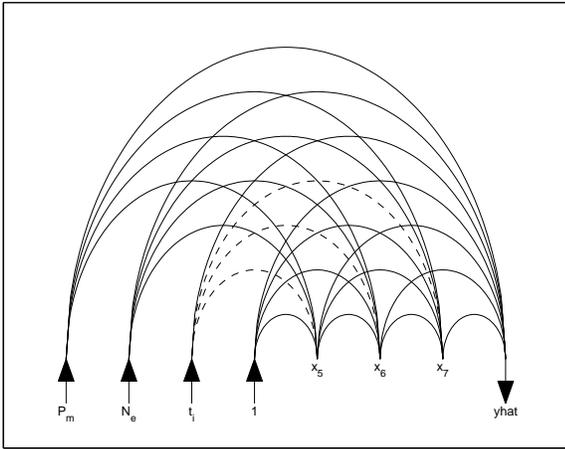


Figure 5. The network used in this problem, with weights connecting t_i to the hidden neurons set to zero (shown by dashed lines).

The fact that t_i can be written as a linear function of P_m and N_e with a nonlinear augmentation term has an additional benefit. The network can be initialized by setting W_1, W_2, W_3 , and W_4 such that the initial (linear) fuel map is plausible. Some a priori knowledge can be applied to picking a good starting point. In most engines, the stoichiometric injector pulse width is approximately proportional to the intake manifold pressure, with a much smaller

contribution from engine speed. Additionally, injectors are usually sized such that they are nearly continually open at the maximum manifold pressure (atmospheric in a naturally aspirated engine) and engine speed. Furthermore, the pressure transducer's output at this maximum pressure will typically be near its full scale reading. Therefore, the initial network was initialized such that the network gives a fuel map of

$$t_i = \frac{(60 \text{ [s/min]})(2 \text{ [rev/injection]})}{K N_{e \text{ max}} P_{FS}} P_m + 0 N_e + 0 \quad (11)$$

where $N_{e \text{ max}}$ is the maximum engine speed [RPM], P_{FS} is the full scale reading of the intake manifold pressure transducer, and K is a constant to take into account the fact that the injectors are not continually open at $N_{e \text{ max}}$ and the pressure transducer will not be quite at its full scale reading at the maximum operating pressure.

The nonlinear neurons (those contributing to Φ) were initialized using Nguyen's method [6], but their contribution to the initial output was kept small by initializing their corresponding weights in the output neuron to small random values.

4 Simulation Results

The control scheme outlined in the previous section was implemented in C and used to control one bank of a simulated General Motors 6L V8 Vortec engine with sequential port injection running on natural gas.

A typical driving cycle was recorded during actual driving, including both city and freeway portions. This driving cycle was then fed to the controller/plant simulation. The network used had 15 hidden neurons and the learning rate was set at 5×10^{-4} for the linear weights of the output neuron, 5×10^{-5} for the nonlinear output neuron weights and 1×10^{-2} for the hidden neurons. The momentum term in the training scheme was 0.3. These values were determined through trial and error and are not optimized. The selected learning rates give a slowly converging, but accurate result. Training speed can be greatly improved at the cost of increased steady state error. The estimated range of delays was previously determined [5] to be between 4 and 24 injection events.

Figures 6 through 12 show the simulated results. Figure 6 shows the relative air-fuel ratio during training (note that the controller only had access to the binary EGO output, but not λ). The controller starts out with larger deviations from the stoichiometric point (λ between 0.7 and 1.3) but converges and begins rich-lean oscillations with decreasing amplitude, as required for proper catalytic converter operation. After 60 minutes of running while training, the root mean squared error in λ has been reduced to approximately 0.02, and at steady state it converges to approximately 3×10^{-4} , as shown in Figure 7. The controller is clearly learning the characteristics of the plant. Figures 8 and 9 shows the progression of the verification error in

the pulse width and λ . This is the root mean squared error for 100 operating points, either randomly selected points from the driving profile or evenly spaced in a grid over the possible operating range. This shows that the controller is *learning* the proper fuel map rather than *tracking* it, as a typical feedback controller would.

Figure 10 is the fuel map after training. While this surface appears linear, it does capture the nonlinear elements of the engine. This can be demonstrated by the fact that the mean error of 100 points from the driving cycle (shown in Figure 11) is approximately 4×10^{-6} s, while the best-fit linear curve has an RMS error of 9.6×10^{-6} s. Figure 12 is a map of the variation in λ over a range of operating conditions.

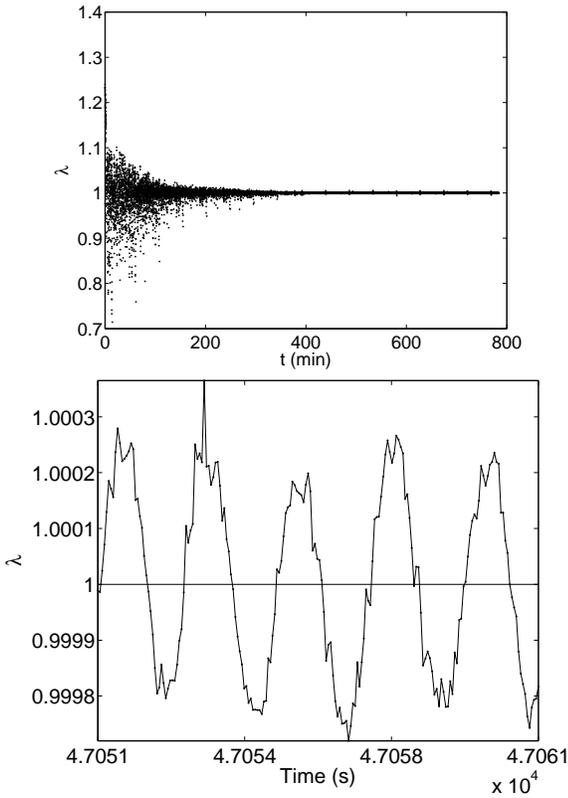


Figure 6. Simulated relative air-fuel ratio λ for the entire driving profile (top) and for the last 100 injections (bottom)

5 Conclusion

This paper presents a simulation of an algorithm which can successfully control an internal combustion engine running on natural gas. The simulated engine initially has large deviations from stoichiometry (λ between 0.7 and 1.3), but improves as the engine is run, such that after 60 minutes of driving the mean error in λ was 0.02, decreasing to 3×10^{-4} at steady state. This algorithm only requires one forward and one backward pass through the algorithm for training (using backpropagation) and one forward pass to determine

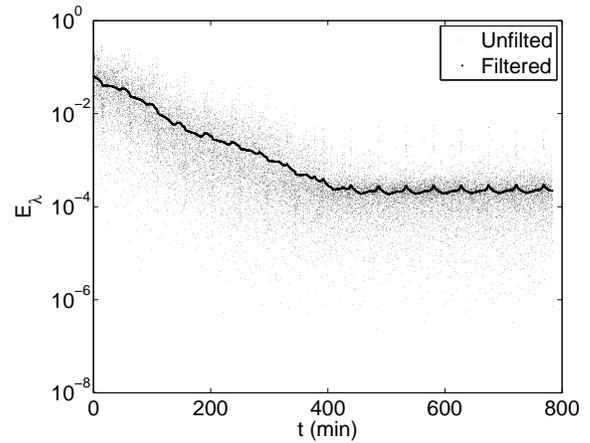


Figure 7. Training error for the controller. This is the instantaneous absolute value of $1-\lambda$ for the data shown in Figure 6.

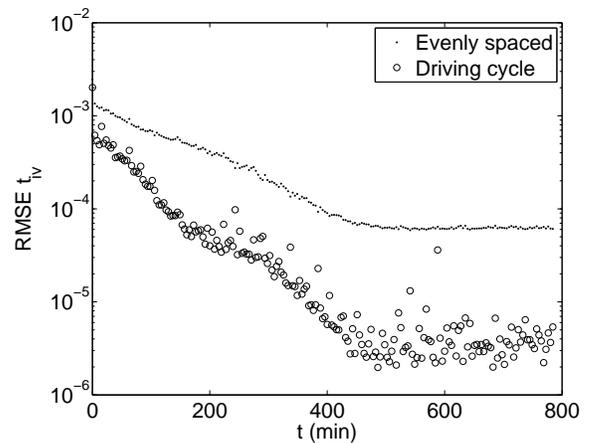


Figure 8. Verification error for the controller. This is the root mean squared error between the stoichiometric pulse width and the controller pulse width for 100 different operating points (no training occurs during verification). Two data sets were used: one was randomly selected from the driving profile and the other is an evenly spaced grid of points covering the possible operating range. This shows the ability of the controller to learn the characteristics of the engine, rather than simply tracking the optimum output for the operating conditions in the recent past.

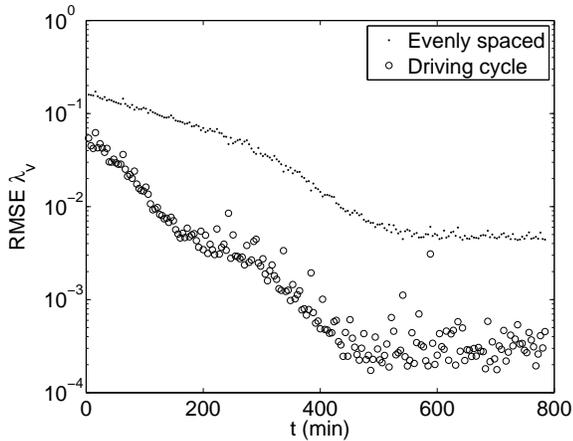


Figure 9. Verification error for the controller. This is the RMS value of $1-\lambda$ for the same data as Figure 8

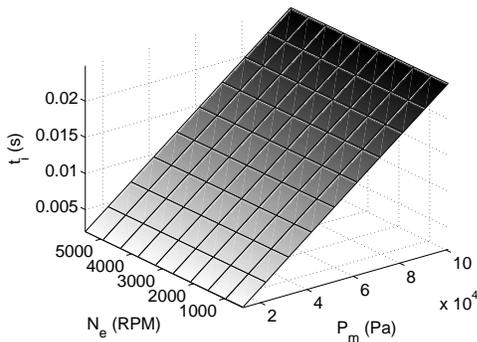


Figure 10. Fuel map showing injector pulse width for the controller at the end of training. Note that while this surface appears linear, it does include the non-linearities in the engine.

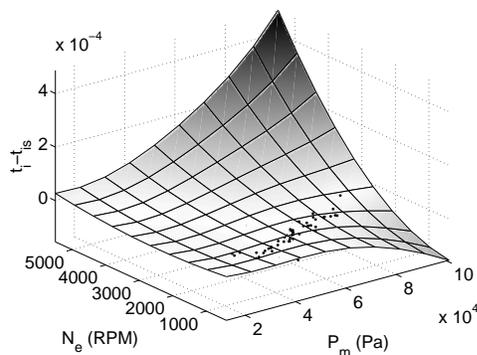


Figure 11. Error map for Figure 10. This is the difference between the fuel map and the stoichiometric fuel map. Dots mark a sample of the training data.

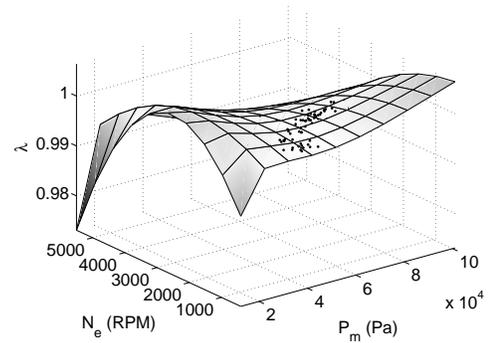


Figure 12. Map of resultant relative air-fuel ratio for the fuel map shown in Figure 10. Dots mark a sample of the training data.

the optimum fuelling, allowing it to be implemented in real time on readily available hardware.

6 Acknowledgements

The authors would like to acknowledge the support of NSERC, in the form of a PGS D Scholarship; the Saskatchewan Research Council for providing equipment and expertise; and General Motors Alternative Fuels for the loan of the test vehicle. Thanks are also given to Natural Resources Canada and Precarn Inc who funded initial work in this area.

References

- [1] R. Isermann and N. Muller. Design of computer controlled combustion engines. *Mechatronics*, 13:1067–1087, 2003.
- [2] Charles Fayette Taylor. *The Internal-Combustion Engine in Theory and Practice*, volume I. M.I.T. Press, Cambridge, second edition, 1985.
- [3] J.B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
- [4] P. Werbos. Backpropagation Through Time: What It Does and How to Do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.
- [5] T. Wiens, R. Burton, G. Schoenau, M. Sulatisky, S. Hill, and B. Lung. Experimental determination of transport delay in a spark ignition engine. Technical report, Saskatchewan Research Council, Saskatoon, 2006.
- [6] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values for the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, pages 21–26, San Diego, 1990.

CHAPTER 6

ALGEBRAIC INVERSION OF AN ARTIFICIAL NEURAL NETWORK CLASSIFIER

Published as:

- Travis Wiens, Rich Burton, Greg Schoenau, “Algebraic Inversion of an Artificial Neural Network Classifier” Proceedings of the ESANN 2007, Bruges, 2007.

6.1 Objectives

This paper presents a method of designing a neural network such that it may be algebraically inverted. This allows one to find the boundary between classes with a calculation that takes approximately the same time as one forward calculation of the network.

6.2 Approaches

If a neural network is used to classify data points into two or more classes, it is often important to be able to determine the location of the boundary between classes. Previously, it was necessary to find the boundary iteratively. Since the network output will usually have large flat areas (representing the classes) separated by relatively sharp transition between classes, gradient methods (e.g. Newton’s Method) are not useful. Using the bisection search algorithm would take 32 iterations to find the transition to the accuracy of a 32-bit integer [16], meaning that the new algebraic inversion method is approximately 32 times faster than such an iterative method.

In order to invert the network algebraically, one starts with a static generalized neural network [10]. This is a network with the maximum number of possible connections between neurons without recurrence (feedback). Using the engine as an example, the output of the last neuron (the network output) is

$$\hat{y} = \text{sig}(W_1P_m + W_2N_e + W_3t_i + W_4 + W_5x_5 + \dots + W_{N-1}x_{N-1}) \quad (6.1)$$

where W_n is the weight on the n^{th} input x_n , in a network with N neurons, and $\text{sig}(x) = (1 - e^{-x})^{-1}$. The network inputs are scaled intake manifold pressure, P_m , engine speed, N_e , and injector pulse width t_i .

In order to find the stoichiometric pulse width, one needs to find a value for t_i on the boundary between the rich and lean classes at the operating point defined by N_e and P_m . Since the sigmoidal activation function varies between 0 and 1, this boundary occurs when $\hat{y} = 0.5 = \text{sig}(0)$. This requires the solution of the equation

$$0 = W_1 P_m + W_2 N_e + W_3 \hat{t}_{is} + W_4 + W_5 x_5 + \dots + W_{N-1} x_{N-1} \quad (6.2)$$

for the estimate of the stoichiometric pulse width, \hat{t}_{is} . If the contributions of the hidden neurons are grouped together in

$$\Phi(P_m, N_e, t_i) = W_5 x_5 + W_6 x_6 + \dots + W_{N-1} x_{N-1}, \quad (6.3)$$

then equation 6.2 can be rewritten as

$$\hat{t}_{is} = \frac{-1}{W_3} (W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e, \hat{t}_{is})). \quad (6.4)$$

Unfortunately, since Φ is a nonlinear function of \hat{t}_{is} , this is as far as one can progress algebraically.

In order to proceed, it is possible to set the hidden neuron weights connecting the input t_i to the hidden neurons to zero, as shown in Figure 5.1. This removes Φ 's nonlinear dependence on \hat{t}_{is} . The closed form algebraic solution to the problem is then

$$\hat{t}_{is} = \frac{-1}{W_3} (W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e)). \quad (6.5)$$

Constraining these weights to zero does have consequences that must be considered. The most important is that the neural network must be monotonically increasing or decreasing with t_{is} , as a non-monotonic function would have two solutions. In the case of the fuel controller this is not a problem as there should only be one optimal injection time for each operating point.

Since the most important feature of this method of neural network inversion is its speed, some tests were performed to evaluate the time taken to perform the calculation. The controller described in Chapter 5 was modified to allow for timing of the inversion calculation. The accuracy that an iterative solver could achieve in the time allowed (since at 6000 RPM, the controller in a V8 engine must calculate an injection amount every 2.5 ms) was also determined.

6.3 Results

The network inversion of the network used was found to take 0.335 ms. This is well within the 2.5 ms constraint. If an iterative solution were to be used, only seven iterations would be allowed in this time window, leaving an error of 0.78%, which is unacceptable for the high precision task of fuel injection.

6.4 Contributions

As mentioned before, the author knows of no other published method of finding a closed form inverse for a neural network. This is a very significant contribution for anyone who is concerned with the time required

to find the class boundaries of a neural network. This can be important for real-time embedded applications or for offline tests with very large data sets which can take long times to complete.

Algebraic Inversion of an Artificial Neural Network Classifier

Travis Wiens, Rich Burton and Greg Schoenau *

University of Saskatchewan - Department of Mechanical Engineering
Saskatoon, Sk, Canada

Abstract. Artificial neural networks are, by their definition, non-linear functions. Typically, this means that it is impossible to find a closed-form solution for the inverse function of a neural network. This paper presents a special form of neural network classifier that allows for its algebraic inversion in order to find the boundary between classes. The control of the fuel-air ratio in a spark ignition engine is given as an example.

1 Introduction

An artificial neural network (hereafter referred to as simply a neural network) is a complex system, made of simple identical non-linear parallel elements [1]. Typically, this complexity and non-linearity do not permit one to find an algebraic solution for the inverse of a neural network; that is, to solve for what inputs will result in a particular output. This paper presents a form of neural classifier which permits one to solve this problem in order to find a closed-form solution for the boundary, under certain conditions. This is achieved by using a generalized neural network [2] with certain weights strategically set to zero. An example is used to illustrate the technique: the control of fuel-air ratio in a spark ignition (SI) engine.

2 Problem Definition

Static neural networks are often used as non-linear function approximators or classifiers. For example, in the control of a spark ignition engine, one may wish to classify whether the fuel-air ratio injected into the engine will be rich (too much fuel for complete combustion) or lean (too little fuel), given operating conditions of intake manifold pressure, P_m , engine speed, N_e , and the control action of fuel injector pulse width, t_i . In this case, a neural network model of the “plant”, G_p , would be set up as

$$\hat{y} = G_p(P_m, N_e, t_i) \quad (1)$$

where \hat{y} is an estimate of a two-state oxygen sensor output, with 0 signifying a lean air-fuel ratio and 1 rich. One would then take a sample of data and train the

*The authors would like to acknowledge the support of NSERC, in the form of a PGS D Scholarship; the Saskatchewan Research Council for providing equipment and expertise; and General Motors Alternative Fuels for the loan of the test vehicle. Thanks are also given to Natural Resources Canada and Precarn Inc who funded initial work in this area.

network weights in G_p so that the error between the estimated sensor output, \hat{y} and the measured sensor output, y is minimized.

Stoichiometric fuel-air control is based on maintaining the fuel-air ratio near the stoichiometric point, where there is just enough fuel for complete combustion, at the boundary between rich and lean. Thus, given operating points P_m and N_e , the controller must find the value for t_i on the “decision boundary”. This injection pulse width is called the stoichiometric pulse width, t_{is} , and the controller’s estimate of it is \hat{t}_{is} . The control function, G_c , is then the inverse of the plant function:

$$\hat{t}_{is} = G_c(P_m, N_e) = G_p^{-1} \quad (2)$$

or the solution of the equation

$$G_p(P_m, N_e, \hat{t}_{is}) = 0.5 \quad (3)$$

for t_{is} . This corresponds to the transition point between rich ($G_p = 1$) and lean ($G_p = 0$) operation.

There are a number of methods of inverting a nonlinear function, but typically they involve iterative numerical solutions, which are not suitable for real-time control. For example, for an 8-cylinder engine running at 6000 RPM, this calculation must be completed at a minimum of every 2.5 ms on processors with low computational power. The ideal solution would be to be able to algebraically invert the network.

3 Proposed Solution

The proposed solution to the problem outlined above involves the use of a special kind of neural network, known as a generalized neural network, or GNN [2]. Unlike the familiar multilayer perceptron (MLP), which has discrete layers, each of which is only connected to the previous layer, the neurons of a GNN are organized in a line, with each neuron connected to all the neurons to the left of it, as shown in Figure 1. Thus, for a network with four inputs (three inputs plus a 1 for bias), the first hidden neuron, x_5 , would have four inputs; the next neuron, x_6 , would have these four inputs plus the output of x_5 , and so on, such that the output neuron, x_N , in an N -neuron network, would have $N - 1$ inputs. Each neuron in the network is a typical artificial neuron: a non-linear sigmoidal function applied to the weighted sum of the inputs. In this case, a $(1 - e^{-x})^{-1}$ sigmoid was used to scale the outputs to a range of 0 to 1, but any sigmoid may be used.

For the example given above, the equation for the network output is

$$\begin{aligned} \hat{y} = \text{sig}(W_1x_1 + W_2x_2 + W_3x_3 + W_4x_4 \dots \\ + W_5x_5 + \dots + W_{N-1}x_{N-1}) \end{aligned} \quad (4)$$

where W_n is the weight on the n^{th} input x_n , in a network with N neurons, and $\text{sig}(x) = (1 - e^{-x})^{-1}$. If one sets up the network as shown in Figure 1, $x_1 = P_m$, $x_2 = N_e$, $x_3 = t_i$, (all scaled to the range (0,1)), $x_4 = 1$ (for a bias) and the

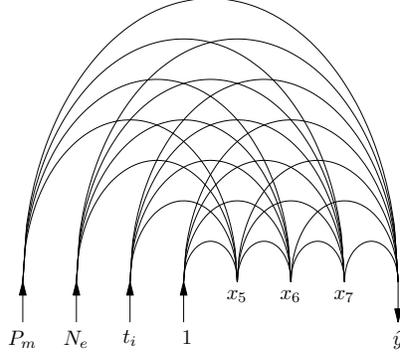


Fig. 1: A Generalized Neural Network (GNN) is a string of neurons, shown here with inputs on the left and outputs on the right. Each neuron is connected to every neuron to left of it. This architecture eliminates the need to determine how many layers to use and how many neurons should be in each layer.

output is scaled such that $y = 0$ is lean and $y = 1$ is rich. Equation 4 now becomes

$$\hat{y} = \text{sig}(W_1 P_m + W_2 N_e + W_3 t_i + W_4 \dots + W_5 x_5 + \dots + W_{N-1} x_{N-1}) \quad (5)$$

The problem is now to solve this equation for \hat{t}_{is} at the boundary, where $\hat{y} = 0.5$. Knowing that $\text{sig}(0) = 0.5$, one may rewrite this equation as

$$\hat{y} = 0.5 = \text{sig}(W_1 P_m + W_2 N_e + W_3 \hat{t}_{is} + W_4 \dots + W_5 x_5 + \dots + W_{N-1} x_{N-1}) \quad (6)$$

$$(7)$$

or

$$0 = W_1 P_m + W_2 N_e + W_3 \hat{t}_{is} + W_4 \dots + W_5 x_5 + \dots + W_{N-1} x_{N-1} \quad (8)$$

If one separates as the hidden neuron outputs into

$$\Phi(P_m, N_e, t_i) = W_5 x_5 + W_6 x_6 + \dots + W_{N-1} x_{N-1}, \quad (9)$$

equation 8 can be manipulated as follows

$$0 = W_1 P_m + W_2 N_e + W_3 \hat{t}_{is} + W_4 + \Phi(P_m, N_e, \hat{t}_{is}) \quad (10)$$

$$\hat{t}_{is} = \frac{-1}{W_3} (W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e, \hat{t}_{is})). \quad (11)$$

Unfortunately, as Φ is a nonlinear function of t_i , this is as far as one can proceed algebraically on a typical network. However, it is possible to proceed if one removes the dependence of Φ on t_i by eliminating (or constraining to zero) the weights that connect t_i to the hidden neurons (but not the output neuron), as shown in Figure 2. Equation 11 then takes the form of

$$\hat{t}_{is} = \frac{-1}{W_3}(W_1P_m + W_2N_e + W_4 + \Phi(P_m, N_e)) \quad (12)$$

which is a closed-form solution for the estimated stoichiometric injection pulse width for operating conditions of intake manifold pressure and engine speed.

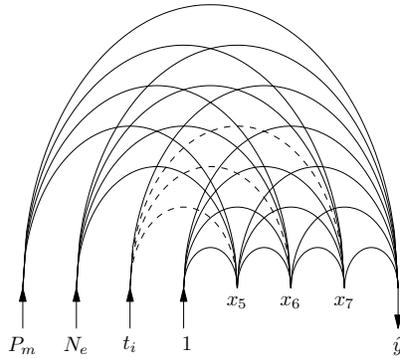


Fig. 2: By setting the weights connecting t_i and the hidden layers to zero (shown as dashed lines), it becomes possible to invert the network. Note that the connection between t_i and the output neuron is not severed.

There are number of a consequences of zeroing the weights connecting one input to the hidden neurons. The weighted sum for the output neuron will be a linear function of t_i . While the neuron output will still be a non-linear function, it will be monotonically increasing or decreasing. This eliminates a class of problems with non-monotonic class boundaries, but eliminates the possibility of multiple solutions. One additional positive consequence of this architecture is that the solution given in Equation 12 can be viewed as a linear equation, augmented by a non-linear term Φ . This can be exploited for initialization or analysis of the network. For example, it is known that the equation for t_{is} is strongly linear with P_m , with a small contribution from N_e and other non-linearities [3][4]. Therefore, plausible initialization values are given by setting $-W_1/W_3$ and $-W_4/W_3$ to match the linear dependence on P_m and setting $W_2, W_5, W_6, \dots, W_N$ to zero or small values. Further details may be found in [5].

4 Simulation Example

The inversion method described in the previous section was used to identify and control a simulated V8 engine following the model in [4], assuming a fuel of

natural gas. The engine was fed inputs of intake manifold pressure and engine speed that were previously recorded from actual city driving. Upon each injection, the controller used online backpropagation[1][6] to train a GNN to match the simulated sensor signal (which included a pure time delay). The controller then used the inverted network (equation 12) to determine an estimate of the stoichiometric pulse width for the next injection. A network with 15 hidden neurons was used for the model.

The results of the simulation study are shown in Figures 3 and 4. Figure 3 shows the relative air-fuel ratio during training. This is the air-fuel ratio divided by the stoichiometric air-fuel ratio. The controller has initially poor performance with approximately 30% error, but quickly improves. After 60 minutes, the error improves to approximately 2%, before eventually reaching a steady state error of approximately 0.03%, as shown in Figure 4.

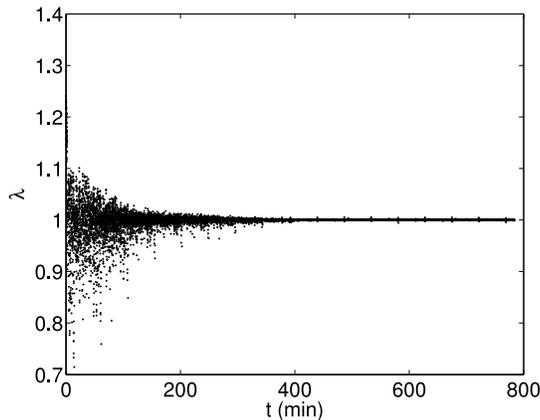


Fig. 3: The relative air-fuel ratio, λ , defined as the measured air-fuel ratio divided by the stoichiometric air fuel ratio, for a simulated V8 engine was controlled using the network inversion scheme introduced in this paper.

With regard to computation speed, in a separate experiment this algorithm was implemented on a Motorola MPC555 microcontroller. The calculation of the stoichiometric pulse width took 0.335 ms, well within the requirements for realtime operation. Since each inversion calculation takes approximately the same time as one forward network calculation, each iteration of a numerical solution can be expected to take at least the same time. Therefore, an iterative solver would only be allowed seven iterations to find its solution in the 2.5 ms time allotted between injections, which would probably not provide the required accuracy for fuel-air control. This is especially true as gradient-based iterative solvers can not be used since they have problems finding the transition point of functions that have a sharp transition with very small slopes in areas away from the transition point. For example, the bisection method would have an unacceptably large error of 0.78% after 7 iterations [7].

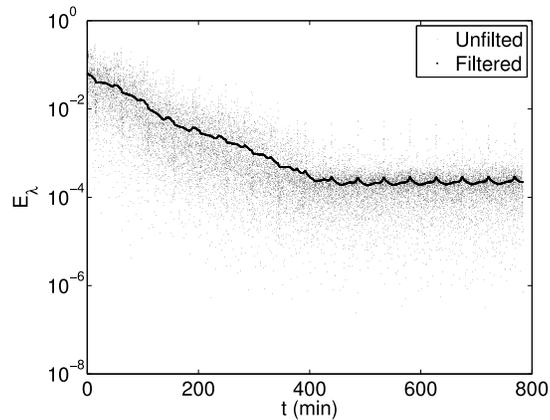


Fig. 4: The error in lambda during training, defined as $|\lambda - 1|$, for the data in Figure 3.

5 Conclusion

This paper has presented a method of inverting a neural network classifier. The closed-form solution developed requires approximately the same number of calculations as one forward pass through the network, eliminating the typical problem of excessive iterations required by numerical solutions. A simulated example of fuel-air ratio control with online learning was presented, although the same method may be used to find any classification boundary that is monotonic. The authors expect to release experimental results for the same controller in the near future.

References

- [1] D. Saad, editor. *On-line Learning in Neural Networks*. Cambridge University Press, Cambridge, 1998.
- [2] P. Werbos. Backpropagation Through Time: What It Does and How to Do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.
- [3] Charles Fayette Taylor. *The Internal-Combustion Engine in Theory and Practice*, volume I. M.I.T. Press, Cambridge, second edition, 1985.
- [4] J.B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
- [5] T. Wiens, R. Burton, G. Schoenau, and M. Sulatisky. Intelligent fuel air ratio control of gaseous fuel SI engines. Technical Report MISC-0168, Saskatchewan Research Council, Saskatoon, 2006.
- [6] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, New Jersey, 2nd edition, 1999.
- [7] W H Press, S A Teukolsky, W T Vetterling, and B P Flannery. *Numerical Recipes in Fortran 77*. Cambridge University Press, Cambridge, 2001.

CHAPTER 7

LIMIT CYCLE BEHAVIOUR OF A NEURAL CONTROLLER WITH DELAYED BANG-BANG FEEDBACK

Published as:

- Travis Wiens, Greg Schoenau, Rich Burton, “Limit Cycle Behaviour of a Neural Controller with Delayed Bang-Bang Feedback” International Journal of Intelligent Technology, vol 2, n 2, 2007.

7.1 Objectives

The objective of this paper is to develop a method of estimating the period and amplitude of the limit cycle of the neural fuel-air controller previously presented. This limit cycle is required for proper operation of the three-way catalyst and the frequency and magnitude of oscillations are critical to the tailpipe emissions of vehicles with tight emissions requirements.

7.2 Approaches

This paper begins with Heywood’s derivation of an equation for the frequency and amplitude of the limit cycle produced by a system with a binary sensor and a delay [3]. His basic model covered a pure integral controller with symmetric gains (i.e. the same gain for rich and lean operation). The paper included here extends this model to include the ability to model the effect of asymmetric gains, which are commonly used to bias the fuel-air ratio slightly rich or lean. Further, the paper derives similar equations for the neural controller, showing that the learning algorithm behaves similarly to an integral controller. This was achieved by linearizing the training algorithm to find the approximate equivalent integral gain of the system.

7.3 Results

It is shown in the paper that the period of oscillation of an integral controller with a delay d is

$$P = d \left(2 + \left| \frac{K_{il}}{K_{ir}} \right| + \left| \frac{K_{ir}}{K_{il}} \right| \right). \quad (7.1)$$

where K_{il} and K_{ir} are the lean and rich integral gains. If these gains are different, the mean injection pulse width t_{inj} will be biased by

$$\overline{t_{inj}} - t_s = \frac{d}{2}(K_{il} + K_{ir}), \quad (7.2)$$

where t_s is the stoichiometric pulse width. The magnitude of the triangular oscillations will then be

$$\max(t_{inj}) - \min(t_{inj}) = d(K_{il} - K_{ir}). \quad (7.3)$$

The integral gain of the integral controller is related to the slope of the triangular limit cycle. If one linearizes the neural network training scheme, it is shown that the slope of the waveform will be approximately

$$dt_{inj} = \pm \frac{\eta t_{isc}}{8W_3} \left(1 + \frac{N-1}{3} \right) \quad (7.4)$$

where η is the learning rate, t_{isc} is the scaling factor for the pulse width, W_3 is the weight connecting the scaled pulse width to the output neuron and N is the number of neurons. More accurate models are also included in the paper, if one has more information about the operating point or internal state of the network.

This information can then be used to estimate the root mean squared error in the pulse width from the stoichiometric value as

$$RMSE = d dt_{inj} \sqrt{\frac{\frac{4}{3}d + 2r}{4d + 2r}} \quad (7.5)$$

where r is the range of values in the delay model. The period of this oscillation can be estimated as

$$P = 4d + 2r. \quad (7.6)$$

which is $2r$ longer than an integral controller with symmetric gains.

Integral and neural controllers were set up for use on the 2003 6.0L V8 Vortec engine (the same as used in Chapters 4 and 8). The results show that the models developed in this paper predict the trends in the characteristics of the limit cycles, although there are some errors between the predicted and measured values.

7.4 Contributions

This paper expands the models available for the limit cycle produced by systems with a delay and binary sensor. This allows an engine control unit designer to more easily tune the limit cycle of the system to the requirements of the catalytic converter, encouraging better tailpipe emissions. Models of the characteristics of linear integral controllers with asymmetric gains are developed as well as similar models for the neural controller developed in this dissertation. The results also verify that the neural training scheme behaves very similarly to an integral controller when linearized about an operating point.

Limit Cycle Behaviour of a Neural Controller with Delayed Bang-Bang Feedback

Travis Wiens, Greg Schoenau, Rich Burton

Abstract—It is well known that a linear dynamic system including a delay will exhibit limit cycle oscillations when a bang-bang sensor is used in the feedback loop of a PID controller. A similar behaviour occurs when a delayed feedback signal is used to train a neural network. This paper develops a method of predicting this behaviour by linearizing the system, which can be shown to behave in a manner similar to an integral controller. Using this procedure, it is possible to predict the characteristics of the neural network driven limit cycle to varying degrees of accuracy, depending on the information known about the system. An application is also presented: the intelligent control of a spark ignition engine.

Keywords—Control and automation, artificial neural networks, limit cycle

I. INTRODUCTION

A commonly encountered form of limit cycle is that which results when a PID compensator is used in a system with a delay and bang-bang¹ sensor. This paper first develops equations describing this classic case, particularly for the application of fuel-air control. Next an analogous analysis is performed for the neural controller presented in [1]. A number of simplifying assumptions allow it to be shown that, for small deviations around an operating point, the training scheme operates as an integral compensator.

II. INTEGRAL CONTROLLER

The combination of a plant modelled as a gain, with a sensor including a bang-bang element and a delay is commonly given as an example of a system exhibiting a limit cycle. The analysis contained in this section is closely based on that found in [2], although similar analyses may be found in control theory textbooks. Consider the fuel-air control of a spark ignition engine via an integral controller, shown in Fig. 1. The plant is modelled as a nonlinear gain representing the fuel-air mixing function, and a delay representing the time taken for the mixture to travel from the injection point to the bang-bang oxygen sensor. The plant transfer function is thus

$$y(k) = F(t_{inj}(k - d)) \quad (1)$$

where y is the oxygen sensor output, $F(\cdot)$ is a nonlinear bang-bang function, t_{inj} is the fuel injector pulse width, k is the index of the sample, and d is the delay (in units of injection events).

Authors are with the Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Canada.

¹A bang-bang sensor is a two-value sensor. For example, the oxygen sensor in a vehicle produces one voltage when the fuel-air ratio is lean and another when it is rich, but little information in between

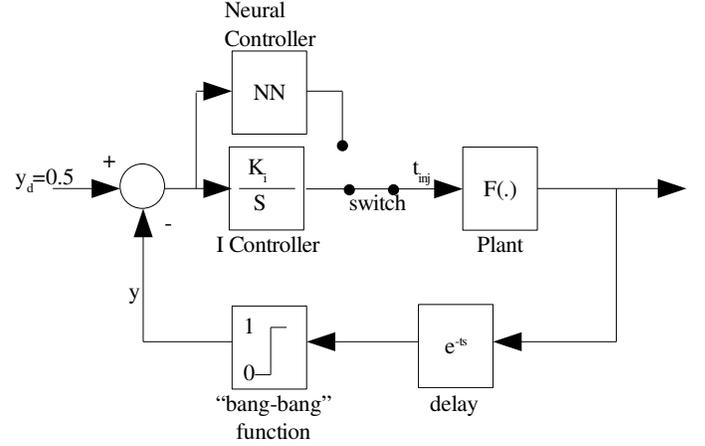


Fig. 1. Block diagram of the system described in this paper (in the Laplace domain). The switch allows the control to be chosen from either an integral controller or a neural network controller.

A discrete time integral controller is used to control the fuel-air ratio such that it oscillates about the stoichiometric ratio² which corresponds to the sensor's transition point. The controller is given by the function

$$t_{inj}(k) = \begin{cases} t_{inj}(k-1) + K_{ir} & y(k) > y_t \\ t_{inj}(k-1) + K_{il} & y(k) < y_t \end{cases} \quad (2)$$

where K_{ir} is the rich integral gain (negative), K_{il} is the lean gain, and y_t is the transition point of the sensor.

The resulting waveform will be a triangular wave with the upward slope equal to K_{il} and the downward slope equal to K_{ir} , as shown in Fig. 2. Because of the delay, the pulse width will overshoot the stoichiometric pulse width for d samples. Thus the rich peak of the curve will be approximately $K_{il}d$ above the stoichiometric pulse width, t_{is} , and the trough will be $-K_{ir}d$ below. This is an approximate value, as depending on the conditions, each section may be $d+1$ samples long rather than d . This is true throughout the analyses in this paper. If matching integral gains are used (i.e. $K_{il} = -K_{ir}$) the time taken for the pulse width to recover from the peak back to stoichiometry will be the same as the time taken to reach the peak, d . Thus, the period of the limit cycle will be $4d$ [2]. If unsymmetrical gains are used, the period, P , can be shown to be

$$P = d \left(2 + \left| \frac{K_{il}}{K_{ir}} \right| + \left| \frac{K_{ir}}{K_{il}} \right| \right). \quad (3)$$

²The stoichiometric ratio is the ratio of fuel to air such that there is just enough oxygen to completely combust the fuel.

as shown in Fig. 3. These unsymmetrical integral gains can be used to bias the fuel-air ratio rich or lean. By integrating the triangles, it can be shown that the mean deviation from stoichiometry is given by

$$\overline{t_{inj}} - t_s = \frac{d}{2}(K_{il} + K_{ir}) \quad (4)$$

and the peak to peak magnitude of the oscillations will be

$$\max(t_{inj}) - \min(t_{inj}) = d(K_{il} - K_{ir}). \quad (5)$$

Plots of (3) and (4) are presented in Fig. 4 and 5. For ease of comparison, the root mean squared error of this curve from the stoichiometric point is

$$RMSE = \left(\frac{d^2}{3 \left(2 - \frac{K_{il}}{K_{ir}} - \frac{K_{ir}}{K_{il}} \right)} \left[K_{il}^2 \left(1 - \frac{K_{il}}{K_{ir}} \right) + \dots \right. \right. \\ \left. \left. K_{ir}^2 \left(1 - \frac{K_{ir}}{K_{il}} \right) \right] \right)^{\frac{1}{2}}. \quad (6)$$

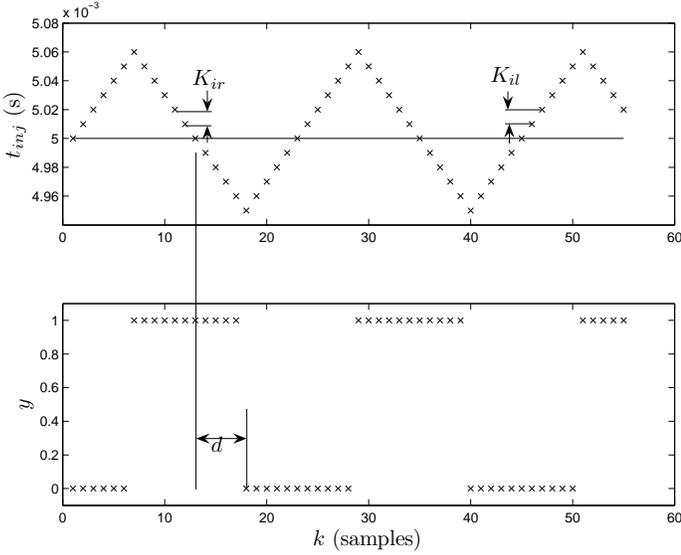


Fig. 2. Limit cycle behaviour in a system with symmetric parameters $K_{ir} = -0.01$ ms and $K_{il} = 0.01$ ms. Notice that the period of the waveform is approximately $4d$. The stoichiometric pulse width is 5.0 ms and the delay is 5 samples.

Therefore, if one knows the delay of the system and the desired bias and period of oscillations for proper catalyst operation, one can solve for K_{il} and K_{ir} by using (3) and (4) to yield

$$K_{ir} = \frac{\overline{t_{inj}} - t_s}{d} \left(\frac{P}{d} - 4 \pm \sqrt{-4 \frac{P}{d} + \left(\frac{P}{d} \right)^2} \right) \quad (7)$$

$$K_{il} = 2 \frac{\overline{t_{inj}} - t_s}{d} - K_{ir} \quad (8)$$

III. NEURAL CONTROLLER

A similar analysis can be performed on the neural controller presented in [1]. This controller uses a type of inverse model

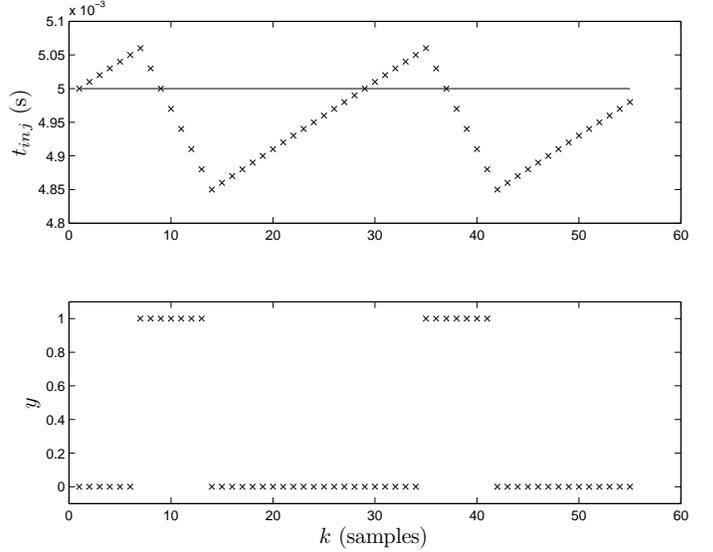


Fig. 3. Limit cycle behaviour in a system with unsymmetrical integral gains $K_{ir} = -0.03$ ms, $K_{il} = 0.01$ ms. Although the delay is the same, the period has increased. The unsymmetrical gains allow the fuel air ratio to be biased, lean in this case.

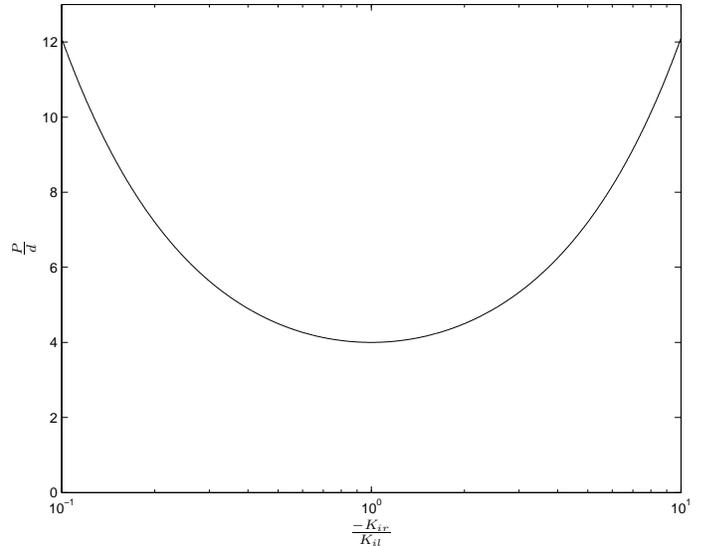


Fig. 4. The period of oscillations for an integral controlled system has a minimum when $K_{ir} = -K_{il}$ and increases as the ratio of integral gains changes. This is a plot of (3).

control to determine an estimate of the stoichiometric injection pulse width. It does this by first identifying the engine using a two-part model.

The first part is a non-linear, but static classifier, which estimates whether the mixture in the intake (no delay) would be rich or lean, given inputs of intake manifold pressure, P_m , engine speed, N_e , and injector pulse width t_{inj} . This part of the model takes the form of a generalized neural network[3] which is updated via online backpropagation training[4] with feedback from the oxygen sensor in the exhaust. This type of network uses the same neurons as a multilayer perceptron, but instead of being organized in layers, it is organized in a row, as shown in Fig. 6. Each neuron has inputs coming from the

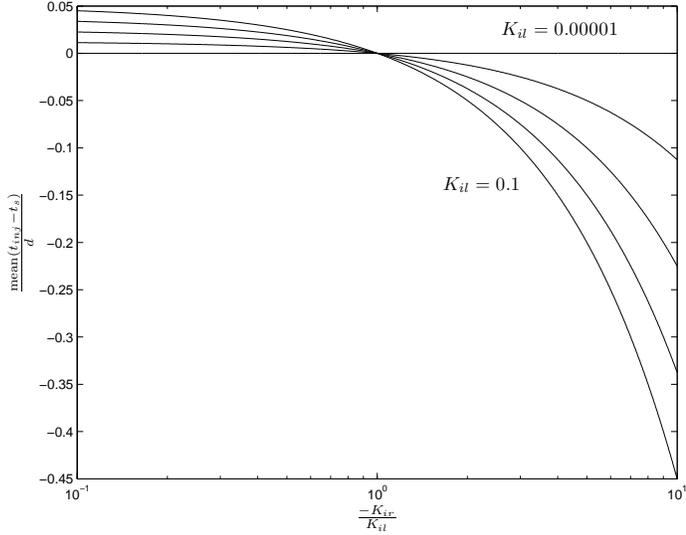


Fig. 5. According to (4), the mean injector pulse width may be biased, as shown in this figure. In all cases, the bias is zero when $K_{ir} = -K_{il}$.

outputs of the neurons to its left.

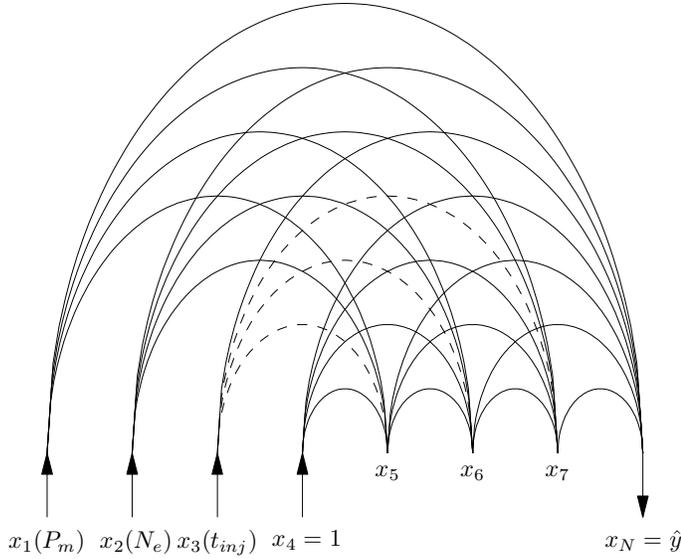


Fig. 6. The generalized neural network architecture uses a string of neurons instead of layers. Each neuron has inputs coming from the neurons to its left. The weights connecting t_{inj} to the hidden neurons (shown with dashed lines) are set to zero to allow the network to be inverted.

The second part of the network is a pure time delay. Since the delay changes based on a number of factors, the model's estimate of the delay is taken to be a range between a minimum and maximum value. The delay model is used to match the delayed oxygen sensor reading to the appropriate inputs. The training algorithm skips any point at which the proper oxygen sensor reading is uncertain. After the training phase is complete, the controller is required to solve the model equations for the stoichiometric pulse width, given the current values for P_m and N_e . By strategically zeroing weights, this can be done algebraically. Further details of the control scheme may be found in [1] and [5].

The learning equation for any arbitrary weight W_i , in the neural network is

$$W_i(k) = W_i(k-1) - \eta \frac{dE}{dW_i} \quad (9)$$

where η is the learning rate and the error term, E , is

$$E = \frac{1}{2}(\hat{y} - y)^2 \quad (10)$$

where \hat{y} is the network's estimate of the bang-bang sensor's output y .

The error gradient can be found by starting at the equation for the the output neuron:

$$\hat{y} = x_N = \text{sig}(\sigma) \quad (11)$$

$$\sigma = W_1x_1 + W_2x_2 + \dots W_{N-1}x_{N-1} \quad (12)$$

where $\text{sig}(\cdot)$ is a unipolar sigmoidal function and x_i is the output of the i th neuron of a total of N neurons.

The gradient can be rewritten as

$$\frac{dE}{dW_i} = \frac{dE}{d\hat{y}} \frac{d\hat{y}}{d\sigma} \frac{d\sigma}{dW_i} \quad (13)$$

using the chain rule, and (10), (11) and (12) may be differentiated as

$$\frac{dE}{d\hat{y}} = \hat{y} - y \quad (14)$$

$$\frac{d\hat{y}}{d\sigma} = \text{sig}(\sigma)(1 - \text{sig}(\sigma)) \quad (15)$$

$$= \hat{y}(1 - \hat{y}) \quad (16)$$

$$\frac{d\sigma}{dW_i} = x_i, \quad (17)$$

so

$$\frac{dE}{dW_i} = (\hat{y} - y)\hat{y}(1 - \hat{y})x_i. \quad (18)$$

An important thing to notice is that the neural network is inverted to find its estimate of the stoichiometric pulse width, so $\hat{y} \approx 0.5$ for any operating point it controls. Also, since the sensor is a bang-bang sensor, it only has two possible values (0 or 1). Therefore, (18) may be simplified as

$$\frac{dE}{dW_i} = (\pm 0.5)(0.5)(1 - 0.5)x_i \quad (19)$$

$$\frac{dE}{dW_i} = \pm \frac{1}{8}x_i. \quad (20)$$

Now consider the effect of this change on the next injection t_{inj} . By inverting the network (see [1]), the scaled pulse width is given by

$$x_3 = \frac{-1}{W_3}(W_1x_1 + W_2x_2 + W_4x_4 + W_5x_5 + \dots W_{N-1}x_{N-1}) \quad (21)$$

where

$$t_{inj} = x_3t_{isc} + t_{ios} \quad (22)$$

with t_{isc} and t_{ios} as constant scaling factors for t_{inj} such that the range of x_3 is constrained to the range of 0 to 1, and x_1 and x_2 are network inputs: manifold pressure and engine speed, also scaled to the range of 0 to 1. x_4 is set to 1 to provide a bias.

If ϵ is defined as the difference $\hat{y} - y$, which would correspond to the control error in a PID controller, the linearized transfer function for the training algorithm can be expressed by $\frac{dt_{inj}}{d\epsilon}$, which is

$$\frac{dt_{inj}}{d\epsilon} = \sum_{i=1}^{N-1} \frac{dt_{inj}}{dx_3} \frac{dx_3}{dW_i} \frac{dW_i}{d\epsilon} \quad (23)$$

if one assumes that the effect of the hidden weights is negligible (because of their indirect influence on the output). This linearized analysis is only valid for small deviations about an operating point. The terms in this equation can be found by differentiation. Substituting (19) into (9), the result is

$$W_i(k) = W_i(k-1) - \eta \epsilon \frac{1}{4} x_i \quad (24)$$

which is differentiated as

$$\frac{dW_i}{d\epsilon} = -\eta \frac{1}{4} x_i. \quad (25)$$

The differentiation of (21) can be shown to be

$$\frac{dx_3}{dW_i} = \frac{-x_i}{W_3}. \quad (26)$$

Finally, (22) can be differentiated as

$$\frac{dt_{inj}}{dx_3} = t_{isc} \quad (27)$$

By substituting (25), (26) and (27) into (23), the result is

$$\frac{dt_{inj}}{d\epsilon} = \frac{\eta t_{isc}}{4W_3} \sum_{i=1}^{N-1} x_i^2. \quad (28)$$

Again, since the value of \hat{y} is constrained to 0.5 and y is constrained to 0 or 1, ϵ must equal ± 0.5 . Thus if one assumes that the system is linear for small deviations about $\epsilon = 0$, $d\epsilon = \pm 0.5$ and the rate of change of the pulse width from one injection to the next is

$$dt_{inj} = \pm \frac{\eta t_{isc}}{8W_3} \sum_{i=1}^{N-1} x_i^2. \quad (29)$$

This provides good results compared to simulation data where all the neuron outputs x_i were known.

If one wishes to predict the performance of a controller, it is necessary to estimate the values for the hidden neurons. If one assumes that the values for x_5, x_6, \dots, x_{N-1} are uniformly distributed between 0 and 1, the sum can be estimated [6] to be

$$\sum_{i=5}^{N-1} x_i^2 \approx \frac{N-5}{3}. \quad (30)$$

Thus, if one knows the scaled values for the intake manifold pressure (x_1), engine speed (x_2) and injector pulse width (x_3), and remembering that $x_4 = 1$, then (29) can be written as

$$dt_{inj} = \pm \frac{\eta t_{isc}}{8W_3} \left(x_1 + x_2 + x_3 + 1 + \frac{N-5}{3} \right). \quad (31)$$

One further simplification can be made if one does not know the operating point a priori. In this case, the values for $x_1, x_2,$

and x_3 are also assumed to be uniformly distributed over the range of 0 to 1 and the result is

$$dt_{inj} = \pm \frac{\eta t_{isc}}{8W_3} \left(1 + \frac{N-1}{3} \right). \quad (32)$$

Any of (29), (31), or (32) give reasonable results; of course, greater accuracy may be obtained as more information is utilized.

Note that dt_{inj} in the above equations plays the same role as K_{il} and K_{ir} in a proportional controller and their values can be directly compared. However the waveform is slightly different. As mentioned above, the training scheme tries to match the neural inputs to the proper delayed feedback value. However, since there is uncertainty in the estimate of the delay, there are times when the training scheme must skip a number of points. Therefore, instead of being a triangular waveform, the waveform has "plateaus", level spots at each peak and valley in t_{inj} . Due to these points being ignored by the training scheme, the limit cycle frequency is decreased and the mean deviation from stoichiometry is increased. If the slope of each of these sections is dt_{inj} , and the range of possible delays values is r , then the root mean squared error from the stoichiometric pulse width will be

$$RMSE = dt_{inj} \sqrt{\frac{\frac{4}{3}d + 2r}{4d + 2r}} \quad (33)$$

and the period is

$$P = 4d + 2r. \quad (34)$$

Note that, as with the integral controller, these values are approximate as the length of time taken for each section can be $d + 1$ samples, rather than d .

It is also possible to bias the control scheme by using different learning rates for lean and rich operation, in the same way as different integral gains may be used to bias the fuel-air ratio.

IV. EXPERIMENTAL VERIFICATION

The controller described above was implemented on a 2001 GM 2500HD truck, fuelled by natural gas (further details may be found in [7]). This truck has a Vortec 6L V8 engine, with each bank controlled independently. The original engine control unit (ECU) maintained control of all functions except fuel control (i.e. spark timing, idle air control, etc.). The controller was programmed such that the relevant parameters could be easily adjusted to examine their effects. For each trial, the truck was run at idle for 60 seconds and data was recorded from the eight injectors, as well as the two heated oxygen sensors, the intake manifold pressure transducer, and a wide-range oxygen sensor. This data was recorded at a rate of 200 kSamples/s. Note that the "bang-bang" heated oxygen sensors were used for feedback and training; the wide-range oxygen sensor's measurements were only used for evaluation of the controller and were not used in its algorithm. All tests were performed with the engine, oxygen sensors and catalysts at their operating temperature.

The first test performed was to determine the effect of the integral gain. The controller was set up with $K_{ir} = -K_{il} =$

K_i and $\eta = 0$ to eliminate the effects of the neural network. Fig. 7 shows the aggregated results of the amplitude of the limit cycle over a variety of integral gains, along with the theoretical curve from (6). One can see that the shape of the curve is accurate for K_i values greater than 0.01 ms, although there is an approximately constant bias error of 0.09 ms. This bias may be due to fluctuations in the operating point which may be attributed to spark timing or idle air control oscillations. Below $K_i = 0.01$ ms these other fluctuations dominate and the integral controller does not cause the same limit cycle.

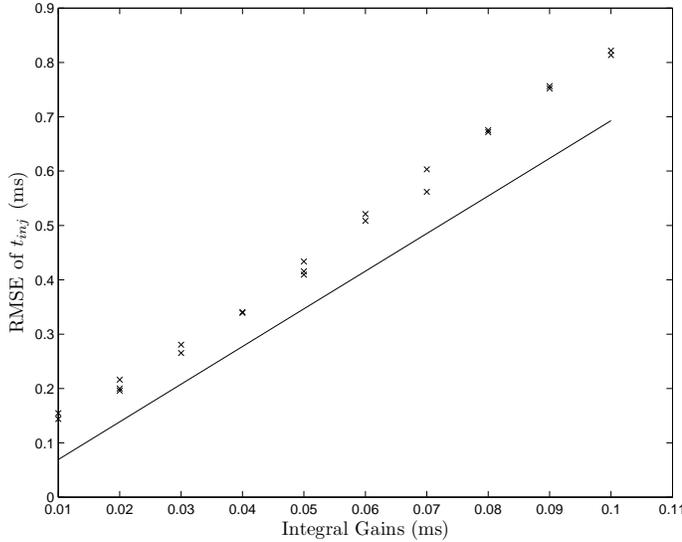


Fig. 7. Effect of integral gain on oscillation amplitude, measured by the standard deviation of 60 seconds of idling at each data point. The integral gains are symmetric ($K_{il} = -K_{ir} = K_i$). These values may be compared to the theoretical value from (6), shown as the solid line.

Fig. 8 demonstrates the ability to bias the controller via asymmetric integral gains. The rich integral gain was held constant at -0.01 ms and the lean integral gain was varied. This test was again performed at idle (the mean t_{inj} was approximately 4.85 ms) with the neural learning algorithm disabled. Fig. 9 shows the period of the limit cycle oscillations, compared to the theoretical curve from (3). Finally Fig. 10 shows an example of the resultant relative fuel-air ratio, λ , defined as

$$\lambda = \frac{m_{is}}{m_i}, \quad (35)$$

where m_i is the mass of fuel injected and m_{is} is the stoichiometric pulse width. Notice the different slopes of the increasing and decreasing portions of the trace.

The next test was performed to verify the theoretical equations for the neural system. In this test the integral controller was disabled and the learning rate, η , was varied. Fig. 11 shows the effect of learning rate on the oscillation magnitude (as quantified by the root mean squared error in pulse width), compared to the results of (33). Fig. 12 shows a sample waveform. Again, the results follow the shape of the theoretical curve for η greater than 0.005, with similar errors as in Fig. 7.

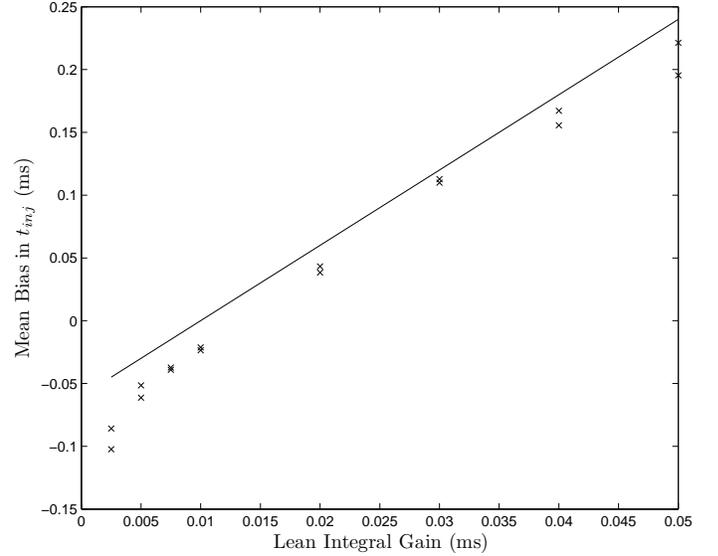


Fig. 8. Effect of asymmetric integral gain on the mean pulse width. The rich integral gain, K_{ir} , is held constant at -0.01 ms while the lean gain, K_{il} , is varied.

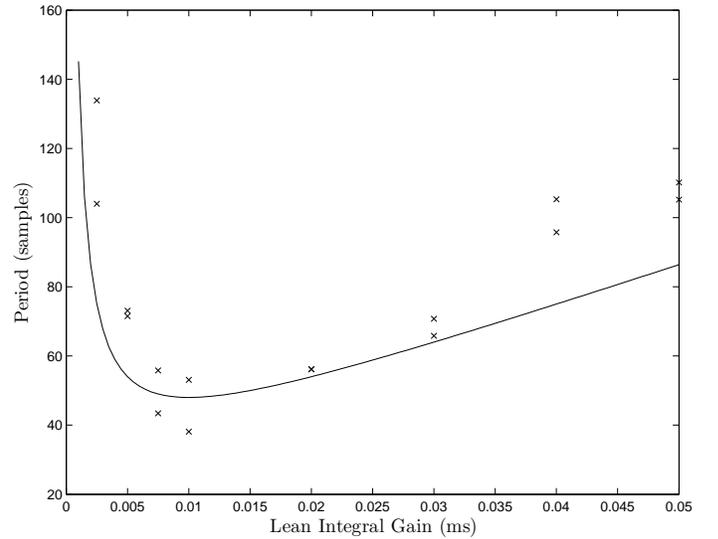


Fig. 9. Effect of asymmetric integral gain on the period of the limit cycle oscillations. The rich integral gain, K_{ir} , is held constant at -0.01 ms while the lean gain, K_{il} , is varied.

V. CONCLUSION

This paper developed equations showing that the training dynamics of the neural controller can be expressed in a linearized form. In particular, it is possible to determine the period, amplitude and bias of the limit cycle of a particular training scheme using “bang-bang” feedback for online training. This development demonstrates that the dynamics of the training scheme parallel those of a classical linear integral controller, and much of the same analysis may be performed. Although there were some errors between the theoretical and experimental values, the theory was verified via experiments performed on a system used to control the fuel-air ratio of a spark-ignition engine.

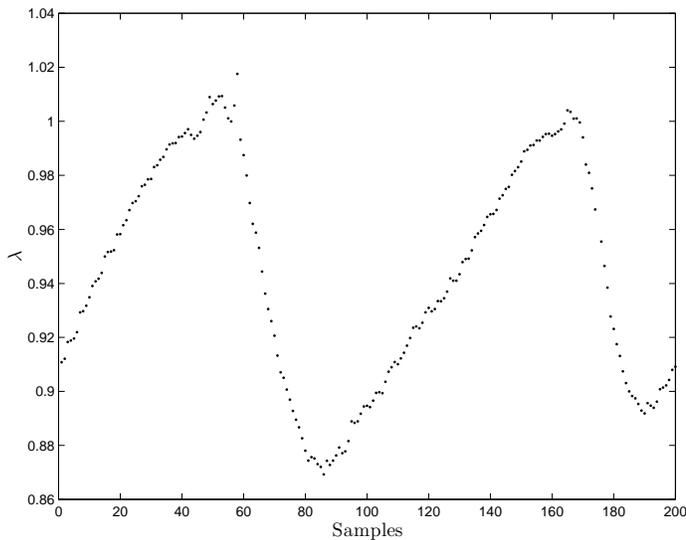


Fig. 10. A sample of the relative air-fuel ratio waveform produced with $K_{ir} = -0.01$ ms and $K_{il} = 0.05$ ms. Note the differing slopes of the increasing and decreasing portions of the trace.

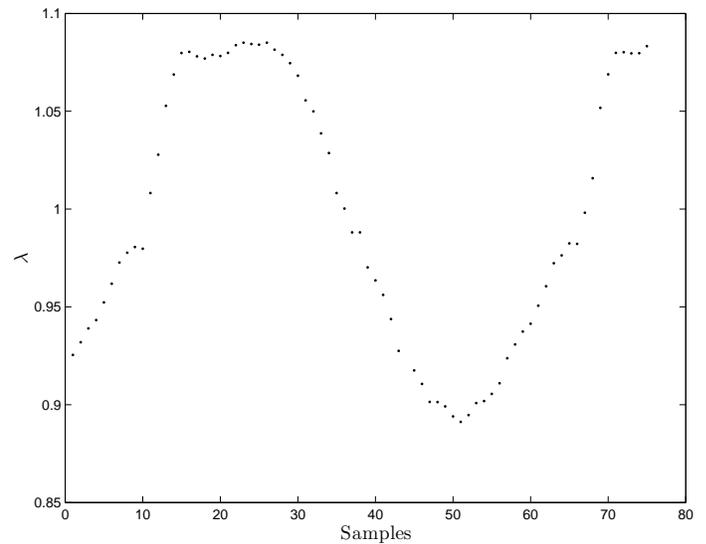


Fig. 12. A sample of the relative air-fuel ratio waveform produced with $\eta = 0.05$.

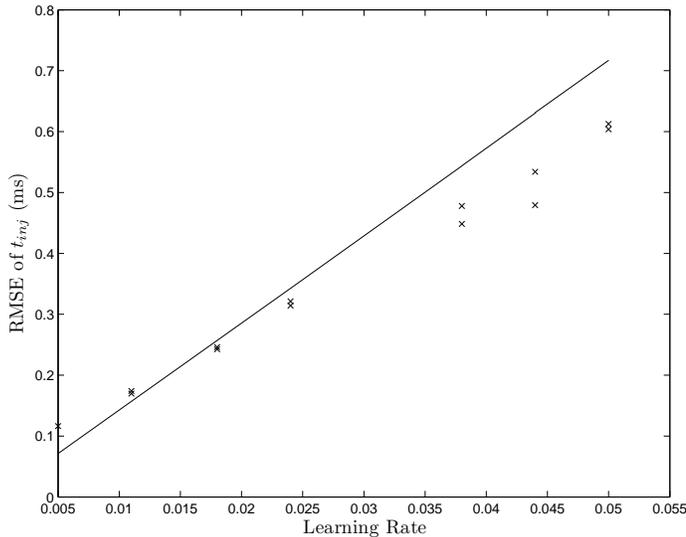


Fig. 11. Effect of learning rate, η , on the amplitude of oscillations.

REFERENCES

- [1] T. Wiens, R. Burton, G. Schoenau, and M. Sulatisky, "Intelligent fuel air ratio control of gaseous fuel SI engines," Tech. Rep. MISC-0168, Saskatchewan Research Council, Saskatoon, 2006.
- [2] J. Heywood, *Internal Combustion Engine Fundamentals*. New York: McGraw Hill, 1988.
- [3] P. Werbos, "Backpropagation Through Time: What It Does and How to Do it," in *Proceedings of the IEEE*, vol. 78, pp. 1550–1560, 1990.
- [4] D. Saad, ed., *On-line Learning in Neural Networks*. Cambridge: Cambridge University Press, 1998.
- [5] T. Wiens, R. Burton, and G. Schoenau, "Algebraic inversion of an artificial neural network classifier," in *Proceedings of the European Symposium on Artificial Neural Networks*, (Bruges), 2007.
- [6] D. Zwillinger, *CRC Standard Mathematical Tables and Formulae*. Boca Raton: Chapman & Hall/CRC press, 2002.
- [7] T. Wiens, R. Burton, G. Schoenau, M. Sulatisky, S. Hill, and B. Lung, "Preliminary experimental verification of an intelligent fuel air ratio controller," Tech. Rep. 2007-01-1339, Society of Automotive Engineers (SAE), 2007.

VI. ACKNOWLEDGEMENT

The authors would like to acknowledge the financial support of NSERC; the Saskatchewan Research Council for providing equipment and expertise; and General Motors Alternative Fuels for the loan of the test vehicle. Thanks are also given to Natural Resources Canada and Precarn Inc who funded initial work in this area.

CHAPTER 8

PRELIMINARY EXPERIMENTAL VERIFICATION OF AN INTELLIGENT FUEL AIR RATIO CONTROLLER

Published as:

- Travis Wiens, Rich Burton, Greg Schoenau, Mike Sulatisky, Sheldon Hill, Bryan Lung, “Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller”. Technical Paper 2007-01-1339, SAE, Warrendale, 2007. Reprinted with permission from SAE Paper 2007-01-1339 Copyright (c) 2007 SAE International.

8.1 Objectives

The objective of this paper was to verify the emissions performance of the neural controller in a commercial vehicle experimental test.

8.2 Approaches

This paper presents an experimental evaluation of the controller installed on a General Motors pickup truck. The truck was mounted on a chassis dynamometer and the tailpipe emissions were measured by an exhaust gas analyzer. The truck was then driven through the “Hot 505” driving cycle of the US Federal Test procedure, which simulates a variety of city and freeway driving conditions. Emissions measurements were recorded, including carbon monoxide (CO) and nitrogen oxides (NO_x), for both the neural controller and the OEM controller.

The truck used was the same as that used for the tests in Chapters 4 and 7: a 2001 GM2500HD chassis with an automatic transmission and a 2003 bifuel Vortec 6.0L V8 engine (model LQ4) running on natural gas supplied by the municipal utility, SaskEnergy. This engine uses parallel gaseous and liquid fuel injectors mounted near the intake ports, although only one set was in use at any time. The exhaust system has two heated oxygen sensors in the exhaust manifold which were used for feedback, as well as Ultra Low Emissions Vehicle (ULEV) catalytic converters. A Bosch wide-range oxygen sensor was mounted in the exhaust pipe as near to the engine as possible (this sensor was only used for monitoring, not as part of the

control scheme). The natural gas was stored in a pressurized tank in the box at a maximum pressure of 3000 psi (20.7 MPa).

It should be noted that the controller used in this trial has a slightly different form from that presented in Chapter 5. It was found that considerably better performance could be achieved by applying a linear integral compensator to the pulse width. This allowed the neural controller to generate the base pulse width while very short term errors were corrected by the integral controller. This allowed slower learning rates to be used in the neural controller, improving the minimum error. The code used is found in Appendix C.

8.3 Results

A plot of the relative air-fuel ratio over the driving cycle is found in Figure 8.1. The mean of this plot is 0.992 with a standard deviation of 0.0140. The OEM controller produced a mean of 0.987 with a standard deviation of 0.0234 over the same driving cycle. This means that the neural controller controlled excursions from the desired fuel-air ratio considerably better than the OEM controller.

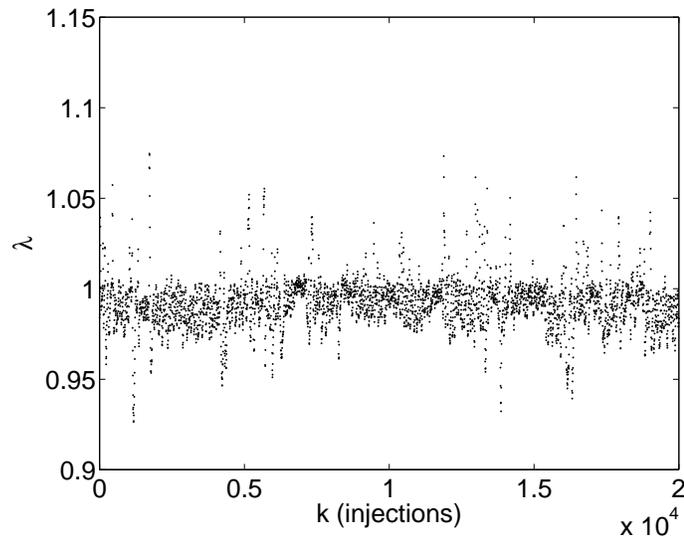


Figure 8.1: Experimentally measured λ over the “hot 505” portion of the Federal Test Procedure driving cycle using natural gas.

While λ is a common diagnostic test parameter, the truly important results are the tailpipe emissions. The maximum and mean of the carbon monoxide and nitrogen oxides concentrations in the exhaust are shown in Table 8.1 for both the neural controller and the OEM ECU. Note that in each case the neural controller has lower emission concentrations than the OEM controller.

Table 8.1: Post-catalyst emissions concentrations over Hot 505 driving cycle.

	Neural Control	OEM Control
Mean CO (ppm)	129.5	306.4
Max CO (ppm)	788	5958
Mean NOx (ppm)	5.11	54.5
Max NOx (ppm)	207	260

8.4 Contributions

This paper presents emissions verification of the control scheme on a standard driving cycle, where most published fuel-air control schemes merely show fuel-air ratio data for stationary engines as verification of the performance of their work. This paper shows that the proposed controller does work in a realistic vehicle application under standard conditions and is effective in reducing tail-pipe emissions.

Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller

Travis Wiens, Rich Burton, Greg Schoenau
Dept. of Mechanical Engineering, University of Saskatchewan

Mike Sulatisky, Sheldon Hill, Bryan Lung
Alternative Energy Products, Saskatchewan Research Council

Copyright © 2007 SAE International

ABSTRACT

This paper presents the experimental verification of a new type of fuel-air ratio controller for spark ignition (SI) engines. The controller does not require any initial calibration before its first use on a new engine beyond some very general information which is used to generate an initial fuel map. The controller then continually updates this nonlinear fuel map in response to changes in the engine or fuel while driving. The controller was implemented on a 2003 Vortec 6L V8 engine in a General Motors 2500HD truck with an OEM (original equipment manufacturer) natural gas conversion. Preliminary results indicate the controller behaves in a manner comparable to OEM controllers in terms of drivability and exhaust emissions, at potentially a much lower development cost.

INTRODUCTION

In modern fuel-injected spark ignition (SI) engines, the vehicle's ECU (engine control unit) is responsible for determining the proper amount of fuel to inject into the intake [1]. Deviation from the optimum fuel-air ratio (also known as the FAR or FA ratio) can lead to reduced power and fuel efficiency and increased emissions [2,3]. Determining the proper fuelling is typically achieved by measuring a number of operating conditions and referring to a look-up table, known as a fuel map, to determine the optimum fuelling. Generating the values in this fuel map is a time-consuming and expensive operation, requiring hours of time on expensive equipment. This process must be repeated for each engine model. In some cases, the fuel map has a basic form of adaptation to changes in vehicle properties from vehicle to vehicle and over time; often, though, it is static in nature.

The objective of this paper is to present the results of an experimental test of a new method of adaptive fuel control. This approach is capable of controlling the fuel-air ratio to a high precision, while eliminating the time-consuming and expensive initial calibration period. As the overall goal of this research program is to reduce the cost of converting vehicles to gaseous fuels (such as

natural gas or hydrogen), the experimental results presented are for a vehicle fuelled by natural gas.

A simulation study was previously performed in order to show the feasibility of the algorithm and is available in reference [4]. As this previous paper presents the form of the controller in detail, the current paper will only briefly describe its operation.

CONTROLLER FORM

The goal of the fuel-air controller is to determine the optimum injection time for which the fuel injectors should remain open. It is typically assumed that this can be achieved by maintaining a fuel-air ratio at the intake that oscillates around the stoichiometric¹ point [3]. This would be a simple matter if one could accurately measure the mass airflow or directly measure the fuel-air ratio at the intake. Unfortunately, it is difficult to measure the instantaneous mass airflow of the rapidly fluctuating flow, and the only practical fuel-air ratio sensors available must be installed in the exhaust, as they indirectly measure the fuel-air ratio by measuring the excess oxygen in the exhaust gases. Locating the oxygen sensor (also known as an exhaust gas oxygen (EGO) sensor or heated oxygen sensor (HO2S)) in the exhaust introduces a delay to the system, which greatly complicates feedback control. This delay is due to the time taken for the charge mixture to travel from the injection point to the cylinder, the time that the mixture resides in the cylinder, and the travel time between the cylinder and sensor.

A typical fuel-air controller assumes that the instantaneous mass airflow can be inferred from such available measurements as intake manifold pressure, engine speed, intake temperature, engine coolant temperature, etc, with corrections made via feedback from the oxygen sensor. The controller presented in this paper makes similar assumptions, although it requires

¹ The stoichiometric ratio is the ratio of fuel to air, such that there is just enough air to completely combust the fuel. This is defined as a relative air-fuel ratio, λ , of 1.

fewer sensors to determine the operating point and requires only minimal initial calibration.

The controller, presented in reference [4], is a modified type of nonlinear model inverse controller. The internal model of the engine is composed of two parts. The first is a non-linear but static model classifying whether the intake air-fuel ratio is rich or lean. This model takes inputs of manifold pressure, P_m , engine speed, N_e , and injector pulse width, t_i , each scaled to the range of 0 to 1, and outputs a value between 0 and 1, transitioning at its estimate of the stoichiometric fuel-air ratio. This model can be thought of as an estimated output of a “bang-bang” oxygen sensor if it could be placed in the intake, rather than the exhaust.

A generalized neural network [5] is used for this model, with some weights strategically set to zero. A generalized neural network (GNN) has an architecture differing from the familiar multilayer perceptron (MLP). While an MLP has a number of layers with each layer's inputs coming from the previous layer's outputs, a GNN can be more easily thought of as a string on neurons, as shown in Figure 1. Each neuron's input is the output of every neuron to the left of it. Thus, with four inputs (including a 1 to enable a bias), the first non-input neuron, x_5 , has four inputs. The next, x_6 , has five inputs and so on. Notice that the output, \hat{y} , has many inputs but, unlike an MLP, it has direct connections to the network inputs, P_m , N_e , and t_i .

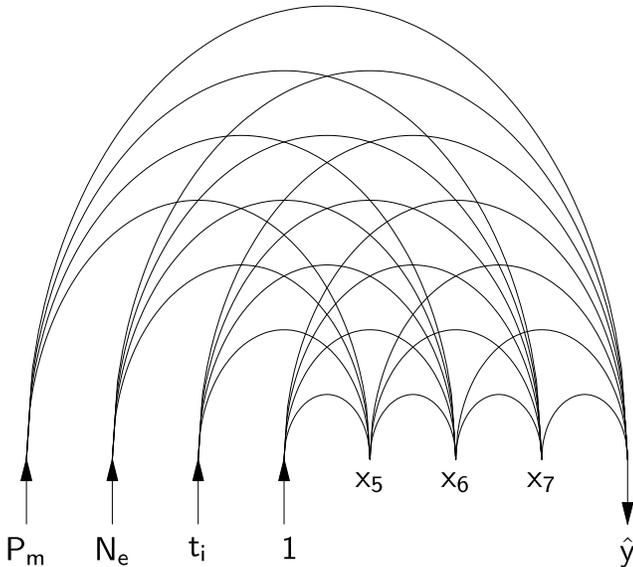


Figure 1: A generalized neural network. The inputs to each neuron are the outputs of each of the neurons to the left of it. This network has 4 input neurons on the left, 3 hidden neurons, and 1 output neuron on the right. The network used for the experimental verification has 14 hidden neurons.

The second part of the model is a delay, which takes into account the transport delays as the fuel travels from the intake, through the engine and to the heated oxygen

sensor (HO2S) mounted in the exhaust, as well as the lag in the sensor itself. As there can be considerably variability of the number of injection events in this delay time, it is treated as a range of probable values.

Upon each injection event, the controller samples inputs of manifold pressure, engine speed, and the HO2S output. The oxygen sensor output is converted to a binary value (1 for rich and 0 for lean) and is placed, together with P_m and N_e , in a circular buffer. The controller then checks if the oxygen sensor output has been constant for the range of injection events in the delay model. If so, the delayed oxygen sensor output is matched to the appropriate delayed inputs of P_m , N_e , and t_i from the buffer and the neural network model is updated using standard back-propagation on-line learning [6,7]. If the oxygen sensor output is not constant for the range of injections in the delay model, no learning takes place for that sample.

The controller then algebraically inverts the neural network to determine the stoichiometric injector pulse width, t_{is} , which it estimates will result in a stoichiometric fuel-air ratio for the currently measured manifold pressure and engine speed. This algebraic inversion is typically impossible to achieve for most types of neural networks, but the generalized neural network architecture may be inverted as long as certain weights are constrained to be zero.

The output of the neural network, \hat{y} , is given by

$$\hat{y} = \text{sig}(W_1x_1 + W_2x_2 + W_3x_3 + W_4x_4 \dots + W_5x_5 + \dots + W_{N-1}x_{N-1}) \quad (1)$$

where W_n is the weight applied to the n th input, x_n , of a network with N neurons. The sigmoid function $\text{sig}(x) = (1 - e^{-x})^{-1}$ is used which ranges from 0 to 1 and with a transition point of $\text{sig}(0) = 0.5$, which corresponds to an estimate of the stoichiometric point. In the specific case of this controller, the inputs are $x_1 = P_m$, $x_2 = N_e$, $x_3 = t_i$ and $x_4 = 1$. The above equation then takes the form of

$$\hat{y} = \text{sig}(W_1P_m + W_2N_e + W_3t_i + W_4 \dots + W_5x_5 + \dots + W_{N-1}x_{N-1}) \quad (2)$$

or

$$\hat{y} = \text{sig}(W_1P_m + W_2N_e + W_3t_i + W_4 + \Phi(P_m, N_e, t_i)) \quad (3)$$

if we combine the contribution of the hidden neuron outputs into Φ .

The goal of inverting the network is to solve for a t_i that results in $\hat{y} = 0.5 = \text{sig}(0)$, given operating points of P_m and N_e . After applying the inverse of $\text{sig}(\cdot)$, Equation 3 can be rearranged algebraically to the form

$$t_{is} = \frac{-1}{W_3}(W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e, t_i)). \quad (4)$$

This cannot be solved analytically because Φ is a nonlinear function of t_i . However, if we constrain the weights associated with t_i to zero, as shown in Figure 2, Φ is now only a function of P_m and N_e . This means that, while the inputs to the last neuron are linear with t_i , the transition boundary is a nonlinear function of P_m and N_e . The closed form solution for the model's estimate of the stoichiometric injection time is

$$t_{is} = \frac{-1}{W_3}(W_1 P_m + W_2 N_e + W_4 + \Phi(P_m, N_e)). \quad (5)$$

This "fuel map" can be thought of as a linear function, augmented by the nonlinear term Φ .

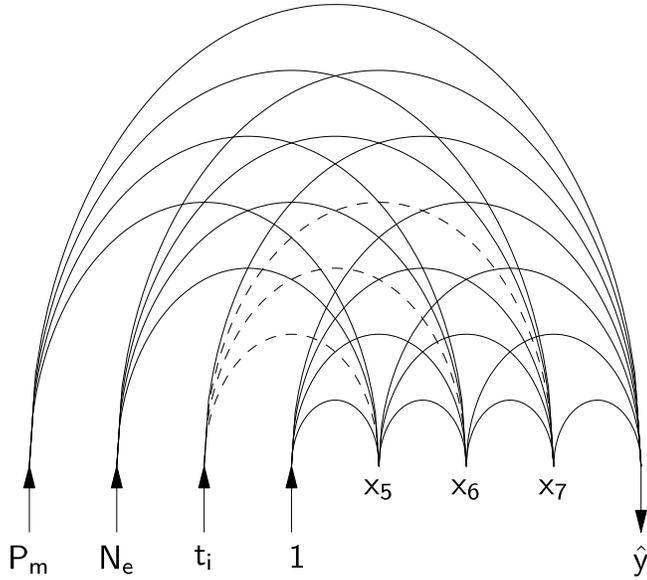


Figure 2: The weights connecting t_i to the hidden neurons are set to zero (shown as dashed lines). This allows the network to be algebraically inverted.

Finally, a proportional-integral (PI) controller value is added to this stoichiometric value to correct for short-term fuelling errors as well as generate the oscillations required for proper exhaust catalyst operation:

$$t_i(k) = t_{is}(k) + K_p(HO2S(k)) + \sum_{i=0}^k K_i(HO2S(i)) \quad (6)$$

where K_p and K_i are the proportional and integral gains, which are both functions of $HO2S(k)$ (positive for lean $HO2S$ readings, negative for rich), and k is the index of the time step. t_{is} is then sent to the injector driver and stored in a circular buffer. The process is repeated for each injection.

One additional advantage to the form of neural network used for this controller is that the linear terms in Equation 5 may be used to set up an initial fuel map. This initial fuel map is a linear one, set such that, at the maximum engine speed and intake manifold pressure, the injectors are open nearly continually (for this experiment, they were set to be open 90% of the maximum), and at zero manifold pressure, the injector pulse width is zero. The hidden neurons are then initialized following Nguyen and Widrow's method [8].

Thus, unlike many engine control units which require a larger number of sensors, such as intake and coolant temperatures, a complete control system for natural gas can be composed of

- manifold pressure sensor,
- crank position sensor (for engine speed measurement and timing of injection pulses),
- exhaust gas oxygen sensor (a wide range sensor is not required),
- processor, and
- associated drivers and wiring.

Furthermore, the only engine-specific data required to develop and initialize a controller for a new engine are

- number of cylinders,
- approximate maximum engine speed,
- approximate maximum intake manifold sensor reading,
- approximate oxygen sensor transition reading, and
- crank position sensor encoding scheme.

EXPERIMENTAL IMPLEMENTATION

The test vehicle selected for this demonstration was a 2001 General Motors 2500HD pickup truck with a 2003 Vortec V8 6L engine and automatic transmission, converted to run on natural gas (see Figure 3). A controller, developed by the Saskatchewan Research Council for a previous project, was programmed with the new algorithm. The controller (shown in Figure 4) includes sensor and injector drivers, three microcontrollers to generate the pulse signals, and a Motorola MPC555 CPU to handle pulse width computation. The truck's original manifold pressure, crank position, and exhaust gas oxygen sensors were used, as well as the standard fuel injectors. The OEM fuel controller was not permanently deactivated, so that direct comparisons could be made between the OEM system and the proposed controller by flipping a switch. At all times the OEM ECU maintained control of all functions other than fuel control, such as spark timing, idle air control, transmission control, etc.



Figure 3: Vehicle used for experimental tests, a 2001 GM 2500HD truck with a Vortec V8 6L natural gas/gasoline bi-fuel engine.

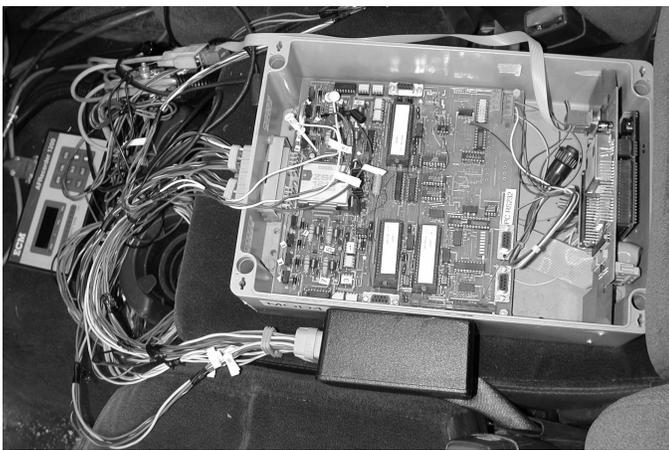


Figure 4: The controller used to implement the algorithm is based on a MPC555 CPU and includes interfacing circuitry for driving the injectors as well as reading various types of sensors.

The neural network control algorithm was implemented with 15 neurons and a training rate of 0.001. The delay was estimated to be between 8 and 18 injections, measured by cutting fuel flow and measuring the time taken for the oxygen sensor to transition from a rich reading to a lean one [9]. The PI controller was set with $K_p=0$ (integral control only) and $K_i=-0.005$ ms/sample for rich HO₂S readings and 0.01 ms/sample for lean. These unsymmetrical integral gains were used to bias the fuel-air ratio slightly rich for the reduction of nitrogen oxides (NO_x).

A wide-range oxygen sensor was installed, but was only used for evaluating the performance of the controller, not as part of the control scheme. A 200 kSamples/s data acquisition card was used to record the manifold pressure, wide-range oxygen sensor output, the two pre-catalyst exhaust gas oxygen sensors, as well as the pulses sent to the injectors.

EXPERIMENTAL RESULTS

This section presents experimental data recorded while driving the "Hot 505" section of the Federal Test Procedure (FTP) driving cycle, which is shown in Figure 5. The neural network was reinitialized immediately before starting the test, which was performed with the engine, oxygen sensors, and catalyst warm.

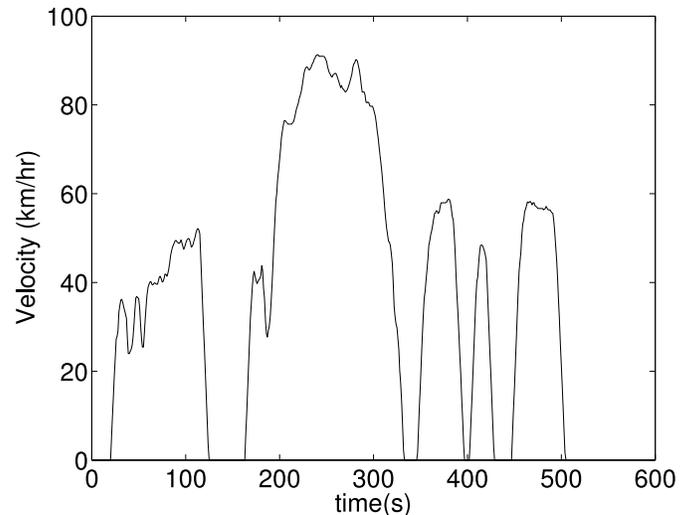


Figure 5: The driving cycle used in this test was the "Hot 505" section of the FTP (Federal Test Procedure) profile. This driving cycle is 505 s long and incorporates moderate acceleration and braking, as well as freeway-style driving.

The trace of the relative fuel-air ratio, λ , is shown in Figure 6, with the same data shown in histogram form in Figure 7. Note that this data was used for evaluation of the controller only and was not available to the control algorithm. Over the course of the test the mean λ was 0.992 with a standard deviation of 0.0140. For comparison, the OEM controller produced a mean of 0.987 and a standard deviation of 0.0234 over the same driving cycle.

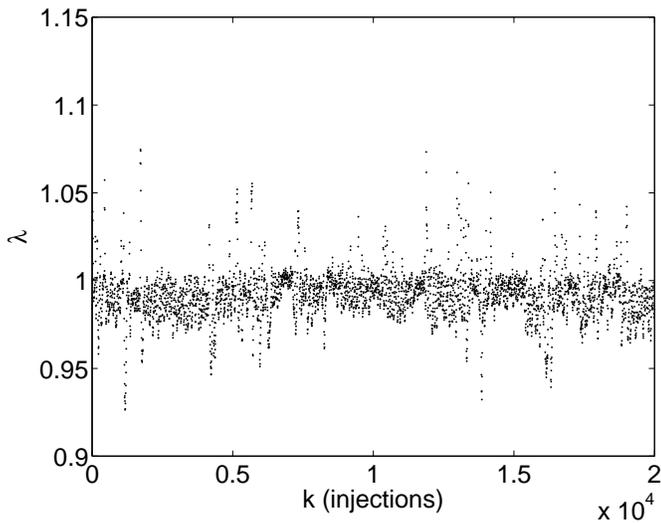


Figure 6: The measured relative fuel-air ratio while driving the profile shown in Figure 3 had a mean of 0.992 and a standard deviation of 0.0140, compared to values of 0.987 and 0.0234 for the OEM controller on the same driving cycle. Note that the wide-range oxygen sensor was used for monitoring the performance of the control scheme only, and is not used for feedback.

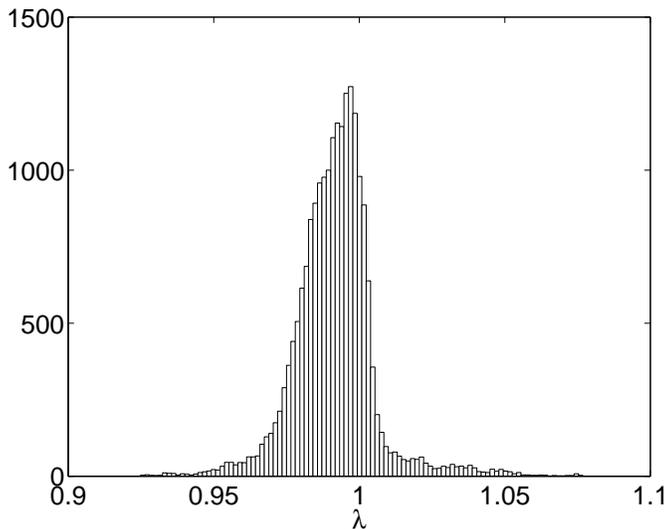


Figure 7: The data set in Figure 6 is shown here in histogram form.

Figure 8 shows the HO2S feedback readings for the same test, demonstrating the expected limit-cycle behaviour required for proper catalyst operation.

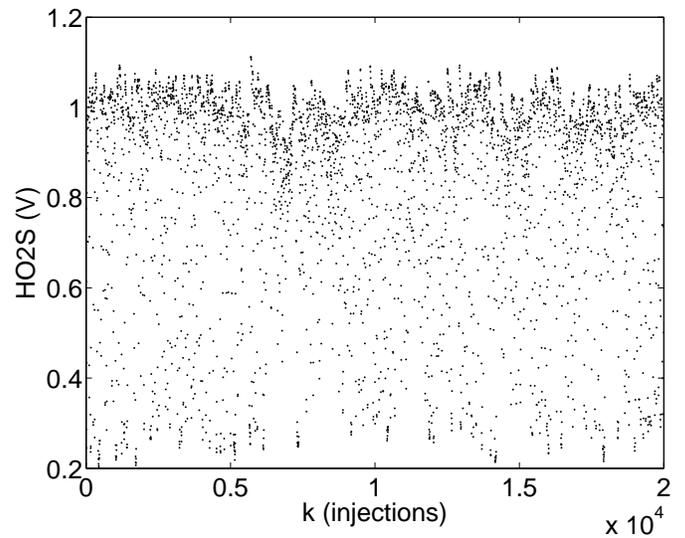


Figure 8: The Heated Oxygen Sensor (HO2S) was used for both feedback to the PI controller as well as for training the neural network. Notice the limit cycle behaviour.

Exhaust emission data was also recorded using an ECOM AC portable emission analyzer and sample dryer. Note that the calibration of this instrument was not certified, so the measurements should be used for comparison purposes only. The two pollutants measured were carbon monoxide (CO) and nitrogen oxides (NOx), as shown in Figures 9 and 10, along with measurements from the OEM controller for comparison. The mean carbon monoxide concentration was 129.5 ppm with a maximum of 788 ppm for the neural-based controller, compared to the OEM controller with a mean of 306.4 ppm and a peak of 5958 ppm. The mean and maximum NOx concentrations were 5.11 ppm and 207 ppm for the neural controller, which may be compared to values from the OEM controller of 54.5 ppm and 260 ppm. These preliminary results indicate that the controller presented in this paper has performance comparable to the OEM controller (if not exceeding) with regards to exhaust emissions. Furthermore, no misfires were detected, and the engine seems to run smoothly, after the first few seconds of rough idle.

With regard to computational performance, the MPC555 CPU could handle the calculation of the pulse width within the RPM limits of the engine. The pulse width calculation will take more processing power than a simple look-up table; measurements indicate that the neural pulse width calculation takes approximately 0.335 ms, the learning phase takes 2.03 ms, and the complete loop takes 2.45 ms per injection. For an eight cylinder engine, this corresponds to a maximum engine speed of 6100 RPM. It should be noted that the code used was relatively unoptimized, and one can expect the possibility of significant improvements through optimization, especially in the learning section. The memory requirements of the algorithm are comparable

to a lookup table; a network with N neurons requires $(N-4)*(N+1)/2+1$ weights. The 15-neuron network used in this experiment required 89 weights, less than a 10×10 lookup table. Additionally, the software requires an N element vector to store the neuron outputs, a temporary $(N-4)*(N+1)/2+1$ item vector to store the gradients of the weights, and four circular buffers with lengths equal to the engine's maximum delay.

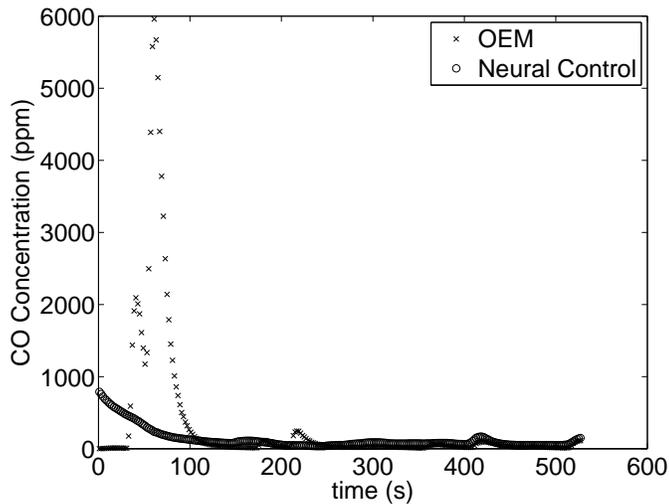


Figure 9: Exhaust gas carbon monoxide concentration measurements were recorded during the same test, shown here with OEM results for comparison. These concentrations are for dry exhaust.

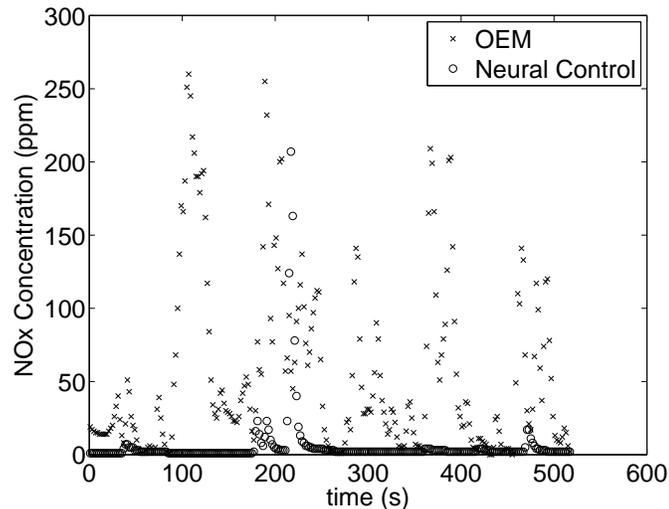


Figure 10: Exhaust gas NOx concentration measurements were recorded during the same test, shown here with OEM results for comparison. These concentrations are for dry exhaust.

CONCLUSION

This paper presents preliminary experimental verification of a new type of adaptive fuel-air controller. These preliminary results demonstrate that the scheme has exhaust emissions performance that is comparable to, or exceeding that of the OEM controller, while eliminating the costly and time-consuming process of calibrating the fuel map. It should be emphasized that these results are preliminary and apply to only one vehicle; however, it is believed that the concept is very sound and can be applied to many other situations. Further, it is expected that these “unoptimized” results can be improved through tuning the algorithm. This can be achieved by improving the learning rate by using more advanced training schemes than the simple back propagation algorithm, and also improving the computational speed so that more neurons may be used in the same calculation period. Further work may also include applying the same controller to various engines to verify the “universal” nature of the controller, and continuing ongoing long-term tests to evaluate the controller’s performance as the engine wears.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of NSERC, in the form of a PGS D Scholarship; the Saskatchewan Research Council for providing equipment and expertise; and General Motors Alternative Fuels for the loan of the test vehicle. Thanks are also given to Natural Resources Canada and Precarn Inc who funded initial work in this area.

REFERENCES

1. Isermann, R. and Muller, N. “Design of Computer Controlled Combustion Engines.” *Mechatronics*, 13:1067–1087, 2003.
2. Taylor, Charles Fayette. *The Internal-Combustion Engine in Theory and Practice*, volume I. M.I.T. Press, Cambridge, second edition, 1985.
3. Heywood, J.B. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
4. Wiens, T., Burton R., Schoenau, G. and Sulatisky, M. *Intelligent Fuel Air Ratio Control of Gaseous Fuel SI engines*. Technical Report, Saskatchewan Research Council, Saskatoon, 2006.
5. Werbos, P. “Backpropagation Through Time: What It Does and How to Do it.” *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.
6. Haykin, S. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, New Jersey, 2nd edition, 1999.
7. Saad, D., editor. *On-line Learning in Neural Networks*. Cambridge University Press, Cambridge, 1998.
8. Nguyen, D. and Widrow, B. “Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values for the Adaptive Weights.”

Proceedings of the International Joint Conference on Neural Networks, pages 21–26, San Diego, 1990.

9. Wiens, T., Burton, R., Schoenau, G., Sulatisky, M., Hill, S., and Lung, B. Experimental Determination of Transport Delay in a Spark Ignition Engine. Technical Report, Saskatchewan Research Council, Saskatoon, 2006.

CHAPTER 9

CONCLUSIONS, CONTRIBUTIONS AND RECOMMENDATIONS

As stated in Chapter 1, the objective of this research program was to reduce the development cost of alternative fuelled vehicles by eliminating the calibration period of the fuel-air controller. The conclusion of this dissertation is that this was achieved through the development of a new type of neural control algorithm. The control algorithm is able to automatically generate a model of the fuel-air and combustion process, which is used to optimize the fuelling for stoichiometric operation.

While this controller was verified using expensive laboratory equipment, the process used to develop it was very inexpensive, not requiring any expensive equipment or sensors such as a dynamometer or emissions measuring equipment. This demonstrates that modern fuel controllers can be developed on a small budget.

The major contributions of this work are:

- collection and summarization of previous work,
- development of a method of automatically determining the pure time delay between the fuel injection event and the feedback measurement,
- development of a more accurate model for the variability of the transport delay in modern port injection engines,
- developing a fuel-air controller requiring minimal knowledge of the engine's parameters,
- development of a method of algebraically inverting a neural network which is much faster than previous iterative methods,
- demonstrating how to initialize the neural model by taking advantage of some important characteristics of the system,
- expansion of the models available for the limit cycle produced by a system with a binary sensor and delay to include integral controllers with asymmetrical gains,
- development of a limit cycle model for the new neural controller, and

- experimental verification of the controller's tailpipe emissions performance, which compares favourably to the OEM controller.

A number of recommendations may be made based on the outcomes of this work. The most important is that this control algorithm be further optimized and tested for long-term use on a variety of vehicles. While the algorithm presented has been verified to work well on the Vortec engine, the control scheme should work on a wide variety of engines, possibly including liquid fuel engines. Also, a number of parameters may be further optimized, such as the number of neurons and learning rate of the network. Finally, although preliminary results show that significant computational speed improvements may be achieved by choosing the activation function wisely (as presented in Appendix B), further research is required to improve the computational performance of the algorithm.

REFERENCES

- [1] M. Sulatisky, S. Hill, J. Lychak, K. Nakamura, T. Matsui, and G. Rideout. Adapting a Geo Metro to Run on Natural Gas Using Fuel-Injection Technology. Technical Report 951942, SAE, Warrendale, 1995.
- [2] H. Nagaishi, T. Nakazawa, K. Abe, and Y. Sasaki. Engine Fuel Injection Control Amount Device. US Pat 699068, 2006.
- [3] J.B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw Hill, New York, 1988.
- [4] S. J. Cornelius. *Modelling and Control of Automotive Catalysts*. PhD thesis, Sidney Sussex College, University of Cambridge, 2001.
- [5] S. Hill, M. Sulatisky, J. Lychak, N. Nakamura, T. Matsui, and G. Rideout. A Lean-burn, Sub-compact Natural Gas Vehicle. Technical Report 961676, SAE, Warrendale, 1996.
- [6] P. Kaidantzis, P. Rasmussen, M. Jensen, T. Vesterholm, and E. Hendricks. Robust, Self-calibrating Lambda Feedback for SI Engines. Technical Report 930860, SAE, Warrendale, 1993.
- [7] S. Hill and B. Lung. The Application of Neural Controls to Ford and Dodge Pickup Trucks Running on Natural Gas. Technical Report 11305-4C02, Saskatchewan Research Council, Saskatoon, 2003.
- [8] D.O. Hebb. *The Organization of Behavior; a Neuropsychological Theory*. Wiley-Interscience, New York, 1949.
- [9] F Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *In Psychological Review*, 65:386–408, November 1958.
- [10] P. Werbos. Backpropagation Through Time: What It Does and How to Do it. *In Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.
- [11] M. Hagan and M. Menhaj. Training Feedforward Networks with the Marquardt Algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.
- [12] J.A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308–313, 1965.
- [13] L. Bottou. *On-line Learning in Neural Networks*, chapter Online Learning and Stochastic Approximations, pages 9–42. Cambridge University Press, Cambridge, 1998.
- [14] G.D. Magoulas, V.P. Plagianakos, and M.N. Vrahatis. Adaptive stepsize algorithms for online training of neural networks. *Nonlinear Analysis*, 47:3425–3430, 2001.
- [15] T. B. DeMarse, D. A. Wagenaar, A.W. Blau, and S. M. Potter. The Neurally Controlled Animat: Biological Brains Acting with Simulated Bodies. *Autonomous Robots*, 11:305–310, 2001.
- [16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran 77*. Cambridge University Press, Cambridge, 2001.

Note that this reference list does not include the references which appear only in the papers.

APPENDIX A

NOMENCLATURE

This appendix includes a list of symbols used throughout the text and included papers. Due to the inclusion of papers which are independent works, there is some repetition of symbols. In the case of repeated symbols, refer to the symbol's definition within the scope of the work in question.

AF	Air-fuel mass ratio
AF_s	Stoichiometric air-fuel mass ratio
A_i	Intake manifold mean cross sectional area
A_{ex}	Exhaust manifold mean cross sectional area
d	Delay in samples
d_{max}	Maximum delay in samples
d_{min}	Minimum delay in samples
\hat{d}	Estimate of delay
E	Training error
\mathbf{g}	Error gradient
k	Sample index
k_d	Delay in samples
K	General purpose constant
K_i	Integral gain
K_{il}	Lean integral gain
K_{ir}	Rich integral gain
K_p	Proportional gain
l_i	Intake manifold effective length
l_{ex}	Exhaust manifold effective length
\dot{m}_a	Mass charge mixture flow
M_a	Molecular weight of air
M_f	Molecular weight of fuel
m_i	Mass of fuel injected
m_{is}	Stoichiometric fuel mass
M_{mix}	Molecular weight of the charge mixture
N	Number of training data points
N	Number of neurons
N_{cyl}	Number of cylinders
N_e	Engine speed (RPM)
N_{emax}	Maximum engine speed (RPM)
P	Period of oscillations
P_{atm}	Ambient atmospheric pressure
P_a	Ambient atmospheric pressure
P_e	Exhaust manifold absolute pressure
P_{FS}	Full scale reading of intake manifold pressure sensor
P_m	Intake manifold absolute pressure
r	Compression ratio
r_c	Compression ratio
t_d	Pure time delay (time)
t_i	Injector pulse width
t_{inj}	Injector pulse width
t_{ios}	Pulse width offset
\hat{t}_{is}	Estimate of stoichiometric pulse width
t_{is}	Stoichiometric pulse width

t_{isc}	Pulse width scale
T_a	Intake air temperature
T_{atm}	Atmospheric temperature
T_c	Engine coolant temperature
T_m	Intake air temperature
u_i	Network input i
\mathbf{u}	Network input vector
V_{cyl}	Displacement of each cylinder
V_{ex}	Volume between exhaust valve and EGO sensor
V_i	Volume between injector and intake valve
x_i	Output of neuron i
X	Neuron Output
$W_{i,j}$	Network weight connecting neurons i and j (generalized neural network)
$W_{i,j}$	Network weight connecting input i and neuron j (single layer perceptron)
W_i	Network weight connecting neuron i to the output neuron (generalized neural network)
\mathbf{X}	Network input vector
y	Network output
\hat{y}	Network output
y_d	Desired network output
y^*	Desired network output
Y	Binary oxygen sensor output
\hat{Y}	Estimate of Y
Y_i	Oxygen sensor output with delay compensation
α	Cutoff ratio
α	Momentum constant
γ	Ratio of specific heats
ε	Difference between network output and desired value
λ	Relative fuel-air ratio
λ_i	Intake relative fuel-air ratio
$\eta_{D,th}$	Diesel cycle theoretical thermal efficiency
$\eta_{O,th}$	Otto cycle theoretical thermal efficiency
η_v	Volumetric efficiency
η	Learning rate
Φ	hidden portion of neural network
σ	Weighted sum of neuron inputs
ρ_i	Intake density
ρ_{atm}	Atmospheric density
ρ_{ex}	Exhaust density
Θ	Vector of network weights

APPENDIX B

BENCHMARKING THE PERFORMANCE OF ACTIVATION FUNCTIONS IN GENERALIZED NEURAL NETWORKS

This appendix includes a paper investigating the performance of a number of activation functions used for artificial neural networks. This was achieved by benchmarking the time required to perform the calculation of each activation function, as well as its derivative, which is required for gradient-based training schemes. In addition, a test was performed to determine whether a faster computational time is offset by requiring more training data to achieve a desired accuracy.

These initial results indicate that there are better choices than the commonly used hyperbolic tangent function. Although these results refer to Generalized Neural Networks, they are believed to be applicable to other neural network forms.

This paper has not yet been submitted for publication

Benchmarking the Performance of Activation Functions in Generalized Neural Networks

Travis Wiens

September 19, 2008

Abstract

In many situations, the speed that an artificial neural network performs calculations is important. A major component of the processor time required to perform a neural computation is devoted to the calculation of the non-linear activation function. This paper investigates the computational performance of a number of activation functions during training and feed-forward operations of networks used for function approximation. Experimental benchmarks were performed using both a high-speed Intel CPU and also on a much slower embedded PowerPC microcontroller. It was shown that the commonly used hyperbolic tangent activation function is generally a poorer choice than other faster functions.

1 Introduction

Artificial neural networks are a type of arbitrary function approximator or classifier, composed of a large number of simple processing elements, termed neurons. The output of each neuron is typically a nonlinear activation function applied to the weighted sum of the neuron's inputs. A number of these processing units are connected in a network which may then be "trained" by adjusting each neuron's weights such that network output approximates the desired output.

The network used in this study is the generalized neural network [Werbos, 1990]. This network is the most general connection scheme with no feedback (as shown in Figure 1). In this scheme, the first N_{input} neurons are input neurons, whose output are the network inputs. The first hidden neuron, numbered $N_{input} + 1$, has inputs of the network inputs. The next neuron is connected to the input neurons, as well as the output from neuron $N_{input} + 1$. This continues throughout the network, with each neuron having inputs from all the neurons before it and providing outputs to all the neurons after it. Finally, the output neuron is used to provide the network output. Due to this connection scheme, the matrix of weights is triangular (a fully populated matrix would require feedback), and has $1/2(N_t + N_{input} + 1)(N_t - N_{input})$ weights for N_t total neurons.

In addition to the calculation of $N_{weights}$ multiply and accumulate operations, N_t nonlinear activation function calls are required. This paper is mainly concerned with the calculation speed of these functions and their derivatives.

While others have performed benchmarking of neural networks, most concentrate on the effect of different training schemes or only consider the number of number of training examples required for convergence, ignoring the time required [Hammadi and Ito, 1998], [Mayoraz, 1990], [Menon et al, 1996], [Sopena, 1999], [Tan, 1995], [Liang 1995]. While in many situations, the amount of data is a limiting factor, there are classes of problems which are time-dependent. This may include problems where the training or calculation of the network must be performed in real time at a specified sampling the rate. One example of this is the calculation of automotive fuel injection pulses, in which the pulse width calculation *must* be completed a rate set by the engine speed [Wiens, 2007]. Another class of example is very large problems which require vast amounts of processor time on expensive hardware. This paper studies a number of activation functions in order to determine their performance in the time domain.

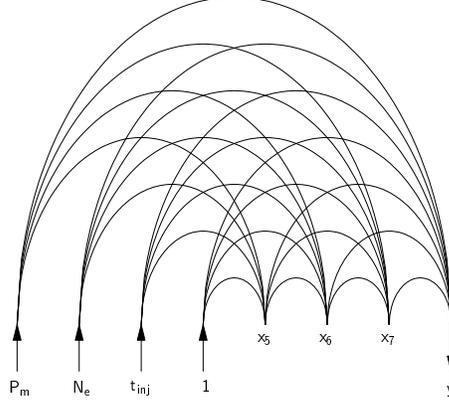


Figure 1: A generalized neural network (GNN) used for automotive applications. This network has four inputs (P_m , N_e , t_{inj} and a constant 1), three hidden neurons (labelled x_5 , x_6 and x_7) and one output neuron (\hat{y}). Each line represents a weight (W_{ij}) connecting the inputs to the neurons as well as connecting the neurons to each other. A multi-layer perceptron may be converted to this form by setting certain weights to zero.

2 Activation Functions

In order to approximate a non-linear function, the neural network must include non-linear elements. This requirement is fulfilled by the activation function, which is applied to the weighted sum of each neuron's inputs. The output of neuron i is given by

$$X_i = \sigma \left(\sum_{j=1}^{j=i-1} W_{ij} X_j \right) \quad (1)$$

where $\sigma(\cdot)$ is the nonlinear activation function and W_{ij} is the weight connecting neurons i and j . In many training schemes, particularly backpropagation, some form of gradient descent is used to optimize the weights. This is typically performed using the chain rule, meaning that the derivative

$$\frac{\partial \sigma(x)}{\partial x} \quad (2)$$

must also be calculated for each neuron. [Werbos [1990]].

In much of the literature, the exponential function hyperbolic tangent,

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}, \quad (3)$$

or the exponential sigmoid,

$$\text{sigmoid}(x) = \frac{1}{e^{-x} + 1}, \quad (4)$$

are used. These functions are generally used because of the speed of calculating the derivative:

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial \tanh(x)}{\partial x} = \text{sech}^2(x) = 1 - \sigma^2(x) \quad (5)$$

for tanh and

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial \text{sig}(x)}{\partial x} = \frac{e^{-x}}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x)) \quad (6)$$

for the exponential sigmoid. While calculation of the derivatives is very fast (one subtraction and one multiplication each), the calculation of $\tanh(x)$ and $\text{sigmoid}(x)$ are both computationally expensive, due to the exponential involved.

An alternative to calculating the exponential function is to use an approximation based on its Taylor series [Elliott, 1993]. The Taylor series of the exponential function is

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \quad (7)$$

If the first order Taylor series is used in the equation for tanh, an approximation may be found:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \approx \frac{(1 + 2x) - 1}{(1 + 2|x|) + 1} = \frac{x}{1 + |x|} \quad (8)$$

which will be known as `tanhtaylor(x)` in this paper. This function is significantly faster to compute (one addition, one absolute value and a division) than `tanh(x)`. The function can be rescaled to the range of (0,1) if a unipolar sigmoid is required:

$$\text{squash}(x) = 0.5 + \frac{0.5x}{1 + |x|}. \quad (9)$$

The derivatives of these functions can be shown to be

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial \text{tanhtaylor}(x)}{\partial x} = \frac{1}{(1 + |x|)^2} = (1 - |\sigma(x)|)^2 \quad (10)$$

and

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial \text{squash}(x)}{\partial x} = \frac{0.5}{(1 + |x|)^2} \quad (11)$$

$$= \begin{cases} 2(1 - \sigma(x))^2 & \text{if } x > 0 \\ 2\sigma(x)^2 & \text{otherwise} \end{cases} \quad (12)$$

which is, like the exponential sigmoids' derivatives, relatively easy to compute, if one has already computed $\sigma(x)$.

One final activation function that has been suggested in the literature [Menon et. al, 1996] is the algebraic sigmoid, given by

$$\text{algsig}(x) = \frac{x}{\sqrt{1 + x^2}}, \quad (13)$$

with a derivative of

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial \text{algsig}(x)}{\partial x} = \frac{1}{(1 + x^2)^{3/2}} = (1 - \sigma(x)^2)^{3/2}. \quad (14)$$

This function is not particularly fast as both the function and its derivative require a square root operation.

3 Benchmarking Procedure

This section introduces the procedures used to evaluate the performance of the previously mentioned activation functions. When discussing the performance of an activation function, a number of metrics are important:

- Time to calculate the function. This is important when the training speed of the network is not as important as the forward calculation speed. This situation occurs when the network is previously trained off-line and only the realtime forward performance is of interest.
- Time to calculate the function and its derivative. This is important when one is interested in minimizing the time to complete a training iteration for a single data point. This may be of interest when training must be performed at a high sample rate or on slower embedded hardware.

- Number of samples required to train to a desired accuracy. Because of the different shape and numerical conditioning, some activation functions converge faster than others. This may be important where data is expensive to generate, but training time is not a large concern.
- Time to train to a desired accuracy. This refers to the total time required to train a neural network to desired approximation error. Since one activation function may be slower to calculate, but allow for the network to be trained in less iterations, this metric measures the overall performance of the activation function.

Since the implementation of the activation functions can have a strong effect on its performance, the C code used to calculate each of the activation functions has been included in Appendix A.

The simplest method of determining the time to calculate a function is to simply measure the CPU time taken to perform the calculation. However, on modern computers, the operating system is not devoted to the single task of running the program, so some variation in time may be expected. Therefore, in this experiment, the calculation was performed 10000 times and elapsed time recorded. The same loop was then run 10000 times with the activation function removed to determine the loop overhead, which is then subtracted from the activation function’s time. This was repeated 10000 times for each function such that confidence limits of the mean could be determined using the standard deviation and Student’s t-distribution for a 95% confidence interval. The results of this test may be found in the next section.

Due to the somewhat complex interactions between the compiler and code, a more practical test was also required. In the next test, a network was trained, using standard backpropagation, to approximate a function with two inputs. At least 1000 training examples were presented and the time taken was measured. This was repeated at least 100 times to determine a confidence interval for the mean. This was performed for networks with a size between 4 and 30 total neurons for each activation function, as well as for a direct linear activation function so that overhead could be compensated for. For each activation function, a linear regression was applied to determine the K constants in

$$t = K_1 N_{activations} + K_2 N_{weights} + K_3 \quad (15)$$

using the regression procedure found in Press et al [2001], which takes into account the varying confidence intervals for each point. In this equation K_1 represents the time taken to calculate each activation function, K_2 represents the time required for the application of the weights, and K_3 is the overhead required for the calculations.

Finally, a test was performed in which a network was trained to approximate the function

$$y = 0.5 \sin(\pi u(1)^2) * \sin(2\pi u(2)) \quad (16)$$

which is the benchmark function introduced by Nguyen and Widrow [1990].

Training was performed using online back-propagation with a learning rate of 0.01 and 25 total neurons, initialized by the Nguyen-Widrow method [Nguyen and Widrow, 1990]. Training was performed on randomly generated samples, with a verification performed every 1000 training iterations on 100 random verification points. This was repeated 100 times for each activation function such that the uncertainty in the mean could be determined. Training was stopped when the root mean squared verification error reached 0.05.

Each test was performed on a MacBook laptop with an Intel Core 2 Duo processor running at a clock speed of 2.160 GHz and the first two tests were repeated on a Motorola (now Freescale) PowerPC MPC555 microcontroller running at a clock speed of 40 MHz. In order to show the effects of compiler optimization, the code run on the MacBook was compiled using the -O0, -O1, -O2 and -O3 flags of the Gnu Compiler Collection (gcc), version 4.0.1 (Apple Computer, Inc. build 5367). The code for the 555 microcontroller was compiled using the default compiler strings for the Intec Project Manager, version 0.1.14, which is -O1 on gcc version 2.95.2. The times are reported in number of clock cycles (rather than seconds) for better ability to compare results between processors.

4 Results

The first test, in which the time taken to simply calculate the activation function was determined, resulted in the values in Tables 1 to 3. The results in these tables for the MacBook were compiled with optimization levels between -O0 (no optimization) to -O3 (heavy optimization) as well as for a Motorola 555 microcontroller (optimization level -O1). Table 1 shows the number of clock cycles taken to calculate the activation functions, Table 2 shows the time to calculate their derivatives, and Table 3 shows the sum of the two. In all the tables in this section, the uncertainties in the mean values were calculated with a 95% confidence interval. From these data, one can see that the fastest clock time during both training and forward operation is the tanhtaylor function, although the squash function is generally very close. The exponential calculations involved in the first three activation functions clearly use a considerable amount of processor time, and one should note that if a tanh activation function must be used, it is generally faster to use a function like `fast_tanh` instead (as shown in Appendix A). The algebraic sigmoid function is comparable to the exponential functions on the MacBook, but suffers considerably on the embedded microcontroller. This is most likely due to the square root function which is called in both the function and its derivative.

Table 1: Activation Function Calculation Time (clocks)

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0	MPC555 -O1
<code>tanh</code>	189.3+/-3.2	183.0+/-3.2	189.3+/-3.3	187.0+/-3.2	969.3+/-16.0
<code>fast_tanh</code>	103.0+/-1.7	132.4+/-2.3	160.1+/-2.8	210.6+/-3.6	867.5+/-14.0
<code>sigmoid</code>	102.1+/-1.7	131.5+/-2.3	131.3+/-2.3	195.9+/-3.4	769.7+/-13.0
<code>tanhtaylor</code>	28.5+/-0.5	44.0+/-0.9	43.4+/-0.9	85.4+/-1.6	88.0+/-2.0
<code>squash</code>	27.9+/-0.5	45.6+/-0.9	47.3+/-0.9	89.8+/-1.6	133.1+/-2.7
<code>alg_sigmoid</code>	84.9+/-1.5	79.2+/-1.5	84.7+/-1.5	130.8+/-2.3	4738.9+/-79.0

Table 2: Activation Function Derivative Calculation Time (clocks)

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0	MPC555 -O1
<code>dtanh</code>	6.0+/-0.1	5.7+/-0.2	6.1+/-0.2	37.9+/-0.8	45.0+/-1.2
<code>dtanh</code>	6.0+/-0.1	5.7+/-0.2	6.1+/-0.2	37.9+/-0.8	45.0+/-1.2
<code>dsigmoid</code>	6.0+/-0.1	6.7+/-0.2	6.5+/-0.2	40.8+/-0.8	44.0+/-1.2
<code>dtanhtaylor</code>	6.0+/-0.1	6.5+/-0.2	6.7+/-0.2	43.9+/-0.9	44.0+/-1.5
<code>dsquash</code>	12.8+/-0.3	16.6+/-0.4	16.8+/-0.4	59.5+/-1.1	62.1+/-1.5
<code>dalg_sigmoid</code>	53.9+/-0.9	48.0+/-0.9	48.0+/-0.9	106.8+/-1.9	4785.9+/-79.0

Table 3: Activation Function Plus Derivative Calculation Time (clocks)

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0	MPC555 -O1
<code>tanh</code>	195.3+/-3.3	188.7+/-3.4	195.4+/-3.5	224.8+/-4.0	1014.3+/-17.2
<code>fast_tanh</code>	109.0+/-1.8	138.1+/-2.5	166.2+/-3.0	248.4+/-4.4	912.5+/-15.2
<code>sigmoid</code>	108.0+/-1.8	138.2+/-2.5	137.8+/-2.5	236.7+/-4.2	813.7+/-14.2
<code>tanhtaylor</code>	34.5+/-0.6	50.5+/-1.1	50.1+/-1.1	129.2+/-2.5	132.0+/-3.5
<code>squash</code>	40.7+/-0.8	62.2+/-1.3	64.0+/-1.3	149.3+/-2.7	195.2+/-4.2
<code>alg_sigmoid</code>	138.8+/-2.4	127.2+/-2.4	132.7+/-2.4	237.6+/-4.2	9523.9+/-158.0

The second set of test results are shown in Table 4. In this test, a linear regression was performed to determine the constants in the equation

$$t = K_1 N_{activations} + K_2 N_{weights} + K_3. \quad (17)$$

Table 4 shows the values for K_1 , which represent the time required for the calculation of each activation function on a MacBook with four levels of optimization as well as on a Motorola 555

microcontroller. In each case the fastest training calculation occurred when using the `tanhtaylor` activation function and the associated derivative. The speed improvement can be as much as four times faster than the typically used `tanh` function. It is also interesting to note that the -O1 optimization level is generally faster than the -O2 level, so one must be careful when using compiler optimization. Also, when comparing the performance of the activation function on different hardware, the microcontroller uses a similar number of clock cycles for the `tanhtaylor` and `squash` functions. The exponentially based `tanh`, `fast_tanh` and `sigmoid` are approximately three times slower on the MPC555 than the Macbook, while the algebraic `sigmoid` requires approximately 30 times more cycles, likely due to the square root function.

Table 4: Calculation Time (clocks)

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0	MPC555 -O1
<code>tanh</code>	288.07+/-0.46	343.05+/-0.26	353.37+/-0.23	466.97+/-0.39	1166.84+/-0.86
<code>fast_tanh</code>	249.08+/-0.09	345.68+/-0.07	301.76+/-0.13	489.76+/-0.18	1066.41+/-0.55
<code>sigmoid</code>	244.36+/-0.06	335.67+/-0.05	320.85+/-0.08	470.31+/-0.14	1043.12+/-0.75
<code>tanhtaylor</code>	128.11+/-0.10	215.23+/-0.07	201.13+/-0.12	387.01+/-0.17	255.09+/-0.02
<code>squash</code>	147.94+/-0.13	232.32+/-0.12	207.72+/-0.14	387.69+/-0.32	368.00+/-0.14
<code>alg_sigmoid</code>	218.67+/-0.05	325.59+/-0.04	323.77+/-0.07	489.37+/-0.13	9636.43+/-0.18

Finally, the last test determined the time and number of training samples taken for each activation function to train a network to an RMS verification error of 0.05. Table 5 shows the mean number of training samples required to train the network such that the verification error reaches 0.05. The exponential `sigmoid` required the most training data, while its approximation, the `squash` function, required the least. Table 6 presents the processor time required for each training example, showing the speed of the activation function and its derivative. The `squash` and `tanhtaylor` functions are the fastest and the `tanh` function is slowest, although there are not very large differences between the fastest and slowest. This is where the compiler optimization level shows its effect; the nonoptimized -O0 code is much slower than the optimized codes (although the level of optimization has little effect). Table 7 is the total training time, which is the product of Tables 5 and 6. This shows the overall effect of each activation function, both in computational time and number of training examples. The fastest activation function is the `squash` function. One surprising activation function is the algebraic `sigmoid`, which performs better than the author expected.

Table 5: Training samples to train network (1e3 samples)

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0
<code>tanh</code>	830.76+/-0.04	870.33+/-0.04	838.70+/-0.04	835.34+/-0.04
<code>fast_tanh</code>	828.02+/-0.05	860.05+/-0.04	913.93+/-0.05	864.29+/-0.05
<code>sigmoid</code>	1995.83+/-0.13	1951.25+/-0.12	1918.88+/-0.11	2012.77+/-0.13
<code>tanhtaylor</code>	738.33+/-0.03	712.71+/-0.04	727.11+/-0.04	693.30+/-0.03
<code>squash</code>	164.44+/-0.01	177.48+/-0.01	175.11+/-0.01	173.54+/-0.01
<code>alg_sigmoid</code>	803.10+/-0.04	752.96+/-0.03	784.20+/-0.04	730.22+/-0.03

Table 6: Training time per sample (1e3 clocks/sample).

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0
<code>tanh</code>	20.15+/-0.05	20.75+/-0.04	21.94+/-0.05	43.95+/-0.05
<code>fast_tanh</code>	18.77+/-0.04	20.10+/-0.05	21.40+/-0.05	43.00+/-0.06
<code>sigmoid</code>	18.61+/-0.03	20.21+/-0.03	21.37+/-0.04	42.30+/-0.03
<code>tanhtaylor</code>	16.66+/-0.04	17.88+/-0.05	19.34+/-0.06	41.07+/-0.05
<code>squash</code>	16.64+/-0.13	17.93+/-0.10	19.37+/-0.11	41.10+/-0.12
<code>alg_sigmoid</code>	18.41+/-0.05	19.83+/-0.05	21.29+/-0.05	43.11+/-0.06

Table 7: Time to train network (1e9 clocks).

	MacBook -O3	MacBook -O2	MacBook -O1	MacBook -O0
<code>tanh</code>	16.74+/-0.80	18.06+/-0.94	18.40+/-0.92	36.72+/-1.68
<code>fast_tanh</code>	15.54+/-0.89	17.31+/-0.85	19.56+/-1.08	37.16+/-2.14
<code>sigmoid</code>	37.13+/-2.33	39.42+/-2.39	41.03+/-2.46	85.13+/-5.44
<code>tanhtaylor</code>	12.30+/-0.53	12.75+/-0.66	14.05+/-0.81	28.46+/-1.42
<code>squash</code>	2.74+/-0.16	3.19+/-0.14	3.40+/-0.21	7.13+/-0.38
<code>alg_sigmoid</code>	14.79+/-0.68	14.94+/-0.68	16.70+/-0.78	31.47+/-1.27

5 Conclusion

The paper presents a number of benchmark results for different activation functions used for artificial neural networks. The general conclusion is that choice of activation function can make a significant difference in computational time. However, the choice of activation function must be driven by consideration of what portion of the calculation speed is important. In order to achieve the fastest forward network calculation, the `tanhtaylor` and related `squash` functions were fastest. However, if the amount of training data is limited, one should note that the `tanhtaylor` function converged more slowly (for the test performed for this paper). In most cases, the commonly used, but computationally exponential functions showed little advantage over faster `tanhtaylor` and `squash` functions. The algebraic sigmoid showed surprisingly good performance, but the calculation of the square root was extremely slow on the embedded microcontroller. Due to the large differences between the embedded processor and the laptop, the best activation function may be different for different hardware and compiler optimization ability.

6 References

- Abu-Khalaf, M.; Lewis, F.L.; Huang, J. "Neural network H-infinity state feedback control with actuator saturation: The nonlinear benchmark problem", "Proceedings of the 5th International Conference on Control and Automation, ICCA'05", pp 1-9, 2005.
- Elliott, D.L. "A Better Activation Function for Artificial Neural Networks", Technical Report 93-8, Institute for Systems Research, University of Maryland, 1993.
- Hammadi, N.C.; Ito, H. "On the activation function and fault tolerance in feedforward neural networks", "IEICE Transactions on Information and Systems", volume E81-D, number 1, pp 66-72, 1998.
- Mayoraz, E. "Benchmark of some learning algorithms for single layer and Hopfield networks", "Complex Systems", v 4, n 4, p 477-90, 1990.
- Menon, A.; Mehrotra, K.; Mohan, C.K.; and Ranka, S. "Characterization of a Class of Sigmoid Functions with Applications to Neural Networks", "Neural Networks", v 9, n 5, pp 819-835, 1996.
- Nguyen, D. and Widrow, B. "Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values for the Adaptive Weights", "Proceedings of the International Joint Conference on Neural Networks", pp 21-26, San Diego, 1990.
- Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; and Flannery, B.P. "Numerical Recipes in Fortran 77", Cambridge University Press, Cambridge, 2001.
- Sopena, J.M.; Romero, E.; Alquezar, R. "Neural networks with periodic and monotonic activation functions: A comparative study in classification problems", "IEE Conference Publication", v 1, n 470, pp 323-328, 1999
- Tan, P.B.; Chua, H.C.; Chen, L.H.; Gong, Y.H. "Comparison of sigmoidal activation functions through maximum weight update", "IES Journal", v 35, n 1, pp 19-23, 1995.
- Werbos, P. "Backpropagation Through Time: What It Does and How to Do It." "Proceedings of the IEEE", v 78, pp 1550-1560, 1990.
- Travis Wiens, Rich Burton, Greg Schoenau, Mike Sulatisky, Sheldon Hill, Bryan Lung, "Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller". Technical Paper

2007-01-1339, SAE, Warrendale, 2007.

Liang, X.; Xuan W. "Study on constructing feedforward neural networks for benchmark problems", "WCNN '95. World Congress on Neural Networks. 1995 International Neural Network Society Annual Meeting", vol.2, pt. 2, pp 253-256 , 1995.

A Code Examples

Six activation functions and their derivatives were included in this study. The C code used for each is as follows.

A.1 Hyperbolic Tangent

This is the standard hyperbolic tangent function contained in the C standard library math.h .

```
double tanh(double x)
```

The derivative is

```
double dtanh(double x)
{
return 1-x*x;
}
```

A.2 Fast tanh

Since the standard tanh code above includes considerable overhead, such as error checking, this simpler function was used to put it on an equal footing with the `sigmoid` function. The derivative is the same as above.

```
double fast_tanh(double x)
{
double x_tmp;
x_tmp=exp(2*x);
return (x_tmp-1)/(x_tmp+1);
}
```

A.3 Exponential Sigmoid

This is the exponential sigmoid function $(1 + e^{-x})^{-1}$.

```
double sigmoid(double x)
{
return 1/(1+exp(-x));
}
```

The derivative is

```
double dsigmoid(double x)
{
return x*(1-x);
}
```

A.4 tanhtaylor

This is an approximation of tanh, using a first order Taylor series for e^x .

```
double tanhtaylor(double x)
{
return x/(((x<0)?-1:1)*x+1);
}
```

Its derivative is

```
double dtanhtaylor(double x)
{
double x_tmp;
x_tmp=1-(((x<0)?-1:1)*x);
return x_tmp*x_tmp;
}
```

A.5 Squash

This is a unipolar version of tanhtaylor, scaled to the range (0,1).

```
double squash(double x)
{
return 0.5+0.5*x/(((x<0)?-1:1)*x+1);
}
```

Its derivative is

```
double dsquash(double x)
{
double x_tmp;
x_tmp=(x<0.5)?(x):(1-x);
return 2*x_tmp*x_tmp;
}
```

A.6 Algebraic Sigmoid

This is the algebraic sigmoid.

```
double alg_sigmoid(double x)
{
return x/sqrt(1+x*x);
}
```

Its derivative is calculated using

```
double dalg_sigmoid(double x)
{
double x_tmp;
x_tmp=1-x*x;
return x_tmp*sqrt(x_tmp);
}
```

APPENDIX C

COMPUTER CODE

C.1 Simulation Code

This code was used to simulate an initial version of the controller. The seven files were compiled together using the GNU Compiler Collection(GCC) compiler, available from {<http://gcc.gnu.org/>}.

C.1.1 main.c

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include "model.h"
#include "controller.h"
#include "training.h"

#define N_POINTS 2000000
#define ETA_LINEAR 5e-4
#define ETA_OUTPUT 5e-5
#define ETA_HIDDEN 1e-1
#define ALPHA_FILT 0.005
#define N_V 10000
#define N_SAVE 1

void main(void)
{
    unsigned int k_mbuf=0;
    double HO2S[K_mbuffer];
    double t_i_s=0;
    double P_m_mbuf[K_mbuffer];
    double N_e_mbuf[K_mbuffer];
    double t_i_mbuf[K_mbuffer];
    double P_m_cbuf[K_cbuffer];
    double N_e_cbuf[K_cbuffer];
    double t_i_cbuf[K_cbuffer];
    double HO2S_cbuf[K_cbuffer];
    double P_m_out;
    double N_e_out;
    double t_i_out;
    double HO2S_out;
    double c;
    unsigned int i, j, ii, jj;
    int k_tmp;
    int k_cbuf=0;
    double W[N_NEURONS][N_NEURONS];
    double alpha=0.7;
    double dW[N_NEURONS][N_NEURONS];
    double eta[N_NEURONS][N_NEURONS];
    double E_t_i_sum=0;
    double E_lambda_sum=0;
```

```

double E_t_i_filt=0;
double E_lambda_filt=0;
unsigned int t_start, t_stop;
unsigned int t_1, t_2, t_3, t_4, t_5, t_6, t_7;
unsigned int t_12, t_23, t_34, t_45, t_56, t_67;
double t_i_s_nodelay=0;
FILE *stream_in;
FILE *stream_out;
FILE *W_stream_out;

init_W_redline(&W,N_NEURONS,N_INPUTS);
if ((stream_in = fopen("P_m_N_e_data.dat","r"))==NULL)
{
printf("Can't open %s \n","P_m_N_e_data.dat");
exit(1);
}
if ((stream_out = fopen("out_data.dat","w"))==NULL)
{
printf("Can't create %s \n","out_data.dat");
exit(1);
}
if ((W_stream_out = fopen("W_out_data.dat","w"))==NULL)
{
printf("Can't create %s \n","W_out_data.dat");
exit(1);
}
for (i=0;i<K_mbuffer;i++)
{
HO2S[i]=0;
P_m_mbuf[i]=80e3;
N_e_mbuf[i]=2000.0;
t_i_mbuf[i]=20e-3;
}
for (i=0;i<K_cbuffer;i++)
{
HO2S_cbuf[i]=0;
P_m_cbuf[i]=80e3;
N_e_cbuf[i]=2000.0;
t_i_cbuf[i]=20e-3;
}
init_eta(&eta,ETA_LINEAR,ETA_OUTPUT,ETA_HIDDEN,N_NEURONS);
for (i=0;i<(N_NEURONS);i++)
{
for (j=0;j<(N_NEURONS);j++)
{
dW[i][j]=0;
}
}
init_eta(&eta,ETA_LINEAR,ETA_OUTPUT,ETA_HIDDEN,N_NEURONS);
t_12=0;
t_23=0;
t_34=0;
t_45=0;
t_56=0;

```

```

t_67=0;
t_start=clock();
for (i=0;i<N_POINTS;i++)
{
t_1=clock();
while(fscanf(stream_in,"%lf
%lf\n",&P_m_cbuf[k_cbuf],&N_e_cbuf[k_cbuf])==EOF)
{
fclose(stream_in);
if ((stream_in = fopen("P_m_N_e_data.dat","r"))==NULL)
{
printf("Can't open %s \n","P_m_N_e_data.dat");
exit(1);
}
}
t_2=clock();
P_m_mbuf[k_mbuf]=P_m_cbuf[k_cbuf];
N_e_mbuf[k_mbuf]=N_e_cbuf[k_cbuf];

t_i_cbuf[k_cbuf]=gnncc_cfti(P_m_cbuf[k_cbuf],N_e_cbuf[k_cbuf],&W,N_NEURONS);
t_i_mbuf[k_mbuf]=t_i_cbuf[k_cbuf];
k_tmp=k_mbuf;
t_3=clock();
targetmodel(&HO2S, &t_i_s, &P_m_mbuf, &N_e_mbuf,
&t_i_mbuf,&k_mbuf);
t_4=clock();
HO2S_cbuf[k_cbuf]=HO2S[k_tmp];
cond_aggr(&P_m_out, &N_e_out, &t_i_out, &HO2S_out, &c,
&P_m_cbuf, &N_e_cbuf, &t_i_cbuf, &HO2S_cbuf,&k_cbuf);
t_5=clock();
if (c==1)
{
update_W(&W, &dW, P_m_out, N_e_out, t_i_out, HO2S_out,
N_NEURONS, &eta, alpha);
}
t_6=clock();

t_i_s_nodelay=targetmodel_nodelay(P_m_out, N_e_out, t_i_out);
if (((double)i/N_SAVE)==(i/N_SAVE))
{
fprintf(stream_out,"%e %e\n",t_i_out,t_i_s_nodelay);
}

t_7=clock();
E_t_i_sum=E_t_i_sum+fabs((t_i_out-t_i_s_nodelay));

E_lambda_sum=E_lambda_sum+fabs(1-(t_i_s_nodelay-T_vn)/(t_i_out-T_vn));

E_t_i_filt=ALPHA_FILT*fabs((t_i_out-t_i_s_nodelay))+(1-ALPHA_FILT)*E_t_i_filt;

E_lambda_filt=ALPHA_FILT*fabs(1-(t_i_s_nodelay-T_vn)/(t_i_out-T_vn))+(1-
ALPHA_FILT)*E_lambda_filt;
t_12=t_12+t_2-t_1;
t_23=t_23+t_3-t_2;

```

```

t_34=t_34+t_4-t_3;
t_45=t_45+t_5-t_4;
t_56=t_56+t_6-t_5;
t_67=t_67+t_7-t_6;
//Verification
if (((double)i/N_V)==(i/N_V))
{
printf("i=%d E_t_i_filt=%e
E_lambda_filt=%e\n",i,E_t_i_filt,E_lambda_filt);

for (ii=0;ii<(N_NEURONS);ii++)
{
for (jj=0;jj<(N_NEURONS);jj++)
{
fprintf(W_stream_out,"%0.4e
",W[ii][jj]);
}
}
fprintf(W_stream_out,"\n");
}
t_stop=clock();
for (i=0;i<(N_NEURONS);i++)
{
for (j=0;j<(N_NEURONS);j++)
{
fprintf(W_stream_out,"%0.4e ",W[i][j]);
}
}
fprintf(W_stream_out,"\n");
printf("RMSE t_i=%e s
lambda=%e\n",E_t_i_sum/N_POINTS,E_lambda_sum/N_POINTS);
printf("Fitlered E t_i=%e s lambda=%e\n",E_t_i_filt,E_lambda_filt);
printf("Elapsed time including engine simulation is %d ticks=%f
s\n", (t_stop-t_start), (double) (t_stop-t_start)/CLOCKS_PER_SEC);
printf("This is %f s per injection or %f s scaled from 1.7GHz to 40
MHz\n", (double) (t_stop-t_start)/CLOCKS_PER_SEC/N_POINTS, (double) (t_stop-t_start)
/CLOCKS_PER_SEC/N_POINTS*1700/40),
printf("\t read \t cntrl \t sim \t aggrg \t train \t write \n");
printf("ticks \t %d \t %d \t %d \t %d \t %d \t %d
\n",t_12,t_23,t_34,t_45,t_56,t_67);
printf("Control, aggregate, and train is %d ticks = %f s or %f
s/injection\n", (t_23+t_45+t_56), (double) (t_23+t_45+t_56)/CLOCKS_PER_SEC, (double)
(t_23+t_45+t_56)/CLOCKS_PER_SEC/N_POINTS);
printf("This is %f, scaled 1700 GHz to 40
MHz\n", (double) (t_23+t_45+t_56)/CLOCKS_PER_SEC/N_POINTS*1700/40);
fclose(stream_in);
fclose(stream_out);
fclose(W_stream_out);
}

```

C.1.2 model.h

```
#define gamma_f 1.292
```

```

//;%specific heat ratio of fuel
#define M_a 28.962
//;%(kg/kmol) Molecular weight of air
#define M_f 16.783
//;%(kg/kmol) Molecular weight of fuel
#define AF_ms 9.21
//;%Molar air fuel ratio (average=stoich hopefully)
#define M_mix 27.7691
//=(gnncc_par.targetmodel.M_a*gnncc_par.targetmodel.AF_ms+gnncc_par.targetmodel.
M_f)/(1+gnncc_par.targetmodel.AF_ms);%(kg/kmol) Molecular weight of AF_s mixture
#define AF_s 15.8935
//=gnncc_par.targetmodel.AF_ms*gnncc_par.targetmodel.M_a/gnncc_par.targetmodel.
M_f;%Mass air fuel ratio
//    %air
#define rho_a 1.21
//;%(kg/m^3) Intake air density (at atmospheric?)
#define P_atm 101e3
//0;%(Pa) Atmospheric pressure
#define T_a 273
#define r_c 9.5
//;%compression ratio
(gm.com/automotive/innovations/altfuel/vehicles/pickup/biFuelCNG/engines.htm)
#define n_cyl 8
//;%Number of cylinders
#define V_eng 5.967e-3
//;%(m^3) Engine displacement (wikipedia)
#define V_cyl 7.4587e-004
//;%(m^3)
#define T_vn 0.9481e-3
//;%injector offset (s) (these from 2003GMInjectotrFlowTests.xls)
#define K_i 354.3
//;%s/kg inverse injector flow coefficient (536.5 from prak)
#define T_c 363.0
//;%(K) coolant temp

//Min and max delay
#define Model_d_1 5
#define Model_d_2 7

//Model buffer
#define K_mbuffer 10

void targetmodel(double *HO2S_mbuf, double *t_i_s, double *P_m_mbuf,double
*N_e_mbuf,double *t_i_mbuf,unsigned int *k);
double targetmodel_nodelay(double P_m,double N_e,double t_i);

```

C.1.3 model.c

```

#include <stdlib.h>
#include <math.h>
#include "model.h"

void targetmodel(double *HO2S_mbuf, double *t_i_s, double *P_m_mbuf,double
*N_e_mbuf,double *t_i_mbuf,unsigned int *k)

```

```

{
unsigned int delay;
int k_d;
double P_e, eta_v;
delay=(unsigned int) (0.5+(double) rand(NULL)/((double)
RAND_MAX)*(Model_d_2-Model_d_1)+Model_d_1);
k_d=*k-delay;
while (k_d>=K_mbuffer)
{
k_d=k_d-K_mbuffer;
}
while (k_d<0)
{
k_d=k_d+K_mbuffer;
}
P_e=133.3*(P_atm/133.3+125*(*(N_e_mbuf+k_d)/4000)*(*(N_e_mbuf+k_d)/4000)*
exp(-(P_atm/133.3-*(P_m_mbuf+k_d)/133.3)/300));
eta_v=(P_m_mbuf+k_d)/P_atm*M_mix/M_a/(1+1/AF_s)*T_c/T_a*(r_c/(r_c-1)-(
P_e/*(P_m_mbuf+k_d)+gamma_f-1)/(gamma_f*(r_c-1)));
*t_i_s=K_i/AF_s*eta_v*V_cyl*rho_a+T_vn;
if (*(t_i_mbuf+k_d)>*t_i_s)
{
*(HO2S_mbuf+k)=1;
}
else
{
*(HO2S_mbuf+k)=0;
}
*k=*k+1;
if (*k >(K_mbuffer-1))
{
*k=0;
}
}

double targetmodel_nodelay(double P_m,double N_e,double t_i)
{
double P_e, eta_v;
double t_i_s;

P_e=133.3*(P_atm/133.3+125*(N_e/4000)*(N_e/4000)*exp(-(P_atm/133.3-P_m/
133.3)/300));
eta_v=P_m/P_atm*M_mix/M_a/(1+1/AF_s)*T_c/T_a*(r_c/(r_c-1)-(P_e/P_m+
gamma_f-1)/(gamma_f*(r_c-1)));
t_i_s=K_i/AF_s*eta_v*V_cyl*rho_a+T_vn;
return t_i_s;
}

```

C.1.4 controller.h

```

#define N_NEURONS 20
#define N_INPUTS 4
#define N_e_scale 6375

```

```

#define N_e_offset -237.5
#define P_m_scale 110625
#define P_m_offset 1.4375e+003
#define t_i_scale 0.0375
#define t_i_offset -3.7499999e-003
#define N_e_redline 6000.0
#define epsilon 1e-4
#define W_3 10
#define overlapfactor 0.7
#define P_max 101e3

void init_W_rand(double *W, unsigned int N_neurons, unsigned int N_inputs);
void init_W_matlab10(double *W, unsigned int N_neurons, unsigned int N_inputs);
void init_W_matlab6(double *W, unsigned int N_neurons, unsigned int N_inputs);
void sim_gnn(double *y_hat, double *x, double *u, double *W, unsigned int
N_neurons, unsigned int N_inputs);
double gnncc_cfti(double P_m, double N_e, double *W, unsigned int N_neurons);
void sim_gnn_linout(double *y_hat, double *x, double *u, double *W, unsigned int
N_neurons, unsigned int N_inputs);
double gnncc_sim(double P_m, double N_e, double t_i, double *W, unsigned int
N_neurons);
double gnncc_sim_linout(double P_m, double N_e, double t_i, double *W, unsigned int
N_neurons);
void gnncc_sim_x(double *y_hat, double *x, double P_m, double N_e, double t_i, double
*W, unsigned int N_neurons);
void init_W_redline(double *W, unsigned int N_neurons, unsigned int N_inputs);

```

C.1.5 controller.c

```

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#include "controller.h"

void init_W_rand(double *W, unsigned int N_neurons, unsigned int N_inputs)
{
    unsigned int i, j, seed;
    seed=time(NULL);
    srand(seed); //seed with time
    for (i=0; i<1000; i++) //generate some random number to get started
    {
        rand();
    }
    for (i=N_inputs; i<N_neurons; i++)
    {
        for (j=0; j<i; j++)
        {
            *(W+j*i*N_neurons)=(2*(double)rand()/RAND_MAX)-1;
        }
    }
    return;
}

```

```

void sim_gnn(
    double *y_hat,
    double *x,
    double *u,
    double *W,
    unsigned int N_neurons,
    unsigned int N_inputs
)
{
    unsigned int i, j;
    double net;
    for (i=0; i<N_inputs; i++)
    {
        *(x+i)=*(u+i);
    }
    for (i=N_inputs; i<N_neurons; i++)
    {
        net=0;
        for (j=0; j<i; j++)
        {
            net=net+*(W+j+i*N_neurons)**(x+j);
        }
        *(x+i)=1/(1+exp(-net));
    }
    *(y_hat)=*(x+N_neurons-1);
return;
};

void sim_gnn_linout(
    double *y_hat,
    double *x,
    double *u,
    double *W,
    unsigned int N_neurons,
    unsigned int N_inputs
)
{
    unsigned int i, j;
    double net;
    for (i=0; i<N_inputs; i++)
    {
        *(x+i)=*(u+i);
    }
    for (i=N_inputs; i<N_neurons; i++)
    {
        net=0;
        for (j=0; j<i; j++)
        {
            net=net+*(W+j+i*N_neurons)**(x+j);
        }
        *(x+i)=1/(1+exp(-net));
    }
*(y_hat)=net; //linear output

```

```

return;
};

double gnncc_sim(double P_m,double N_e,double t_i,double *W,unsigned int
N_neurons)
{
double u[N_INPUTS]={(P_m-P_m_offset)/P_m_scale,
(N_e-N_e_offset)/N_e_scale, (t_i-t_i_offset)/t_i_scale, 1};//maybe eliminate
division
double *px;
double y_hat;
int i;
px=(double *) malloc(N_neurons*sizeof(double));//pointer to x
sim_gnn(&y_hat,px,&u,W,N_neurons,N_INPUTS);
free(px);
return y_hat;
}

double gnncc_sim_linout(double P_m,double N_e,double t_i,double *W,unsigned int
N_neurons)
{
double u[N_INPUTS]={(P_m-P_m_offset)/P_m_scale,
(N_e-N_e_offset)/N_e_scale, (t_i-t_i_offset)/t_i_scale, 1};//maybe eliminate
division
double *px;
double y_hat;
int i;
px=(double *) malloc(N_neurons*sizeof(double));//pointer to x
sim_gnn_linout(&y_hat,px,&u,W,N_neurons,N_INPUTS);
free(px);
return y_hat;
}

double gnncc_cfti(double P_m,double N_e,double *W,unsigned int N_neurons)
{
double u[N_INPUTS]={(P_m-P_m_offset)/P_m_scale,
(N_e-N_e_offset)/N_e_scale, 0, 1};//maybe eliminate division
double *px;
double phi,t_i;
int i;
double *W_cfti;
W_cfti=(double *) malloc(N_neurons*N_neurons*sizeof(double));
for (i=0;i<N_neurons*N_neurons;i++)
{
*(W_cfti+i)=*(W+i);//copy W to W_cfti
}
for (i=0;i<N_INPUTS;i++)
{
*(W_cfti+(N_neurons-1)*N_neurons+i)=0;//clear linear output
neurons
}
for (i=0;i<N_neurons;i++)
{
*(W_cfti+i*N_neurons+2)=0;//clear t_i weights
}
}

```

```

}
px=(double *) malloc(N_neurons*sizeof(double)); //pointer to x
sim_gnn_linout (&phi, px, &u, W_cfti, N_neurons, N_INPUTS);
t_i=-t_i_scale/ *(W+(N_neurons-1)*N_neurons+2)*
*(W+(N_neurons-1)*N_neurons+0) *(P_m-P_m_offset)/P_m_scale+
*(W+(N_neurons-1)*N_neurons+1) *(N_e-N_e_offset)/N_e_scale+
*(W+(N_neurons-1)*N_neurons+3)+phi)+t_i_offset;
free(px);
free(W_cfti);
return t_i;
}

void gnncc_sim_x(double *y_hat, double *x, double P_m, double N_e, double t_i, double
*W, unsigned int N_neurons)
{
double u[N_INPUTS]={ (P_m-P_m_offset)/P_m_scale,
(N_e-N_e_offset)/N_e_scale, (t_i-t_i_offset)/t_i_scale, 1}; //maybe eliminate
division
int i;
int j;
sim_gnn(y_hat, x, &u, W, N_neurons, N_INPUTS);
return;
}

void init_W_redline(double *W, unsigned int N_neurons, unsigned int N_inputs)
{
unsigned int i, j, seed;
double W_1, W_4;
double *W_temp;
double W_mag, W_mag_sq, W_sum, W_lin_P_m;
unsigned int N_inputs_i;

W_temp=(double*)malloc(N_neurons*sizeof(double));

seed=time(NULL);
srand(seed); //seed with time
for (i=0; i<1000; i++) //generate some random number to get started
{
rand();
}
for (i=0; i<N_neurons; i++)
{
for (j=0; j<N_neurons; j++)
{
*(W+j+i*N_neurons)=0.0;
}
}

for (i=N_inputs; i<(N_neurons-1); i++)
{
N_inputs_i=i-2; //inputs except for t_i and 1
W_mag_sq=0;
W_sum=0;
for (j=0; j<N_inputs_i; j++)

```

```

{
W_temp[j]=(2*(double)rand()/RAND_MAX)-1;
W_mag_sq=W_mag_sq+W_temp[j]*W_temp[j];
}
W_mag=overlapfactor*pow((N_neurons-1)/2,(1/N_inputs_i));
for (j=0;j<N_inputs_i;j++)
{
W_temp[j]=W_mag/sqrt(W_mag_sq)*W_temp[j];
W_sum=W_sum+W_temp[j];
}
for (j=0;j<N_inputs-2;j++)
{
*(W+i*N_neurons+j)=2*W_temp[j];
}
for (j=N_inputs;j<N_inputs_i+2;j++)
{
*(W+i*N_neurons+j)=2*W_temp[j+N_inputs-6];
}

*(W+i*N_neurons+N_inputs-1)=((2*(double)rand()/RAND_MAX)-1)*W_mag-W_sum;
*(W+i*N_neurons+N_inputs-2)=0;//t_i
}
W_lin_P_m=60.0*2/N_e_redline/P_max;
W_1=-W_lin_P_m*W_3*P_m_scale/t_i_scale;//%weight for P_m
W_4=-t_i_offset*W_3/t_i_scale+W_1*P_m_offset/(P_m_scale*t_i_scale);//
bias.
*(W+(N_neurons-1)*N_neurons+0)=W_1;
*(W+(N_neurons-1)*N_neurons+1)=0;
*(W+(N_neurons-1)*N_neurons+2)=W_3;
*(W+(N_neurons-1)*N_neurons+3)=W_4;
for (i=N_inputs;i<N_neurons-1;i++)
{

*(W+(N_neurons-1)*N_neurons+i)=epsilon*((2*(double)rand()/RAND_MAX)-1);
}
free(W_temp);
return;
}

```

C.1.6 training.h

```

//model estimate of delay range
#define Cont_d_1 4
#define Cont_d_2 8
#define K_cbuffer 9

void gnn_static_gradient(double *F_w, double F_Y_hat, double *W, double *x,
const unsigned int n);
void cond_aggr(double *P_m_out, double *N_e_out, double *t_i_out, double
*HO2S_out, double *c, double *P_m_cbuf, double *N_e_cbuf, double *t_i_cbuf,
double *HO2S_cbuf,int *k_cbuf);
void update_W(double *W, double *dW,double P_m, double N_e, double t_i, double
HO2S, unsigned int N_neurons, double *eta, double alpha);
void init_eta(double *eta,double eta_linear,double eta_output,double

```

```
eta_hidden,unsigned int N_neurons);
```

C.1.7 training.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "training.h"
#include "controller.h"

void gnn_static_gradient(
    double *F_w,
    double F_Y_hat,
    double *W,
    double *x,
    const unsigned int n
)
{
    int i,j;
    double *F_x;
    double *F_net;

    F_x=(double *) malloc(n * sizeof(double));//allocate array
    F_net=(double *) malloc(n * sizeof(double));
    for (i=0;i<(n);i++)
    {
        F_x[i]=0;
        F_net[i]=0;
    }
    F_x[n-1]=F_Y_hat;
    for (i=n-1;i>=0;i--)
    {
        for (j=i+1;j<(n);j++)
        {
            F_x[i]=F_x[i]+*(W+j*n+i)*F_net[j];
        }
        if(i>3)//skip input neurons
        {
            F_net[i]=F_x[i]**(x+i)*(1-(x+i));//for sigmoid only
            for (j=0;j<=(i-1);j++)
            {
                *(F_w+i*n+j)=F_net[i]**(x+j);
            }
        }
    }
    free((void *) F_x);//free memory
    free((void *) F_net);
    return;
};
```

```
void cond_aggr(double *P_m_out, double *N_e_out, double *t_i_out, double *H02S_out, double *c, dou
{
int k_d;
```

```

int i;
*HO2S_out=*(HO2S_cbuf+*k_cbuf);
k_d=*k_cbuf+1;
while (k_d>=K_cbuffer)
{
k_d=k_d-K_cbuffer;
}
*P_m_out=*(P_m_cbuf+k_d);
*N_e_out=*(N_e_cbuf+k_d);
*t_i_out=*(t_i_cbuf+k_d);
*c=1;//Start with confidence=1
for (i=0;i<(Cont_d_2-Cont_d_1+1);i++)
{
k_d=*k_cbuf+i;
while (k_d>=K_cbuffer)
{
k_d=k_d-K_cbuffer;
}
if (*(HO2S_cbuf+k_d)!=*HO2S_out)
{
*c=0;
i=(Cont_d_2-Cont_d_1+1);//exit loop
}
}
*k_cbuf=*k_cbuf+1;
while (*k_cbuf>=K_cbuffer)
{
*k_cbuf=*k_cbuf-K_cbuffer;
}
}

void update_W(double *W, double *dW,double P_m, double N_e, double t_i, double HO2S, unsigned int N_neurons)
{
double F_Y_hat;
double *px;
double *pF_w;
double HO2S_hat;
int i,j;

pF_w=(double *) malloc(N_neurons*N_neurons*sizeof(double));//pointer to F_w
px=(double *) malloc(N_neurons*sizeof(double));//pointer to x
gnncc_sim_x(&HO2S_hat, px,P_m,N_e, t_i, W,N_neurons);
F_Y_hat=HO2S_hat-HO2S;
gnn_static_gradient(pF_w,F_Y_hat,W,px,N_neurons);
for (i=N_INPUTS;i<N_neurons;i++)
{
for (j=0;j<i;j++)
{
*(dW+j+i*N_neurons)=-*(eta+j+i*N_neurons) * *(pF_w+j+i*N_neurons) -alpha* *(dW+j+i*N_neurons);
*(W+j+i*N_neurons)=*(W+j+i*N_neurons)+*(dW+j+i*N_neurons);
}
}
free((void *) pF_w);
}

```

```

free((void *) px);
}

void init_eta(double *eta,double eta_linear,double eta_output,double eta_hidden,unsigned int N_neu
{
int i,j;
for (i=N_INPUTS;i<(N_neurons-1);i++)
{
for (j=0;j<(i);j++)
{
*(eta+i*N_neurons+j)=eta_hidden;
}
}
for (j=0;j<N_INPUTS;j++)
{
*(eta+(N_neurons-1)*N_neurons+j)=eta_linear;
}

for (j=N_INPUTS;j<(N_neurons-1);j++)
{
*(eta+(N_neurons-1)*N_neurons+j)=eta_output;
}

for (i=0; i<(N_neurons-1); i++)
{
*(eta+i*N_neurons+2)=0;//t_i
}
}

```

C.2 Microcontroller Code

This code was used to control the air-fuel ratio of the experimental vehicle. The code was compiled using the Intec Project Manager development environment and was implemented on a Motorola (now Freescale) Power PC MPC555 microcontroller. This microcontroller was only responsible for calculating the value of pulse widths; other input and output driver functionality was handled by other circuits on the control board.

```

/* _____

FILE:    GMTruckMAP.c
PROJECT: online.ipj
Version: 5.02
Copyright(c) October 28, 2003: Saskatchewan Research Council
March 19,2006- June 27, 2007: Travis Wiens

PURPOSE: MAP NN based speed density equation for GM Trucks
         Added online learning

ANB-1 EGO1
ANB-2 EGO2
ANB-3 GM MAF (not used)
ANB-4 IAT (or Fuel Temp for GM) (not used)
ANB-5 MAP
MPIO - 15 RPM (not used)
        MPIO - 14 REQ

```

```

                MPIO - 11 CYL6 GASOLINE INJECTOR (labelled as 5 on board)
ANB-1 EGO1
ANB-2 EGO2

//prettify with
//astyle --style=ansi <GMTruckMAP.c> outfile.c
_____/
//----- #INCLUDE -----

#include "m_common.h"
#include "timers.h" // functions for the PIT
#include "irq.h" // Functions for
IRQ0:7/FRZ&*IRQOUT
#include "ss_mios.h" // Functions for to the MIOS
#include "qadc.h"
#include "timing.h"
#include <stdio.h> // printf
#include <string.h>
#include<stdlib.h>
#include<math.h>

//----- GLOBAL VARIABLES -----
int reset_RPM;
int RPM_index;
int crank_sig;
int crank_flag;
int PIC_req_flag;
UInt64 RPM_start_period;
UInt64 RPM_end_period;
double RPM;
double RPM_new;
double RPM_1;
double RPM_2;

const double version=5.02;

//for get_bank()
#define N_CYL 8
int gbank_matrix[N_CYL]={2,1,2,1,1,2,1,2}; //bank for each injection,
{6,5,4,3,1,8,7,2}
int gMPIO_old=0; //storage of last reading

//----- # DEFINES -----
#define SC1SR *(pInt16) (0x30500C + INTERNAL_MEMORY_BASE)
#define SC2SR *(pInt16) (0x305024 + INTERNAL_MEMORY_BASE)
#define LED_FLASH_PERIOD 1000000
//delay to wait between data updates
#define MMFT_DELAY 2000
#define N_NEURONS 14
//was 15
#define N_INPUTS 4
#define N_e_scale 6875.0
#define N_e_offset -687.5
#define P_m_scale 6250.0

```

```

#define P_m_offset -625.0
//adjust for 0-30 ms output, not s
#define t_i_scale 37.5
#define t_i_offset -3.75
#define N_e_redline 6000.0
#define epsilon 1e-4
#define W_3 10.0
#define overlapfactor 0.7
//adjust for 5000 mv fullscale
#define P_max 5000.0
//transition point from rich to lean
#define HO2S_CUTOFF_DEFAULT 2500.0
// fudge factor for initial linear fueling
#define W_LIN_FUDGE_FACTOR 1.1
//value to assign to rich HO2S readings
#define RICH 1
#define LEAN 0
#define N_PARAMS ((N_NEURONS-4)*(N_NEURONS+1)/2+1)
//model estimate of delay range
#define Cont_d_1 8
#define Cont_d_2 18
#define K_cbuffer (Cont_d_2+1)
// was 5,10
//buffer must be 1 larger than delay for cond_aggr to work
#define ETA_LIN_DEFAULT 1e-4
#define ETA_OUT_DEFAULT 1e-4
#define ETA_HIDDEN_DEFAULT 1e-4
#define ALPHA_DEFAULT 0.0
#define EGO_PORT_1 2
#define EGO_PORT_2 1
//2 for EGO1, 1 for EGO2 (I think this was backwards)
#define LEARNING_DELAY 30000000
//time to wait before learning (us)
#define HO2S_START_1 5000
#define HO2S_START_2 5000
//voltage to reach before learning (mV)
#define RICH_BIAS_DEFAULT 1.0
#define K_P_RICH_DEFAULT -0.05
#define K_P_LEAN_DEFAULT 0.05
#define K_I_RICH_DEFAULT -0.005
#define K_I_LEAN_DEFAULT 0.01
#define K_P_RICH_PROP_DEFAULT 0
#define K_P_LEAN_PROP_DEFAULT 0
#define K_I_RICH_PROP_DEFAULT 0
#define K_I_LEAN_PROP_DEFAULT 0
#define K_I_LEAK_DEFAULT 0
#define INJ_MPIO 11
//MPIO PIN connected to injector
#define REQ_MPIO 14
//MPIO PIN connected to master REQ
#define BANK_SYNCH_MAX_RPM 1000
#define BANK_SYNCH_MAX_MAP 2500
//m_f=INJECTOR_SLOPE*(PULSE-INJECTOR_OFFSET)
#define INJECTOR_OFFSET 0.619

```

```

//ms
#define INJECTOR_SLOPE 2.703e-3
//g/ms
#define P_DITHER_DEFAULT 90
#define PW_DITHER_DEFAULT 45
#define AMP_DITHER_DEFAULT 0
#define BIAS_DITHER_DEFAULT 0.0
#define AUTONOMOUS_MODE_DEFAULT FALSE
#define VERBOSE_DEFAULT 0

typedef struct param_struct_
{
    int *pvehicle;
    double *pMAP;
    double *pRPM;
    double *pPULSE;
    double *pW_gnn_1;
    double *pW_gnn_2;
    double *peta;
    double *pHO2S_1;
    double *pHO2S_2;
    int *pk_datalog;
    double *palpha;
    double *pdW_1;
    double *pdW_2;
    int *plearning_flag_1;
    int *plearning_flag_2;
    double *prich_bias;
    int *pP_dither;
    int *ppw_dither;
    double *pbias_dither;
    double *pamp_dither;
    float *pHO2S1_cutoff;
    float *pHO2S2_cutoff;
    double *pK_p_rich;//PI constants
    double *pK_p_lean;//PI constants
    double *pK_i_rich;//PI constants
    double *pK_i_lean;
    double *pM_f;
    double *pI_1;
    double *pI_2;
    double *pK_p_rich_prop;//proportional PI constants
    double *pK_p_lean_prop;
    double *pK_i_rich_prop;
    double *pK_i_lean_prop;
    double *pK_i_leak;//term bringing K_i back to zero
    int *pautonomous_mode;//runs without master
    int *pverbose;//print warnings
}
param_struct;

//----- FUNCTIONS -----
float Logsig(float);
void IO_setup( void ); // Initialize MPIO/IRQ/PWM/ADC pins

```

```

void PWM_setup( void ); // Initializes PWM pins for PWM
operation
void ADC_setup( void ); // Initializes ADC pins for ADC
operation
void TB_setup(void); // Initializes PIT timer
void RS_COMM_SERVICE_struct(param_struct *params);
void POLL_PIC (UInt8);
void SET_DATA_READY (UInt8);
void SEND_PULSE_DATA_func(double PULSE_fn);
UInt64 FLASH_LED (UInt64 LED_old);
void GET_RPM_GM(void);
void GET_INPUTS(double *MAP_fn,double *HO2S_1_fn,double *HO2S_2_fn,UInt64
*DEC_fn, float HO2S1_cutoff, float HO2S2_cutoff);
void init_W_redline_lomem(double *W);
double gnn_pulse_calc_lomem(double P_m,double N_e,double *W);
void cond_aggr(double *P_m_out, double *N_e_out, double *t_i_out, double
*HO2S_out, double *c, double *P_m_cbuf, double *N_e_cbuf, double *t_i_cbuf,
double *HO2S_cbuf,int *k_cbuf);
void update_W_mom(double *pW,double P_m, double N_e, double t_i, double HO2S,
double *peta,double *pdW, double alpha);
void sim_gnn_lomem(double *py_hat, double *px,double *pu,double *pW);
void gnncc_sim_x_lomem(double *py_hat,double *px,double P_m,double N_e,double
t_i,double *pW);
int idx_lomem(int r, int c);
void gnn_static_gradient_lomem(double *F_w, double F_Y_hat, double *W, double
*x, const unsigned int n);
int adc_seed(int port);
void init_eta(double * p_eta, double eta_lin, double eta_out, double
eta_hidden);
int get_bank(int * k_fire,double P_m, double N_e, int verbose);
void bias_eta(double *peta_biased,double *peta_base,double factor, int
N_params);
double PI_control(double PULSE_base,double *I,double HO2S_old, double
HO2S,double K_p_rich,double K_p_lean,double K_i_rich,double K_i_lean);
double PI_control_prop(double PULSE_base,double *I,double HO2S_old, double
HO2S,double K_p_rich,double K_p_lean,double K_i_rich,double K_i_lean,double
K_p_rich_prop,double K_p_lean_prop,double K_i_rich_prop,double K_i_lean_prop);
void PI_leak(double *I,double K_i_leak);
double dither(double PULSE,int *k_dither,int P_dither, int pw_dither,double
amp_dither);
double dither_sin(double PULSE,int *k_dither,int P_dither, double
bias_dither,double amp_dither);
double fast_sin(double x);

//----- Mainline Program -----
int main ( void )
{
    double W_gnn_1[N_PARAMS];
    double W_gnn_2[N_PARAMS];
    double dW_1[N_PARAMS]={0};
    double dW_2[N_PARAMS]={0};
    double MAF;
    double MAP;
    double FT;

```

```

double PULSE=10;//pulse sent to master
double HO2S_1=RICH;
double HO2S_2=RICH;
double HO2S_old_1=RICH;
double HO2S_old_2=RICH;
int vehicle;
double P_m_cbuf_1[K_cbuffer]={0};
double N_e_cbuf_1[K_cbuffer]={0};
double t_i_cbuf_1[K_cbuffer]={0};
double HO2S_cbuf_1[K_cbuffer]={0};
double P_m_cbuf_2[K_cbuffer]={0};
double N_e_cbuf_2[K_cbuffer]={0};
double t_i_cbuf_2[K_cbuffer]={0};
double HO2S_cbuf_2[K_cbuffer]={0};
double P_m_out=3000;
double N_e_out=100;
double t_i_out=10;
double HO2S_out=RICH;
int k_cbuf_1=0;
int k_cbuf_2=0;
double c=0;
double PULSE_old=10;
double eta[N_PARAMS];
double alpha=ALPHA_DEFAULT;
int k_datalog=0;//number of injections b/w data writes
int i_datalog=0;
int learning_flag_1=0;
int learning_flag_2=0;
int bank=1;
int k_fire=0; //firing order number
double rich_bias=RICH_BIAS_DEFAULT;//this constant adjusts eta to bias
lambda rich when >1
double K_p_rich=K_P_RICH_DEFAULT;//PI constants
double K_p_lean=K_P_LEAN_DEFAULT;//PI constants
double K_i_rich=K_I_RICH_DEFAULT;//PI constants
double K_i_lean=K_I_LEAN_DEFAULT;//PI constants
double K_p_rich_prop=K_P_RICH_PROP_DEFAULT;//PI constants
double K_p_lean_prop=K_P_LEAN_PROP_DEFAULT;//PI constants
double K_i_rich_prop=K_I_RICH_PROP_DEFAULT;//PI constants
double K_i_lean_prop=K_I_LEAN_PROP_DEFAULT;//PI constants
double K_i_leak=K_I_LEAK_DEFAULT;//LEAKY I constant
double I_1=0;//Integrator
double I_2=0;
int P_dither=P_DITHER_DEFAULT;//period of "dither: signal
int pw_dither=PW_DITHER_DEFAULT;//pulse width
double amp_dither=AMP_DITHER_DEFAULT;//amplitude of dither (on unit signal)
double bias_dither=BIAS_DITHER_DEFAULT;//bias to add
int k_dither=0;//counter
float HO2S1_cutoff=HO2S_CUTOFF_DEFAULT;
float HO2S2_cutoff=HO2S_CUTOFF_DEFAULT;
double M_f;//mass of fuel used (g)
int autonomous_mode=AUTONOMOUS_MODE_DEFAULT;
int verbose=VERBOSE_DEFAULT;
param_struct params;

```

```

    UInt32 i;
    UInt64 LED_old;
    UInt64 DEC_mmft; //timer for get_mmft
    UInt64
t1=0,t2=0,t3=0,t4=0,t5=0,t6=0,t7=0,t8=0,t9=0,t10=0,t11=0,t1_old=0,t3_old=0;

    TMCLK_set (OSCM);
    DEC_set (0xFFFFFFFFFFFFFFFF);
    DEC_mmft= DEC_get ();
    LED_old = DEC_get ();
    IO_setup ();
    PWM_setup ();
    ADC_setup ();
    TB_setup ();

    params.pvehicle=&vehicle;
    params.pMAP=&MAP;
    params.pRPM=&RPM;
    params.pPULSE=&PULSE;
    params.pW_gnn_1=&W_gnn_1[0];
    params.pW_gnn_2=&W_gnn_2[0];
    params.peta=&eta[0];
    params.pHO2S_1=&HO2S_1;
    params.pHO2S_2=&HO2S_2;
    params.pk_datalog=&k_datalog;
    params.palpha=&alpha;
    params.pdW_1=&dW_1[0];
    params.pdW_2=&dW_2[0];
    params.plearning_flag_1=&learning_flag_1;
    params.plearning_flag_2=&learning_flag_2;
    params.prich_bias=&rich_bias;
    params.pP_dither=&P_dither;
    params.ppw_dither=&pw_dither;
    params.pbias_dither=&bias_dither;
    params.pamp_dither=&amp_dither;
    params.pHO2S1_cutoff=&HO2S1_cutoff;
    params.pHO2S2_cutoff=&HO2S2_cutoff;
    params.pK_p_rich=&K_p_rich;
    params.pK_p_lean=&K_p_lean;
    params.pK_i_rich=&K_i_rich;
    params.pK_i_lean=&K_i_lean;
    params.pK_p_rich_prop=&K_p_rich_prop;
    params.pK_p_lean_prop=&K_p_lean_prop;
    params.pK_i_rich_prop=&K_i_rich_prop;
    params.pK_i_lean_prop=&K_i_lean_prop;
    params.pK_i_leak=&K_i_leak;
    params.pM_f=&M_f;
    params.pI_1=&I_1;
    params.pI_2=&I_2;
    params.pautonomous_mode=&autonomous_mode;
    params.pverbose=&verbose;

    //printf("      .ad000000000ba.\n  d00000000000000b\n  d000000000000000b\n
d0000000000000000b\n|000000000000000000|\nOP'~'Y0000000000P'~'YO\n00

```

```

`Y00000P'      00\n00b      `000'      d00\nY00o      000      o00P\n`000o
000      o000'\n");
    //printf(" `000b._,d000b._,d000'\n `0000000000000000'\n
0000000o0o000000\n Y000000000000000P\n `00000I``I00000'\n
`0000I,,,I0000'\n `00000000000'\n `~00000~'\n");
    printf("\nWelcome Puny Earthlings\n");
    printf("_____/\\_____/\\o/_____\nAhhh, Sharks!!!\n\n");
    //printf"Please visit zombo.com for technical details\n";

    vehicle = 7;//current default is the GM RED NCS (most others probably won't
work

    if (vehicle==4)
    {
        printf("Vehicle not supported\n");
        //GM_BLACK_NCS_setup_func(&W[0][0][0],&Wb[0][0]);
    }
    else if (vehicle==5)
    {
        printf("Vehicle not supported\n");
        //GM_RED_NCS_setup_func(&W[0][0][0],&Wb[0][0]);
    }
    else if (vehicle==6)
    {
        //initialize gnn
        init_W_redline_lomem(&W_gnn_1[0]);
        init_W_redline_lomem(&W_gnn_2[0]);
    }
    else if (vehicle==7)
    {
        init_W_redline_lomem(&W_gnn_1[0]);
        init_W_redline_lomem(&W_gnn_2[0]);
        init_eta(&eta[0], ETA_LIN_DEFAULT, ETA_OUT_DEFAULT, ETA_HIDDEN_DEFAULT);
    }

    PIC_req_flag = 1;
    PULSE = 12.0; // October 28, 2003
    MAF = 1200.0; // July 31, 2003
    MAP = 3500.0; // initial MAP reading
    FT = 400.0; // initial Fuel Temp reading
    RPM = 100.0; // initial RPM reading
    RPM_new = 100;
    RPM_1 = 100;
    RPM_2 = 100;
    RPM_index = 0;
    reset_RPM = 1;
    MPIO_port_set(0x0400); // All outputs low except Ready_Flag
HIGH
    PWM_pin_set(0,LOW); // Turn LED ON
    // wait 100ms
    for ( i = 0 ; i <= 150000 ; i++ )
    {}
    PWM_pin_set(0,HIGH); // TURN LED OFF
    // wait 100ms

```

```

for ( i = 0 ; i <= 150000 ; i++ )
{}
PWM_pin_set(0,LOW); // TURN LED ON
printf("This is Travis' Rewrite, version %2.2f\n",version);
if (vehicle == 0)
{
    printf("\nDefault Vehicle Type: DODGE NCS. NOT WORKING\n");
}
else if (vehicle == 1)
{
    printf("\nDefault Vehicle Type: FORD NCS NOT WORKING\n");
}
else if (vehicle == 2)
{
    printf("\nDefault Vehicle Type: GM MAP NOT WORKING\n");
}
else if (vehicle == 3)
{
    printf("\nDefault Vehicle Type: GM MAF NOT WORKING\n");
}
else if (vehicle ==4)
{
    printf("\nDefault Vehicle Type: BLACK GM NN NOT WORKING\n");
}
else if (vehicle ==5)
{
    printf("\nDefault Vehicle Type: RED GM NN NOT WORKING\n");
}
else if (vehicle ==6)
{
    printf("\nDefault Vehicle Type: RED GM gnn\n");
}
else if (vehicle ==7)
{
    printf("\nDefault Vehicle Type: RED GM online gnn\n");
}
else
{
    printf("\nVehicle not supported\n");
}

for ( ; ; )
{
    t1_old=t1;
    t1=DEC_get();
    LED_old=FLASH_LED(LED_old);
    RS_COMM_SERVICE_struct (&params);

    if ((vehicle==7))
    {
        t2=DEC_get();

GET_INPUTS (&MAP, &HO2S_1, &HO2S_2, &DEC_mmft, HO2S1_cutoff, HO2S2_cutoff);
        if ((MPIO_pin_get(REQ_MPIO)==0) || (autonomous_mode))//is there new

```

```

injector
    //data or autonomous mode
    {
        t3_old=t3;
        t3=DEC_get();
        i_datalog++;
        bank=get_bank(&k_fire,MAP,RPM,verbose);
        PIC_req_flag = 1;
        SET_DATA_READY(LOW); // tell Kim I'm processing
        MPIO_pin_set(0,HIGH); // set test pin high output
        if (bank==1)
        {
            t4=DEC_get();
            PULSE=gnn_pulse_calc_lomem(MAP,RPM,&W_gnn_1[0]);
            t5=DEC_get();
            if (amp_dither)
            {
                PULSE=dither_sin(PULSE,&k_dither,P_dither,bias_dither,amp_dither);
            }
            if (learning_flag_1)
            {
                PULSE=PI_control_prop(PULSE,&I_1,HO2S_old_1,HO2S_1,K_p_rich,K_p_lean,K_i_rich,
                K_i_lean,K_p_rich_prop,K_p_lean_prop,K_i_rich_prop,K_i_lean_prop);
                PI_leak(&I_1,K_i_leak);
                HO2S_old_1=HO2S_1;
            }
        }
        else
        {
            t4=DEC_get();
            PULSE=gnn_pulse_calc_lomem(MAP,RPM,&W_gnn_2[0]);
            t5=DEC_get();
            if (amp_dither)
            {
                PULSE=dither_sin(PULSE,&k_dither,P_dither,bias_dither,amp_dither);
            }
            if (learning_flag_2)
            {
                PULSE=PI_control_prop(PULSE,&I_2,HO2S_old_2,HO2S_2,K_p_rich,K_p_lean,K_i_rich,
                K_i_lean,K_p_rich_prop,K_p_lean_prop,K_i_rich_prop,K_i_lean_prop);
                PI_leak(&I_2,K_i_leak);
                HO2S_old_2=HO2S_2;
            }
        }
        MPIO_pin_set(0,LOW); // set test pin high output
        M_f=M_f+(PULSE-INJECTOR_OFFSET)*INJECTOR_SLOPE;
        SEND_PULSE_DATA_func(PULSE);
        t6=DEC_get();
        if (bank==1)
        {

```

```

if (learning_flag_1)
{
    PULSE_old=PULSE;
    HO2S_cbuf_1[k_cbuf_1]=HO2S_1;
    t_i_cbuf_1[k_cbuf_1]=PULSE;
    P_m_cbuf_1[k_cbuf_1]=MAP;
    N_e_cbuf_1[k_cbuf_1]=RPM;

    t7=DEC_get ();
    cond_aggr (&P_m_out, &N_e_out, &t_i_out, &HO2S_out, &c,
&P_m_cbuf_1[0], &N_e_cbuf_1[0], &t_i_cbuf_1[0], &HO2S_cbuf_1[0],&k_cbuf_1);
    t8=DEC_get ();
    if (c==1)
    {
        update_W_mom (&W_gnn_1[0],P_m_out, N_e_out, t_i_out,
HO2S_out, &eta[0],&dW_1[0],alpha);
    }
    t9=DEC_get ();
}
else //check if we can learn yet
{
    if (((float) (adc_get (PORTB,
EGO_PORT_1))*5000/1023)>HO2S_START_1)
    {
        learning_flag_1=1;
        printf("learning 1\n");
    }
    else if
((int) (0xFFFFFFFFFFFFFFFF-DEC_get ())>LEARNING_DELAY)
    {
        learning_flag_1=1;
        printf("learning 1\n");
    }
}
}
else //bank2
{
    if (learning_flag_2)
    {
        PULSE_old=PULSE;
        HO2S_cbuf_2[k_cbuf_2]=HO2S_2;
        t_i_cbuf_2[k_cbuf_2]=PULSE;
        P_m_cbuf_2[k_cbuf_2]=MAP;
        N_e_cbuf_2[k_cbuf_2]=RPM;
        t7=DEC_get ();
        cond_aggr (&P_m_out, &N_e_out, &t_i_out, &HO2S_out, &c,
&P_m_cbuf_2[0], &N_e_cbuf_2[0], &t_i_cbuf_2[0], &HO2S_cbuf_2[0],&k_cbuf_2);
        t8=DEC_get ();
        if (c==1)
        {
            update_W_mom (&W_gnn_2[0],P_m_out, N_e_out, t_i_out,
HO2S_out, &eta[0],&dW_2[0],alpha);
        }
        t9=DEC_get ();
    }
}
}

```

```

    }
    else //check if we can learn yet
    {
        if (((float)(adc_get(PORTB,
EGO_PORT_2))*5000/1023)>HO2S_START_2)
        {
            learning_flag_2=1;
            printf("learning 2\n");
        }
        else if
((int)(0xFFFFFFFF-DEC_get())>LEARNING_DELAY)
        {
            learning_flag_2=1;
            printf("learning 2\n");
        }
    }
    }
    t10=DEC_get();
    POLL_PIC(HIGH); // wait for REQ_MPIO to go high (end of Kim's
request)

    SET_DATA_READY(HIGH); // tell Kim I'm ready
    t11=DEC_get();
    }
}
else if ((vehicle==4)||(vehicle==5))
{
    printf("This vehicle not supported!\n");
}
else if (vehicle==6)
{
    printf("This vehicle not supported!\n");
}
if ((i_datalog>=k_datalog)&&(k_datalog!=0))
{
    printf("%d, %e, %e, %e, %e, %e, %e, %e, %e, %e", (int) DEC_get(),
MAP, RPM, HO2S_1, PULSE, c, P_m_out, N_e_out, t_i_out, HO2S_out);
    for (i=0;i<N_PARAMS;i++)
    {
        printf(", %e",W_gnn_1[i]);
    }
    for (i=0;i<N_PARAMS;i++)
    {
        printf(", %e",W_gnn_2[i]);
    }
    printf("\n");

printf("=====\nt1=%d,t21=%d,t32=%d,t43=%d,t54=%d,t65=%d,t76=%d,t87=%d,t98=
%d,t109=%d,t1110=%d,t111=%d,fast_cycle=%d,calc_cycle=%d\n====\n",
        (int) t1, (int) (t2-t1), (int) (t3-t2), (int) (t4-t3),
(int) (t5-t4), (int) (t6-t5),
        (int) (t7-t6), (int) (t8-t7), (int) (t9-t8), (int)
(t10-t9), (int) (t11-t10), (int) (t11-t1), (int) (t1-t1_old), (int) (t2-t3_old));
    i_datalog=0;
}
}

```

```

    }
    return( 0 );
}
//***** End Main Routine *****

//***** Flash LED*****
UInt64 FLASH_LED (UInt64 LED_old)
{
    UInt64 LED_new;
    LED_new = DEC_get();
    if ((LED_old - LED_new) >= LED_FLASH_PERIOD)
    {
        PWM_port_set(PWM_port_get()^0x01);
        LED_old = LED_new;
    }
    return(LED_old);
}

//***** Get RPM GM routine *****
void GET_RPM_GM(void)
//[PIC_req_flag,crank_flag,reset_RPM,RPM2,RPM1,RPM_new,RPM,RPM_start_period]=f(
PIC_req_flag,crank_sig,crank_flag,reset_RPM,RPML,RPM_new,RPM_start_period
{
    UInt64 DEC_temp;
    crank_sig = MPIO_pin_get(15);
    DEC_temp = DEC_get();
    if (PIC_req_flag == 1)
    {
        if (crank_sig == 0) // GM
        {
            PIC_req_flag = 0;
            crank_flag = 0;
            reset_RPM = 1;
        }
    }
    else
    {
        if ((crank_sig == 1)&&(crank_flag == 0))
        {
            crank_flag = 1;
            if (reset_RPM == 1)
            {
                reset_RPM = 0;
            }
            else
            {
                RPM_end_period = DEC_temp;
                RPM_2 = RPM_1;
                RPM_1 = RPM_new;
                //GM 24 teeth
                RPM_new = 2.5/((float)(RPM_start_period - RPM_end_period)*1e-6);
                // new for GM average 3 readings
                RPM = (RPM_2 + RPM_1 + RPM_new)/3;
            }
        }
    }
}

```

```

        }
        RPM_start_period = DEC_temp;
    }
    else if ((crank_sig == 0)&&(crank_flag == 1))
    {
        crank_flag = 0;
    }

    else
    {}

}

}
//***** End Get_RPM_GM*****

void GET_INPUTS(double *MAP_fn,double *HO2S_1_fn,double *HO2S_2_fn,UInt64
*DEC_fn, float HO2S1_cutoff, float HO2S2_cutoff)
{
    UInt64 DEC_new;
    double HO2S_raw;
    DEC_new = DEC_get();
    GET_RPM_GM();
    if ((*DEC_fn - DEC_new) >= MMFT_DELAY)
    {
        *DEC_fn = DEC_new;
        /*** Get MAP ***/
        // get MAP reading from AN5-B
        *MAP_fn = (float)(adc_get(PORTB, 5))*5000/1023;
        // convert to mV
        if (*MAP_fn < 20)
        {
            *MAP_fn = 400;
        }

        // get HO2S reading from port
        HO2S_raw = (float)(adc_get(PORTB, EGO_PORT_1))*5000/1023;
        // convert to mV
        if (HO2S_raw < 20)
        {
            HO2S_raw= 400;
        }
        if (HO2S_raw<HO2S1_cutoff)
        {
            *HO2S_1_fn=LEAN;
        }
        else
        {
            *HO2S_1_fn=RICH;
        }

        // get HO2S reading from port
        HO2S_raw = (float)(adc_get(PORTB, EGO_PORT_2))*5000/1023;
        // convert to mV
        if (HO2S_raw < 20)

```

```

        {
            HO2S_raw= 400;
        }
        if (HO2S_raw<HO2S2_cutoff)
        {
            *HO2S_2_fn=LEAN;
        }
        else
        {
            *HO2S_2_fn=RICH;
        }
    }
}

//***** Send Pulse Data *****
void SEND_PULSE_DATA_func(double PULSE_fn)
{
    UInt8 pulse_low_byte;
    UInt8 pulse_high_byte;
    UInt16 rd_port;
    UInt16 pulse_int;

    // 05/22/02 convert pulse to 0.02844 ms base
    pulse_int = (UInt16) (PULSE_fn/0.02844);
    pulse_low_byte = (UInt8) (pulse_int);
    pulse_high_byte = (UInt8) (pulse_int >> 8);
    rd_port = (MPIO_port_get() & 0xFF00);
    MPIO_port_set(((UInt16) (pulse_low_byte)) + rd_port);
    MPIO_pin_set(8,HIGH);
    MPIO_pin_set(8,LOW);
    rd_port = (MPIO_port_get() & 0xFF00);
    MPIO_port_set(((UInt16) (pulse_high_byte)) + rd_port);
    MPIO_pin_set(9,HIGH);
    MPIO_pin_set(9,LOW);
}
//***** End Send Pulse Data *****

//***** Set Data Ready Status *****
void SET_DATA_READY(UInt8 status)
{
    MPIO_pin_set(10,status);
}
//***** End Set Data Ready Status *****

void POLL_PIC(UInt8 level)
{
    UInt8 level1;
    UInt8 MPIO_req;
    level1 = (UInt8) level;
    MPIO_req = (UInt8) (MPIO_pin_get(REQ_MPIO));

    while (MPIO_req!=level1)

```

```

    {
        MPIO_req = (UInt8) (MPIO_pin_get (REQ_MPIO));
    }
}

void RS_COMM_SERVICE_struct (param_struct *params)
//int *vehicle_fn, double *MAP_fn, double *MAF_fn, double *FT_fn, double
*RPM_fn, double *PULSE_fn, float *pW, float *pWb, double *pW_gnn_1, double
*pW_gnn_2, double *peta, double *pH02S_1, double *pH02S_2, int *p_k_datalog,
double *palpha, double *pdW_1, double *pdW_2, int *plearning_flag_1, int
*plearning_flag_2, double *prich_bias)
{
    char input_data[40];
    int command_val;
    char *echo_command_char[5];
    char *command_val_char[5];
    int echo_command;
    int rd_bit_1;
    int i;
    double eta_lin, eta_out, eta_hidden;
    int *vehicle_fn;
    double *MAP_fn;
    double *RPM_fn;
    double *PULSE_fn;
    double *pW_gnn_1;
    double *pW_gnn_2;
    double *peta;
    double *pH02S_1;
    double *pH02S_2;
    int *p_k_datalog;
    double *palpha;
    double *pdW_1;
    double *pdW_2;
    int *plearning_flag_1;
    int *plearning_flag_2;
    double *prich_bias;
    int *pP_dither;
    int *ppw_dither;
    double *pbias_dither;
    double *pamp_dither;
    float *pH02S1_cutoff;
    float *pH02S2_cutoff;
    double *pK_p_rich;
    double *pK_p_lean;
    double *pK_i_rich;
    double *pK_i_lean;
    double *pK_p_rich_prop;
    double *pK_p_lean_prop;
    double *pK_i_rich_prop;
    double *pK_i_lean_prop;
    double *pK_i_leak;
    int *pautonomous_mode;
    int *pverbose;
}

```

```

rd_bit_1 = SC1SR;
rd_bit_1 = rd_bit_1 >> 6;
rd_bit_1 = rd_bit_1 & 1;
if (rd_bit_1 == 1)
{
    vehicle_fn=params->pvehicle;
    MAP_fn=params->pMAP;
    RPM_fn=params->pRPM;
    PULSE_fn=params->pPULSE;
    pW_gnn_1=params->pW_gnn_1;
    pW_gnn_2=params->pW_gnn_2;
    peta=params->peta;
    pH02S_1=params->pH02S_1;
    pH02S_2=params->pH02S_2;
    p_k_datalog=params->pk_datalog;
    palpha=params->palpha;
    pdW_1=params->pdW_1;
    pdW_2=params->pdW_2;
    plearning_flag_1=params->plearning_flag_1;
    plearning_flag_2=params->plearning_flag_2;
    prich_bias=params->prich_bias;
    pP_dither=params->pP_dither;
    ppw_dither=params->ppw_dither;
    pbias_dither=params->pbias_dither;
    pamp_dither=params->pamp_dither;
    pH02S1_cutoff=params->pH02S1_cutoff;
    pH02S2_cutoff=params->pH02S2_cutoff;
    pK_p_rich=params->pK_p_rich;
    pK_p_lean=params->pK_p_lean;
    pK_i_rich=params->pK_i_rich;
    pK_i_lean=params->pK_i_lean;
    pK_p_rich_prop=params->pK_p_rich_prop;
    pK_p_lean_prop=params->pK_p_lean_prop;
    pK_i_rich_prop=params->pK_i_rich_prop;
    pK_i_lean_prop=params->pK_i_lean_prop;
    pK_i_leak=params->pK_i_leak;
    pautonomous_mode=params->pautonomous_mode;
    pverbose=params->pverbose;

    gets(input_data); /* Read port clear data */
    printf("\nEnter Command, (1 for list)\n");
    gets(input_data); /* Read data from port */
    /* 1st character should be command */
    *echo_command_char=strtok(input_data, " ");
    /* data input from keyboard */
    *command_val_char=strtok(NULL, " ");
    echo_command = strtol(*echo_command_char, NULL, 10);
    command_val = strtol(*command_val_char, NULL, 10);
    while (echo_command > 0)
    {
        if (echo_command ==1)
        {
            printf("4:print weights\n5:enter alpha\n6:enter eta\n7:change
vehicle\n8:log data\n9:print OP\n10:learning flag\n11:rich

```



```

    }
    //GM COMBO
    else if (*vehicle_fn == 5)
    {
        printf("\nMODE = RED GM NN BASED PULSEWIDTH\nNOT
SUPPORTED!!!\n");
    }
    //GM COMBO
    else if (*vehicle_fn == 6)
    {
        printf("\nMODE = RED GM gnn BASED PULSEWIDTH\n");
        //initialize
        init_W_redline_lomem(pW_gnn_1);
    }
    //GM COMBO
    else if (*vehicle_fn == 7)
    {
        printf("\nMODE = RED GM online gnn BASED PULSEWIDTH\n");
        //initialize
        init_W_redline_lomem(pW_gnn_1);
        init_W_redline_lomem(pW_gnn_2);
        init_eta(peta, ETA_LIN_DEFAULT, ETA_OUT_DEFAULT,
ETA_HIDDEN_DEFAULT);
        for (i=0;i<N_PARAMS;i++)
        {
            *(pdW_1+i)=0;
            *(pdW_2+i)=0;
        }
    }
    else
    {
        printf("\nERROR!!! Re-enter MODE using command 7\n");
    }

}
else if (echo_command == 8)
{
    printf("Enter interval to log data\n");
    scanf("%d",p_k_datalog);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 9)
{
    if ((*vehicle_fn == 4)||(*vehicle_fn==5))
    {
        printf("Vehicle= %d\nNOT SUPPORTED!!!\n",*vehicle_fn);
    }
    else if ((*vehicle_fn == 6)||(*vehicle_fn==7))
    {
        printf("Vehicle= %d\n",*vehicle_fn);
        printf("HO2S_1 (binary)= %f\n",*pHO2S_1);
        printf("HO2S_2 (binary)= %f\n",*pHO2S_2);
        printf("MAP = %10.5f\n",*MAP_fn);
    }
}

```

```

        printf("RPM = %10.5f\n", *RPM_fn);
        printf("PULSE = %10.5f\n", *PULSE_fn);
    }
}
else if (echo_command == 10)
{
    printf("Enter learning flag 1(%d)\n", *plearning_flag_1);
    scanf("%d",plearning_flag_1);
    printf("Enter learning flag 2(%d)\n", *plearning_flag_2);
    scanf("%d",plearning_flag_2);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 11)
{
    printf("Enter rich bias factor(%f)\n", *prich_bias);
    scanf("%lf",prich_bias);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 12)
{
    printf("Enter dither period(%d)\n", *pP_dither);
    scanf("%d",pP_dither);
    //printf("Enter dither pulsewidth(%d)\n", *ppw_dither);
    //scanf("%d",ppw_dither);
    printf("Enter bias (%f)\n", *pbias_dither);
    scanf("%lf",pbias_dither);
    printf("Enter dither unit amplitude (%f)\n", *pamp_dither);
    scanf("%lf",pamp_dither);
    gets(input_data); /* Read data from port */
}
else if (echo_command ==13)
{
    printf("Enter HO2S1 cutoff (%f)\n", *pHO2S1_cutoff);
    scanf("%f",pHO2S1_cutoff);
    printf("Enter HO2S2 cutoff (%f)\n", *pHO2S2_cutoff);
    scanf("%f",pHO2S2_cutoff);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 14)
{
    printf("Enter rich K p(%f)\n", *pK_p_rich);
    scanf("%lf",pK_p_rich);
    printf("Enter lean K p(%f)\n", *pK_p_lean);
    scanf("%lf",pK_p_lean);
    printf("Enter rich K i(%f)\n", *pK_i_rich);
    scanf("%lf",pK_i_rich);
    printf("Enter lean K i(%f)\n", *pK_i_lean);
    scanf("%lf",pK_i_lean);
    gets(input_data); /* Read data from port */
    *params->pI_1=0;
    *params->pI_2=0;
}
else if (echo_command == 15)
{

```

```

        printf("Fuel used=%f g\n", *params->pM_f);
    }
else if (echo_command == 16)
{
    *params->pM_f=0;
    printf("Resetting Fuel used=%f g\n", *params->pM_f);
}
else if (echo_command == 17)
{
    printf("Enter rich K p prop(%f)\n", *pK_p_rich_prop);
    scanf("%lf", pK_p_rich_prop);
    printf("Enter lean K p prop(%f)\n", *pK_p_lean_prop);
    scanf("%lf", pK_p_lean_prop);
    printf("Enter rich K i prop(%f)\n", *pK_i_rich_prop);
    scanf("%lf", pK_i_rich_prop);
    printf("Enter lean K i prop(%f)\n", *pK_i_lean_prop);
    scanf("%lf", pK_i_lean_prop);
    printf("Enter leak coefficient(%f)\n", *pK_i_leak);
    scanf("%lf", pK_i_leak);
    gets(input_data); /* Read data from port */
    *params->pI_1=0;
    *params->pI_2=0;
}
else if (echo_command == 18)
{
    printf("Enter autonomous mode(%d)\n", *pautonomous_mode);
    scanf("%d", pautonomous_mode);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 19)
{
    printf("Enter verbosity level(%d)\n", *pverbose);
    scanf("%d", pverbose);
    gets(input_data); /* Read data from port */
}
else if (echo_command == 999)
{
    printf("\nTravis' Rewrite Version: %3.2f\n", version);
}
else
{
    printf("\nCommand not supported\n");
}
printf("Enter Command\n");
gets(input_data); /* Read data from port */
/* 1st character should be command */
*echo_command_char=strtok(input_data, " ");
/* data input from keyboard */
*command_val_char=strtok(NULL, " ");
echo_command = strtol(*echo_command_char, NULL, 10);
command_val = strtol(*command_val_char, NULL, 10);
}
printf("\nExit Command Mode ...DONE\n");
}

```

```

}

// ***** Logsig Computation*****
float Logsig(float logsign) // LogSig Computation ((1/(1+exp(-alpha*x)))-1)
{
    float logsigout; //Logsig Value Out
    logsigout=(2/(1 + exp(-2*logsign)))-1;
    return (logsigout);
}
// ***** END LogSig Computation*****

void PWM_setup ( void )
{
    UInt8 x = 0;
    MIOS_init( 16 ); // Initialize the MIOS clock prescaler
to 2
    // DO NOT SET BELOW 2
    for ( x = 0 ; x < 4 ; x++ )
    {
        PWM_init( x , OUTPUT );
    }
}

void IO_setup ( void )
{
    IRQ_init( GPIO , GPIO ); // Set IRQ pins for GPIO operation
    // Set all pins as Output
    IRQ_port_direction_set( 0xFF );
    // Set MPIO bits input/output
    MPIO_port_direction_set ( 0x7FF );
    // 0x3FFF Set MPIO bits 0-13 as outputs, bits 14-15 as inputs
    // 0x7FF Set MPIO bits 0-10 as outputs, bits 11-15 as inputs
}

void ADC_setup( void ) // Initializes ADC pins for ADC
operation
{
    // depending on compiler, one may need to use SCAN_1 instead of SCAN_16CH
    adc_init( SCAN_16CH , PORTA , 0 ); // 1 scan of 16 PORTA pins
    //adc_init( SCAN_1 , PORTA , 0 );
    adc_init( SCAN_16CH , PORTB , 0 ); // 1 scan of 16 PORTB pins
    //adc_init( SCAN_1 , PORTB , 0 );
}

void TB_setup( void ) // Initializes TB
{
    TMBCLK_set(OSCM); // setup clock for 4MHz
    TB_set(0xfffffff); // set to fast mode
    TMBCLK_en( ENABLE ); // Start the Time Base
}

void init_W_redline_lomem(double *W)
{
    unsigned int i, j, seed;

```

```

double W_1,W_4;
double W_temp[N_NEURONS];
double W_mag, W_mag_sq, W_sum, W_lin_P_m;
unsigned int N_inputs_i;
int idx=0;

seed=adc_seed(EGO_PORT_2); //seed from oxygen sensor noise
srand(seed);
for (i=0;i<1000;i++) //generate some random number to get
started
{
    rand();
}
for (i=N_INPUTS;i<(N_NEURONS-1);i++)
{
    N_inputs_i=i-2; //inputs except for t_i and 1
    W_mag_sq=0;
    W_sum=0;
    for (j=0;j<N_inputs_i;j++)
    {
        W_temp[j]=(2*(double)rand()/RAND_MAX)-1;
        W_mag_sq=W_mag_sq+W_temp[j]*W_temp[j];
    }
    W_mag=overlapfactor*pow((N_NEURONS-1)/2,(1/N_inputs_i));
    for (j=0;j<N_inputs_i;j++)
    {
        W_temp[j]=W_mag/sqrt(W_mag_sq)*W_temp[j];
        W_sum=W_sum+W_temp[j];
    }

    for (j=0;j<N_INPUTS-2;j++)
    {
        *(W+idx)=2*W_temp[j];
        idx++;
    }
    *(W+idx)=((2*(double)rand()/RAND_MAX)-1)*W_mag-W_sum;
    idx++;
    for (j=N_INPUTS;j<N_inputs_i+2;j++)
    {
        *(W+idx)=2*W_temp[j+N_INPUTS-6];
        idx++;
    }
}

//this is for t_i in ms, not s
W_lin_P_m=60.0*2/N_e_redline/P_max/W_LIN_FUDGE_FACTOR*1000;
//%weight for P_m
W_1=-W_lin_P_m*W_3*P_m_scale/t_i_scale;
//W_2=0;//N_e
//bias.
W_4=+t_i_offset*W_3/t_i_scale+W_1*P_m_offset/(P_m_scale);
*(W+idx)=W_1;
idx++;
*(W+idx)=0;

```

```

idx++;
*(W+idx)=W_3;
idx++;
*(W+idx)=W_4;
idx++;
for (i=N_INPUTS;i<N_NEURONS-1;i++)
{
    *(W+idx)=epsilon*((2*(double)rand()/RAND_MAX)-1);
    idx++;
}

return;
}

double gnn_pulse_calc_lomem(double P_m,double N_e,double *W)
{
    double u[N_INPUTS]= //maybe eliminate division
    {
        (P_m-P_m_offset)/P_m_scale, (N_e-N_e_offset)/N_e_scale, 0, 1
    };
    double x[N_NEURONS];
    double phi,t_i;
    unsigned int i,j;
    double net;
    unsigned int idx=0;
    unsigned int idx_lin=0;
    for (i=0;i<N_INPUTS;i++)
    {
        x[i]=u[i];
    }
    for (i=N_INPUTS;i<N_NEURONS;i++)
    {
        net=0;
        if (i<(N_NEURONS-1)) //hidden layers
        {
            for (j=0;j<i;j++)
            {
                if (j!=2) //skip t_i
                {
                    net=net+ *(W+idx)* x[j];
                    idx++;
                }
            }
            x[i]=1/(1+exp(-net));
        }
        else //output layer skips hidden neurons
        {
            idx_lin=idx;//save index of first linear weight
            idx=idx+4;
            for (j=N_INPUTS;j<i;j++)
            {
                if (j!=2) //skip t_i
                {

```

```

        net=net+*(W+idx)* x[j];
        idx++;
    }
}

}

}
phi=net;
t_i=-t_i_scale/ *(W+idx_lin+2)* (*(W+idx_lin+0) *(P_m-P_m_offset)/P_m_scale+
*(W+idx_lin+1)*(N_e-N_e_offset)/N_e_scale+ *(W+idx_lin+3)+phi)+t_i_offset;
return t_i;
}

```

```

void cond_aggr(double *P_m_out, double *N_e_out, double *t_i_out, double
*H02S_out, double *c, double *P_m_cbuf, double *N_e_cbuf, double *t_i_cbuf,
double *H02S_cbuf,int *k_cbuf)
{
    int k_d;
    unsigned int i;
    //as long as we use a circular buffer of size Cont_d_2+1, the data at k_d+1
    //is the first possible delay point
    *H02S_out=*(H02S_cbuf+*k_cbuf);
    k_d=*k_cbuf+1;
    while (k_d>=K_cbuffer)
    {
        k_d=k_d-K_cbuffer;
    }
    *P_m_out=*(P_m_cbuf+k_d);
    *N_e_out=*(N_e_cbuf+k_d);
    *t_i_out=*(t_i_cbuf+k_d);
    *c=1; //Start with confidence=1
    for (i=0;i<(Cont_d_2-Cont_d_1+1);i++)
    {
        k_d=*k_cbuf+i;
        while (k_d>=K_cbuffer)
        {
            k_d=k_d-K_cbuffer;
        }
        if (*(H02S_cbuf+k_d)!=*H02S_out)
        {
            *c=0;
            //exit loop
            i=(Cont_d_2-Cont_d_1+1);
        }
    }
    *k_cbuf=*k_cbuf+1;
    while (*k_cbuf>=K_cbuffer)
    {
        *k_cbuf=*k_cbuf-K_cbuffer;
    }
}

```

```

void update_W_mom(double *pW,double P_m, double N_e, double t_i, double H02S,
double *peta,double *pdW, double alpha)

```

```

{

double F_w[N_PARAMS];
double F_Y_hat;
double x[N_NEURONS];
double HO2S_hat;
unsigned int i;

gnncc_sim_x_lomem(&HO2S_hat, &x[0],P_m,N_e, t_i, pW);
F_Y_hat=HO2S_hat-HO2S;
gnn_static_gradient_lomem(&F_w[0],F_Y_hat,pW,&x[0],N_NEURONS);
for (i=0;i<N_PARAMS;i++)
{
*(pdW+i)=alpha* *(pdW+i) -(peta+i)*F_w[i];
*(pW+i)=*(pW+i)+ *(pdW+i);
}
}

void gnncc_sim_x_lomem(double *py_hat,double *px,double P_m,double N_e,double
t_i,double *pW)
{
double u[N_INPUTS]= //maybe eliminate division
{
(P_m-P_m_offset)/P_m_scale, (N_e-N_e_offset)/N_e_scale,
(t_i-t_i_offset)/t_i_scale, 1
};
sim_gnn_lomem(py_hat,px,&u[0],pW);
return;
}

void sim_gnn_lomem(double *py_hat, double *px,double *pu,double *pW)
{
unsigned int i,j;
double net;
int idx=0;
for (i=0;i<N_INPUTS;i++)
{
*(px+i)= *(pu+i);
}
for (i=N_INPUTS;i<(N_NEURONS-1);i++)
{
net=0;
for (j=0;j<i;j++)
{
if (j!=(N_INPUTS-2)) //ignore t_i
{
net=net+ *(pW+idx)* *(px+j);
idx++;
}
}
*(px+i)=1/(1+exp(-net));
}
//output neuron
{

```

```

        net=0;
        for (j=0;j<i;j++)
        {
            {
                net=net+ *(pW+idx)* *(px+j);
                idx++;
            }
        }
        *(px+i)=1/(1+exp(-net));
    }
    *(py_hat)= *(px+N_NEURONS-1);
    return;
}

void gnn_static_gradient_lomem(double *F_w, double F_Y_hat, double *W, double
*x, const unsigned int n)
{
    int i;
    int j;
    double *F_x;
    double *F_net;

    //allocate array
    F_x=(double *) malloc(n * sizeof(double));
    if (F_x==NULL)
    {
        printf("Out of memory on F_x");
        exit (1);
    }
    F_net=(double *) malloc(n * sizeof(double));
    if (F_net==NULL)
    {
        printf("Out of memory on F_net");
        exit (1);
    }

    for (i=0;i<(int) n;i++)
    {
        F_x[i]=0;
        F_net[i]=0;
    }
    F_x[n-1]=F_Y_hat;

    for (i=n-1;i>=0;i--)
    {
        for (j=(i+1);j<((int) n);j++)
        {
            if ((i!=2 || j==(N_NEURONS-1)) && (j>(N_INPUTS-1)))
            {
                F_x[i]=F_x[i]+*(W+idx_lomem(j,i))*F_net[j];
            }
        }
        if(i>(N_INPUTS-1)) //skip input neurons
        {

```

```

        //for sigmoid only
        F_net[i]=F_x[i]* *(x+i) * (1.0-*(x+i));
        for (j=0;(int) j<=(i-1);j++)
        {
            if ((j!=2) || (i==(N_NEURONS-1)))
            {
                *(F_w+idx_lomem(i, j))=F_net[i]* *(x+j);
            }
        }
    }
}
free((void *) F_x); //free memory
free((void *) F_net);
return;
};

int idx_lomem(int r, int c)
{
    int idx=NULL;
    if (r==(N_NEURONS-1))
    {
        idx= c+0.5*(N_NEURONS-5)*(N_NEURONS);
    }
    else if (c<2)
    {
        idx=c+0.5*(r-4)*(r+1);
    }
    else if (c>2)
    {
        idx=c-1+0.5*(r-4)*(r+1);
    }
    return(idx);
}

int adc_seed(int port)
{
    //generates a seed from the adc on port b
    int seed=0;
    int i,j;
    const int n_bits=16;
    const int n_wait=400;
    int lsb;
    ADC_setup();
    for (i=0;i<n_bits;i++)
    {
        lsb=(int)((unsigned short)(adc_get(PORTB, port)<<15))>>15;
        seed=seed+(lsb<<i);
        for (j=0;j<n_wait;j++)
        {}
    }
    return seed;
}

```

```

void init_eta(double * p_eta, double eta_lin, double eta_out, double eta_hidden)
{
    int i;
    int n_neurons;
    n_neurons=(int) (1.5+sqrt(4.25+2.0*N_PARAMS));
    for (i=0;i<N_PARAMS;i++)
    {
        if (i<(0.5*(n_neurons-5)*n_neurons))
        {
            *(p_eta+i)=eta_hidden;
        }
        else if (i>(0.5*(n_neurons-5)*n_neurons+3))
        {
            *(p_eta+i)=eta_out;
        }
        else
        {
            *(p_eta+i)=eta_lin;
        }
    }
}

int get_bank(int * k_fire,double P_m, double N_e, int verbose)
{
    //somehow figures out which bank to use for each injection
    //this may appear to be magic, but it's really just sufficiently advanced
    int bank;
    int MPIO_current;

    MPIO_current=MPIO_pin_get(INJ_MPIO);

    if (N_e<BANK_SYNCH_MAX_RPM&&P_m<BANK_SYNCH_MAX_MAP)
    {
        if (MPIO_current&&(!gMPIO_old))//if inj just turned on
        {
            if ((*k_fire)!= (N_CYL-1))
            {
                if(verbose>0)
                {
                    printf("%.%d\n",*k_fire);//warning takes too long to print
                }
            }
            *k_fire=0;
        }
        else
        {
            *k_fire=(*k_fire)++;
        }
        if (*k_fire>(N_CYL-1))
        {
            if (verbose>0)
            {

```

```

        printf("%d\n", *k_fire);
    }
    *k_fire=0;
}
}
else
{
    *k_fire=(k_fire)++;
    if (*k_fire>(N_CYL-1))
    {
        *k_fire=0;
    }
}
bank=gbank_matrix[*k_fire];
gMPIO_old=MPIO_current;//save MPIO in global variable

return bank;
}

void bias_eta(double *peta_biased,double *peta_base,double factor, int N_params)
{
    int i;
    for (i=0;i<N_params;i++)
    {
        *(peta_biased+i)=*(peta_base)*factor;
    }
}

double PI_control(double PULSE_base,double *I,double HO2S_old, double
HO2S,double K_p_rich,double K_p_lean,double K_i_rich,double K_i_lean)
{
    double PULSE;
    if (HO2S==RICH)
    {
        *I=*I+K_i_rich;//integrator
        PULSE=PULSE_base+*I;
    }
    else
    {
        *I=*I+K_i_lean;//integrator
        PULSE=PULSE_base+*I;
    }
    if ((HO2S==RICH) & (HO2S_old==LEAN))
    {
        *I=*I+K_p_rich;
    }
    else if ((HO2S==LEAN) & (HO2S_old==RICH))
    {
        *I=*I+K_p_lean;
    }
    return PULSE;
}

double dither(double PULSE,int *k_dither,int P_dither, int pw_dither,double

```

```

amp_dither)
{
    //multiplies PULSE by PWM signal
    if (*k_dither<pw_dither)
    {
        PULSE=PULSE * (1+amp_dither);
    }
    else
    {
        PULSE=PULSE * (1-amp_dither);
    }
    (*k_dither)++;
    if (*k_dither>=P_dither)
    {
        *k_dither=0;
    }
    return PULSE;
}

double dither_sin(double PULSE,int *k_dither,int P_dither, double
bias_dither,double amp_dither)
{
    //adds sinusoidal dither to PULSE
    const double pi=3.141592653589793238;
    PULSE=PULSE*(1+bias_dither+amp_dither*fast_sin(((double)
*k_dither)/((double) P_dither)*2*pi));
    (*k_dither)++;
    if (*k_dither>=P_dither)
    {
        *k_dither=0;
    }
    return PULSE;
}

double fast_sin(double x)
{
    const double pi=3.141592653589793238;
    const double B=4/pi;
    const double C=-4/(pi*pi);
    double y;
    x=fmod(x,(2.0*pi));//wrap at 2*pi
    y= B*x+C*x*abs(x);
    return y;
}

double PI_control_prop(double PULSE_base,double *I,double HO2S_old, double HO2S
, double K_p_rich, double K_p_lean,double K_i_rich,double K_i_lean,double
K_p_rich_prop,double K_p_lean_prop,double K_i_rich_prop,double K_i_lean_prop)
{
    double PULSE;
    if (HO2S==RICH)
    {
        *I=*I+K_i_rich+K_i_rich_prop*PULSE_base;//integrator
    }

```

```

        //PULSE=PULSE_base+*I;
    }
    else
    {
        *I=*I+K_i_lean+K_i_lean_prop*PULSE_base;//integrator
    }
    if ((HO2S==RICH) & (HO2S_old==LEAN))
    {
        *I=*I+K_p_rich+K_p_rich_prop*PULSE_base;
    }
    else if ((HO2S==LEAN) & (HO2S_old==RICH))
    {
        *I=*I+K_p_lean+K_p_lean_prop*PULSE_base;
    }
    PULSE=PULSE_base+*I;
    return PULSE;
}

void PI_leak(double *I,double K_i_leak)
{
    *I=*I*(1-K_i_leak);
}

```

APPENDIX D

COPYRIGHT INFORMATION

Due to the manuscript nature of this dissertation, the author has assigned copyright to the publisher of a number of papers. This appendix documents the copyright holders' permission to reproduce the papers in this dissertation.

D.1 Limit Cycle Behaviour of a Neural Controller with Delayed Bang-Bang Feedback, Chapter 7

Date: Wed, 28 Mar 2007 14:06:09 +0300
From: editor <info@enformatika.org>
To: Travis Wiens <tkw954@mail.usask.ca>
Subject: Re: Paper Code:20190 (registration and final submission)

You can include this paper to your PhD thesis.

D.2 Preliminary Experimental Verification of an Intelligent Fuel Air Ratio Controller, Chapter 8

Date: Thu, 11 Jan 2007 15:35:49 -0500
From: Martha Swiss <MSwiss@sae.org>
To: Travis Wiens <tkw954@mail.usask.ca>
Subject: RE: FW: Copyright Assignment for Paper 2007-01-1339

Dear Travis,
Permission to use SAE Paper 2007-01-1339 in your Ph.D. thesis, tentatively titled, Online Training of a Neural Fuel-Air Ratio Controller, and with an estimated publish date of early 2008, is hereby granted, and we request that the following credit line be used:

"Reprinted with permission from SAE Paper 2007-01-1339 Copyright (c) 2007 SAE International."

Permission is for this one-time use only. New requests are required for subsequent editions, for reprints or excerpts, or for other uses of the material.

Thank you for contacting SAE and for your cooperation.

Sincerely,

Martha

Martha Swiss
Intellectual Property Manager
SAE International
400 Commonwealth Drive
Warrendale, PA 15096-0001

Voice: +01 724-772-4049
Fax: +01 724-776-3036
Email: swiss@sae.org

D.3 Computer Code in Appendix C

Date: Thursday, November 29, 2007 4:26 PM
From: Travis Wiens [mailto:tkw954@mail.usask.ca]
To: Sulatisky, Mike
Subject: Copyright Permission

Mike,
I'm putting together my thesis and discovered that, while I rewrote most of the code that I used on the red truck microcontroller, there are still a couple of functions that I used from SRC's old code, such as the RPM calculation function. Can I have your permission to reproduce these bits in my thesis?

Date: Fri, 30 Nov 2007 16:52:01 -0600
From: "Sulatisky, Mike" <sulatisky@src.sk.ca>
To: Travis Wiens <tkw954@mail.usask.ca>
Subject: RE: Copyright Permission

Yes, you have my permission. Thanks for asking.