REINFORCEMENT LEARNING CONTROL APPLIED TO DIESEL ENGINES

By

ORLANDO DE JESUS

Engineer in Electronics
Universidad Simón Bolivar
Caracas, Venezuela
1985

Project Management Specialist
Universidad Simón Bolívar
Caracas, Venezuela
1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1998

REINFORCEMENT LEARNING CONTROL APPLIED

TO DIESEL ENGINES

Thesis Approved:

Martin T. Hagan

Thesis Adviser

R. Ramakumar

[illegible]

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank my major adviser, Dr. Martin Hagan, for his guidance, encouragement, patience and friendship.

My thanks also goes to Salim A. Jaliwala of Cummins Engine Company for his support in this endeavor.

I would like to express my deep appreciation to my wife, Marisela, for her years of loving support, encouragement and patience. I wish to thank my parents Manuel and Nelli for all they have done to support all my years of study and work, especially the encouragement for this one. Special thanks to my family Jose, Gilberto, Pedro, Ernesto, Nelli, Juan, Genesis, Elier, Daniela, Debbi, Maria de los Angeles, Belkys, Debora, Elia, Maria, Jorge.

Special regards to my friends from AETI C.A. I hope their efforts to maintain the company will success. Thanks to my friends for your support Aitzol, Xabier, Hector, Manuel, Magaly, Francis, Carmen Elena, Felipe, Fidelia, Jose Abraham, Jose Luis, Yamirca, Noe, Rossany, Victor, Tamar, Gustavo, Sandra, Antonio, Carmen, Alejandro, Adriana, Elsa, Rene, Paloma, Francisca, Hanz, Eliecer, Janice, Marta, Daniel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

a	Pole location for the derivative action of the PID controller.
a_t	Action at time t.
BDC	Bottom dead centre.
b_1	Viscous friction for basic engine model.
CA	Crank angle.
CI	Compression-ignition engine.
CMAC	Cerebellar model articulation controller.
C_{pa}	Specific heat for air.
C_{pe}	Exhaust specific heat.
C_{pwall}	Specific heat of the exhaust manifold wall.
CR	Crank radius.
γ	Specific heat ratio.
γ	Reward discount factor.
d	Diameter of cylinder.
D	Exhaust manifold diameter.
DI	Direct injection.

DP	Dynamic programming methods.
dQ_L	Heat transfer term which defines the heat transfer from cylinder gas to wall and vice versa.
$E\{\ \}$	Expected value.
$\vec{e}_t$	Eligibility traces vector.
EVC	Exhaust valve closes.
EVO	Exhaust valve opens.
ε	Effectiveness of the intercooler.
f	Engine fueling rate (mm^3 $stroke$).
F	Equivalence ratio.
fix	Rounds toward zero.
$fmep$	Friction mean effective pressure.
f_s	Stoichiometric[1] fuel air ratio
$(F/A)_{actual}$	Actual fuel air ratio.
GPI	Generalized policy iteration.
h	Specific enthalpy.
h_{oj}	Specific stagnation enthalpy
ht	Heat transfer coefficient ($kw/{}^{\circ}K \cdot m^2$)
ID	Ignition delay.

1. Stoichiometric. "pertaining to or involving substances that are in the exact proportions required for a given reaction" (19)

IDI	Indirect injection.
IVC	Intake valve closes.
IVO	Intake valve opens.
I	Engine Inertia ($lb\text{-}ft\text{-}sec^2$).
$J(s_t)$	*Cost-to-go* function starting at state s_t.
$J(\theta)$	Varying inertia of crankshaft.
K_d	Derivative gain if PID controller.
K_i	Integral gain of PID controller.
K_p	Proportional gain of PID controller.
Leg	Lower error gain.
Lel	Lower error load.
$\dot{m}$	mass flow rate.
m_a	Air mass.
$\dot{m}_c$	Compressor mass flow.
MC	Monte Carlo methods.
$\dot{m}_{corr}$	Corrected mass flow rate.
m_{em}	Mass in the exhaust manifold.
$\dot{m}_{ex}$	Exhaust mass flow rate.
$\dot{m}_f$	Engine fueling rate.
$\dot{m}_{fb}$	Burned fuel mass changing rate.

$\dot{m}_{fburn}$	Fuel mass burning rate.
m_{im}	Mass in the intake manifold.
m_{wall}	Mass of the exhaust manifold wall.
$\dot{m}_3$	Air mass flow after the intercooler.
$\dot{m}_4$	Intake manifold mass flow.
$\dot{m}_5$	Engine outlet mass flow.
N	Actual engine speed (*rpm*).
N_{corr}	Corrected turbocharger speed.
$Nref$	Reference Engine Speed.
N_{tc}	Turbocharger rotor speed (*rpm*).
η_t	Turbine efficiency.
η_v	Volumetric efficiency.
η_{ind}	Indicated efficiency (percent).
μ	Gas viscosity.
P	Pressure.
P_{cyl}	Cylinder pressure.
P_d	Down stream pressure.
P_{std}	Standard pressure.

$p_{s_t s_{t+1}}(a_t)$	Probability that the system changes from the state s_t to the state s_{t+1} if the action a_t is executed.
P_u	Up stream pressure.
P_1	Ambient pressure.
P_4, P_{im}	Intake manifold pressure.
P_6	Exhaust manifold pressure.
π	3.14159265358979.
$\pi(s, a)$	Agent's policy as function of the possible states s and actions a.
ρ_3	Air density after the intercooler.
r_t	Reward or cost at time t.
$Q(s_t, a_t)$	Q-factor function or cost function starting at the state s_t and executing the action a_t.
Q_{com}	Heat released by combustion.
Q_{cool}	Heat rejected to charge air cooler.
Q_e	Heat rejected from exhaust manifold.
$\dot{Q}_{ht}$	Heat transfer rate to cylinder walls.
Q_{LHV}	Fuel lower heating value (kJ/kg).
R	Gas constant ($kJ/kg \cdot {}^\circ K$).
s	Laplace transform variable.
s_t	System state at time t.

SARSA	On policy TD method based on s_t, a_t, r_t, s_{t+1} and a_{t+1}.
SI	Spark-ignition engine.
S_p	Mean piston speed.
T	Sampling time.
T_c	Compressor Torque.
T_{cyl}	Temperature at the cylinder.
TD	Temporal difference methods.
TDC	Top dead centre.
T_e	Engine Torque.
T_f	Friction Torque.
T_i	Indicated Torque.
T_{im}	Intake manifold temperature.
T_{load}	External load torque.
T_{std}	Standard temperature.
T_t	Turbine torque.
T_u	Up stream temperature.
T_w	Coolant temperature.
T_{wall}	Wall temperature.
T_1	Ambient temperature.

T_2	Compressor temperature.
T_4	Intake manifold temperature.
T_5	Engine outlet temperature.
T_6	Exhaust manifold temperature.
τ	Fueling delay in milliseconds.
u	Specific internal energy.
Ueg	Upper error gain.
Uel	Upper error load.
V	Volume.
V_d	Displacement volume.
V_{em}	Exhaust manifold volume.
V_{im}	Intake manifold volume.
W_c	Compressor work.
W_{pis}	Piston work.
W_t	Turbine work.
ω_{tc}	Turbocharger speed.
z	Z-transform variable.

CHAPTER 1

INTRODUCTION

The principal objective of this research has been to investigate the potential of using reinforcement learning techniques for the speed control of diesel engines. The first learning technique is called genetic reinforcement learning. We will apply this technique to the optimization of PID controller parameters. We will begin the training with base-line controller parameters for a general engine configuration. However, each specific engine will have different characteristics. Therefore the controller may not be suitable for all engines. With the genetic reinforcement learning we will optimize the controller parameters based on specific engine behavior. We will work with analog and digital controllers and different engine configurations.

The second learning algorithm we will investigate is called reinforcement learning. Reinforcement learning is an approximate form of dynamic programming, in which a neural network controller is trained to optimize a specific performance function. At each iteration the algorithm receives a certain reward or penalty, and attempts to control the system so as to maximize future rewards or minimize penalties.

Let us now outline the flow of this thesis. Chapter 2 has a description of the basic diesel engine operation. Classifications are based on the injection type and how the gas exchange process is performed. Chapter 3 describes the implementation of a diesel engine model to be used in testing the various reinforcement learning algorithms. We will consider a pseudo-linear system and a system based on neural networks.

Chapter 4 has a discussion of a linear adaptive control technique for the diesel engine. We will use the self-tuning regulator technique. The results of this chapter will be used as base-line to compare the results for the reinforcement learning algorithms.

Chapter 5 has a description of the GENITOR algorithm. This genetic reinforcement learning algorithm will be used to optimize the parameters of PID controllers for different engine configurations.

Chapter 6 is a discussion of the general reinforcement learning framework. Reinforcement learning is an approximate form of dynamic programming. The objective is to determine a control action which optimizes future performance. Reinforcement learning is an excellent strategy for the intelligent control of systems which are difficult to model but easy to simulate.

Reinforcement learning involves a two-stage process. First, a model must be developed to predict future performance. Next, an appropriate action must be determined to optimize the performance. In Chapter 6 the basic framework for reinforcement learning is presented, and several variations of reinforcement learning are described. Simulations are used to illustrate the operation of the various algorithms.

Chapter 7 has different simulations based on the reinforcement learning algorithm. The first cases demonstrate how the algorithm will learn to change the engine speed from an initial condition to a desired speed. After that we will present cases which illustrate engine speed tracking. We will consider reward schemes based on penalty per step on the episode and instant absolute error. Other experiments will be related to the use of multiple neural networks.

Chapter 8 will contain a summary of the main results and contributions of this thesis. This will be followed by recommendations for future work.

Appendix A describes different diesel engine mathematical models from different researchers for use in simulation.

CHAPTER 2

DIESEL ENGINE OPERATION.

2.1. Historical review.

The history of diesel engines started in the last years of the 19th century with the work of Dr. Rudolf Diesel. Until WWI "the diesel engine was used primarily in stationary and ship propulsion applications in the form of relatively low speed four-stroke normally aspirated engines"$_{(8)}$. WWI spurred the use of diesel engines in transportation and WWII increased the development of highly supercharged diesel engines. From that time, we have seen a continuous process of improvement in the design of diesel engines and the application of electronic modules and computer algorithms in their design.

2.2. Classification of the diesel engines

The first classification principle is the compression-ignition principle. In contrast to spark-ignition (SI) engines, "the compression-ignition (CI) engine operates with a heterogeneous charge of previously compressed air and a finely divided spray of liquid fuel"$_{(8)}$. That mix is injected into the cylinder engine, mixed with the air inside the cylinder and compressed until combustion by the self ignition properties of the fuel. According to the combustion process we have the following categories:

- a. Direct Injection (DI) systems. When the fuel is injected directly inside the cylinder.

• b. Indirect Injection (IDI) systems. The fuel is injected in a prechamber and is transferred at high speed to the cylinder through a narrow passage. With this arrangement a high degree of air motion is obtained. This implies a faster air fuel mixing.

A second division is based in the way in which the gas exchange process is performed. We have two periods called closed and open periods, where the combustion or power generation occurs and the exhaust gases are expelled from the combustion chamber respectively. This division is similar to that applied to spark ignition engines. We can divide the engines as:

• a. Two-stroke engines. The combustion occurs in the region of top dead centre (TDC) and the gas exchange is made in the region of bottom dead centre (BDC) of each revolution. The scavenging or gas exchange process at the BDC takes from $100°$ to $150°$ of the crank angle (CA) period of $360°$. We can subdivide two-stroke engines into: loop scavenged engines, uniflow scavenge single piston engines and uniflow scavenge opposed piston engines.

We can summarize the two-stroke cycle as:

1-2	compression	Closed Period
2-3	heat release associated with combustion	
3-4	expansion	
4-5	blowdown	Open Period
5-6	scavenging	
6-1	supercharge	

• b. Four-stroke engines. For this type of engines the combustion and the gas exchange occur in alternate revolutions. As seen in Figure 2.1 the combustion occurs in TDC region with all the valves closed. After that, the exhaust valve opens (EVO) just before the BCD region, then the inlet valve opens (IVO) just before the TDC region. Just after the TDC region the exhaust valve closes (EVC) and the inlet valve closes (IVC) just after the BDC region. Here the engine starts the closed period where the combustion occurs, continuing with the next cycle. For this type of engine the crank angle (CA) period is 720°.

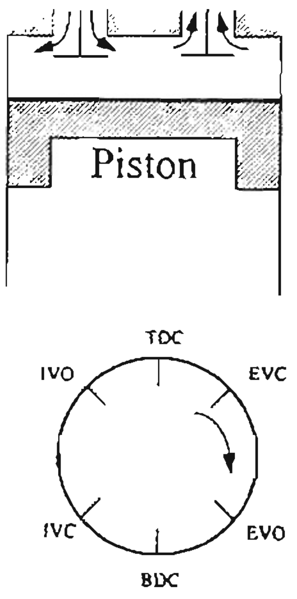


Figure 2.1: *Four-stroke engine (turbocharged)*(8).

We can summarize the four-stroke cycle as shown in Figure 2.2.

1-2	compression	Closed Period
2-3	heat release associated with combustion	
3-4	expansion	
4-5	blowdown	Open Period
5-6	exhaust	
6-7	overlap	
7-8	induction	
8-1	precompression	

Figure 2.2: *Four-stroke cycle for diesel engine* (8).

We can study the engine cycle based on air standard cycles, as shown in Figure 2.3. The first case is the constant pressure or diesel cycle (Figure 2.3-a), where the combustion process is modeled by a constant pressure heat addition (points 2-3). This was the description for "classical" diesel engines, with little relevance today. The second case is the constant volume or Otto cycle (Figure 2.3-b), where the combustion process is modeled by a constant volume heat addition (points 2-3). This cycle is normally used for spark ignition engines, but is valid for diesel engines with light load conditions. The third case is the dual combustion or composite cycle (Figure 2.3-c), where the combustion process is a combination of the previous cases. This cycle is closer to the actual operation of diesel engines. Other important theoretical cycles are the Atkinson cycle and the Carnot cycle.

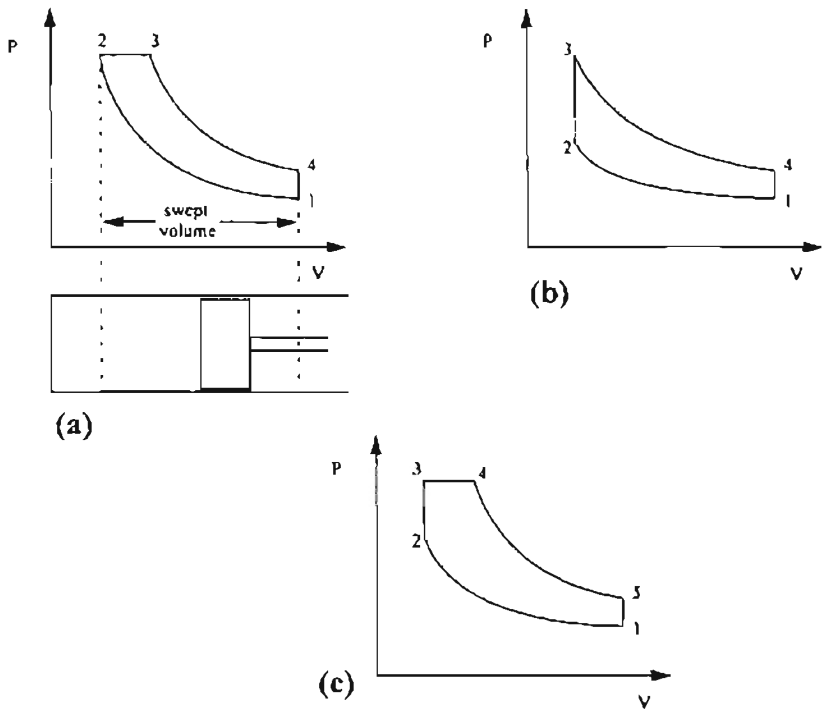


Figure 2.3: *Air standard cycles: (a) constant pressure cycle; (b) constant volume cycle; (c) dual combustion or composite cycle* (8).

The real processes of a diesel engine are different from the ideal cycles from the previous page. The combustion process occurs in the closed period, that is similar for two-stroke and four-stroke engines. We can say that the combustion process has three periods, as shown in Figure 2.4:

- (i) The delay period.

- (ii) The premixed burning phase (chemically controlled).

- (iii) The diffusion burning phase (controlled by mixing rate).

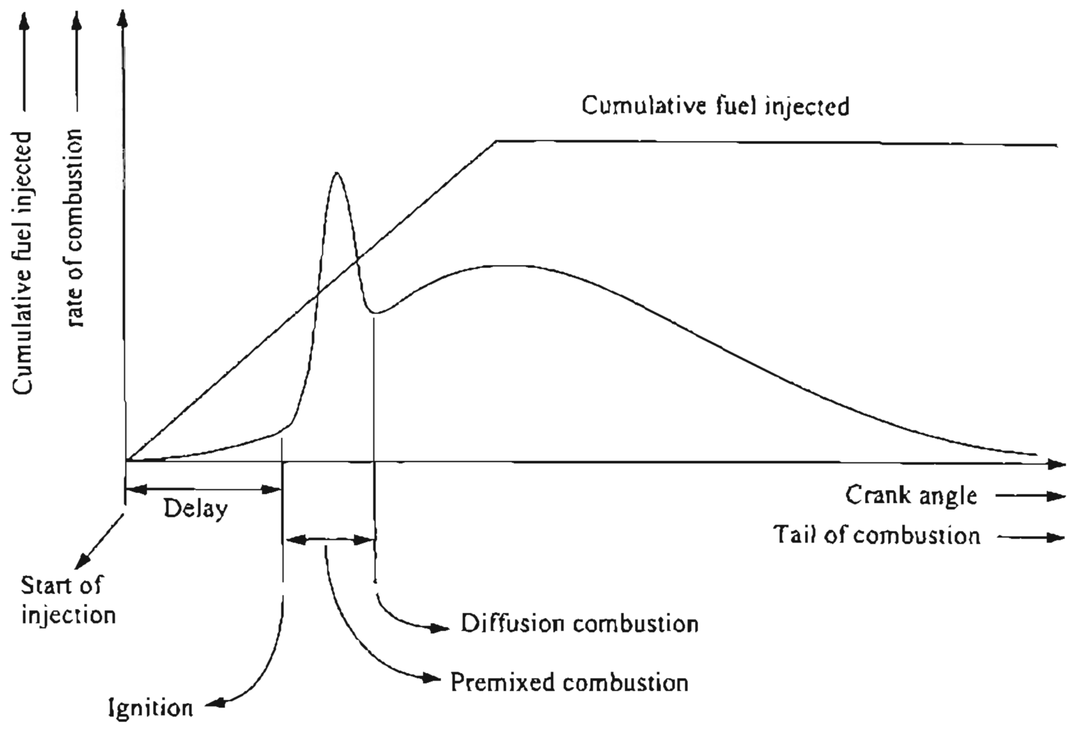


Figure 2.4: *Phases of combustion process* (8).

For the open period we have a gas exchange process as shown in Figure 2.5 for the case of four-stroke engine. The numbers at each step are related with the four-stroke cycle shown in Figure 2.2 and Figure 2.1.

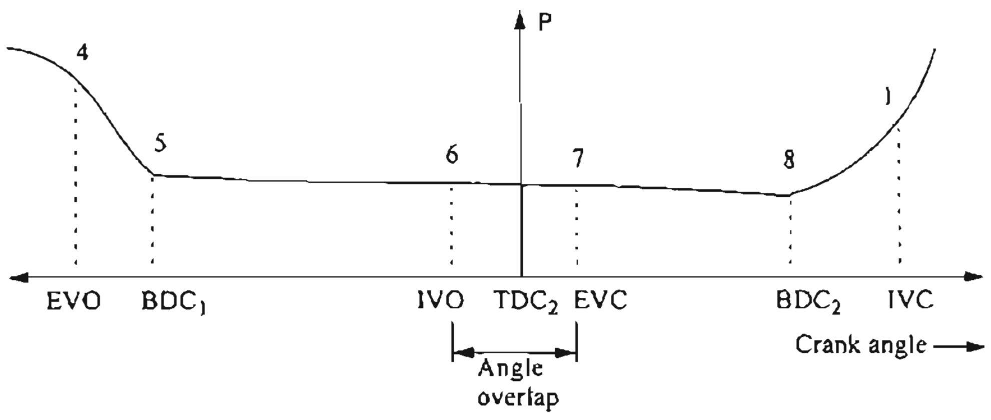


Figure 2.5: *Gas exchange four-stroke engine* (8).

To analyze the engine cycles in detail we can use a step by step basis, using small crank angle increments $d\theta$ (usually $0.5° < d\theta < 2°$ CA). The step could change according to the phase in the cycle. Another important term in the calculations is the heat transfer term dQ_L which defines the heat transfer from cylinder gas to wall and vice versa.

In the Appendix A the reader can see several different detailed mathematical models for the diesel engine. Each model is intended to define mathematically the combustion process inside the diesel engine. There models could be applied to the design and control of diesel engines.

CHAPTER 3

ENGINE MODEL.

Initially we tried to implement the Kao and Moskwa mean torque model $_{(7)}$ as shown in Figure A.2. That model requires specific parameters of the engine that were unavailable at the time of this project. Initial trials were conducted using some typical parameters found in different papers and books, but the model normally fails in its operation.

Cummins suggested a simplified model of the diesel engine that is shown in Figure 3.1. This model has a fueling delay defined by $e^{-\tau s}$ where s is the Laplace transform and τ is the time delay.

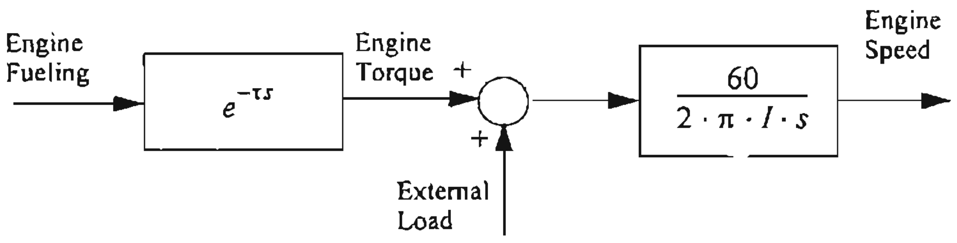
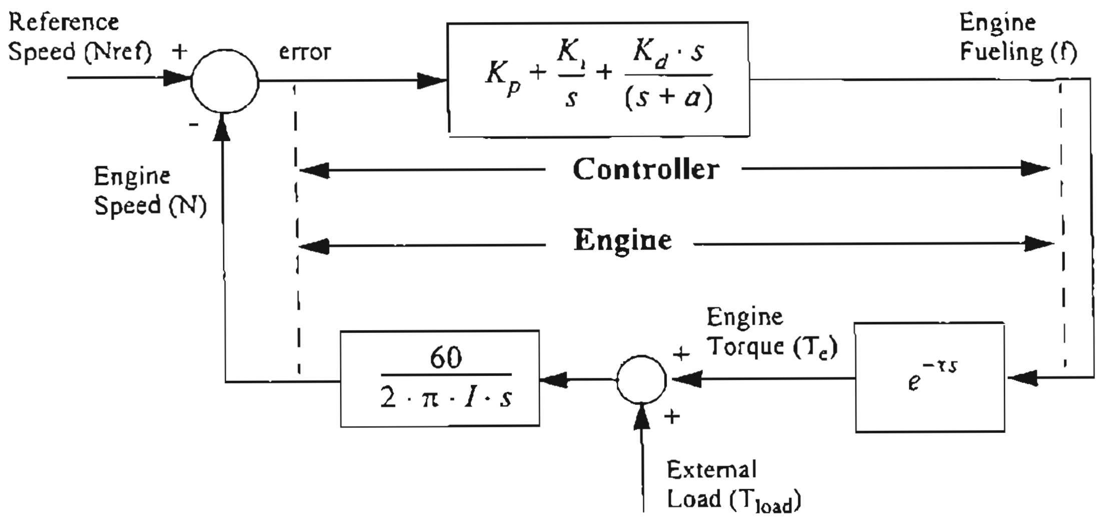


Figure 3.1: *Simple Engine model. Block diagram in s-domain.*

The model shown in Figure 3.1 has an initial PID controller proposed by Cummins for the nominal values of fueling delay $\tau = 80\ ms$ and engine inertia $I = 2\ lb\text{-}ft\text{-}sec^2$. The basic engine and the controller are shown in Figure 3.2. This model does not include any limitation in fueling and engine speed. Also, it does not include any friction. If the engine is working at a given speed without load, and we set the fueling to zero, the engine will continue at the same speed for unlimited time.



Reference Speed (Nref)
+
-
error
$K_p + \frac{K_i}{s} + \frac{K_d \cdot s}{(s+a)}$
Engine Fueling (f)
Engine Speed (N)
Controller
Engine
Engine Torque (T_e)
+
+
$\frac{60}{2 \cdot \pi \cdot I \cdot s}$
$e^{-\tau s}$
External Load (T_{load})

Figure 3.2: *Simple Engine Control System.*

For this basic system, we have the following parameters are:

s = Laplace Transform variable.	π = 3.14159265358979.
$Nref$ = Reference Engine Speed.	f = fueling mm^3 /stroke.
N = Actual Engine Speed in rpm.	τ = 80 msec delay.
$error = (Nref - N)$ = Speed error in rpm.	$K_p = 2$.
T_e = Engine Torque in lb-ft.	$K_i = 0.5$.
T_{load} = External load torque in lb-ft.	$K_d = 0.05$.
I = Engine Inertia = 2 lb-ft-sec^2.	a = 10 or 20.

To obtain a more realistic engine model, we included a simple gain block that multiplies the engine speed by b_l substracting the resulting value to the total load applied to the engine, as shown in Figure 3.3. This block represents viscous friction. We also included two saturation blocks to avoid negative or excessive engine speeds or fueling. The first block limits the fueling applied to the engine. That fueling was limited between 0 and 2240 $mm^3/stroke$. The second block limits the engine speed. The engine speed was limited between 0 and a top speed of 2000 rpm. For our experiments we tested with two friction values intended for low friction ($b_l = 0.01$) and high friction ($b_l = 0.1229$).

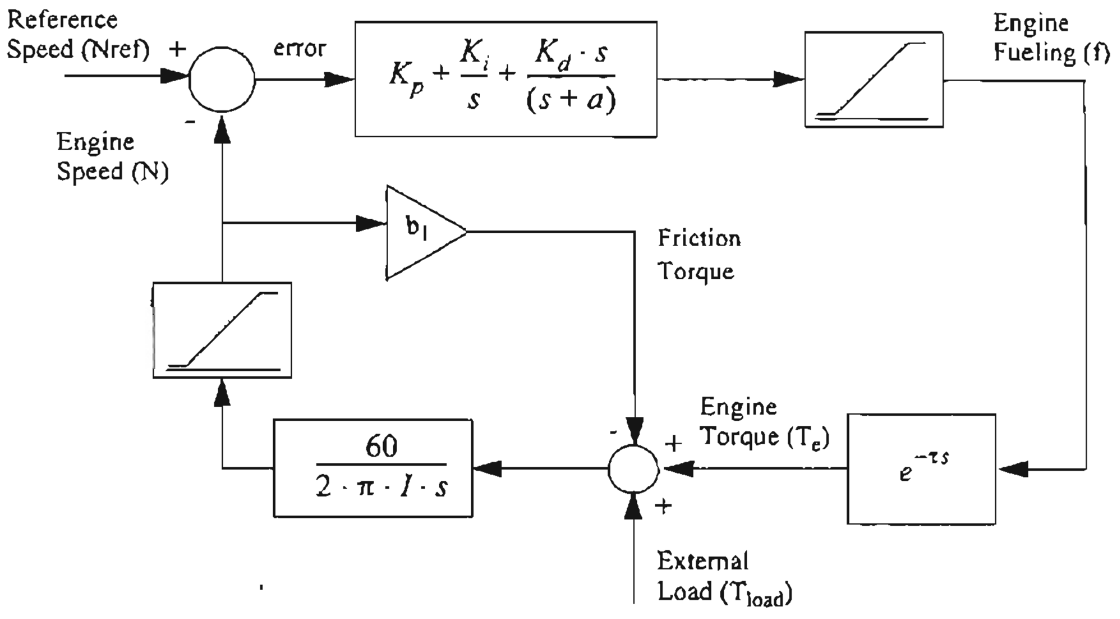


Figure 3.3: *Engine Control System with engine friction and limited speed and fueling.*

The basic PID controller has the parameter values $K_p = 2$, $K_i = 0.5$, $K_d = 0.05$ and two possible values for $a = 10$ and $a = 20$. We simulated the diesel engine with the basic PID controller for different conditions of inertia and fueling delay. The friction and the external load were constant and equal to $b_1 = 0.01$ and $T_{load} = 150\ lb\text{-}ft$ respectively.

If we unify the controller transfer function we obtain:

$$G(s) = \frac{(Kp + Kd)s^2 + (Kp \cdot a + Ki)s + Ki \cdot a}{s^2 + a \cdot s} \tag{3.1}$$

Since we have two values for **a**, the initial transfer functions for the PID controller are:

$$\text{For } a = 10 \Rightarrow \mathrm{G}(s) = \frac{2.05s^2 + 20.5s + 5}{s^2 + 10s} \tag{3.1}$$

$$\text{For } a = 20 \Rightarrow \mathrm{G}(s) = \frac{2.05s^2 + 40.5s + 10}{s^2 + 20s} \tag{3.1}$$

For the simulations we have assumed that the external load could change from 0 to 600 ft-lb. The maximum fueling rate was defined as 150 mm^3/stroke. The reference engine speed will change between 600 rpm and 650 rpm. We found that limitations in fueling were found for a load of 150 lb-ft. If we increase the load we need more fueling. If we use zero load, we found that negative speed or negative fueling will be needed, making this model unrealizable. Also, simulations with zero load and zero fuel will run forever for a fixed speed. A simulink representation of the model is shown in Figure 3.4.

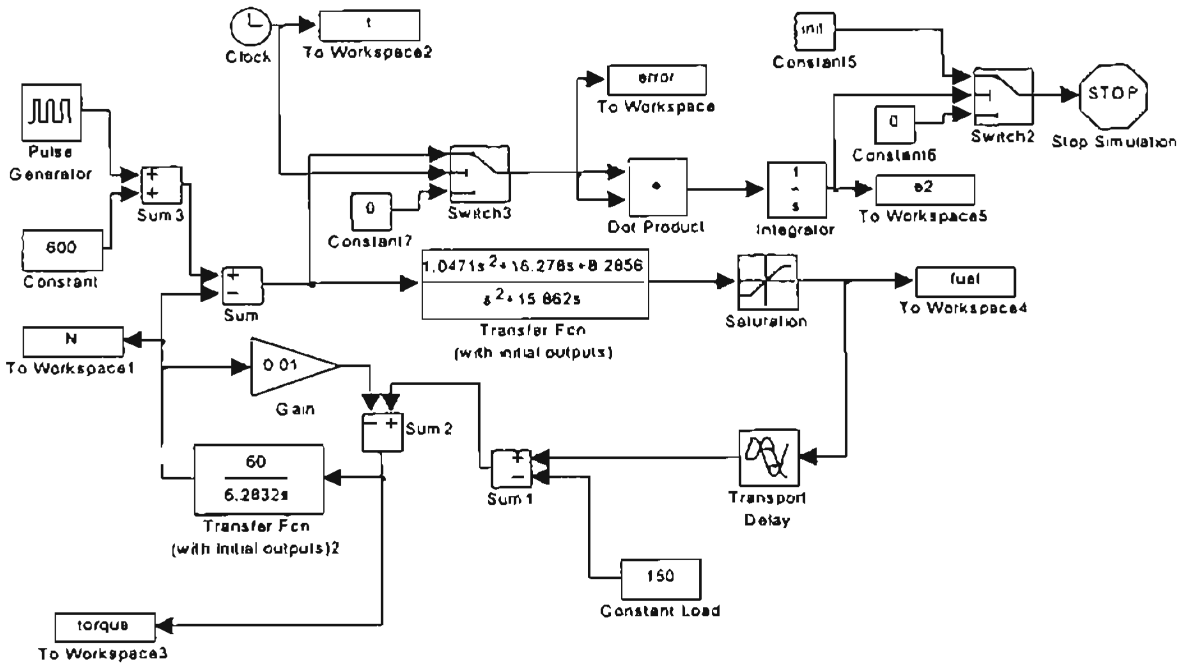


Figure 3.4: *Basic Engine model with limits in fuel and internal losses.*

Figure 3.5 shows the speed transition from 600 *rpm* to 650 *rpm* for different values of fueling delay and fixed engine inertia $I = 2\ lb\text{-}ft\text{-}sec^2$. We noticed how the percent overshoot and the mean square error increases as the fueling delay increases. Larger fueling delays implies that each action due to the controller takes more time to influence the engine response. For that reason an oscillatory response is observed. Figure 3.6 shows the same speed transition from 600 *rpm* to 650 *rpm* for different values engine inertia and fixed fueling delay $\tau = 80\ ms$. We noticed how the oscillatory response increases as the inertia reduces. The simulation results show how variations in the parameters affected the final system response. The subsequent chapters will discuss different alternatives to optimize the controller or to generate a controller to reduce the overshoot or the mean square error for the engine response.

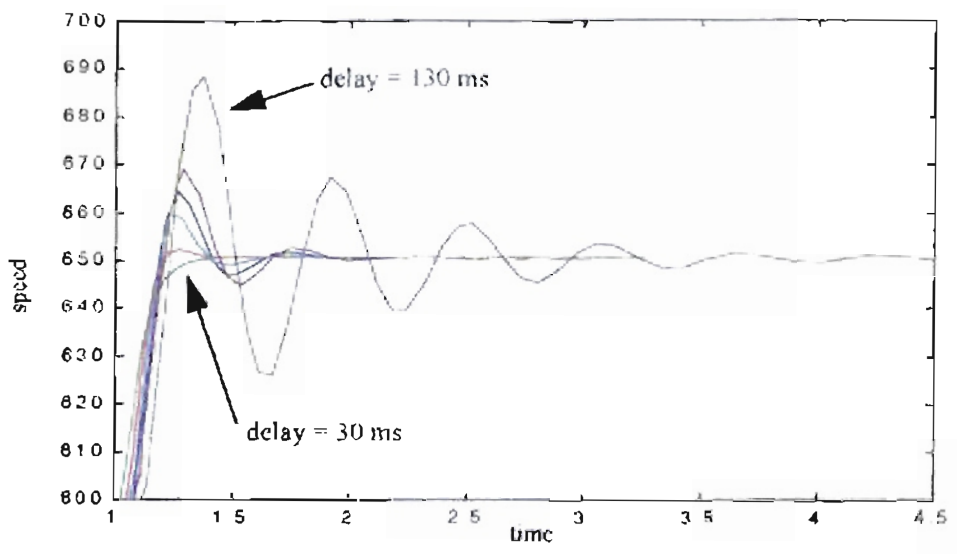


Figure 3.5: *Engine response for different fueling delay using basic PID controller.*

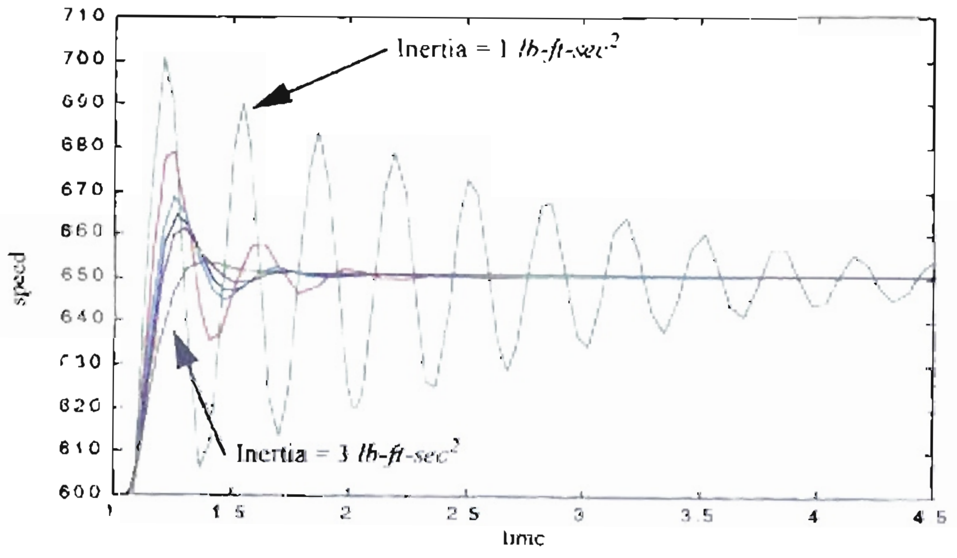


Figure 3.6: *Engine response for different engine inertia using basic PID controller.*

In addition to the engine model shown in Figure 3.2, we also modeled the engine with neural networks as shown in Figure 3.7 (subsystems are shown from Figure 3.8 to Figure 3.10). Due to all the interactions of the subsystems, we preferred the simulink representation of the previously referenced figures. For the training process we used data obtained from an engine simulation from a Cummins diesel engine. With that data we trained two neural networks: one for combustion and torque subsystem and the other for air mass generation subsystem. Those neural networks were based on the engine model shown in Figure A.2 in the appendix A. The combustion and torque production subsystem has as inputs the engine speed N, the engine fueling $\dot{m}_f$ and the fuel-air ratio, which is based on the mass flow $\dot{m}_4$ and the engine fueling $\dot{m}_f$. The same block produces the indicated torque T_i. The air compression subsystem depends of the engine speed N and the engine fueling $\dot{m}_f$ to generate the air mass flow $\dot{m}_4$.

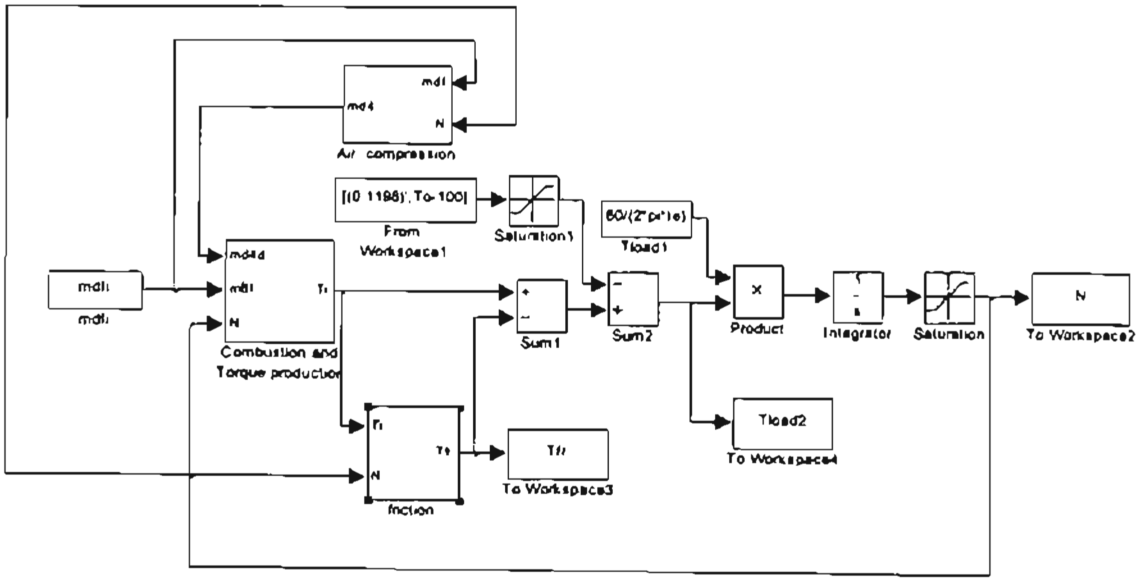


Figure 3.7: *Neural network based engine model.*

Since the fuel-air ratio and the fuel ratio have a smaller dynamic range than the engine speed (see Table 3.1), we divided the speed input by 1000 for the combustion and torque production neural network. A similar operation was made with the torque output. Figure 3.8 shows a representation of the combustion and torque production neural network. Similar considerations were applied to the air compression subsystem shown in Figure 3.9.

Input or Output	minimum	maximum	Training range
fuel-air ratio	0	0.0988	0 to 0.1
fuel ratio	0	2.9132	0 to 0.3
engine speed (rpm)	572.40	1972.70	0 to 2

Table 3.1: *Input-Output range for combustion and torque production subsystem.*

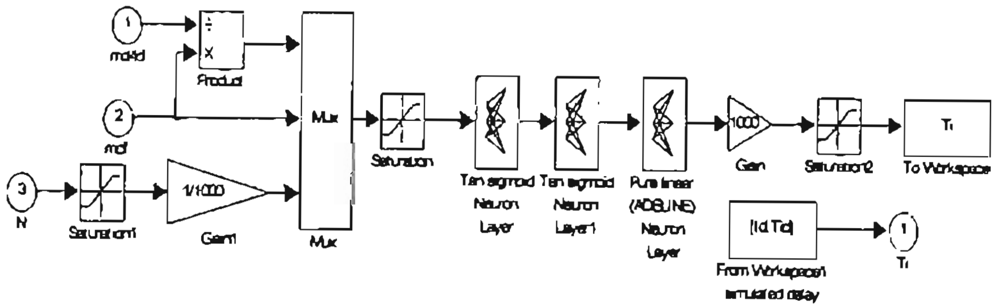


Figure 3.8: *Combustion and Torque production subsystem.*

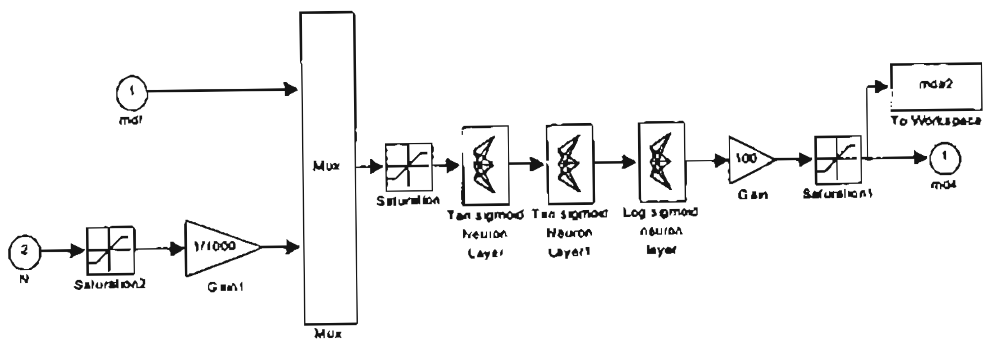


Figure 3.9: *Air compression subsystem.*

The engine friction subsystem is based on the Eq. (A.21) from appendix A. For the model of Figure 3.10, we replaced the mean piston speed S_p by the engine speed N.

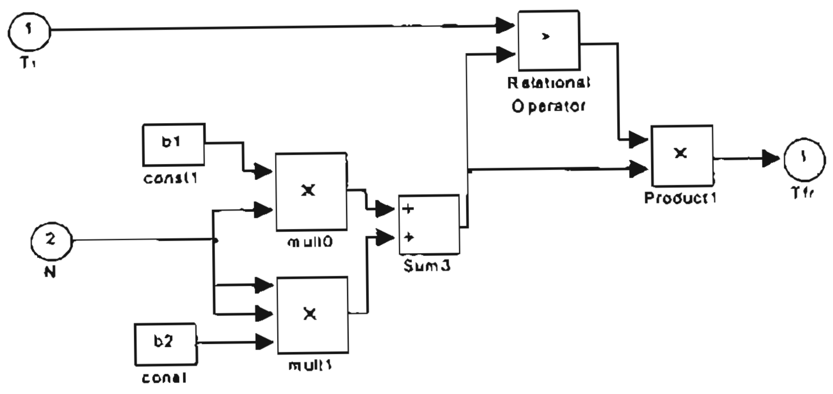


Figure 3.10: *Engine friction subsystem.*

We will see in the following chapters how the models previously described were applied for adaptive control, genetic reinforcement learning and reinforcement learning.

CHAPTER 4

SELF-TUNING REGULATOR APPLIED TO DIESEL ENGINES.

4.1. Introduction.

In order to provide a standard with which to compare the reinforcement learning algorithms which will be presented in later chapters, this chapter will apply the self-tuning regulator (1) to diesel engine control. This is a standard linear adaptive control technique, with block diagram as shown in Figure 4.1.

The self-tuning regulator is an "indirect" method; a model of the process is developed (in the estimation block), and this model is used to determine the controller (in the Controller Design block).

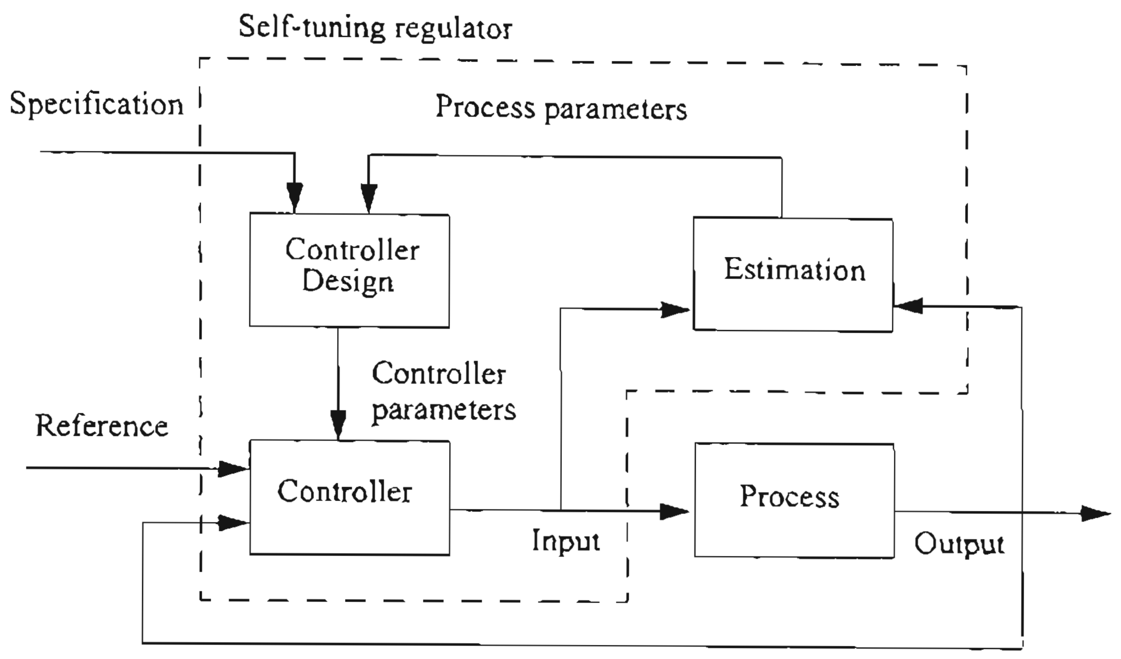


Figure 4.1: *Block diagram of a self-tuning regulator* (1).

4.2. Pole Placement Design.

There are several different types of self-tuning regulator. One is adaptive pole placement. The idea of this method is to design a controller to meet the specified closed-loop poles specifications. If we take our diesel engine model from Figure 4.2 we can model the input-output relation as:

$$N = \frac{A \cdot S}{B \cdot R + A \cdot S} \cdot Nref - \frac{A \cdot R}{B \cdot R + A \cdot S} \cdot T_{load} \tag{4.1}$$

The idea is to adjust the controller parameters to obtain the desired pole locations.

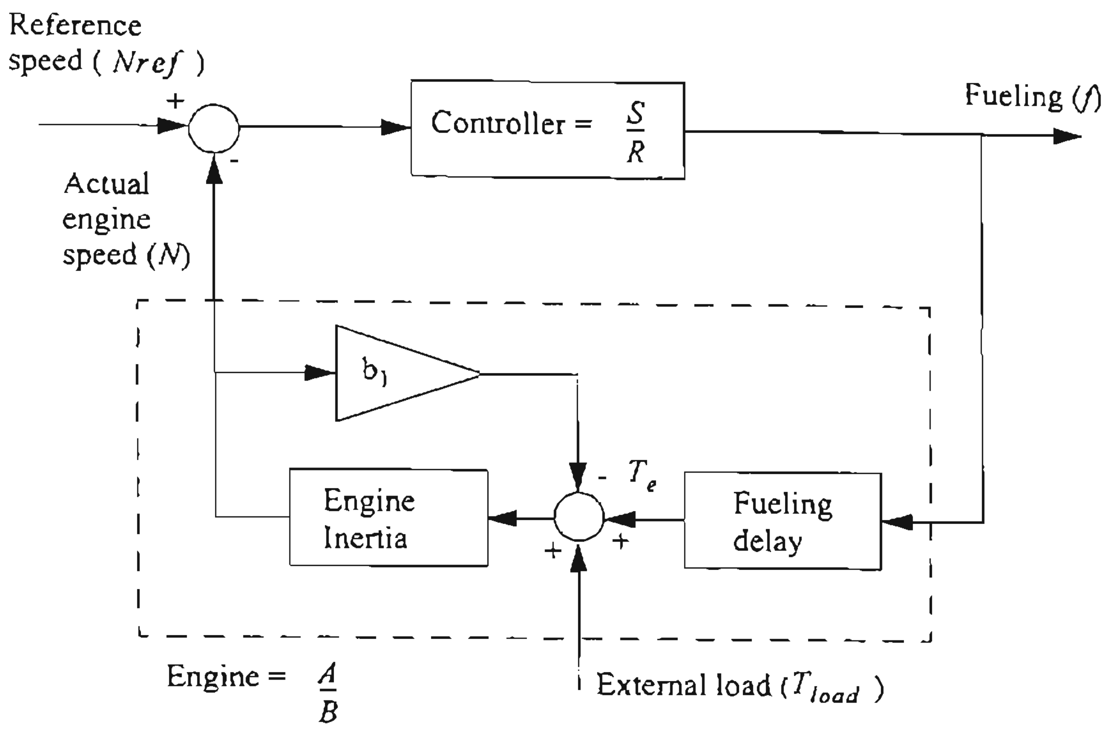

Reference speed ($Nref$)
+
Controller = $\frac{S}{R}$
Fueling (f)
-
Actual engine speed (N)
b_1
- T_e
Engine Inertia
+
+
Fueling delay
Engine = $\frac{A}{B}$
External load (T_{load})

Figure 4.2: *Engine model.*

For the self-tuning regulator we need to define the system structure. The engine transfer function from fueling f to engine speed N is:

$$\frac{N(s)}{f(s)} = \frac{60/(2\pi I)}{s + (60 \cdot b_1)/(2\pi I)} e^{-\tau s} \tag{4.2}$$

The simplified system model is shown in Figure 4.3, where the load is before the engine delay. We will be using a digital controller, therefore we need to obtain the discrete-time transfer function of the diesel engine with a zero-order hold:

$$Z\left[\frac{1-e^{-Ts}}{s} \cdot \frac{60/2\pi I}{s + 60 \cdot b_1/2\pi I} \cdot e^{-\tau s}\right] = \frac{(1/b_1)\left(1 - e^{-\frac{60 b_1 T}{2\pi I}}\right)}{z - e^{-\frac{60 b_1 T}{2\pi I}}} \cdot z^{-\frac{\tau}{T}} = \frac{N(z)}{f(z)} \tag{4.3}$$

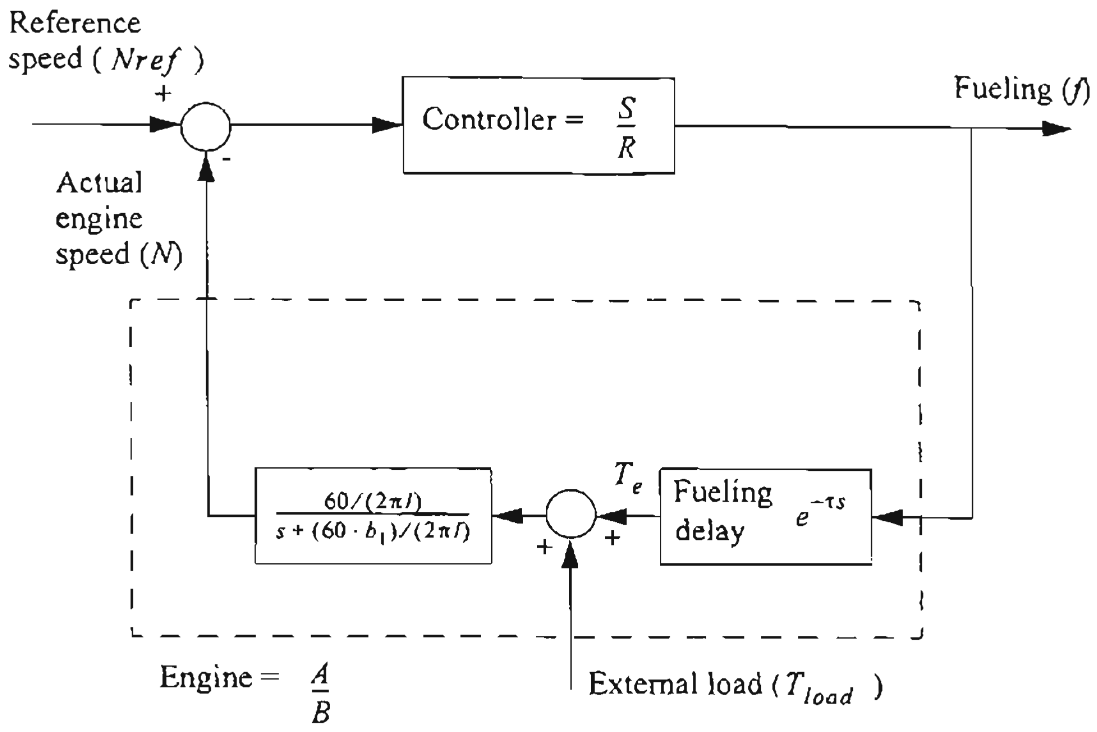

Reference speed ($Nref$)
+
Controller = $\frac{S}{R}$
Fueling (f)
-
Actual engine speed (N)
$\frac{60/(2\pi I)}{s+(60\cdot b_1)/(2\pi I)}$
T_e
Fueling delay $e^{-\tau s}$
+
+
Engine = $\frac{A}{B}$
External load (T_{load})

Figure 4.3: *Simplified engine model.*

For our experiments we want to select an appropriate sample time T. The engine model will change according to the fueling delay τ and the engine inertia I, among other parameters. If we define for our experiments that the fueling delay τ will change between 30 ms and 130 ms we want a sample time that will cover those variations using a reasonable system order. We evaluated some transfer functions for different sample times T and values of b_1, as seen in Table 4.1.

Table 4.1: *Engine transfer functions for different values of sampling time T and engine friction b_1.*

T (ms)	b_1	Transfer function
10	0.01	$\frac{0.0477 \cdot z^{-100\tau}}{z-0.9995}$
10	0.1229	$\frac{0.0476 \cdot z^{-100\tau}}{z-0.9941}$
50	0.01	$\frac{0.2384 \cdot z^{-20\tau}}{z-0.9976}$
50	0.1229	$\frac{0.2353 \cdot z^{-20\tau}}{z-0.9711}$
100	0.01	$\frac{0.4763 \cdot z^{-10\tau}}{z-0.9952}$
100	0.1229	$\frac{0.4637 \cdot z^{-10\tau}}{z-0.9430}$

We want to define a system structure that supports different variations in the engine delay and inertia. From Eq. (4.3) we can see that system order variations were due to the engine delay. We can estimate a system structure of the form:

$$\frac{N(z)}{f(z)} = \frac{a_m + a_{m-1}z^{-1} + a_{m-2}z^{-2} + \ldots + a_1 z^{-m+1} + a_0 z^{-m}}{z-b} \tag{4.4}$$

or

$$\frac{N(z)}{f(z)} = \frac{a_m z^m + a_{m-1}z^{m-1} + a_{m-2}z^{m-1} + \ldots + a_1 z + a_0}{z^m(z-b)} \tag{4.5}$$

We now want to define a mechanism to identify the parameters described in Eq. (4.4) and Eq. (4.5). That mechanism is defined by the parameter estimation process of the next section. This is the procedure which will be performed by the Estimation block of Figure 4.1.

4.3. Parameter estimation.

For parameter estimation we have our linear model defined as:

$$Z(k) = H(k)\theta + V(k) \tag{4.6}$$

where for each instant k, Z is a vector with the measurements, H is the data matrix, θ is a vector with unknown parameters and V represents noise or variations in the parameters that we cannot explain. For example, if our system is represented by $y_j = f(x_{1j}, x_{2j}, x_{3j})$ we can say that the linear model is:

$$y_j = \theta_1 x_{1j} + \theta_2 x_{2j} + \theta_3 x_{3j} + V(j) \tag{4.7}$$

resulting in:

$$Z(k) = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_k \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \quad H(k) = \begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \\ x_{13} & x_{23} & x_{33} \\ \dots & \dots & \dots \\ x_{1k} & x_{2k} & x_{3k} \end{bmatrix} \quad V(k) = \begin{bmatrix} V(1) \\ V(2) \\ V(3) \\ \dots \\ V(k) \end{bmatrix} \tag{4.8}$$

We want to obtain a function $\hat{Z}(k) = H(k)\hat{\theta}$ by minimizing:

$$\min_{\text{respect } \theta} [Z(k) - \hat{Z}(k)]^T [Z(k) - \hat{Z}(k)] = \text{Sum Squared Error} \tag{4.9}$$

where we will obtain the least squared estimate of the parameter vector $\hat{\theta}_{LS}$. If we have our system represented by:

$$y(t) = \phi_1 y(t-1) + \ldots + \phi_p y(t-p) + a(t) \tag{4.10}$$

then:

$$Z(k) = \begin{bmatrix} y(p+1) \\ y(p+2) \\ \ldots \\ y(N) \end{bmatrix} \qquad \theta = \begin{bmatrix} \phi_1 \\ \varphi_2 \\ \phi_3 \end{bmatrix} \tag{4.11}$$

$$H(k) = \begin{bmatrix} y(p) & y(p-1) & \ldots & y(1) \\ y(p+1) & y(p) & \ldots & y(2) \\ \ldots & \ldots & \ldots & \ldots \\ y(N-1) & y(N-2) & \ldots & y(N-p) \end{bmatrix} \qquad V(k) = \begin{bmatrix} a(p+1) \\ a(p+2) \\ \ldots \\ a(p+N) \end{bmatrix}$$

where we minimize:

$$\tilde{Z}^T\tilde{Z} = [Z(k) - H(k)\theta]^T[Z(k) - H(k)\theta] \Rightarrow Z^TZ - 2Z^TH\theta + \theta^TH^TH\theta = J \tag{4.12}$$

where $\tilde{Z} = Z - \hat{Z}$. To find the minimum we must calculate the gradient using the following properties:

$$\nabla_x[x^Ty] = y$$

$$\nabla_x[y^Tx] = y$$

$$\nabla_x[x^TAx] = Ax + A^Tx$$

The minimum of $\tilde{Z}^T\tilde{Z}$ is found by the relation:

$$\nabla_\theta J = -2H^TZ + 2H^TH\theta = 0 \tag{4.13}$$

which implies the normal equation:

$$H^TH\theta = H^TZ \Rightarrow \hat{\theta}_{LS} = [H^TH]^{-1}H^TZ \tag{4.14}$$

which is the batch form of the least squared estimate.

In general we have that:

$$Z(k) = H(k)\theta + V(k) \tag{4.15}$$

for k measurements. If we take an additional $k+1$ measurement then Z grows in one element and H grows in one row:

$$H(k+1) = \begin{bmatrix} H(k) \\ \psi_{k+1}^T \end{bmatrix} \Rightarrow H^T(k+1) = \begin{bmatrix} H^T(k) & \psi_{k+1} \end{bmatrix} \tag{4.16}$$

where:

$$\psi_{k+1}^T = \begin{bmatrix} y(N) & y(N-1) & \dots & y(N-p+1) \end{bmatrix}$$

then:

$$H^T(k+1)H(k+1) = \begin{bmatrix} H^T(k) & \psi_{k+1} \end{bmatrix} \begin{bmatrix} H(k) \\ \psi_{k+1}^T \end{bmatrix}$$

$$= [H^T(k)H(k) + \psi_{k+1}\psi_{k+1}^T]$$

We want to find $[H^T(k)H(k) + \psi_{k+1}\psi_{k+1}^T]^{-1}$ where $P_k^{-1} = H^T(k)H(k)$ then we would find P_{k+1} from P_k:

$$[H^T(k)H(k) + \psi_{k+1}\psi_{k+1}^T]^{-1} = [P_k^{-1} + \psi_{k+1}\psi_{k+1}^T]^{-1} = P_{k+1} \tag{4.17}$$

then we can use the matrix inversion lemma:

$$(A + BB^T)^{-1} = A^{-1} - A^{-1}B[I + B^TA^{-1}B]^{-1}B^TA^{-1} \tag{4.18}$$

$$\Rightarrow P_{k+1} = P_k - P_k\psi_{k+1}[1 + \psi_{k+1}^T P_k \psi_{k+1}]^{-1}\psi_{k+1}^T P_k \tag{4.19}$$

If we define the scalar α_k as:

$$\alpha_k = \frac{1}{1 + \psi_{k+1}^T P_k \psi_{k+1}} \tag{4.20}$$

then:

$$\Rightarrow P_{k+1} = [P_k^{-1} + \psi_{k+1}\psi_{k+1}^T]^{-1} = P_k - \alpha_k P_k \psi_{k+1}\psi_{k+1}^T P_k \tag{4.21}$$

We need another relation to find the parameters. From Eq. (4.14) we have for the time k that:

$$\begin{aligned}\hat{\theta}(k) &= [H^T(k)H(k)]^{-1}H^T(k)Z(k) \\ &= P(k)H^T(k)Z(k)\end{aligned} \tag{4.22}$$

and for the time $k+1$ that:

$$\begin{aligned}\hat{\theta}(k+1) &= P(k+1)H^T(k+1)Z(k+1) \\ &= P(k+1)\begin{bmatrix} H^T(k) & \psi_{k+1} \end{bmatrix}\begin{bmatrix} Z(k) \\ z(k+1) \end{bmatrix} \\ &= P(k+1)\Big[H^T(k)Z(k) + \psi_{k+1}z(k+1)\Big] \\ &= [P_k - \alpha_k P_k \psi_{k+1}\psi_{k+1}^T P_k]\Big[H^T(k)Z(k) + \psi_{k+1}z(k+1)\Big] \\ &= P_k H^T(k)Z(k) + P_k\psi_{k+1}z(k+1) \\ &\quad - \alpha_k P_k \psi_{k+1}\psi_{k+1}^T P_k H^T(k)Z(k) \\ &\quad - \alpha_k P_k \psi_{k+1}\psi_{k+1}^T P_k \psi_{k+1} z(k+1) \\ &= \theta(k) + P_k\psi_{k+1}\alpha_k\left[\frac{z(k+1)}{\alpha_k} - \psi_{k+1}^T\theta(k) - \psi_{k+1}^T P_k \psi_{k+1} z(k+1)\right]\end{aligned}$$

$$= \theta(k) + P_k\psi_{k+1}\alpha_k[z(k+1) + \cancel{\psi_{k+1}^T P_k\psi_{k+1}z(k+1)}$$
$$- \psi_{k+1}^T\theta(k) - \cancel{\psi_{k+1}^T P_k\psi_{k+1}z(k+1)}]$$

$$\Rightarrow \hat{\theta}(k+1) = \hat{\theta}(k) + \alpha_k P_k\psi_{k+1}[z(k+1) - \psi_{k+1}^T\theta(k)] \tag{4.23}$$

where the value $\psi_{k+1}^T\theta(k) = \hat{z}(k+1)$ is the prediction of the $z(k+1)$ value. The value $z(k+1) - \psi_{k+1}^T\theta(k)$ is the prediction error. The gain matrix $K(k)$ is defined as the value $\alpha_k P_k\psi_{k+1}$. We can see in Eq. (4.23) that the new estimate of $\hat{\theta}(k+1)$ is based on the previous value $\hat{\theta}(k)$ and the correction term $\Delta\theta(k) = \alpha_k P_k\psi_{k+1}[z(k+1) - \psi_{k+1}^T\theta(k)]$.

To initialize the algorithm, typically $P_0 = \beta I$ and $\theta = zero$. Eq. (4.21) and Eq. (4.23) make up the recursive least squares method for parameter estimation.

If the parameter changes with time, we need a factor to forget older data, especially for adaptive filtering. We are minimizing $\tilde{Z}^T\tilde{Z} = \sum_{i=1}^{k}\tilde{z}^2(i)$, but we want to weight the last errors more than the older ones, then we can use the weighted least squares as:

$$\sum_{i=1}^{k}\lambda^{k-i}\cdot\tilde{z}^2(i) = \tilde{z}^2(k) + \lambda\cdot\tilde{z}^2(k-1) + \lambda^2\cdot\tilde{z}^2(k-2) + \dots \tag{4.24}$$

where $0 < \lambda < 1$. The general weighted least squares is $\tilde{Z}^T W\tilde{Z}$. For the λ case W has the form:

$$W(k) = \begin{bmatrix} \cdot & & & 0 \\ & \cdot & & \\ & & \lambda^2 & \\ & & & \lambda \\ 0 & & & & 1 \end{bmatrix}$$

If we want to minimize $\tilde{Z}^T W \tilde{Z}$ then:

$$\hat{\theta}_{WLS} = [H^T W H]^{-1} H^T W Z$$

As the least squares we will estimate $\hat{\theta}_{WLS}(k+1)$ from $\hat{\theta}_{WLS}(k)$. We have the following relations:

$$H(k+1) = \begin{bmatrix} H(k) \\ \psi_{k+1}^T \end{bmatrix} \qquad Z(k+1) = \begin{bmatrix} Z(k) \\ z(k+1) \end{bmatrix} \qquad W(k+1) = \begin{bmatrix} \lambda W(k) & 0 \\ 0 & 1 \end{bmatrix}$$

$$P_{k+1} = [H^T(k+1) W(k+1) H(k+1)]^{-1}$$

$$= \left[\begin{bmatrix} H^T(k) & \psi(k+1) \end{bmatrix} \begin{bmatrix} \lambda W(k) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} H(k) \\ \psi_{k+1}^T \end{bmatrix} \right]^{-1}$$

$$= \left[\begin{bmatrix} \lambda H^T(k) W(k) & \psi_{k+1} \end{bmatrix} \begin{bmatrix} H(k) \\ \psi_{k+1}^T \end{bmatrix} \right]^{-1}$$

$$= [\lambda H^T(k) W(k) H(k) + \psi_{k+1} \psi_{k+1}^T]^{-1}$$

where $H^T(k) W(k) H(k) = P_k^{-1}$. Using the matrix inversion lemma (see Eq. (4.18)):

$$P_{k+1} = \frac{1}{\lambda}P_k - \frac{1}{\lambda}P_k\psi_{k+1}\left[1 + \frac{1}{\lambda}\psi_{k+1}^T P_k\psi_{k+1}\right]^{-1}\psi_{k+1}^T P_k\frac{1}{\lambda}$$

$$= \frac{1}{\lambda}[P_k - P_k\psi_{k+1}[\lambda + \psi_{k+1}^T P_k\psi_{k+1}]^{-1}\psi_{k+1}^T P_k] \tag{4.25}$$

where $\alpha_k = \dfrac{1}{\lambda + \psi_{k+1}^T P_k\psi_{k+1}}$. We can rewrite Eq. (4.25) as:

$$P_{k+1} = \frac{1}{\lambda}[P_k - \alpha_k P_k\psi_{k+1}\psi_{k+1}^T P_k] \tag{4.26}$$

For the parameter estimation we have that:

$$\theta(k+1) = P_{k+1}H^T(k+1)W(k+1)Z(k+1)$$

$$= P_{k+1}\begin{bmatrix} H^T(k) & \psi_{k+1}\end{bmatrix}\begin{bmatrix}\lambda W(k) & 0 \\ 0 & 1\end{bmatrix}\begin{bmatrix} Z(k) \\ z(k+1)\end{bmatrix}$$

$$= P_{k+1}[\lambda H^T(k)W(k)Z(k) + \psi_{k+1}z(k+1)]$$

$$= [P_k - P_k\psi_{k+1}\alpha_k\psi_{k+1}^T P_k]\frac{1}{\lambda}[\lambda H^T(k)W(k)Z(k) + \psi_{k+1}z(k+1)]$$

$$= \hat{\theta}(k) + P_k\psi_{k+1}\alpha_k\Big[\alpha_k^{-1}z(k+1)\frac{1}{\lambda} - \psi_{k+1}^T P_k H(k)W(k)Z(k) - \psi_{k+1}^T P_k z(k+1)\frac{1}{\lambda}\Big]$$

By substitution of α_k we obtain:

$$\hat{\theta}(k+1) = \hat{\theta}(k) + P_k\psi_{k+1}\alpha_k[z(k+1) - \psi_{k+1}^T\hat{\theta}(k)] \tag{4.27}$$

We have a new set of equations defined by Eq. (4.26) and Eq. (4.27) to update $\hat{\theta}(k+1)$ and P_{k+1} from $z(k+1)$ and ψ_{k+1}. To choose λ we have two options:

λ too small $\rightarrow$ increases the variance of the estimate (more oscillation).

λ too large $\rightarrow$ increases the bias of the estimate.

For initialization we can choose $P_0 = \alpha I$ or $P_0 = [H^T H]^{-1}$ for the first data set of points. For our case we must use the ARX model for exogenous inputs:

$$y(t) - \phi_1 y(t-1) - \ldots - \phi_p y(t-p) = b_0 u(t) + b_1 u(t-1) + \ldots + b_m u(t-m) + a(t)$$

where the data matrix and the vector of parameters is defined by:

$$H = \begin{bmatrix} y(p-1) & \ldots & y(1) & u(p) & \ldots & u(p-m) \\ y(p) & \ldots & y(2) & u(p+1) & \ldots & u(p-m+1) \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ y(N) & \ldots & y(N-p+2) & u(N+1) & \ldots & u(N-m+1) \end{bmatrix}$$

$$\theta = \begin{bmatrix} \phi_1 \\ \ldots \\ \phi_p \\ b_0 \\ \ldots \\ b_m \end{bmatrix}$$

4.4. Parameter estimation for the diesel engine.

The objective of this section is to find a generic transfer function model that could be used in later sections in the implementation of an adaptive controller. Looking at Eq. (4.4) and Eq. (4.5) we can note that if we use a very fast sampling time we would obtain a system with a large order and eventually more difficult to manage. If we use a very slow sample time we would obtain a reduced system order, but we also reduce the capability of modeling different time delays with the same transfer function model. We will try to find a trade-off between sampling time and system order which allow us the use of a unique transfer function for different fueling delays and inertia values.

We simulated the engine with the original PID controller for different engine delay conditions using the model shown in Figure 4.4. We saved different data files of fueling versus engine speed for diverse values of fueling delay and different engine load. For identification purposes the speed reference was changed between 600 and 650 rpm. The engine load was simulated with a normal random number generator with the mean value equal to the desired load and variance equal to one. The engine has a friction denoted by a block with the same description. A variable called b_1 could be adjusted for different friction values.

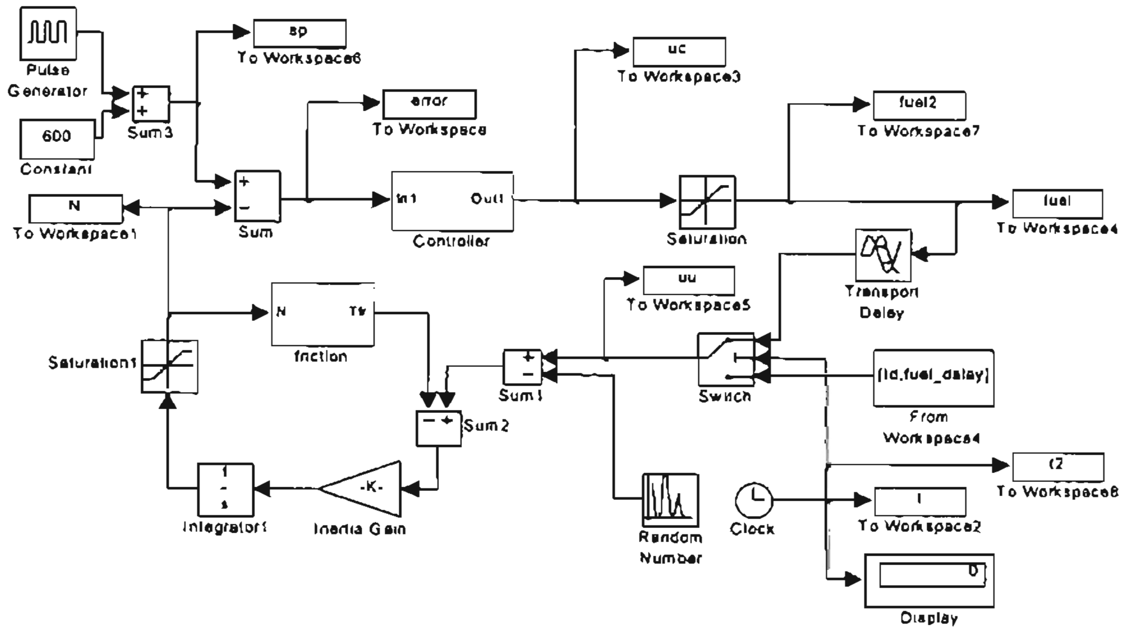


Figure 4.4: *Simulink model for self-tuning control.*

For the first case we used a sample time of $T = 100\ ms$, with friction $b_1 = 0.1229$ and a engine transfer function model:

$$\frac{N(z)}{f(z)} = \frac{a_2 z^2 + a_1 z + a_0}{z^3 - b z^2} \tag{4.28}$$

Knowing the average engine load we applied the identification process described by Eq. (4.26) and Eq. (4.27) for different fueling delay and engine load. Figure 4.5 to Figure 4.8 have the identified parameters for different engine load plotted versus fueling delay. There are 16 different curves in each figure, one for each engine load (0 to 150 lb-ft). In most cases the curves directly overlap.

We can see in Figure 4.5 how the parameter a_2 decreases from a value close to 0.4637 for $\tau = 0$ to a value near to zero for $\tau = 100\ ms$. Similar results were obtained for the parameter a_1, that reaches its maximum value at $\tau = 100\ ms$ as shown in Figure 4.6. The parameter a_0 increases after $\tau = 100\ ms$ as shown in Figure 4.7. The parameter b oscillates between 0.9418 and 0.9430 adjusting its value for the different values in the engine delay as shown in Figure 4.8. From the figures we can see where each parameter of the numerator (a_2 to a_0) has its maximum value with respect to the fueling delay. For example, a_2 reaches its maximum value for zero fueling delay and a_1 has its maximum value for $\tau = 100\ ms$.

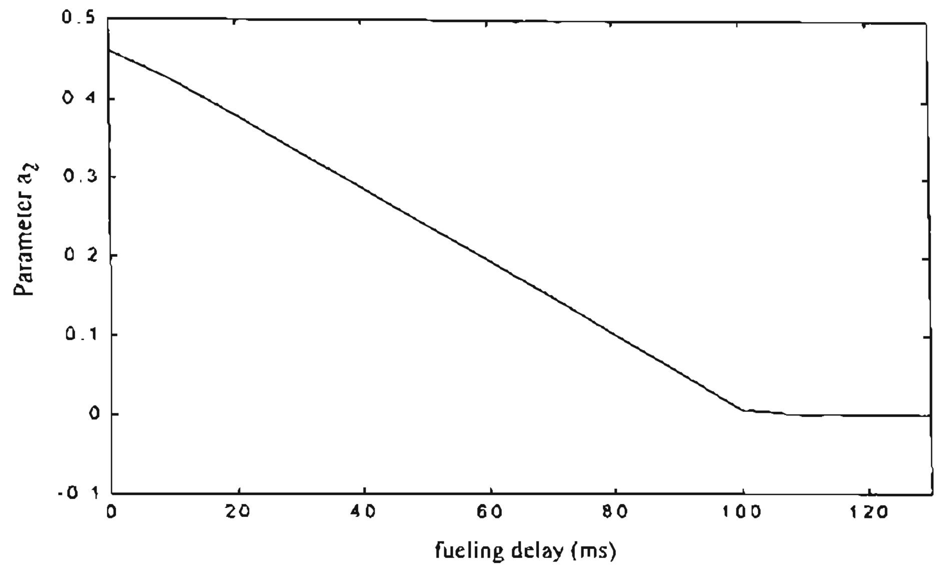


Figure 4.5: *Parameter* a_2 *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 100ms$).

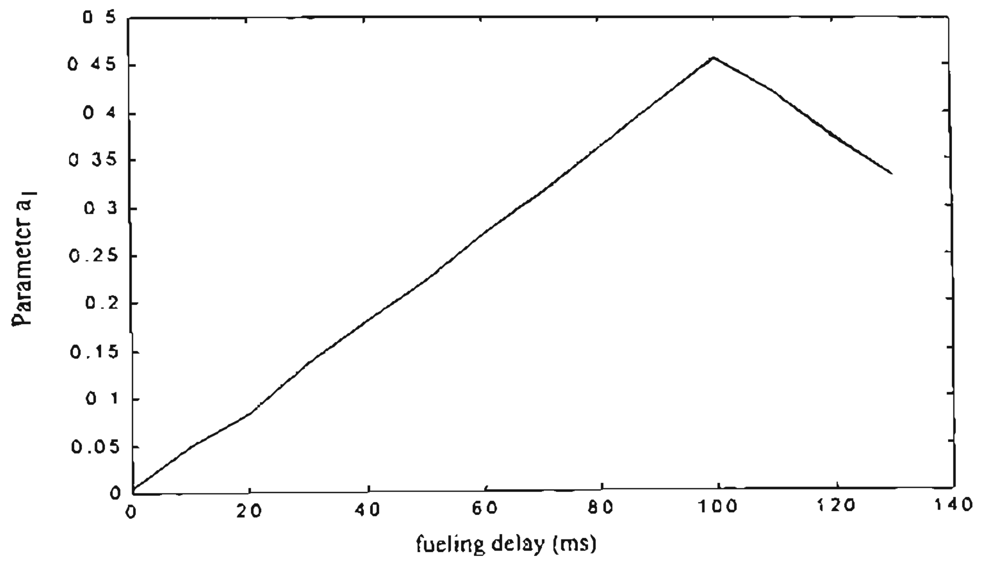


Figure 4.6: *Parameter* a_1 *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 100ms$).

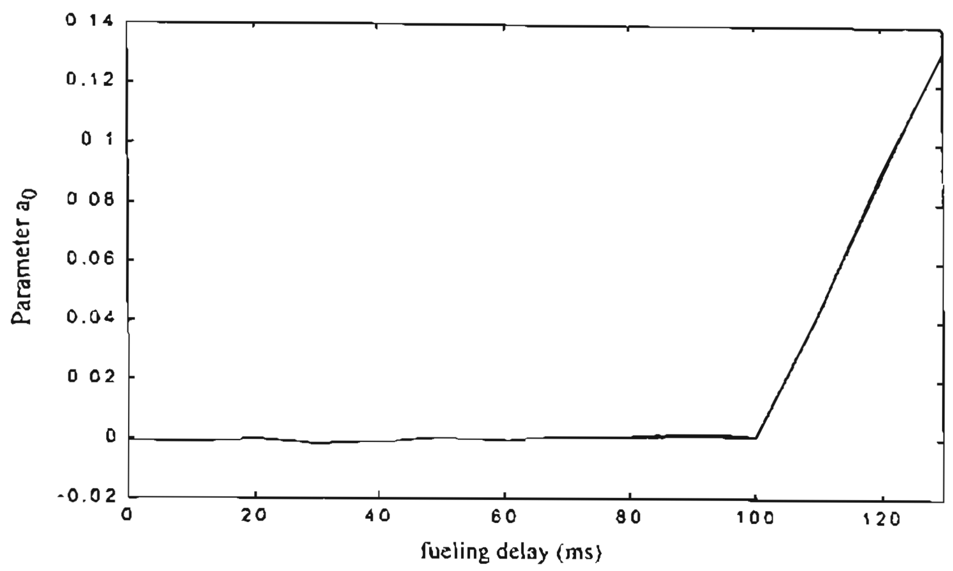


Figure 4.7: *Parameter* a_0 *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 100ms$).

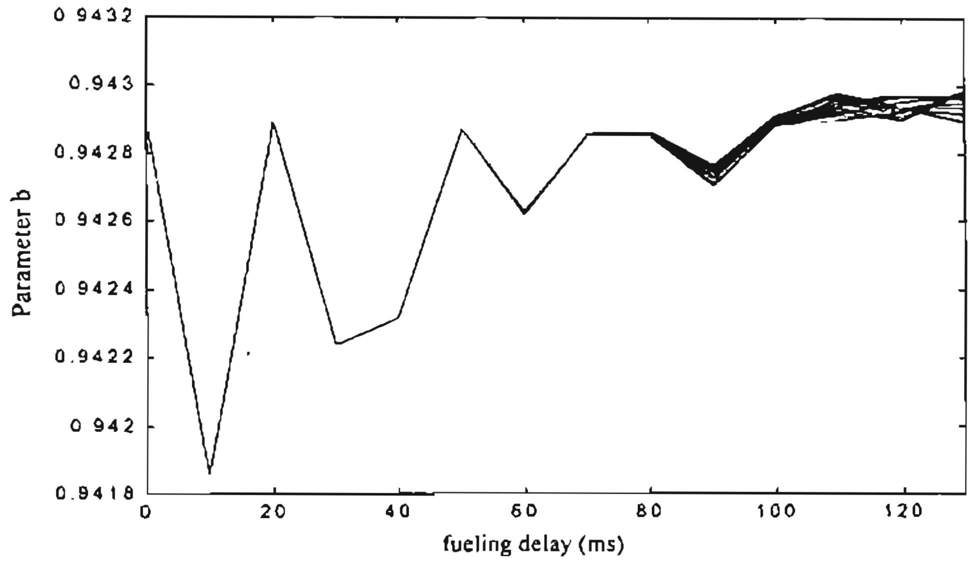


Figure 4.8: *Parameter* b *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 100ms$).

For the next case we changed the sample time to $T = 50\ ms$ maintaining the same transfer function structure shown in Eq. (4.28). From Figure 4.9 we can see for the new sample time that the parameter a_2 has an initial lower value changing to a value close to zero for $\tau = 50\ ms$. However we note negatives values after $\tau = 100\ ms$. The parameter a_1 has its maximum value at $\tau = 50\ ms$ as seen in Figure 4.10. The parameter a_0 increases after $\tau = 50\ ms$ but continues increasing after $\tau = 100\ ms$ as shown in Figure 4.11. From Figure 4.12 we can see that the parameter b is close to the estimated value of 0.9711 but blows up after $\tau = 100\ ms$. The values after fueling delays of $100\ ms$ for a_2, a_0 and b are due to the lack of an additional term that represents fueling delays greater than $100\ ms$.

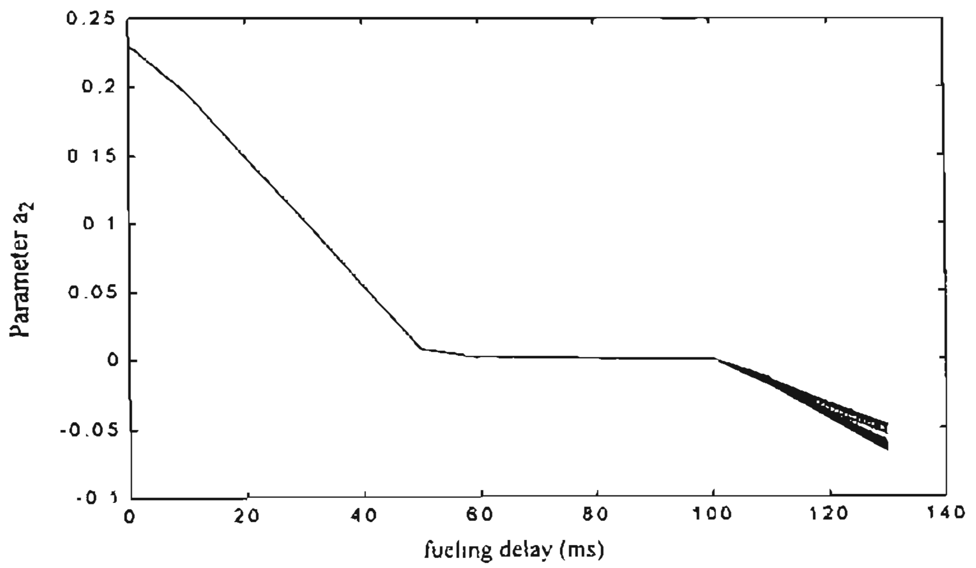


Figure 4.9: *Parameter a_2 for $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ versus fueling delay ($T = 50ms$).*

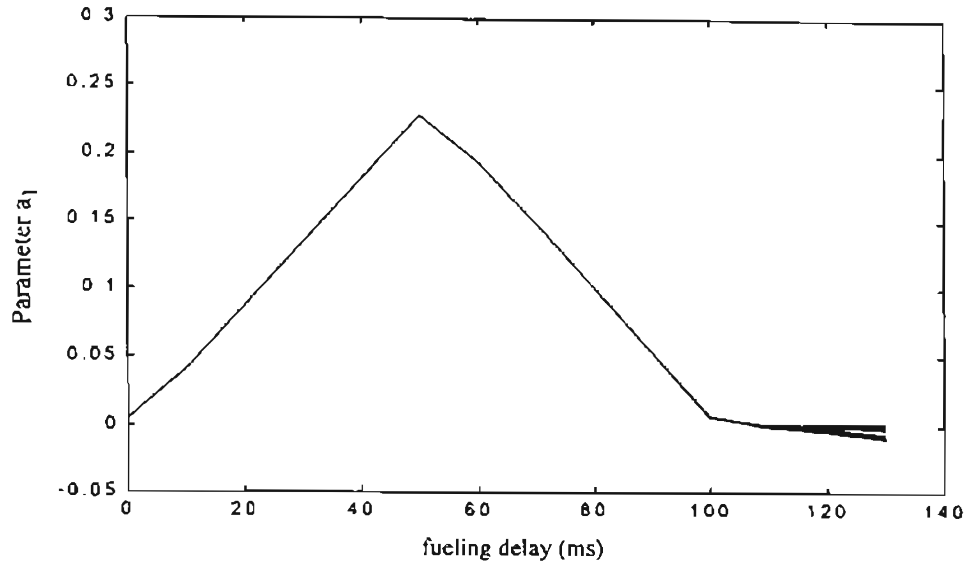


Figure 4.10: *Parameter* a_1 *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 50ms$).

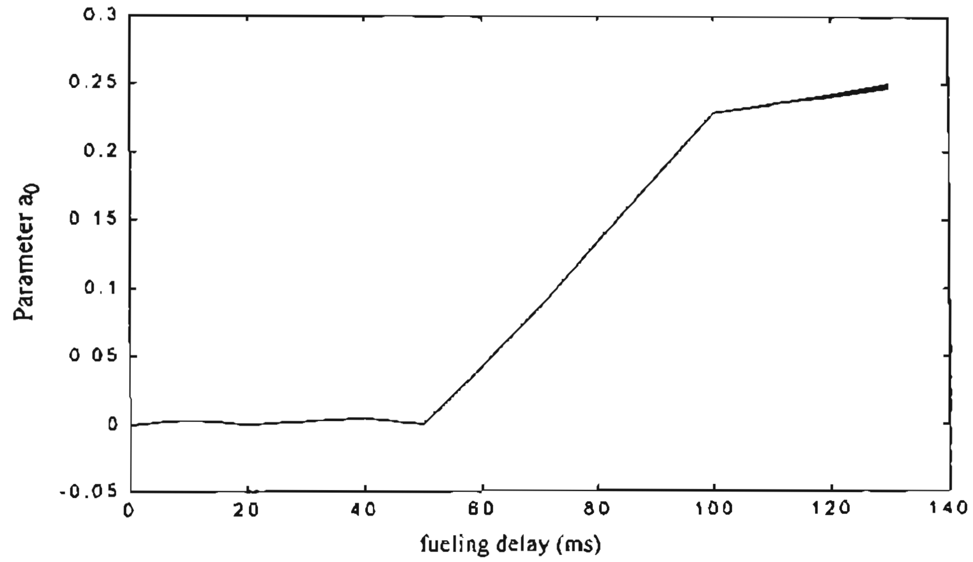


Figure 4.11: *Parameter* a_0 *for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay* ($T = 50ms$).

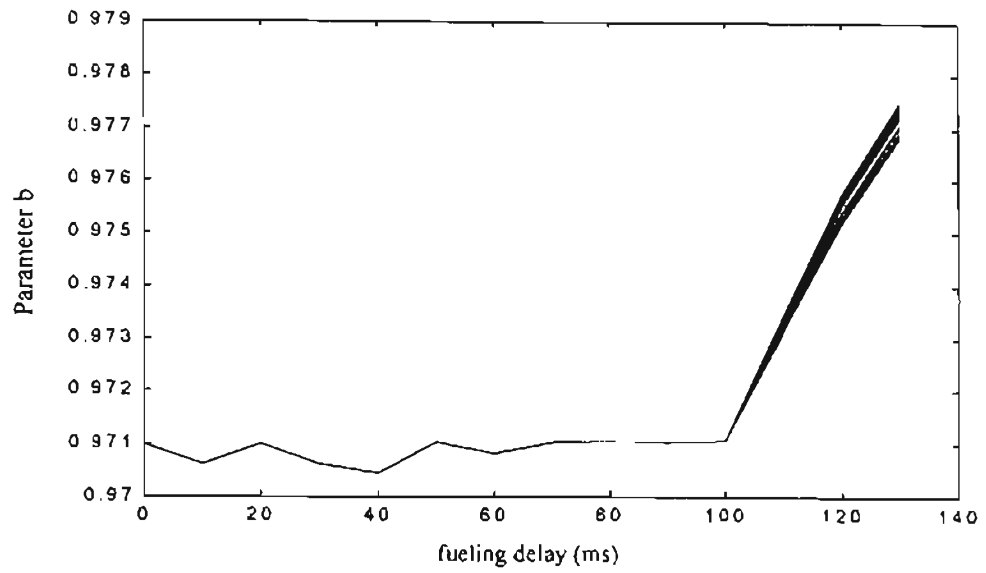


Figure 4.12: *Parameter b for* $\frac{a_2z^2 + a_1z + a_0}{z^3 - bz^2}$ *versus fueling delay (T = 50ms).*

For our last case we decided to increment the system order for the engine model:

$$\frac{N(z)}{f(z)} = \frac{a_3z^3 + a_2z^2 + a_1z + a_0}{z^4 - bz^3} \tag{4.29}$$

using the same sample time $T = 50\ ms$. From Figure 4.13 to Figure 4.16 we can see that the parameters a_3, a_2, a_1 and a_0 reach their maximum value for different engine fueling delays τ that were proportional to the sample time T. The parameter b changes between values 0.9704 and 0.9712 that were close to the calculated estimate of 0.9711. In Figure 4.17 we can see that b has variations for different loads after $T = 100\ ms$.

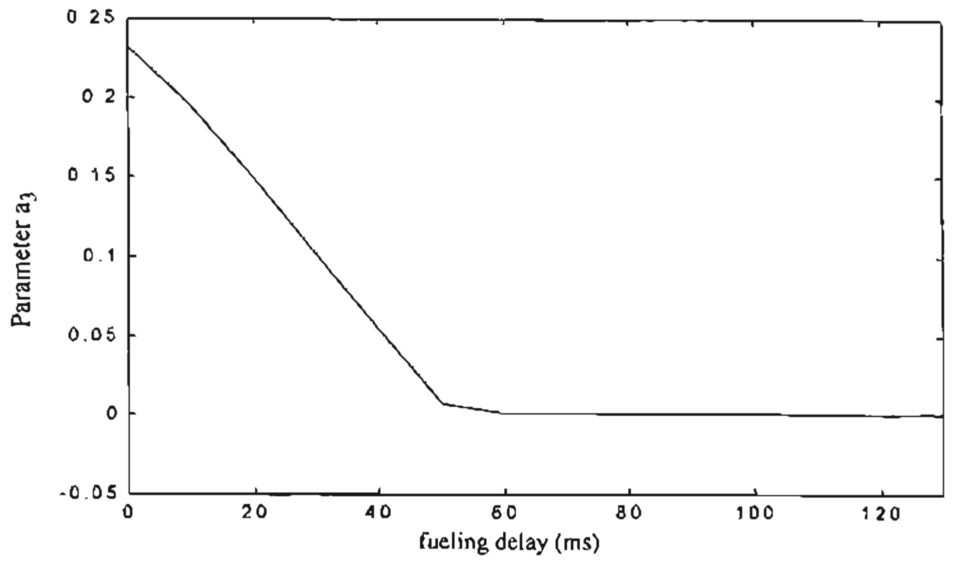


Figure 4.13: *Parameter* a_3 *for* $\dfrac{a_3z^3 + a_2z^2 + a_1z + a_0}{z^4 - bz^3}$ *versus fueling delay*

$(T = 50ms)$.

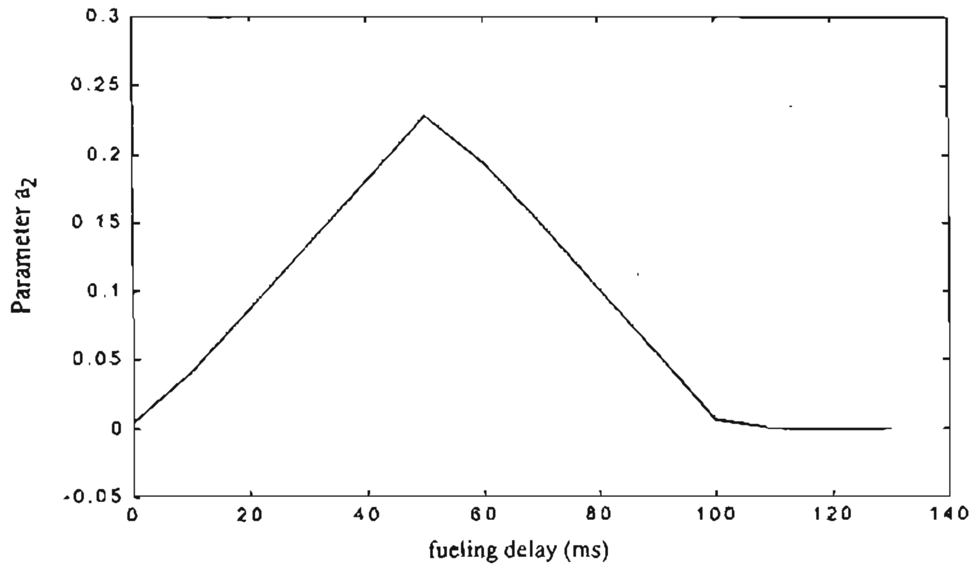


Figure 4.14: *Parameter* a_2 *for* $\dfrac{a_3z^3 + a_2z^2 + a_1z + a_0}{z^4 - bz^3}$ *versus fueling delay*

$(T = 50ms)$.

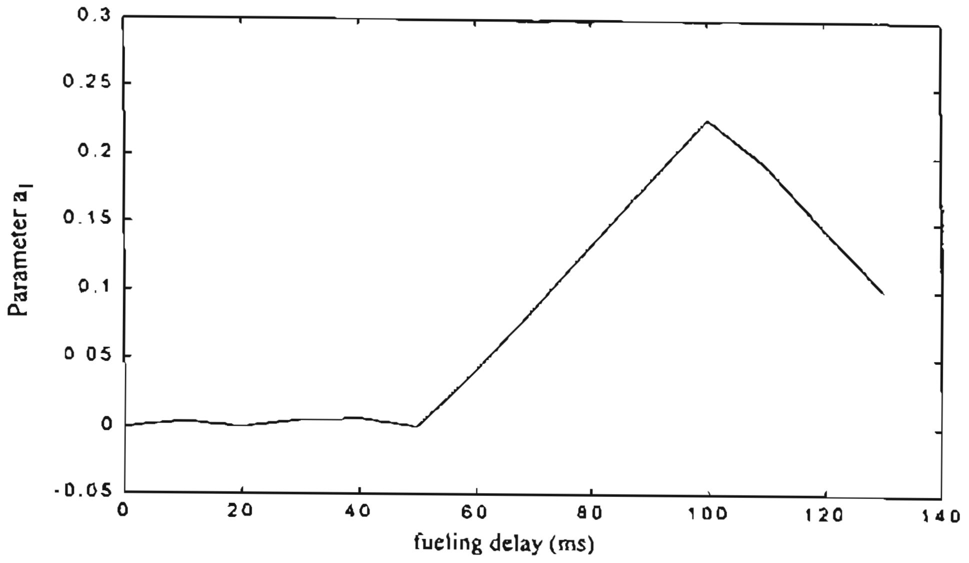


Figure 4.15: *Parameter* a_1 *for* $\dfrac{a_3z^3 + a_2z^2 + a_1z + a_0}{z^4 - bz^3}$ *versus fueling delay*

(T = 50ms).

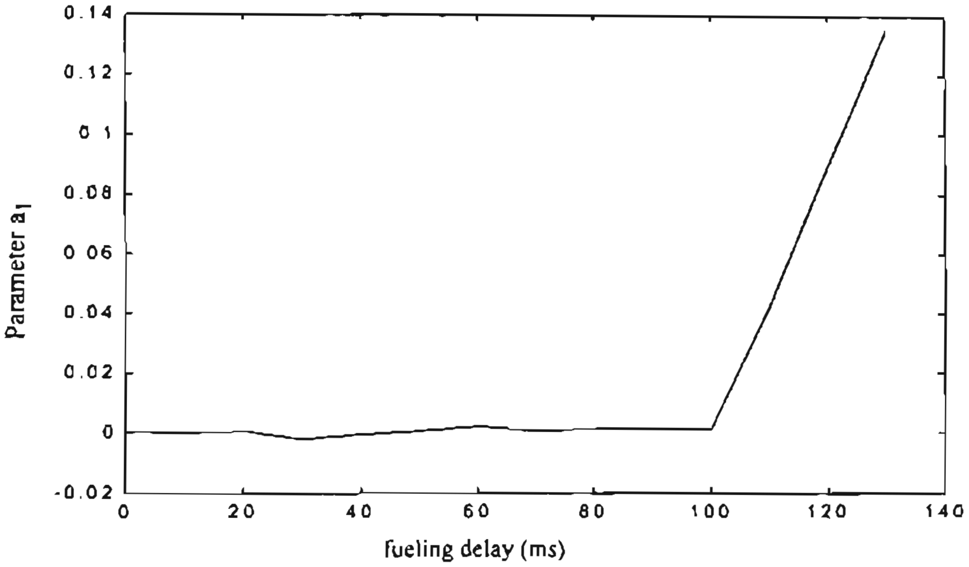


Figure 4.16: *Parameter* a_0 *for* $\dfrac{a_3z^3 + a_2z^2 + a_1z + a_0}{z^4 - bz^3}$ *versus fueling delay (T = 50ms).*

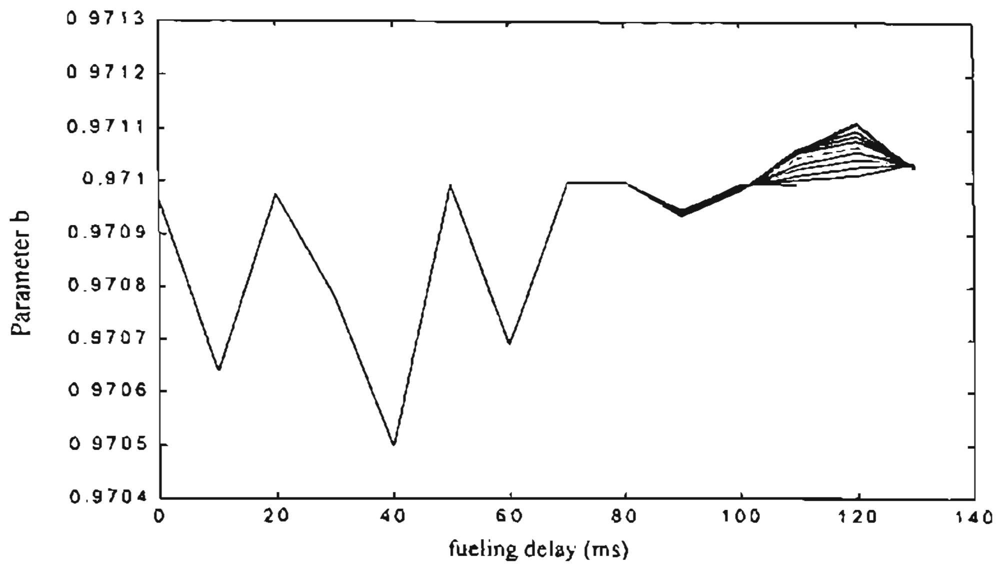


Figure 4.17: *Parameter b for* $\frac{a_3 z^3 + a_2 z^2 + a_1 z + a_0}{z^4 - b z^3}$ *versus fueling delay* ($T = 50ms$).

4.5. Controller design.

This section presents an off-line design of a controller that could be used to control the speed engine. The objective is develop some constraints that could be used in the self-tuning controller in the next section.

For the controller design we select the following transfer function:

$$\frac{S}{R} = \frac{S_1 z^2 + S_2 z + S_3}{z^2 + R_2 z + R_3} \tag{4.30}$$

If we combine the controller transfer function of Eq. (4.30) with the system transfer function of Eq. (4.29) to obtain the input-output relation of the closed loop system described in Eq. (4.1) we obtain:

$$B \cdot R + A \cdot S = (z^4 - bz^3)(z^2 + R_2 z + R_3) + (a_3 z^3 + a_2 z^2 + a_1 z + a_0)(S_1 z^2 + S_2 z + S_3)$$

$$B \cdot R + A \cdot S = z^6 + z^5(R_2 - b + a_3 S_1) + z^4(R_3 - bR_2 + a_3 S_2 + a_2 S_1) + z^3(-bR_3 + a_3 S_3 + a_2 S_2 + a_1 S_1) + z^2(a_2 S_3 + a_1 S_2 + a_0 S_1) + z(a_1 S_3 + a_0 S_2) + (a_0 S_3) \tag{4.31}$$

If we select our closed loop characteristic equation as:

$$Ac = z^6 + z^5 a_{c5} + z^4 a_{c4} + z^3 a_{c3} + z^2 a_{c2} + z a_{c1} + a_{c0} \tag{4.32}$$

then we could define the following relation:

$$\begin{bmatrix} 1 & 0 & a_3 & 0 & 0 \\ -b & 1 & a_2 & a_3 & 0 \\ 0 & -b & a_1 & a_2 & a_3 \\ 0 & 0 & a_0 & a_1 & a_2 \\ 0 & 0 & 0 & a_0 & a_1 \\ 0 & 0 & 0 & 0 & a_0 \end{bmatrix} \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} a_{c5} + b \\ a_{c4} \\ a_{c3} \\ a_{c2} \\ a_{c1} \\ a_{c0} \end{bmatrix} \tag{4.33}$$

Initially we could solve this relation by applying the relation $RS = (A^T A)^{-1} A^T a_c$. However we must include some constraints to obtain the desired system response. We can find constraints if we apply the final value theorem to Eq. (4.1). We want the final value of the engine speed with respect to the reference speed to be close to one. We also want the final value of the engine speed with respect to the engine load to be close to zero. From both conditions we can define:

$$\lim_{z \to 1}\left(\frac{A \cdot S}{B \cdot R + A \cdot S}\right) \approx 1 = errorgain \tag{4.34}$$

and

$$\lim_{z \to 1}\left(\frac{A \cdot R}{B \cdot R + A \cdot S}\right) \approx 0 = errorload \tag{4.35}$$

For practical purposes we could define a lower and upper bound for $errorgain$ as $Leg < errorgain < Ueg$, where Leg could be 0.9999999 and Ueg could be 1.00000001. A similar relation could be applied to $errorload$ as $Lel < errorload < Uel$, where Lel could be -0.00000001 and Uel could be 0.00000001. Solving for Eq. (4.34):

$$\begin{bmatrix} [(1-b)Leg] \\ [(1-b)Leg] \\ [(\sum a_i)(Leg-1)] \\ [(\sum a_i)(Leg-1)] \\ [(\sum a_i)(Leg-1)] \end{bmatrix}^T \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \le -Leg(1-b) \tag{4.36}$$

where $\sum a_i = a_3 + a_2 + a_1 + a_0$. For the upper limit of $errorgain$ we obtained:

$$\begin{bmatrix} [-(1-b)Ueg] \\ [-(1-b)Ueg] \\ [-(\sum a_i)(Ueg-1)] \\ [-(\sum a_i)(Ueg-1)] \\ [-(\sum a_i)(Ueg-1)] \end{bmatrix}^T \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \le Ueg(1-b) \tag{4.37}$$

Solving for Eq. (4.35):

$$\begin{vmatrix} [-(\sum a_i) + (1-b)Uel] \\ [-(\sum a_i) + (1-b)Uel] \\ [(\sum a_i)(Uel)] \\ [(\sum a_i)(Uel)] \\ [(\sum a_i)(Uel)] \end{vmatrix}^T \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \le -Uel(1-b) + (\sum a_i) \tag{4.38}$$

$$\begin{vmatrix} [(\sum a_i) - (1-b)Lel] \\ [(\sum a_i) - (1-b)Lel] \\ [-(\sum a_i)(Lel)] \\ [-(\sum a_i)(Lel)] \\ [-(\sum a_i)(Lel)] \end{vmatrix}^T \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \le Lel(1-b) - (\sum a_i) \tag{4.39}$$

We must consider a case where:

$$\lim_{z \to 1} (R) = 0 \tag{4.40}$$

resulting in an equality for Eq. (4.34) and Eq. (4.35). If Eq. (4.40) is satisfied, this does not mean a final gain equal to one with respect the reference speed or a minimization in the influence of the external load. To avoid Eq. (4.40) we included the following condition:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} R_2 \\ R_3 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} \ne -1 \tag{4.41}$$

To solve the relations given by Eq. (4.33), Eq. (4.36), Eq. (4.37), Eq. (4.38), Eq. (4.39) and Eq. (4.41) we can use the Matlab function **conls** in the form:

$$result = conls(A, b, C, d) \tag{4.42}$$

where A and b correspond with the matrix and vector given in Eq. (4.33). C and d correspond with the matrices and vectors given from Eq. (4.36) to Eq. (4.39) and Eq. (4.41). This function solves the constrained linear least-squares problem:

$$\min_{x}\left(\frac{1}{2}\|Ax-b\|^2\right) \qquad \text{subject to} \qquad Cx \le d \tag{4.43}$$

From Eq. (4.43) we notice that the condition described by Eq. (4.42) could not be reached. Therefore we replaced Eq. (4.42) by the following expression:

$$\begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}R_2\\R_3\\S_1\\S_2\\S_3\end{bmatrix} \le -1.000000001 \tag{4.44}$$

that allow us to find an expression close to -1. Another possibility could be:

$$\begin{bmatrix}-1 & -1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}R_2\\R_3\\S_1\\S_2\\S_3\end{bmatrix} \le 0.999999999 \tag{4.45}$$

We must define the desired transfer function (desired closed loop poles) to be used in Eq. (4.42). For different experiments we found that fixing a transfer function generally originates an approximation that generally has one or more of the poles outside of the unit circle. This condition produces an undesired response for the system. To reduce this problem we first identified the closed loop transfer function for the original controller. From that transfer function we determined if the complex poles were predominant with respect the real poles. If that condition occurs then we defined the least dominant real pole with the same size of the real part of the dominant complex pole. After that the complex poles were reduced in size.

For example in the case of a diesel engine with delay $\tau = 110\ ms$ we identified the system transfer function as:

$$\frac{N(z)}{f(z)} = \frac{0.0005z^3 - 0.0012z^2 + 0.1930z + 0.0428}{z^4 - 0.9711z^3} \tag{4.46}$$

where the sample time was $T = 50\ ms$ and the average load 150 lb-ft. For the original PID controller:

$$\frac{S}{R} = \frac{2.05z^2 - 2.575z + 0.5437}{z^2 - 1.25z + 0.25} \tag{4.47}$$

we obtained the closed loop poles:

0.9879

0.7940 + 0.4972i

0.7940 - 0.4972i

-0.3130 + 0.0378i

-0.3130 - 0.0378i

0.2701

We applied different combinations in the reduction of real poles and complex poles. The combination which reduced the mean square error was by reducing the real poles size by 0.8 and the complex poles by 0.98. For that reduction the desired new poles must be:

0.2701

-0.3067 - 0.0371i

-0.3067 + 0.0371i

0.7782 - 0.4873i

0.7782 + 0.4873i

0.7903

By using the function **conls** we obtained final poles at:

0.9890

0.6838 + 0.3588i

0.6838 - 0.3588i

-0.3087 + 0.0201i

-0.3087 - 0.0201i

0.3318

with $errorgain = 1.00000001$ and $errorload = -8.1349x10^{-8}$. The controller obtained was:

$$\frac{S}{R} = \frac{1.4893z^2-1.9144z + 0.4374}{z^2-1.1005z + 0.1005} \tag{4.48}$$

If we use the condition given by Eq. (4.45) we obtained the controller transfer function:

$$\frac{S}{R} = \frac{1.4884z^2-1.9150z + 0.4366}{z^2-1.1005z + 0.1005} \tag{4.49}$$

We simulated the system with the original PID controller (Eq. (4.47)) and both controllers (Eq. (4.48) and Eq. (4.49)) for the nominal load of 150 lb-ft and a second load of 100 lb-ft. From Figure 4.18 we can see that the system response for the new controllers has lower overshoot and is immune to load variations. With the original PID controller we have a higher overshoot and more variation in the response due to the engine load. For this system we tested various combinations of pole locations. We found that the lower mean squared error was obtained for a reduction of 80 % in the dominant real pole and 98 % in the complex poles. The movement in the pole location is shown in Figure 4.19.

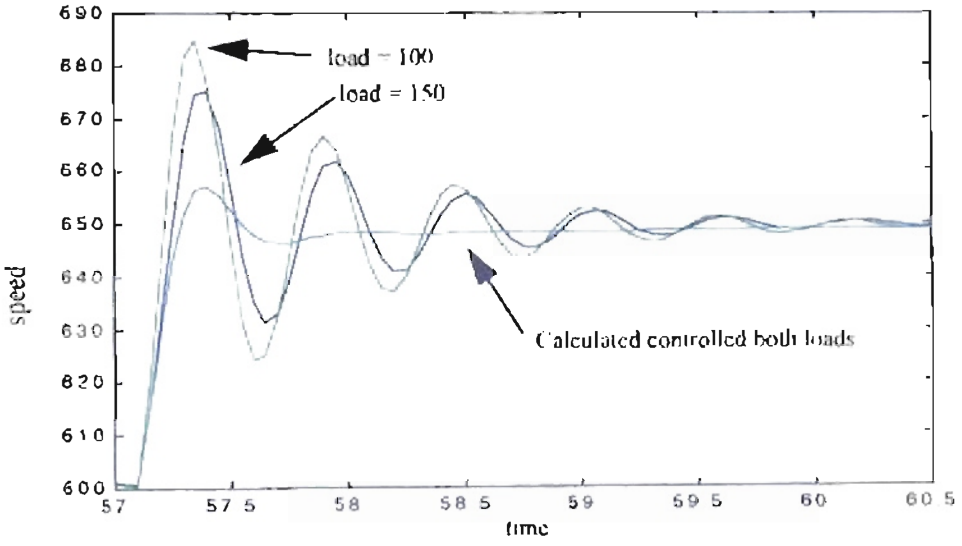


Figure 4.18: *Engine response for different controllers and loads.*

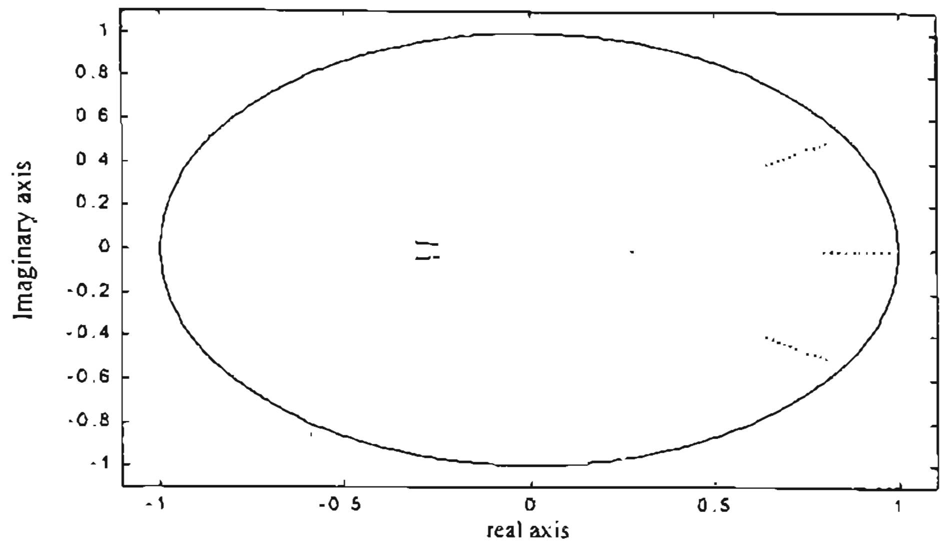


Figure 4.19: *Final pole location for different values in the magnitude reduction of original real and complex poles.*

4.6. Adaptive control.

In this section we will try to use the results of the previous section in the design of an adaptive controller for the engine. As shown in Figure 4.1, the adaptive controller has two blocks. A first block is dedicated to estimate the parameters of the engine as described by Eq. (4.29). Using those parameters, the controller is defined in a second block. The controller design block will adjust the controller parameters to achieve a desired set of pole locations. We will use the pole locations that were developed in the previous section.

The identification block was performed by using Eq. (4.26) and Eq. (4.27):

$$P_{k+1} = \frac{1}{\lambda}[P_k - \alpha_k P_k \psi_{k+1} \psi_{k+1}^T P_k]$$

$$\hat{\theta}(k+1) = \hat{\theta}(k) + P_k \psi_{k+1} \alpha_k [z(k+1) - \psi_{k+1}^T \hat{\theta}(k)]$$

For the identification block we used $\lambda = 0.99$. We tried with lower values, but the estimated parameters had too much variation. For initialization, we choose $P_0 = [H^T H]^{-1}$ and $\theta = zero$ for the first data set of points. The system started with the original PID controller, and after 1 second the adaptive process was started.

The first approach was defining a desired response that the closed loop system must follow. Using that definition, we found that the engine response was normally saturated at maximum or minimum speeds. The second approach consisted in reducing the size of the dominant real pole by a given percentage from the original controller. Results from last section suggested that we use a reduction in the complex poles of 98 % and a reduction in the real pole dominant of 80 %. The controller was obtained by solving the relations given by Eq. (4.33), Eq. (4.36), Eq. (4.37), Eq. (4.38), Eq. (4.39) and Eq. (4.41), using the Matlab function **conls**.

In Figure 4.20 and Table 4.2 we can see the different responses for variations in the size of the dominant real pole for an engine with fueling delay of 110 ms and two friction values. We note that the best response was obtained with a reduction of 80 %. Similar results are observed for an engine with fueling delay of 130 ms in Figure 4.21 and Table 4.3, except for the percent overshoot that was better with a reduction of 70 %. A special case was for a fueling delay of 80 ms. As seen in Figure 4.22, the maximum reduction was for 85 % of the dominant real pole. When we tried a larger reduction, the response of the resulting engine was saturated at zero. However, we noted better responses for 85 % and 90 % reduction for the percent overshoot. The mean square error has an small increment. In Figure 4.23 we can see the case for a fueling delay of 50 ms. Here we obtained a percent overshoot improvement for the dominant real pole reduced to 90 %. If we continue with the reduction we don't see further improvement and the mean square error increases.

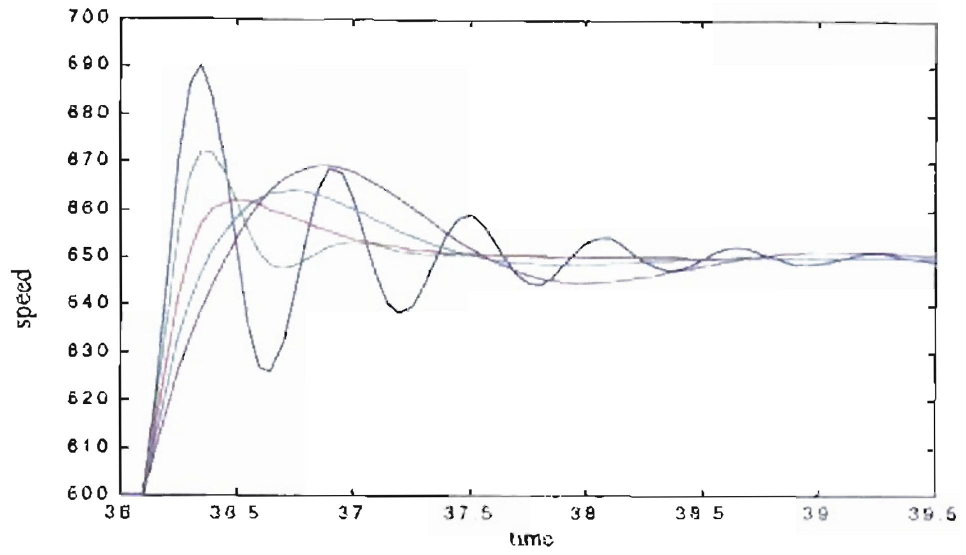


Figure 4.20: *Engine response for different size reduction of the real poles for fueling delay of 110 ms, engine inertia of 2 lb-ft-sec^2, friction b_f = 0.1229. Blue = Original System, Green = 90 %, Red = 80 %, Cyan = 70 %, Magenta = 60 %.*

Size reduction of the real poles	Mean square error		Percent overshoot	
	b_f = 0.1229	b_f = 0.01	b_f = 0.1229	b_f = 0.01
Original response	1529.52	1873.36	81.87	94.31
90 %	934.70 (61 %)	1072.66 (57 %)	45.49 (56 %)	57.85 (61 %)
80 %	920.91 (60 %)	1059.53 (56 %)	24.35 (30 %)	36.91 (39 %)
70 %	1088.70 (71 %)	1301.91 (69 %)	28.17 (34 %)	41.41 (44 %)
60 %	1438.96 (94 %)	1881.43 (101 %)	38.75 (47 %)	55.02 (58 %)

Table 4.2: *Engine response mean square error and percent overshoot for different size reduction of the real poles for fueling delay of 110 ms and engine inertia of 2 lb-ft-sec^2.*

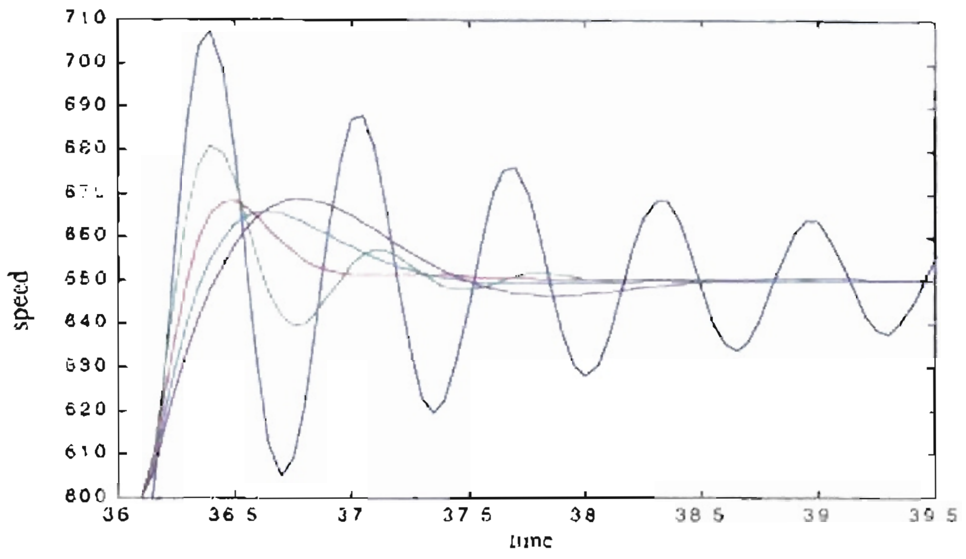


Figure 4.21: *Engine response for different size reduction of the real poles for fueling delay of 130 ms, engine inertia of 2 lb-ft-sec^2, friction b_l = 0.1229. Blue = Original System. Green = 90 %. Red = 80 %. Cyan = 70 %. Magenta = 60 %.*

Size reduction of the real poles	Mean square error		Percent overshoot	
	b_l = 0.1229	b_l = 0.01	b_l = 0.1229	b_l = 0.01
Original response	4965.88	6818.79	117.06	119.69
90 %	1292.81 (26 %)	1555.00 (23 %)	63.54 (54 %)	77.19 (64 %)
80 %	1084.30 (22 %)	1272.48 (19 %)	37.08 (32 %)	51.67 (43 %)
70 %	1118.29 (23 %)	1420.07 (21 %)	31.48 (27 %)	46.48 (39 %)
60 %	1394.30 (28 %)	1750.87 (26 %)	37.57 (32 %)	53.81 (45 %)

Table 4.3: *Engine response mean square error and percent overshoot for different size reduction of the real poles for fueling delay of 130 ms and engine inertia of 2 lb-ft-sec^2.*

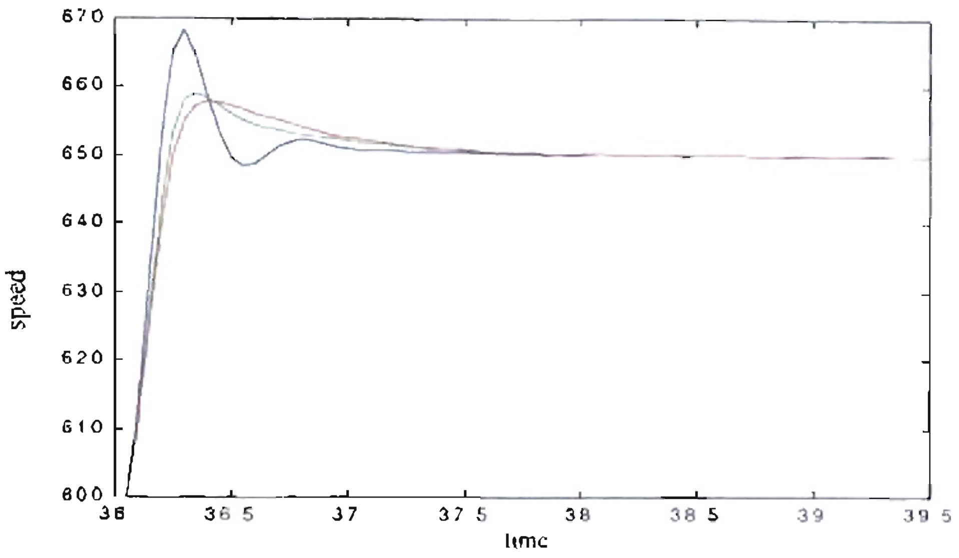


Figure 4.22: *Engine response for different size reduction of the real poles for fueling delay of 80 ms, engine inertia of 2 lb-ft-sec^2, friction b_f 0.1229. Blue Original System. Green 90 %, Red 85 %.*

Size reduction of real poles	Mean square error		Percent overshoot	
	$b_f = 0.1229$	$b_f = 0.01$	$b_f = 0.1229$	$b_f = 0.01$
Original response	660.74	725.81	34.76	43.79
90 %	661.98 (100 %)	728.07 (100 %)	17.44 (50 %)	26.32 (60 %)
85 %	689.61 (104 %)	765.93 (106 %)	16.00 (46 %)	25.40 (58 %)

Table 4.4: *Engine response mean square error and percent overshoot for different size reduction of the real poles for fueling delay of 80 ms and engine inertia of 2 lb-ft-sec^2*

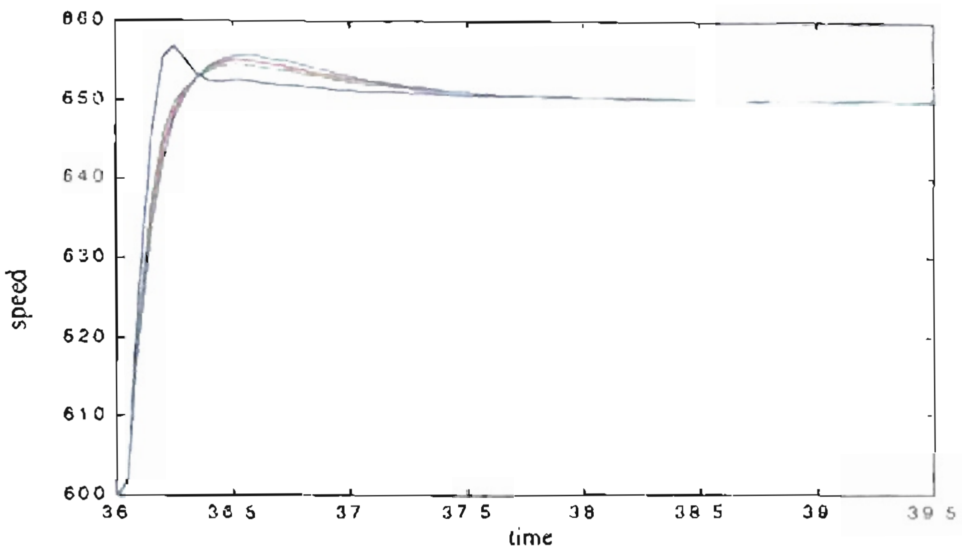


Figure 4.23: *Engine response for different size reduction of the real poles for fueling delay of 50 ms, engine inertia of 2 lb-ft-sec^2, friction b_f = 0.1229. Blue = Original System. Green = 90 %, Red = 85 %, Cyan = 80 %.*

Size reduction of the real poles	Mean square error		Percent overshoot	
	b_f = 0.1229	b_f = 0.01	b_f = 0.1229	b_f = 0.01
Original response	445.51	479.11	13.38	19.78
90 %	493.81 (111 %)	537.07 (112 %)	9.08 (68 %)	16.25 (82 %)
85 %	515.23 (116 %)	563.17 (118 %)	10.21 (76 %)	17.65 (89 %)
80 %	539.92 (121 %)	593.43 (124 %)	11.48 (86 %)	19.25 (93 %)

Table 4.5: *Engine response mean square error and percent overshoot for different size reduction of the real poles for fueling delay of 50 ms and engine inertia of 2 lb-ft-sec^2.*

The objective of this chapter was to introduce a standard adaptive controller with which to compare the reinforcement learning algorithms to be presented in later chapters. The self-tuning regulator was chosen as the base-line controller.

There are many variations of the self-tuning regulator. We used the pole-positioning STR. The first stage in the development of this STR is to choose a set of desired pole locations. If these are not chosen carefully, the resulting system may not be stable. Through experimentation, we found that the best approach was to start with the closed-loop locations of the base-line PID controller. We then identified the dominant poles and reduced them in magnitude by a specified percentage (a reduction of 80% provided the best performance for the engine speed control).

CHAPTER 5

GENETIC REINFORCEMENT LEARNING FOR DIESEL ENGINES CONTROL.

5.1. Introduction.

In this chapter we will describe the GENITOR algorithm for the optimization of the parameters of a diesel engine controller. We will explain the basics of that algorithm and how it can be used to adjust the controller parameters.

5.2. GENITOR Algorithm.

The GENITOR Algorithm was developed by Dr. Whitley and his students at Colorado State University and publications are available starting in 1988 (9, 23 to 32). Initially, GENITOR was an algorithm to solve binary genetic applications(32). After some updates, Whitley et. al proposed the GENITOR algorithm as an application using real numbers for training neural networks for reinforcement learning and "neurocontrol" applications in a term they called Genetic Reinforcement Learning (25).

Traditional Genetic Algorithms apply biologic ideas to the solution of a problem. We can encode a solution in a string, where each parameter solution is consider as a bit of that string. If we manipulate that string we can obtain new solutions based on the survival of the fittest. Researchers used manipulation methods related to chromosomal recombination, such as crossover, mutation, etc.

Normally the initial population is generated randomly. By random selection two members of that population are selected and we apply "crossover" to obtain a new member of the population or offspring. We will consider the parents **0101010001010101** and **yxyyxyyxyxyxyxyx**, where the bit representation (**0**, **1**, **x**, **y**) was choosen to recognize each parent. If we apply a crossover at the 4th bit of the parents, it means the recombination:

0101\ /010001010101⇒0101xyyxyxyxyxyx

yxyy\ /xyyxyxyxyxyx⇒yxyy010001010101

After the Crossover operation, we can perform the mutation operation, where some bits of the offspring are randomly selected and the values complemented. For example, if we select the bit 2, 7 and 10 of the first offspring we will obtain:

0101xyyxyxyxyxyx ⇒ **0001xyxxyxxxyxyx**

↑ ↑ ↑ ↑ ↑ ↑

An important feature from GENITOR to obtain an improvement in the quality of the population is the tendency to select the best parents more frequently. The difference with gradient search methods is that genetic algorithms will search randomly in all of the hyperplanes.

We can define a hyperplane that represents the binary encoding as seen in Figure 5.1. If we have a 3-bit string the search is performed in the upper hypercube of Figure 5.1. If we have a 4-bit string the search is made in the hypercube of four dimensions shown in the lower part of the same figure. The difference between the subspaces is the first bit, where 1 represents the inner cube and 0 the outer cube. The concept of implicit parallelism means an efficient search in those numerous hyperplanes. This feature permits the search of nonlinear functions without gradient calculation.

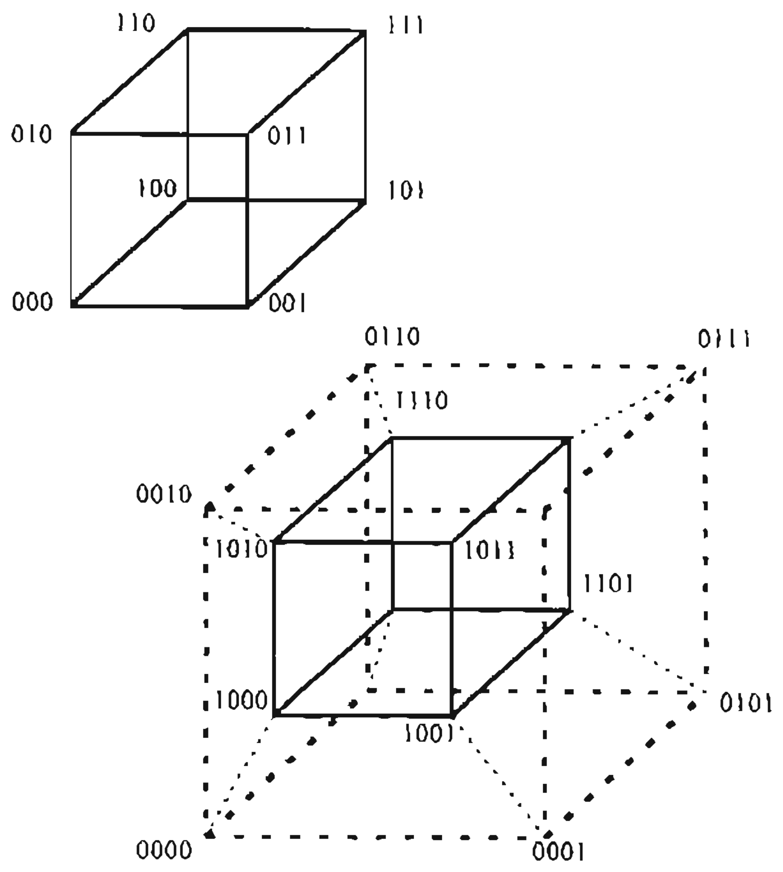


Figure 5.1: *A 3-dimensional and a 4-dimensional hypercube* (24).

The GENITOR algorithm, developed by Whitley and his students, generates an initial population of random strings. Each member of the population is evaluated and the population is sorted according to their fitness. Two parents are selected at random from the population. That selection process uses a bias ranking selection algorithm allowing a higher probability of selection to the best parents. The bias ranking selection is implemented with the relation:

$$parent = \frac{\mathrm{fix}(Populationsize(bias - \sqrt{bias^2 - 4(bias - 1)\mathrm{rand}}))}{2(bias - 1)} + 1$$

that represents the probability density function:

$$f(p) = bias - 2(bias - 1)p$$

where p is the parent ranking. We can see a plot of this function for Bias = 1.9 in Figure 5.2. We notice that parents with higher fitness (lower position in sorted population) have higher probability to be selected.

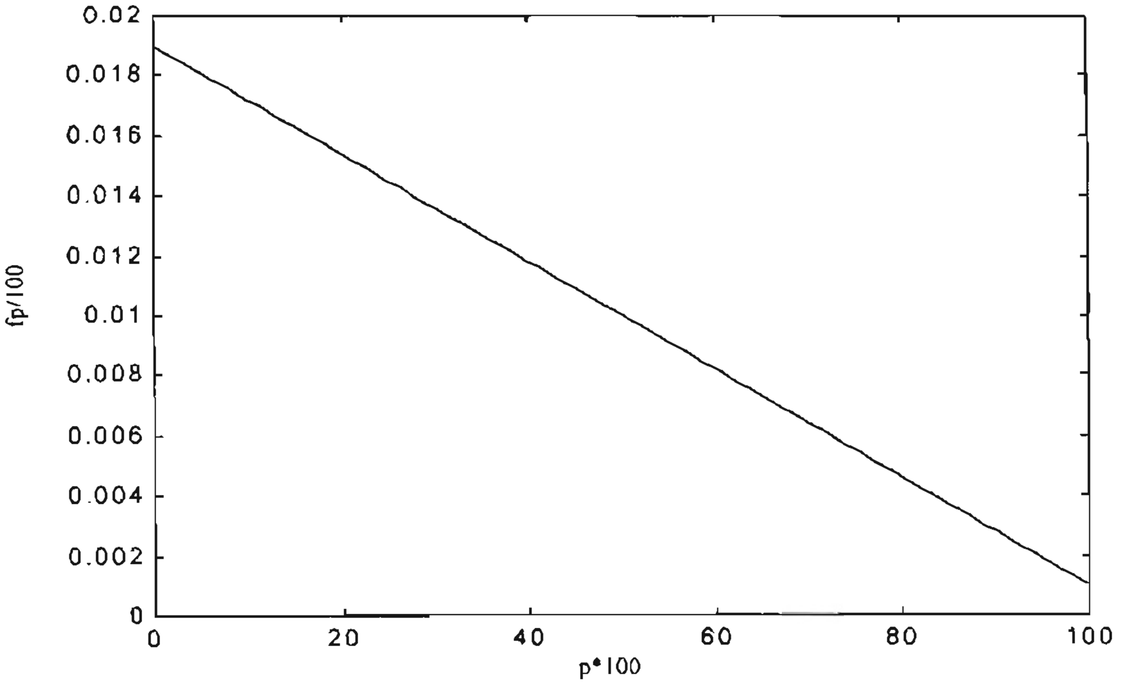


Figure 5.2: *Probability density function for bias = 1.9.*

A crossover process permits the recombination of the parents. From both offspring, we select one and discard the other. We evaluate the new offspring and it is placed according to its fitness, replacing the lowest ranked parent.

The improvements that Whitley and his students defined for the application of GENITOR in the training of Neural Networks are:

- 1.- The Neural Network problem is encoded as real-valued strings instead of binary strings.
- 2.- A different procedure for mutation is used. "Traditional genetic algorithms are largely driven by recombination, not mutation"(25).
- 3.- The algorithm uses an small population (e.g. 50 individuals) to reduce the exploration of dissimilar representations for the same neural network.

We can see the GENITOR implementation in Figure 5.3.

1. Initialization Phase.

- Set all the weights in the network to a random value between ± 2.5
- Set one allele representing the probability of crossover to a random value between 0 and 1.
- Evaluate each individual and sort the population according to the fitness.

2. Iteration phase.

- Select two individuals according to relative fitness using linear-bias selection.
- Crossover with probability determined by the crossover probability allele of the string selected as parent 1; otherwise perform mutation on parent 1.
- The offspring always inherits the crossover probability of parent 1. If parent has higher fitness than the offspring, increment the offspring crossover probability by a factor of 0.10 (to maximum 0.95); otherwise decrease the crossover probability by a factor of 0.10 (to minimum 0.05).
- Evaluate new offspring and insert in the population according to fitness.
- Continue "iteration" until error is acceptable or MAX-ITERATIONS = True.

Operator.

Mutation: Mutate all weights on the first selected individual by adding a random value with range ± 10.0.

Crossover: Perform no crossover if the parents differ by two or fewer alleles. Otherwise, recombine the strings one-point crossover between the first and the last positions at which the parents have different weight values.

Figure 5.3: *Original GENITOR algorithm (D. Whitley, et. al.)* $_{(25)}$.

From Figure 5.3 we can see that one allele is the probability of crossover. As the algorithm converges, the probability of crossover decreases. We can only perform crossover or mutation for a new offspring. Also, the mutation operator creates a new random offspring near the selected parent.

5.3. Model Description.

In this section we will briefly review the basic diesel engine model and baseline PID controller which were discussed in Chapter 3. The simplified engine diesel control model is described from Figure 3.1 to Figure 3.4.

With the model of the Figure 3.4 we made a simulation with constant load of 150 ft-lb. The speed engine was changed between 600 rpm and 650 rpm every 5 seconds for a total time of 20 seconds. We can see in Figure 5.4 and Figure 5.5 how the PID controller changes the engine speed between 600 rpm and 650 rpm, and the fueling is maintained between 0 and 300 $mm^3/stroke$. We will make the future simulations based on this model for the engine.

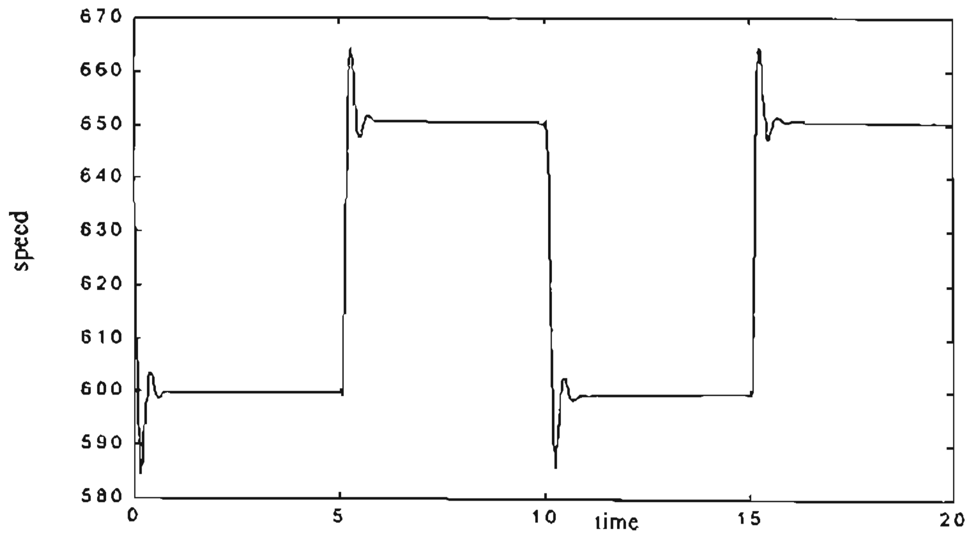


Figure 5.4: *Original engine response for Kp=2, Ki=0.5, Kd=0.05, a=15.*

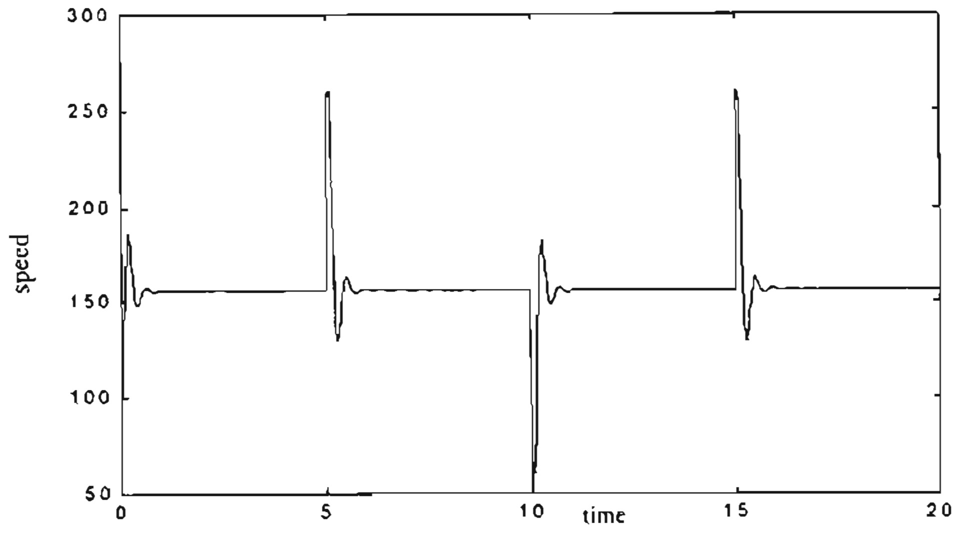


Figure 5.5: *Original fueling response for Kp=2, Ki=0.5, Kd=0.05, a=15.*

5.4. Genetic Reinforcement Learning applied to PID controller.

We applied Genetic Reinforcement Learning to the system shown in Figure 3.4. The simulation will run with constant load of 150 ft-lb changing speed between 600 rpm and 650 rpm every 5 seconds, for a total simulation time of 20 seconds. We defined as fitness function the mean square error of the desired speed response. This parameter is related to the rise time for the engine speed. For the original PID controller we have the responses of Figure 5.4 and Figure 5.5. For $a = 10$ and $a = 20$ we have a mean square error of 1000.34 and 999.34, respectively. We want to minimize the mean square error and indirectly the rise time and the tracking error. The initial population used for training was set around the basic PID parameters and $a = 15$.

5.5. Initial results.

We can see the training results in Table 5.1. For each experiment we started the training with random values around the basic PID controller parameters. We can see an improvement in the responses due to the PID controller based on the mean square error. The mean square error was calculated by direct integration of the square error in the simulink model. We obtained approximate improvements for the mean square error from 3.63 % to 10.00 %. From the two initial rows of Table 5.1, we notice that the best results, for the same number of epochs, were obtained for the smaller population. After increasing the number of epochs to 30000, we obtained best results for the population of 50, as shown in the last row of Table 5.1, Figure 5.9 and Figure 5.10. We can see an improvement with longer training, but good results can be obtained after a few epochs.

Table 5.1: *Controller's results for mean square error based fitness.*

Best Fitness $(error)^2$.	**% improvement**	**Pop. Size**	**Resulting Parameters**	**Notes**

920.64	7.87	5	Kp=1.7718
			Ki= 0.0788
			Kd=0.6835
			a = 20.8593
				Random initial conditions around basic PID controller. 3000 epochs.

963.01	3.63	50	Kp=1.8662
			Ki=0.0117
			Kd=0.1455
			a = 17.9139
				Random initial conditions around basic PID controller. 3000 epochs.

899.38	10.00	50	Kp=1.7975
			Ki=0.1417
			Kd=1.0868
			a = 25.9656
				Random initial conditions around basic PID controller. 30000 epochs.

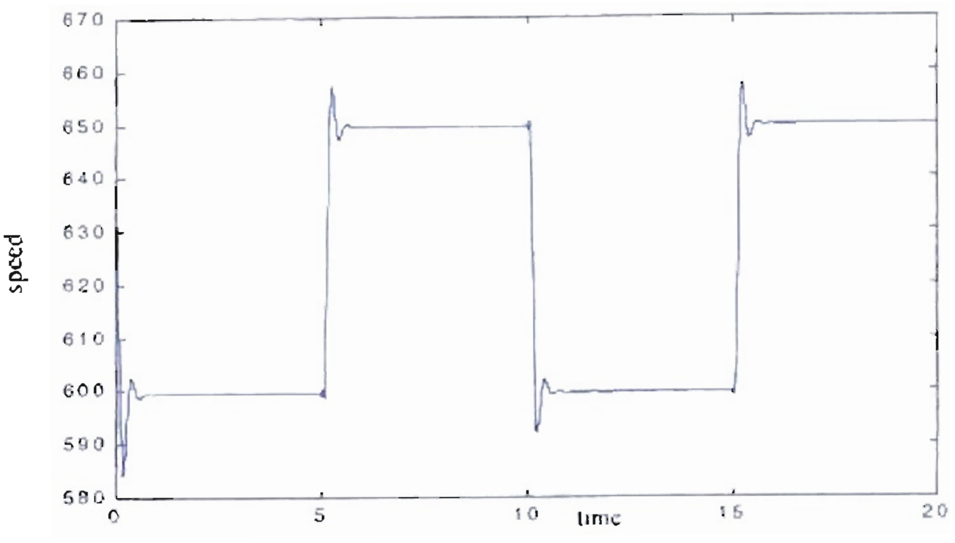


Figure 5.6: *Optimal engine response for training with population 5, epochs 3000, resulting: Kp 1.7718, Ki 0.0788, Kd 0.6835, α 20.8593.*

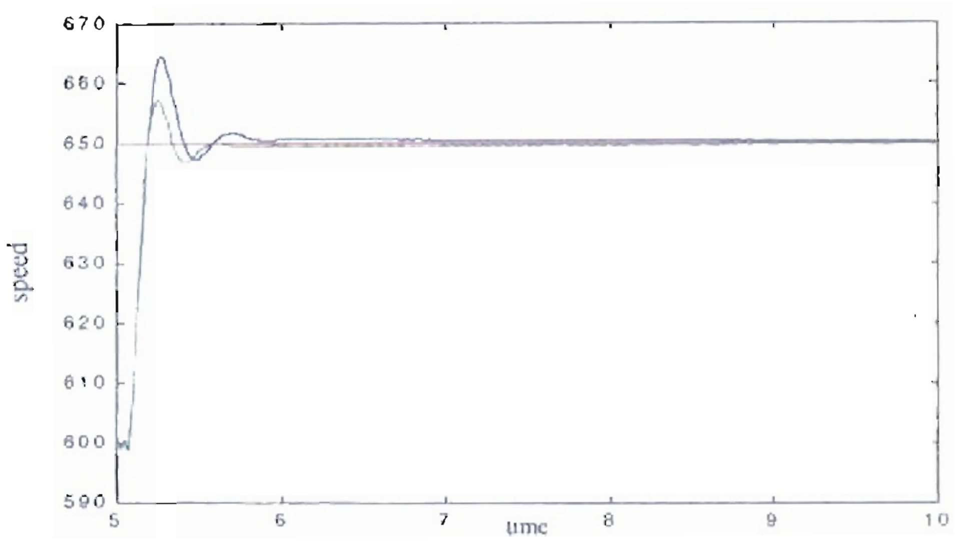


Figure 5.7: *Detail of transition from 600 to 650 rpm. Blue original PID parameters. Green Genitor optimized parameters. Population 5, epochs 3000.*

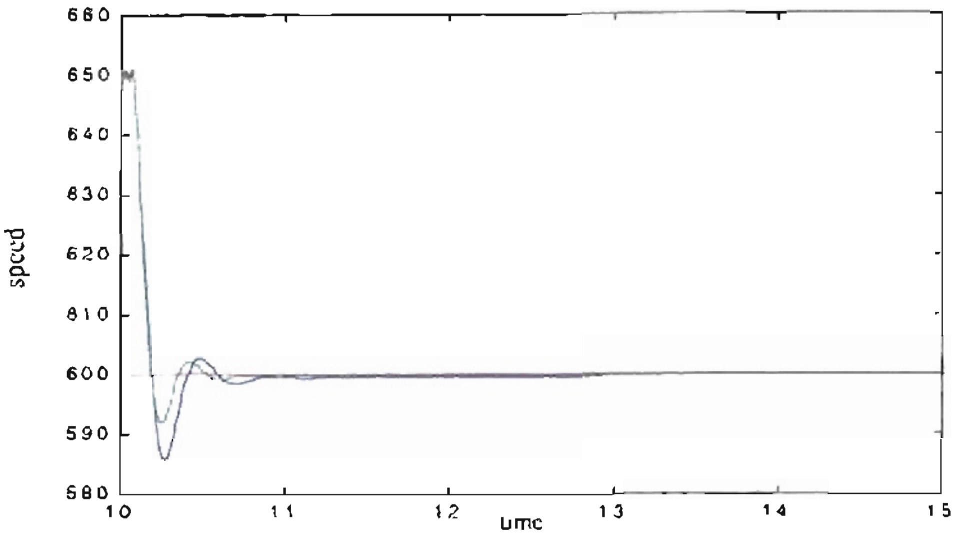


Figure 5.8: *Detail of transition from 650 to 600 rpm. Blue Original PID parameters, Green Genitor optimized parameters. Population 5, epochs 3000.*

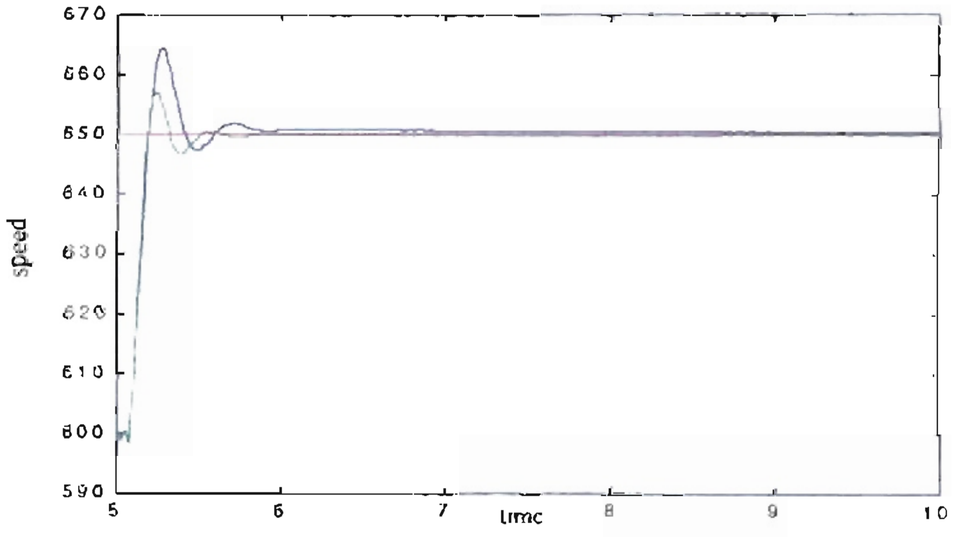


Figure 5.9: *Detail of transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters. Population 50, epochs 30000.*

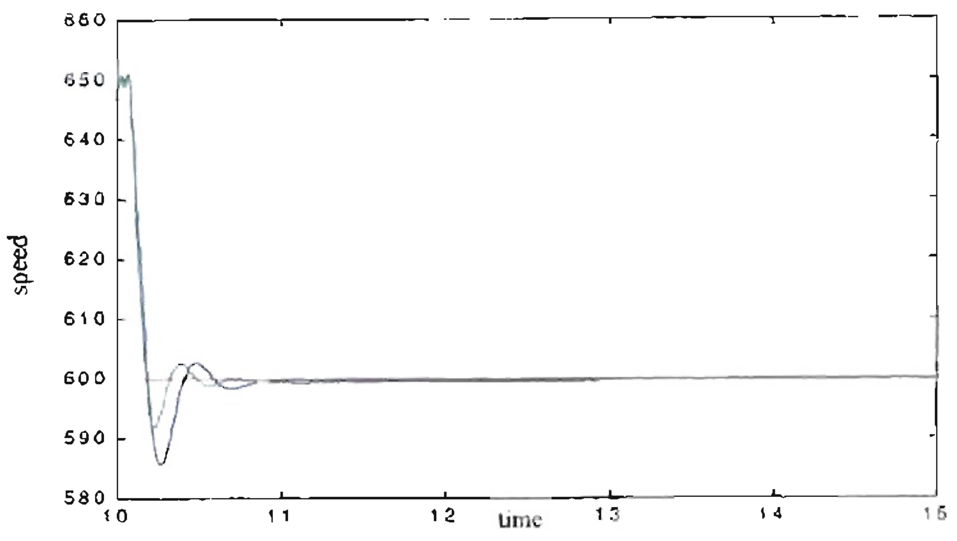


Figure 5.10: *Detail of transition from 650 to 600 rpm. Blue Original PID parameters, Green Genitor optimized parameters. Population 50, epochs 30000.*

5.6. GENITOR applied to analog PID controller.

5.6.1. Results for different engine inertia.

For the next case we repeated the training for different values of engine inertia *I* For each case the initial response changes according to the engine inertia value Optimization is needed to adjust the PID controller values to match the engine inertia We use the mean square error as our fitness value. To reduce the training time, we changed the training scheme to two speed transitions. The first transition is from 600 to 650 rpm at 1 second. The second transition is from 650 to 600 at 3.5 seconds. The engine and the PID controller are initialized with the engine conditions for 600 rpm. The simulation runs from 0 to 1 second without error measurements, then the training is started. After the first set of simulations we obtained negative values of integral gain. Those values were due to the limited simulation time for each speed, during which the tendency of an increased accumulation of integral error could not be noticed. To overcome this, we increased the simulation time to 8 seconds with the second transition from 650 to 600 rpm at 4.5 seconds. To ensure the correct values of the parameters for future training, we changed the mutation process in the GENITOR algorithm by accepting only positive values of the PID parameters.

For the mean square error as fitness we obtained the results shown in Table 5.2 for six different inertia values and two different population sizes. For very oscillatory engine responses we could reduce the mean square error to 20 % of its original value for inertia equal to 1 lb-ft-sec^2, as seen in first row of Table 5.2 and Figure 5.11. As inertia increases, the mean square error was reduced to 77 % of its original value for inertia equal to 1.4 lb-ft-sec^2 (see the second row of Table 5.2 and Figure 5.13). In the last four rows of Table 5.2 we can see mean square error reduction ranging from 91 % to 95 % of their original values for inertia between 1.8 lb-ft-sec^2 and 3 lb-ft-sec^2 (see also Figure 5.15, Figure 5.17, Figure 5.19 and Figure 5.21). If we compare the results related to the population size we noticed small differences in the results for 1500 epochs. For lower inertia values (see Figure 5.12 and Figure 5.14) the learning process is faster with smaller population. For the remaining cases the learning rate is similar for both of the populations used. A special case is for inertia 2.2 lb-ft-sec^2 (see Figure 5.18) where a good initial value in the population generated a better response for the case of higher population. We must remember that the population initialization and the recombination process are random in nature, therefore, for a specific experiment we could obtain results that are not consistent with overall trends.

Table 5.2: *Engine with different inertia. Controller's results for mean square error based fitness.*

Inertia lb-ft-sec^2.	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 50
1	2939.12	600.18 (20.42 %)	Kp=0.9651 Ki=0.5399 Kd=0.0481 a = 13.0029	592.71 (20.16 %)	Kp=0.9964 Ki=0.3470 Kd=0.0422 a = 16.5038
1.4	761.07	591.34 (77.69 %)	Kp=1.3283 Ki=0.4298 Kd=0.0549 a = 15.6957	597.15 (78.46 %)	Kp=1.4847 Ki=0.4459 Kd=0.0510 a = 13.4670
1.8	626.52	599.19 (95.63 %)	Kp=1.9398 Ki=0.5016 Kd=0.0543 a = 14.2661	597.16 (95.31 %)	Kp=1.9493 Ki=0.4472 Kd=0.0505 a = 14.8449
2.2	625.57	594.90 (95.09 %)	Kp=2.0588 Ki=0.4409 Kd=0.0583 a = 13.2913	592.23 (94.67 %)	Kp=2.0697 Ki=0.4436 Kd=0.0496 a = 15.4919
2.6	648.53	596.43 (91.96 %)	Kp=2.2861 Ki=0.4830 Kd=0.0494 a = 16.7868	606.42 (93.50 %)	Kp=2.1127 Ki=0.4758 Kd=0.0476 a = 14.6991
3	665.24	621.36 (93.40 %)	Kp=2.0800 Ki=0.4801 Kd=0.0473 a = 14.9868	618.70 (93.00 %)	Kp=2.0795 Ki=0.4097 Kd=0.0546 a = 14.9168

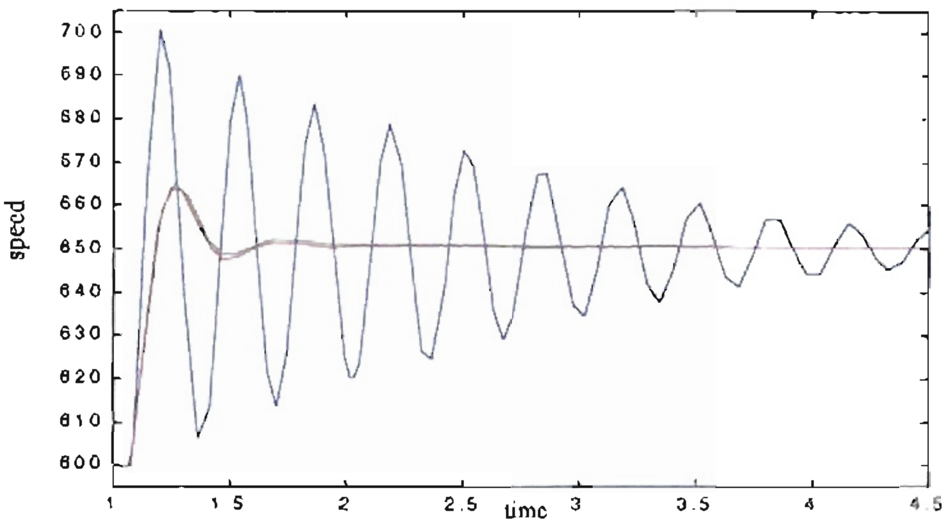


Figure 5.11: *Detail transition from 600 to 650 rpm. Blue – Original PID parameters, Green – Genitor optimized parameters (Population = 5), Red – Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 1 lb-ft-sec^2.*

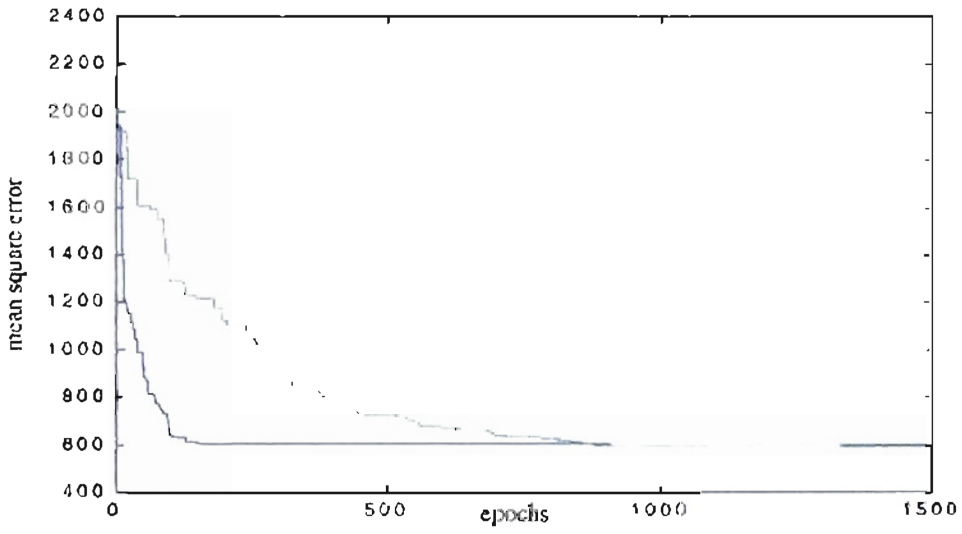


Figure 5.12: *Learning rate for engine inertia = 1 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

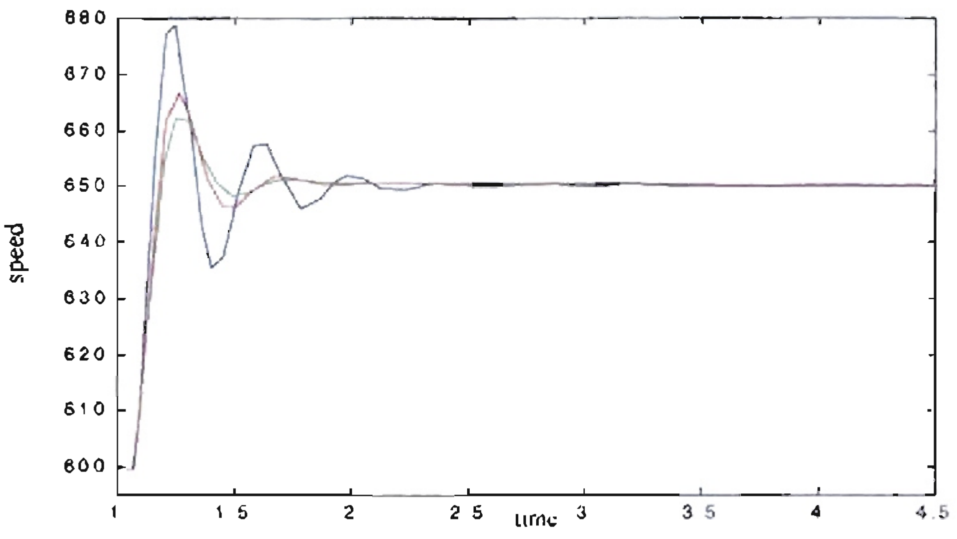


Figure 5.13: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 1.4 lb-ft-sec^2.*

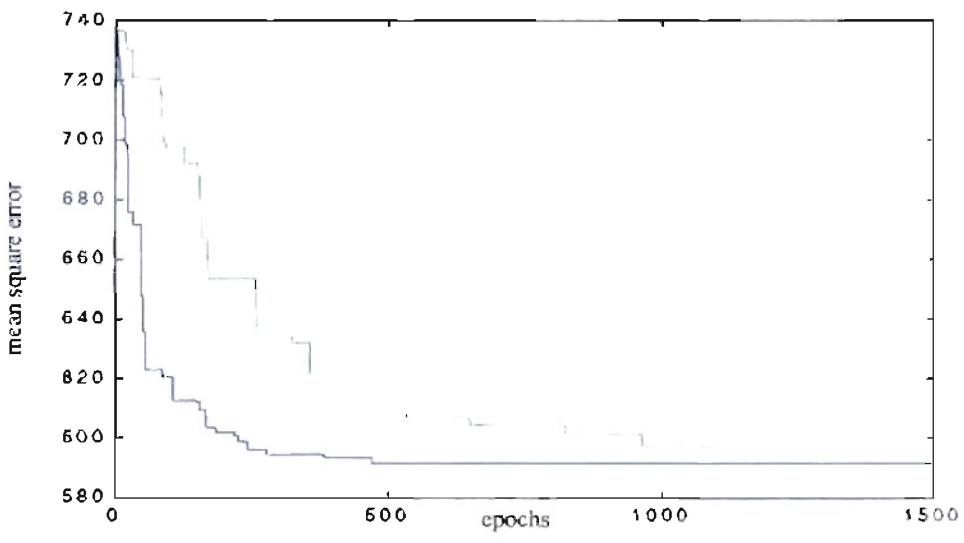


Figure 5.14: *Learning rate for engine inertia 1.4 lb-ft-sec^2. Blue: Population 5, Green: Population 50.*

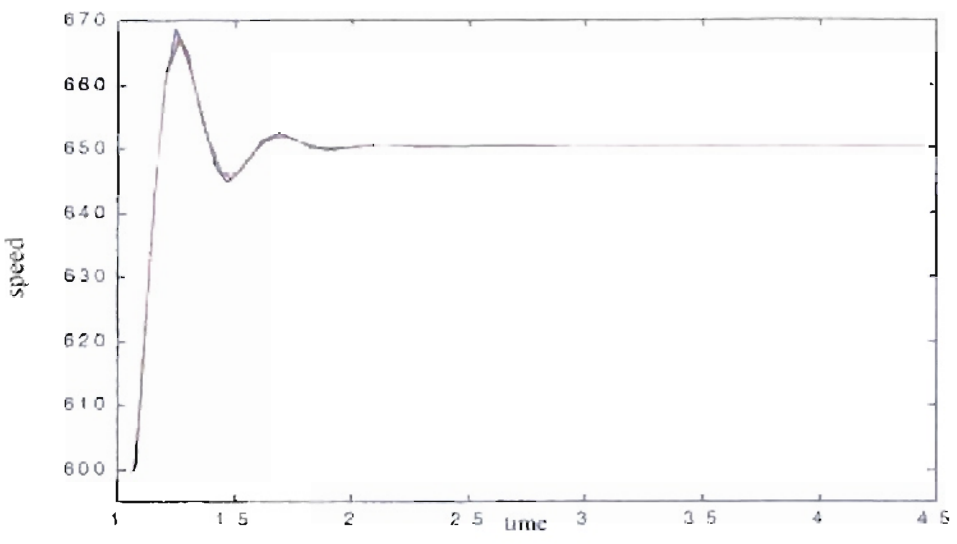


Figure 5.15: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500. Inertia 1.8 lb-ft-sec^2.*

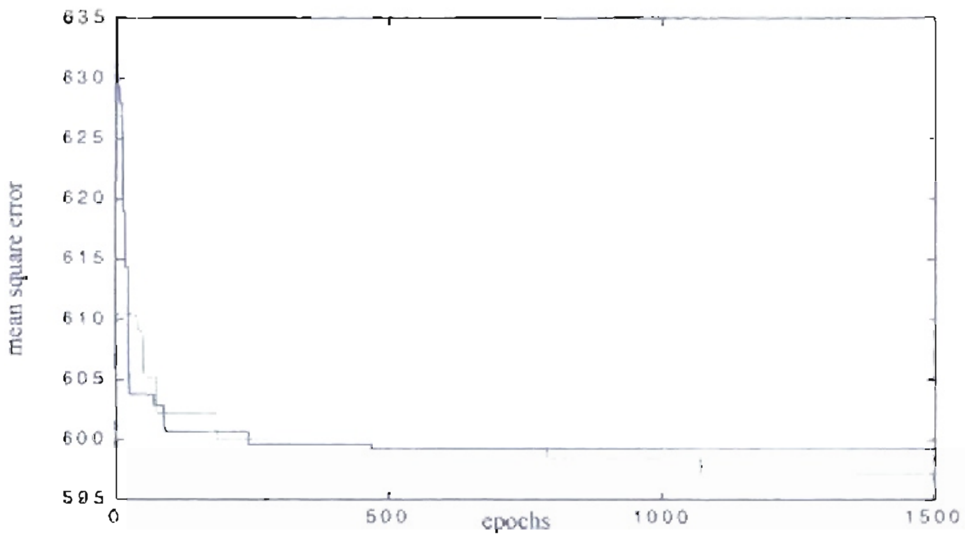


Figure 5.16: *Learning rate for engine inertia 1.8 lb-ft-sec^2. Blue: Population 5, Green: Population 50.*

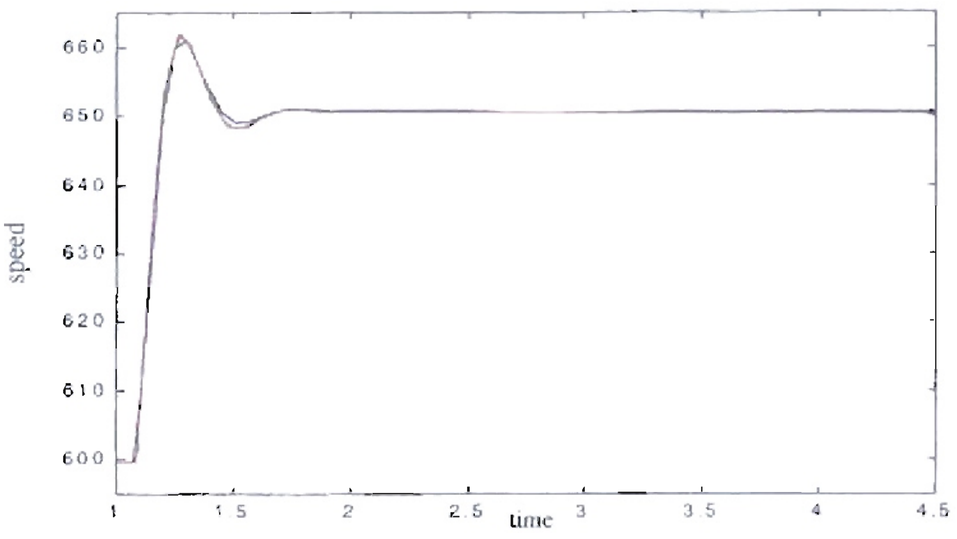


Figure 5.17: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 2.2 lb-ft-sec^2.*

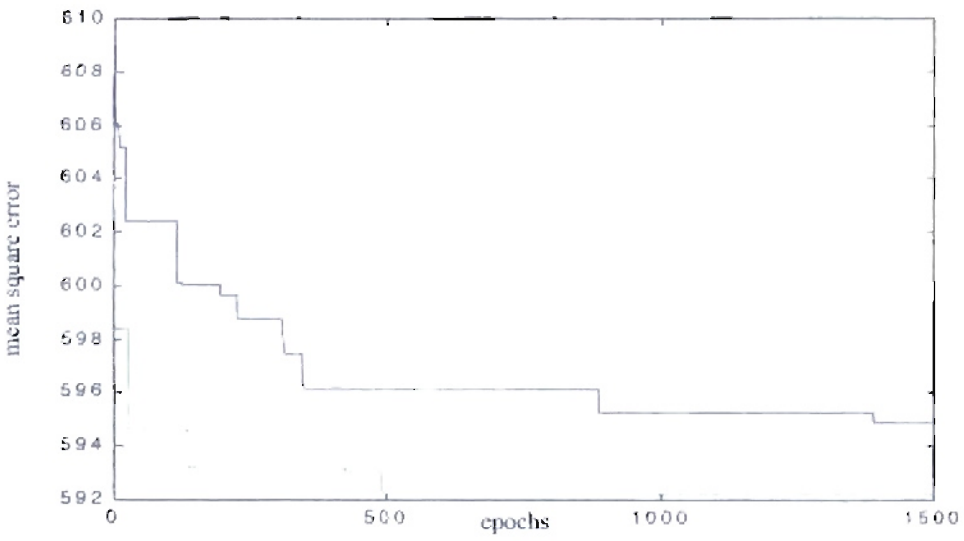


Figure 5.18: *Learning rate for engine inertia = 2.2 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

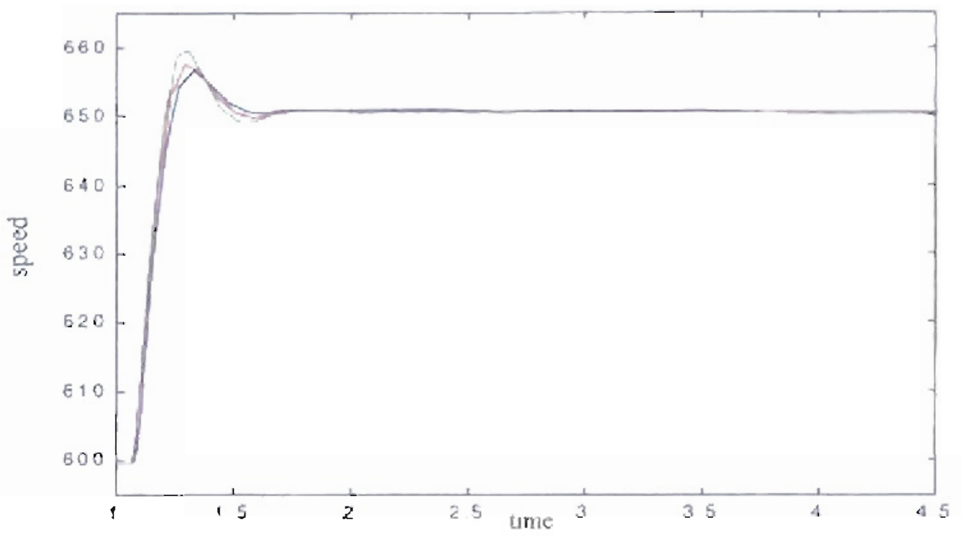


Figure 5.19: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 2.6 lb-ft-sec^2.*

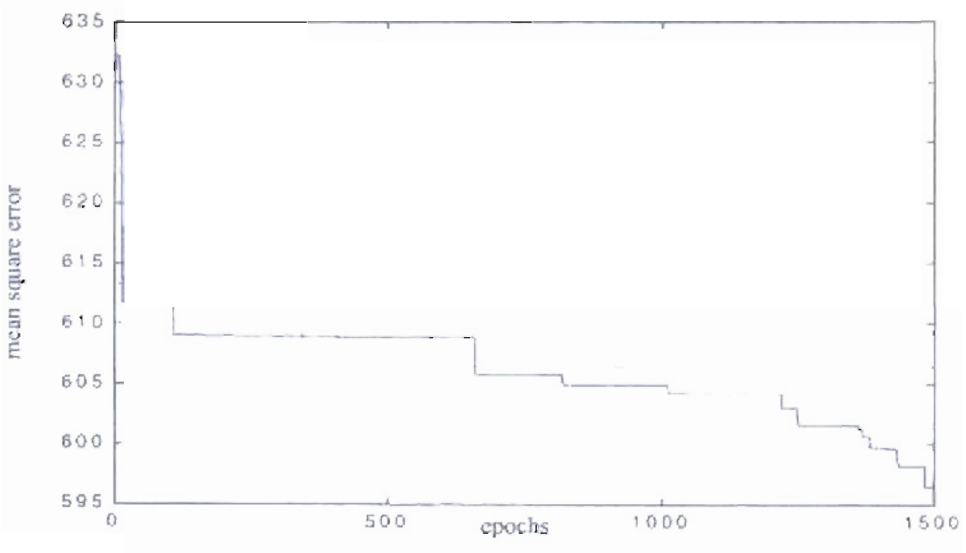


Figure 5.20: *Learning rate for engine inertia = 2.6 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

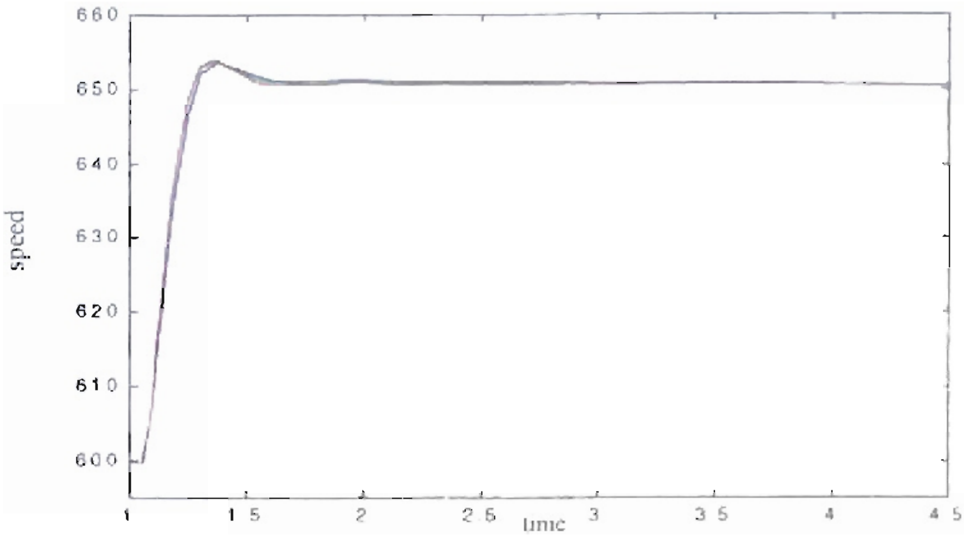


Figure 5.21: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 3 lb-ft-sec^2.*

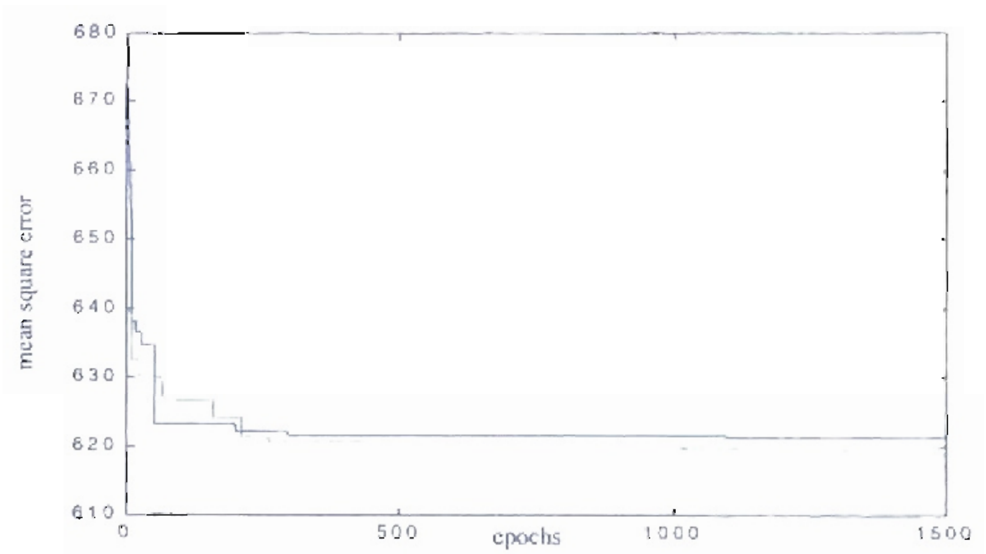


Figure 5.22: *Learning rate for engine inertia = 3 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

For the next case we repeated the training for different engine inertias, but we changed the fitness function from mean square error to percent overshoot. For that fitness function we obtained the results shown in Table 5.3. From that table we can see that the percentage response improvement depends on the initial overshoot. More impressive results were obtained with more initial overshoot and oscillatory responses. Due to the selected fitness function, we can see in the odd figures from Figure 5.23 to Figure 5.35 that the resulting responses tend to be more flat. If we compare the training processes, we obtained a better response for small populations, as seen in the even figures from Figure 5.24 to Figure 5.36. This must be due to the less restrictive fitness function (overshoot) allowing faster mutation for lower populations.

Table 5.3: *Engine with different inertia. Controller's results for percent overshoot based fitness.*

Inertia *lb-ft-sec^2*.	**Initial Fitness (overshoot)**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 50**
1	103.1990	0.4311 (0.41 %)	Kp=0.5323 Ki=0.0670 Kd=0.0767 a = 14.8190	0.9894 (0.95 %)	Kp=0.5595 Ki=0.1045 Kd=0.0475 a = 16.3213
1.4	57.6517	0.3047 (0.52 %)	Kp=0.6509 Ki=0.0628 Kd=0.0307 a = 11.9443	1.2924 (2.24 %)	Kp=0.7971 Ki=0.1319 Kd=0.0514 a = 13.6600
1.8	36.6645	0.3939 (0.41 %)	Kp=0.7736 Ki=0.0625 Kd=0.0507 a = 13.9364	0.4040	Kp=0.9222 Ki=0.0796 Kd=0.0691 a = 14.8195
2.2	22.2402	0.3906 (1.07 %)	Kp=0.9513 Ki=0.0675 Kd=0.0494 a = 13.9331	0.4233 (1.15 %)	Kp=1.1102 Ki=0.0830 Kd=0.0479 a = 14.8503
2.6	12.9828	0.3604 (2.77 %)	Kp=1.3477 Ki=0.1014 Kd=0.0698 a = 17.8313	0.3329 (2.56 %)	Kp=1.2580 Ki=0.0872 Kd=0.0561 a = 13.0939
3	7.1080	0.4237 (5.96 %)	Kp=1.5058 Ki=0.0971 Kd=0.0440 a = 15.1023	0.3842 (5.40 %)	Kp=1.4804 Ki=0.0955 Kd=0.0449 a = 13.6163

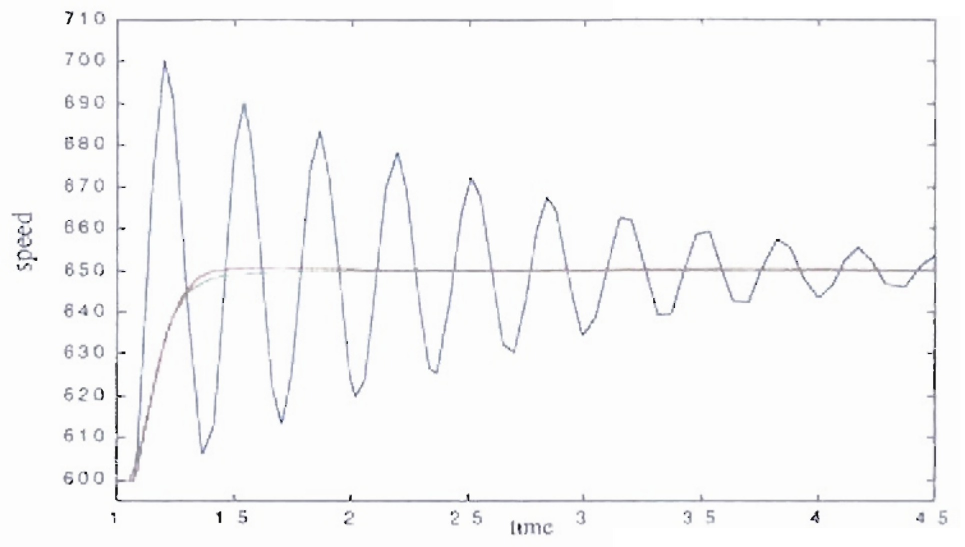


Figure 5.23: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500. Inertia 1 lb-ft-sec^2.*

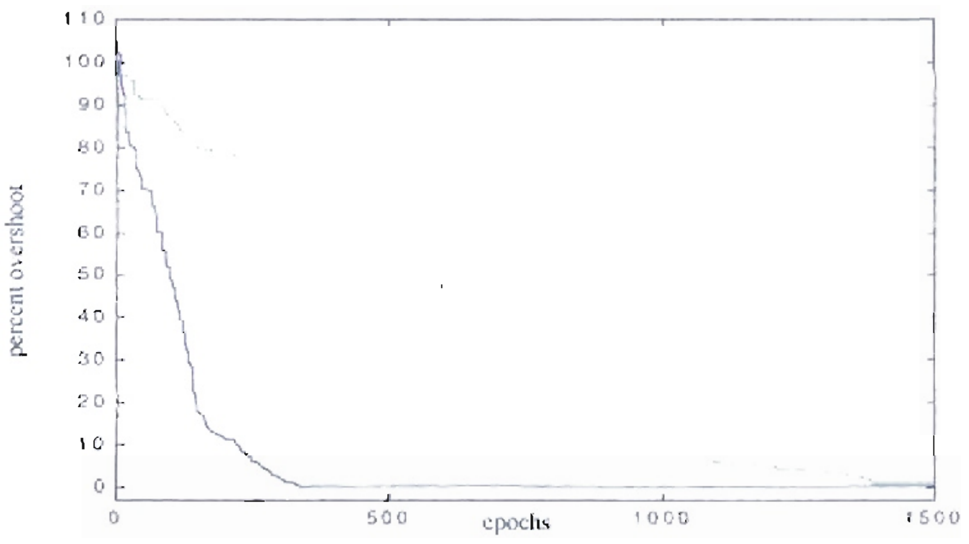


Figure 5.24: *Learning rate for engine inertia 1 lb-ft-sec^2. Blue: Population 5, Green: Population 50.*

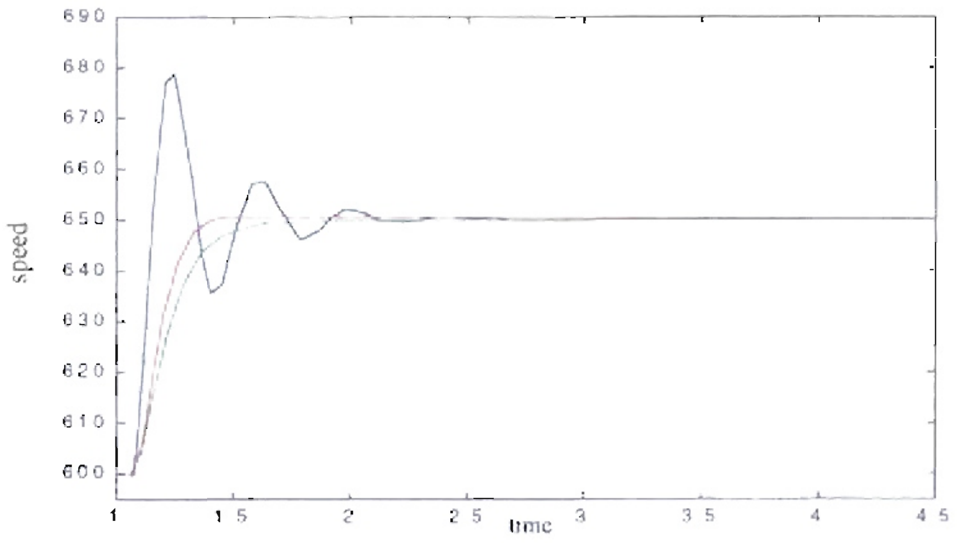


Figure 5.25: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5). Red = Genitor optimized parameters (Population = 50). Epochs = 1500. Inertia = 1.4 lb-ft-sec^2.*

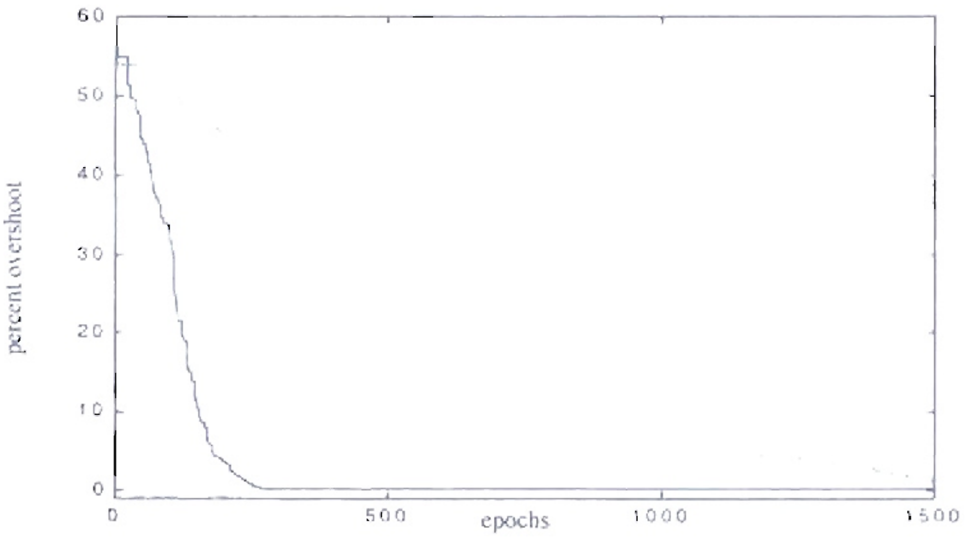


Figure 5.26: *Learning rate for engine inertia = 1.4 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

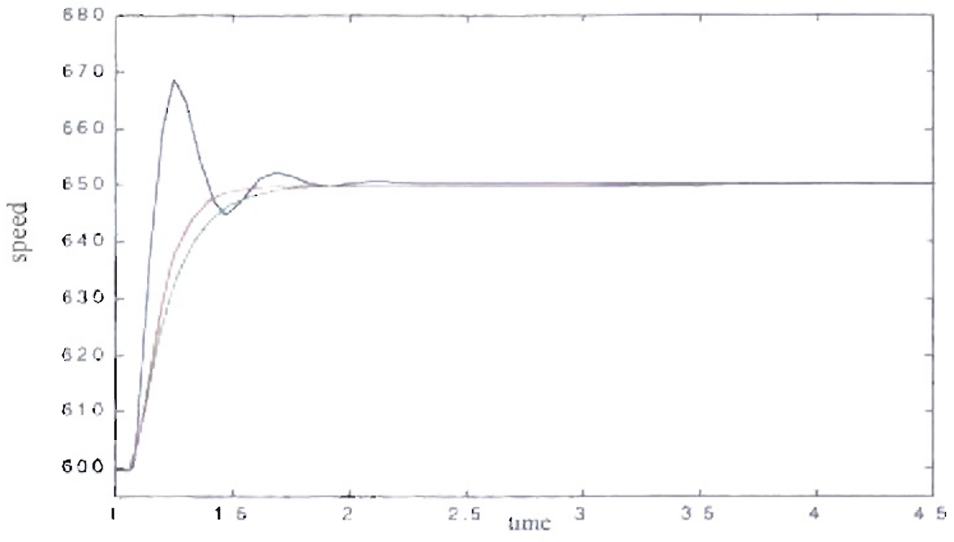


Figure 5.27: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters. Green = Genitor optimized parameters (Population = 5). Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 1.8 lb-ft-sec^2.*

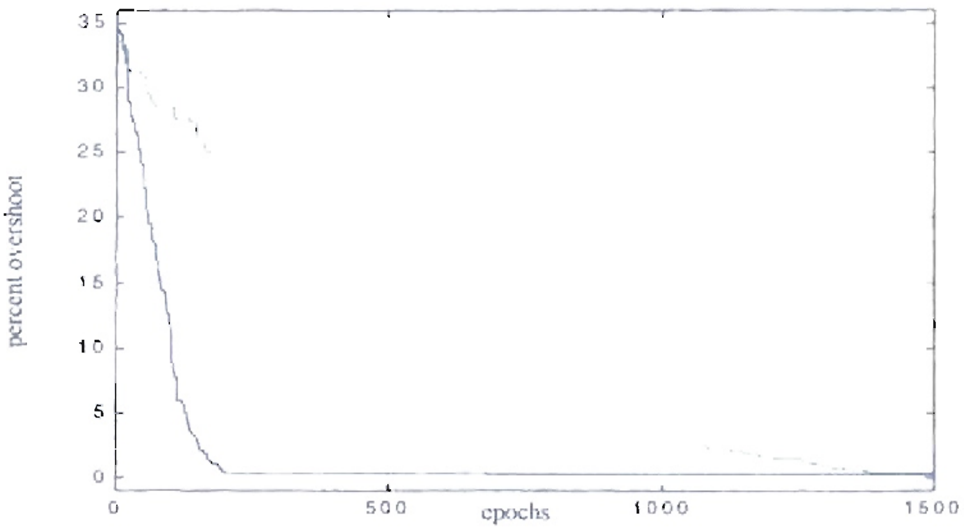


Figure 5.28: *Learning rate for engine inertia = 1.8 lb-ft-sec^2. Blue: Population = 5. Green: Population = 50.*

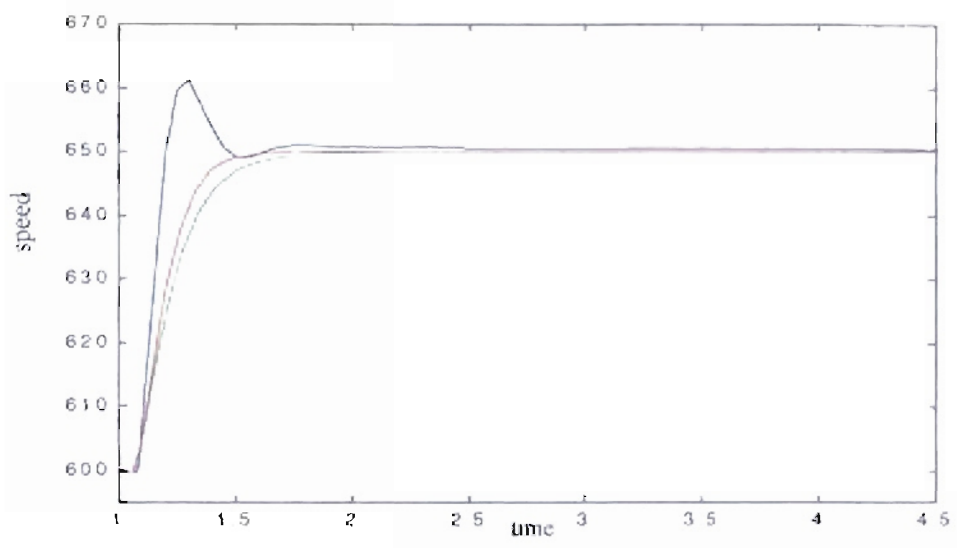


Figure 5.29: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 2.2 lb-ft-sec^2.*

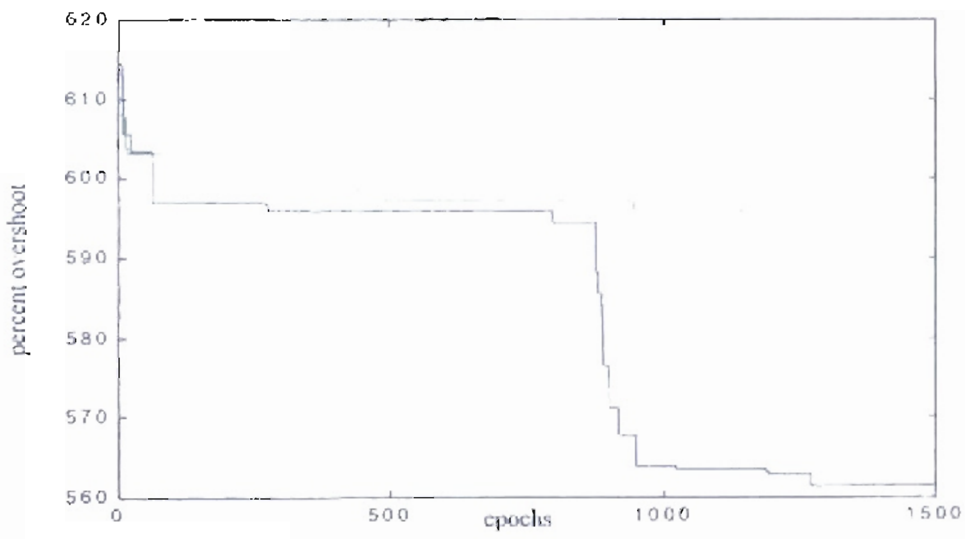


Figure 5.30: *Learning rate for engine inertia 2.2 lb-ft-sec^2. Blue: Population 5. Green: Population 50.*

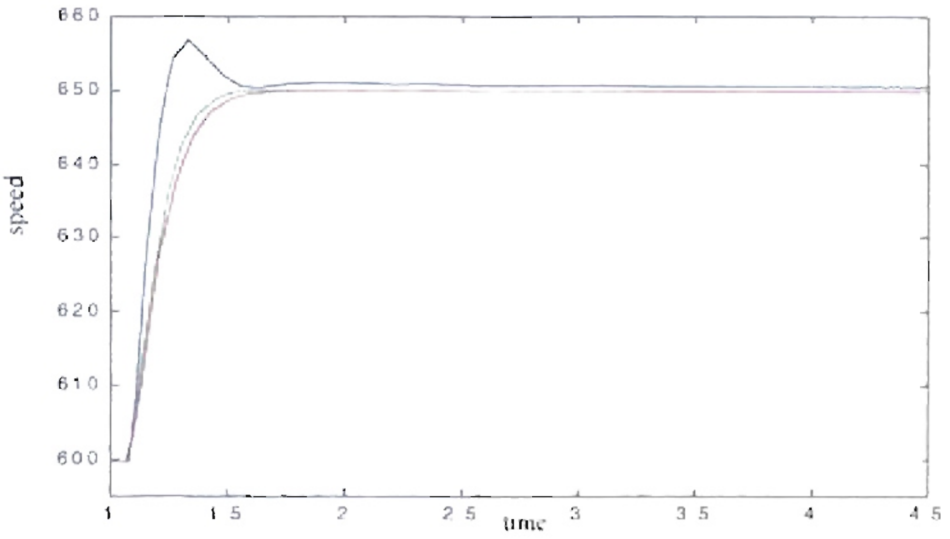


Figure 5.31: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 2.6 lb-ft-sec^2.*

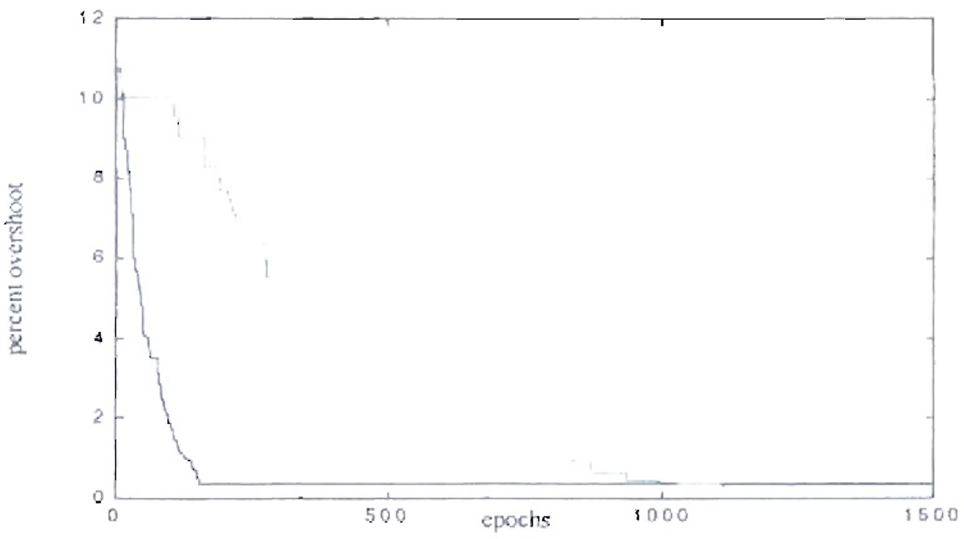


Figure 5.32: *Learning rate for engine inertia = 2.6 lb-ft-sec^2. Blue: Population = 5, Green: Population = 50.*

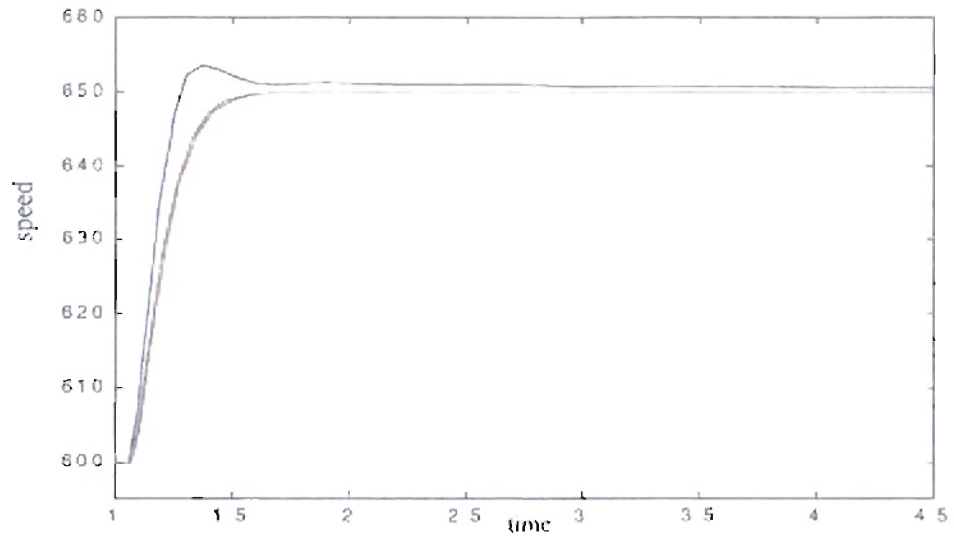


Figure 5.33: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500. Inertia 3 lb-ft-sec^2.*

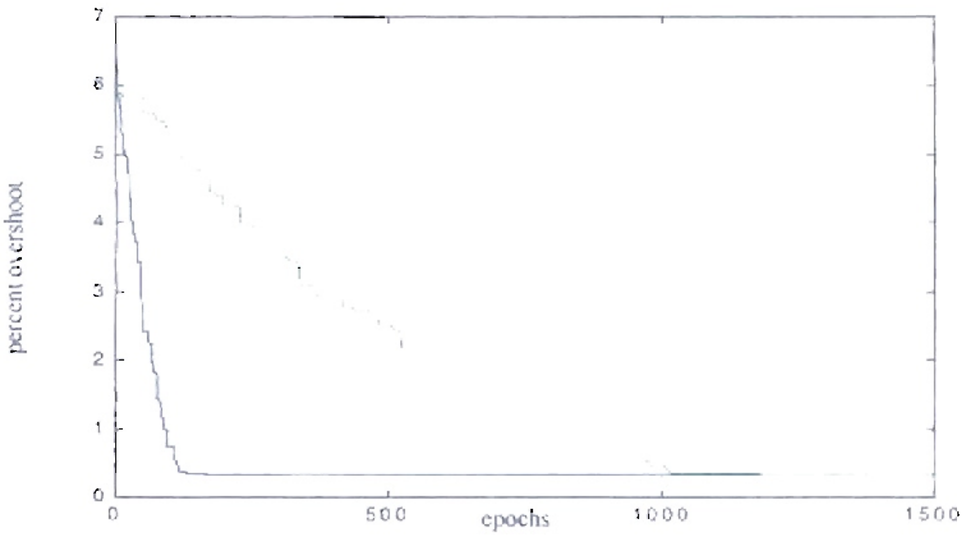


Figure 5.34: *Learning rate for engine inertia 3 lb-ft-sec^2. Blue: Population 5, Green: Population 50.*

5.6.2. Results for different fueling delays.

For the next case we repeated the training for different engine fueling delays τ. For each case the initial response changed according to the fueling delay value with a fixed engine inertia of 2 lb-ft-sec^2. An optimization is needed to adjust the PID controller values to match the engine fueling delay.We use the mean square error as our fitness value. Like the previous case, we changed the training scheme to two speed transitions to reduce the training time process. The first transition is from 600 rpm to 650 rpm at 1 second. The second transition is from 650 rpm to 600 rpm at 4.5 seconds. The engine and the PID controller are initialized with the engine conditions for 600 rpm. The simulation runs from 0 to 1 second without error measurements, then the training is started.

Using the percent overshoot as fitness, we obtained the results shown in Table 5.4 for six different fueling delay values and two different population sizes. If we look at Table 5.4 and the odd figures from Figure 5.35 to Figure 5.45, we notice how the response improves from the closest values to the delay of 80 msec to the extreme delay values. This must be due to the fact that the original PID parameters were optimized for the delay of 80 msec. As we move far from that delay, the original PID response needs more improvement. If we look at the training process (even figures from Figure 5.36 to Figure 5.46), we notice a faster response for the smaller population. However, in the majority of the responses the final values obtained for the large population were better. Figure 5.40 show a special case, where the training for smaller population apparently arrived at a local minimal and then future training does not improve the engine response.

Table 5.4: *Engines with different fueling delay. Controller's results for mean square error based fitness.*

Delay (msec)	Initial Fitness $(error)^2$.	Best Fitness $(error)^2$. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness $(error)^2$. Pop. = 50	Resulting Parameters Pop. = 5
30	355.32	257.26 (72.36 %)	Kp=3.0428 Ki= 0.2787 Kd=0.0527 a = 15.0048	268.68 (75.57 %)	Kp=2.7465 Ki= 0.4947 Kd=0.0425 a = 15.7296
50	432.08	385.81 (89.29 %)	Kp=2.2091 Ki=0.4515 Kd=0.0568 a = 14.5073	380.24 (88.00 %)	Kp=2.5555 Ki= 0.4656 Kd=0.0496 a = 14.1181
70	544.42	521.53 (95.79 %)	Kp=1.9492 Ki=0.4857 Kd=0.0533 a = 16.3126	517.01 (94.96 %)	Kp=1.9840 Ki=0.4314 Kd=0.0485 a = 13.3124
90	729.64	660.49 (90.52 %)	Kp=1.7835 Ki=0.2725 Kd=0.0609 a = 13.6584	669.07 (91.69 %)	Kp=1.7831 Ki=0.4787 Kd=0.0496 a = 13.6043
110	1007.13	818.17 (81.23 %)	Kp= 1.5797 Ki=0.3228 Kd=0.0523 a = 17.3795	818.07 (81.22 %)	Kp= 1.4269 Ki=0.2924 Kd=0.0519 a = 14.7027
130	1579.99	1006.44 (63.69 %)	Kp= 1.3023 Ki=0.5150 Kd=0.0347 a = 13.9807	972.48 (61.54 %)	Kp= 1.2753 Ki=0.2998 Kd=0.0482 a = 14.2724

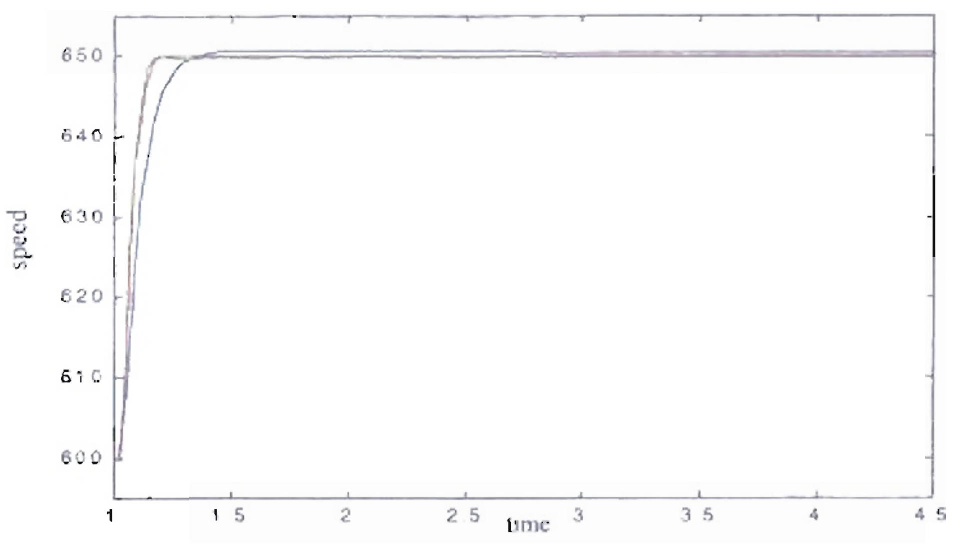


Figure 5.35: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 30 msec.*

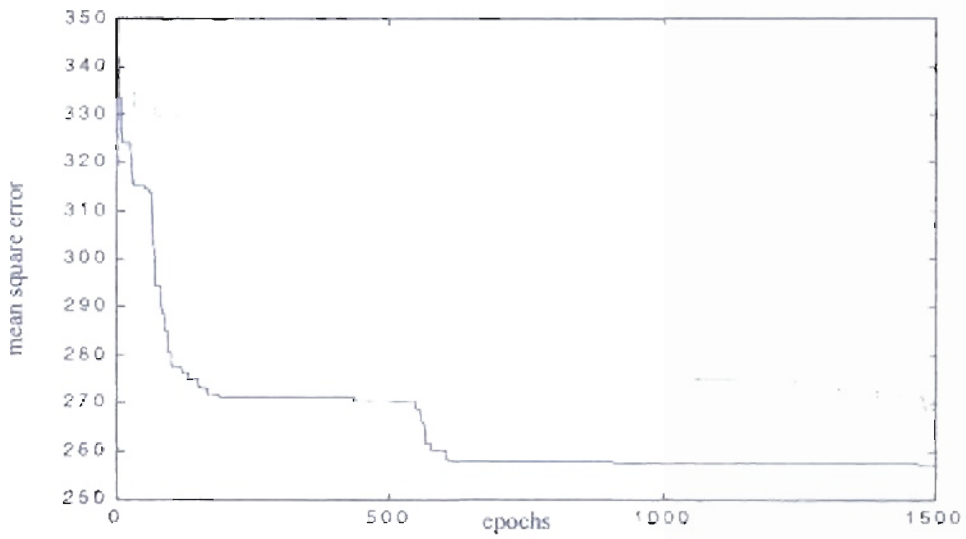


Figure 5.36: *Learning rate for engine delay = 30 msec. Blue: Population = 5, Green: Population = 50.*

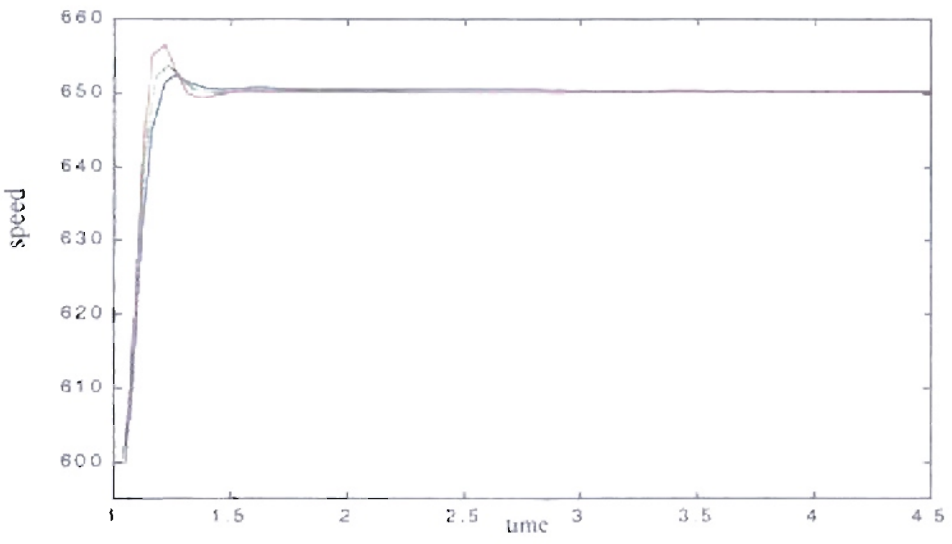


Figure 5.37: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 50 msec.*

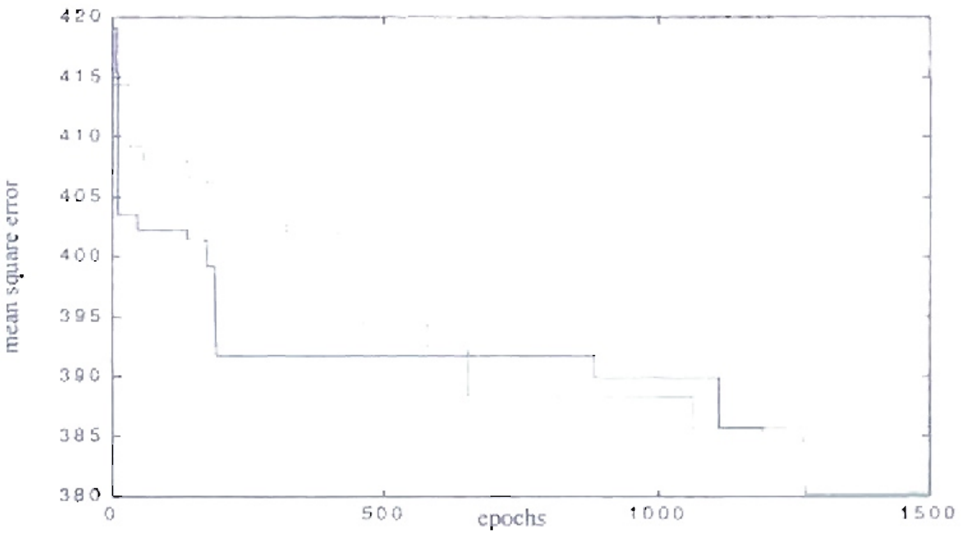


Figure 5.38: *Learning rate for engine delay = 50 msec. Blue: Population = 5, Green: Population = 50.*

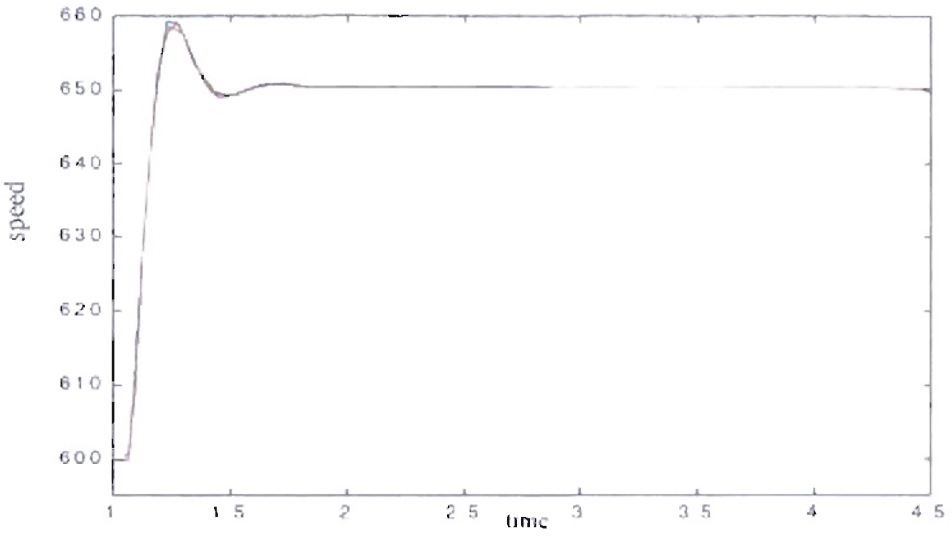


Figure 5.39: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 70 msec.*

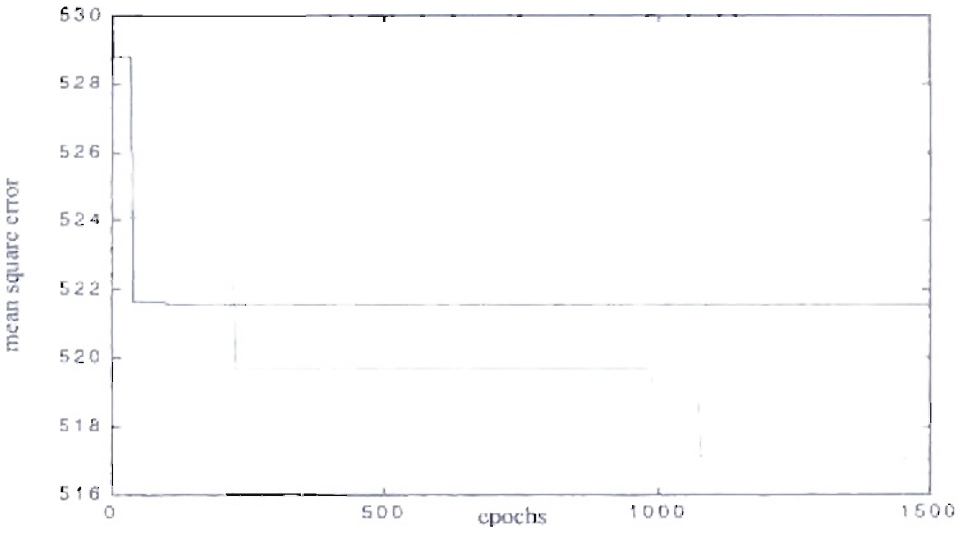


Figure 5.40: *Learning rate for engine delay 70 msec. Blue: Population 5, Green: Population 50.*

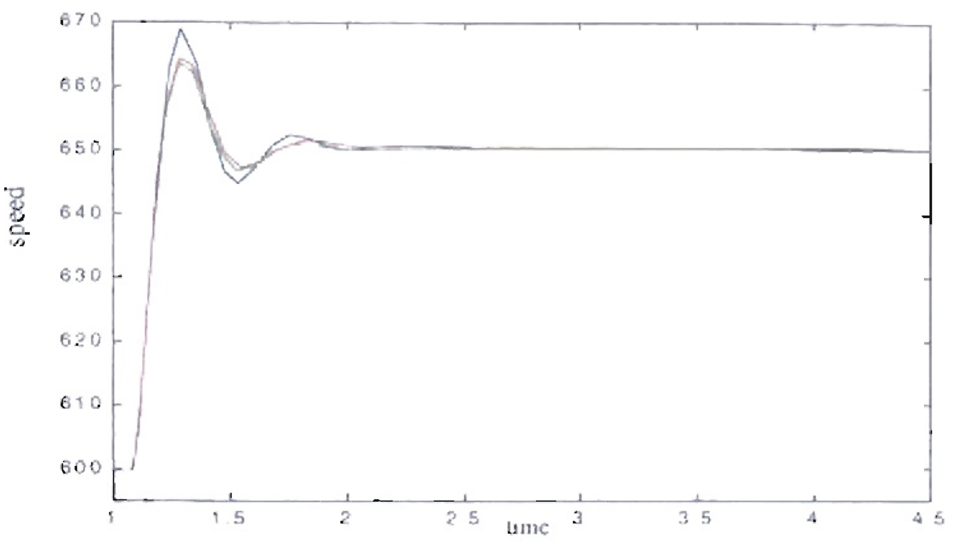


Figure 5.41: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters. Green = Genitor optimized parameters (Population = 5). Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 90 msec.*

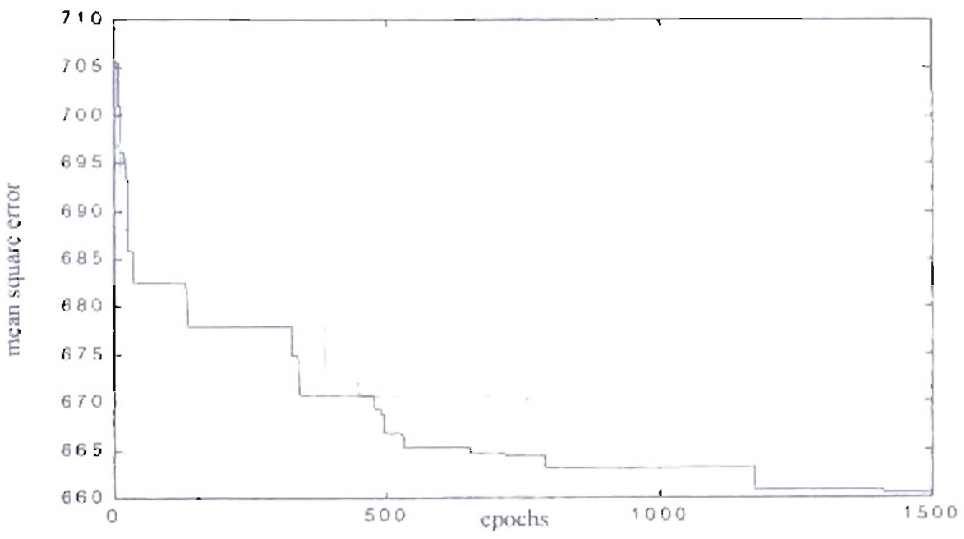


Figure 5.42: *Learning rate for engine delay = 90 msec. Blue: Population = 5. Green: Population = 50.*

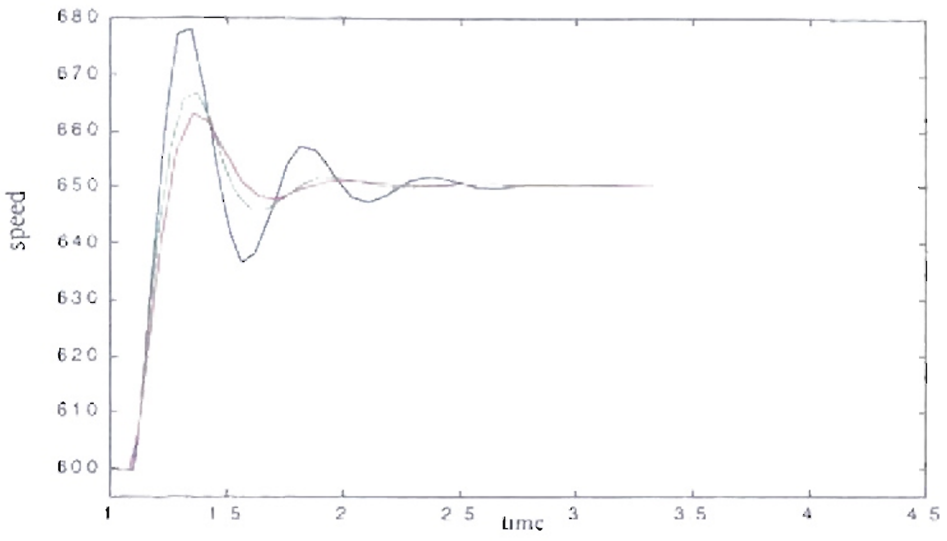


Figure 5.43: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 110 msec.*

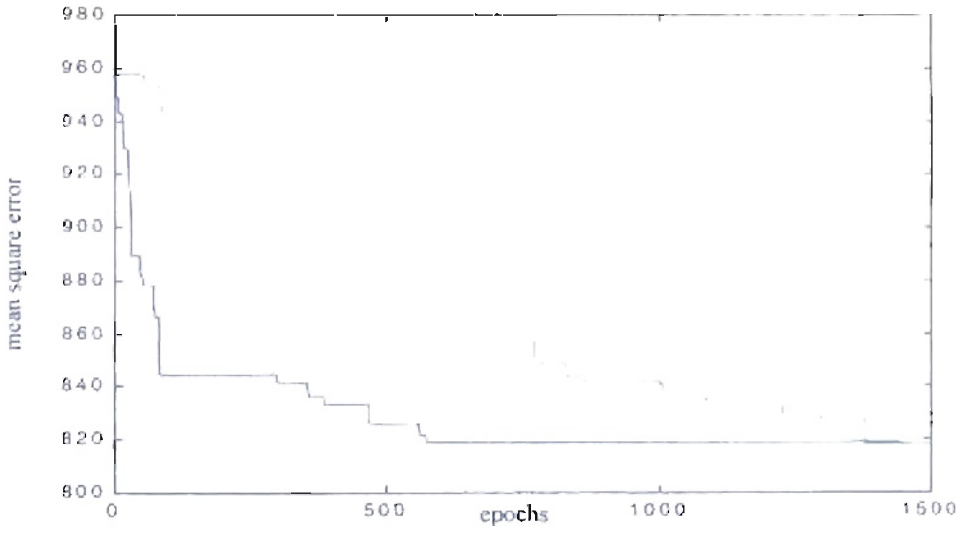


Figure 5.44: *Learning rate for engine delay 110 msec. Blue: Population 5, Green: Population 50.*

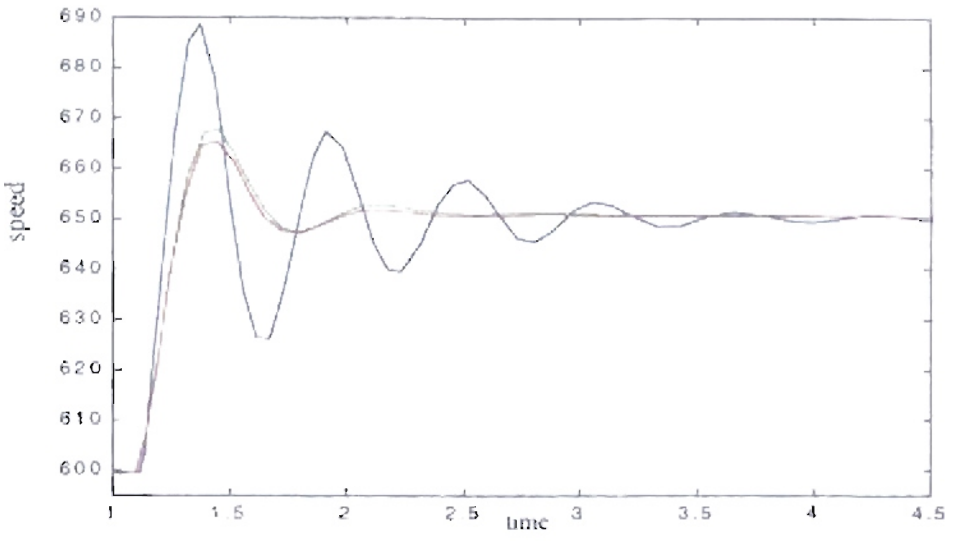


Figure 5.45: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500. Delay = 130 msec.*

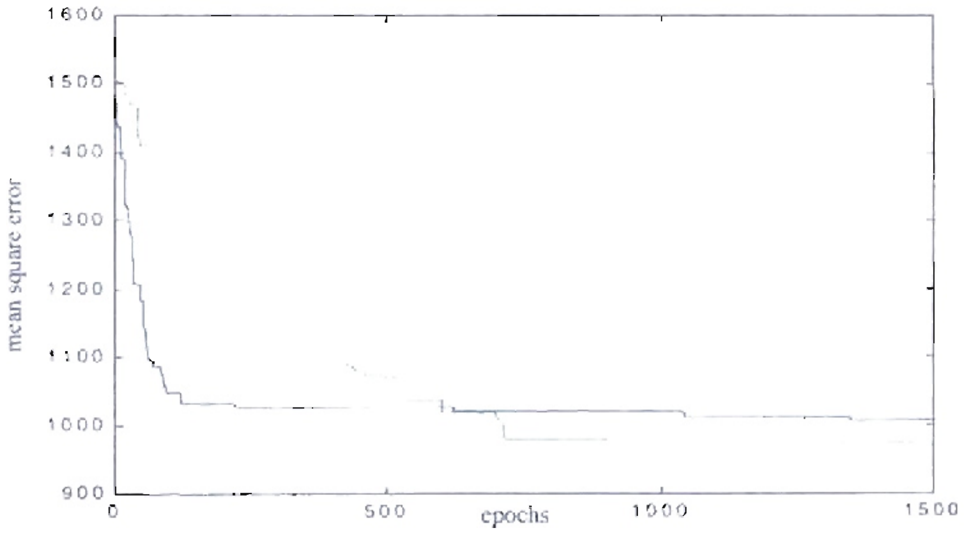


Figure 5.46: *Learning rate for engine delay = 130 msec. Blue: Population = 5, Green: Population = 50.*

For the next case we repeated the training for different fueling delays τ. changing the fitness function to the percent overshoot. For that fitness function we obtained the results shown in Table 5.5. As we can see from Figure 5.47 to Figure 5.58. the resulting responses tend to be flatter.

Table 5.5: *Engines with different fueling delay. Controller's results for percent overshoot based fitness.*

Delay (msec)	Initial Fitness (overshoot).	Best Fitness (overshoot). Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (overshoot). Pop. = 50	Resulting Parameters Pop. = 5
30	1.3080	0.3443 (26.32 %)	Kp=1.9959 Ki=0.2174 Kd=0.0431 a=16.0838	0.3612 (27.61 %)	Kp=2.2170 Ki=0.2454 Kd=0.0490 a=14.8958
50	4.5210	0.4311 (9.53 %)	Kp=1.6536 Ki=0.1636 Kd=0.0546 a=18.3008	0.3864 (8.54 %)	Kp=1.6156 Ki=0.1564 Kd=0.0481 a=15.1590
70	18.7506	0.4243 (2.26 %)	Kp=1.1677 Ki=0.0990 Kd=0.0591 a=15.1457	0.3374 (1.79 %)	Kp=1.2237 Ki=0.1097 Kd=0.0577 a=15.0712
90	37.9684	0.3627 (0.95 %)	Kp=0.8812 Ki=0.0698 Kd=0.0526 a=17.7431	0.4373 (1.15 %)	Kp=0.9625 Ki=0.0770 Kd=0.0601 a=13.8594
110	56.3349	0.3243 (0.57 %)	Kp=0.7657 Ki=0.0548 Kd=0.0520 a=13.4557	0.3109 (0.55 %)	Kp=0.7764 Ki=0.0587 Kd=0.0601 a=13.3248
130	76.8723	0.2631 (0.34 %)	Kp=0.5320 Ki=0.0335 Kd=0.0322 a=14.8544	1.0819 (1.40 %)	Kp=0.6909 Ki=0.0609 Kd=0.0473 a=15.2094

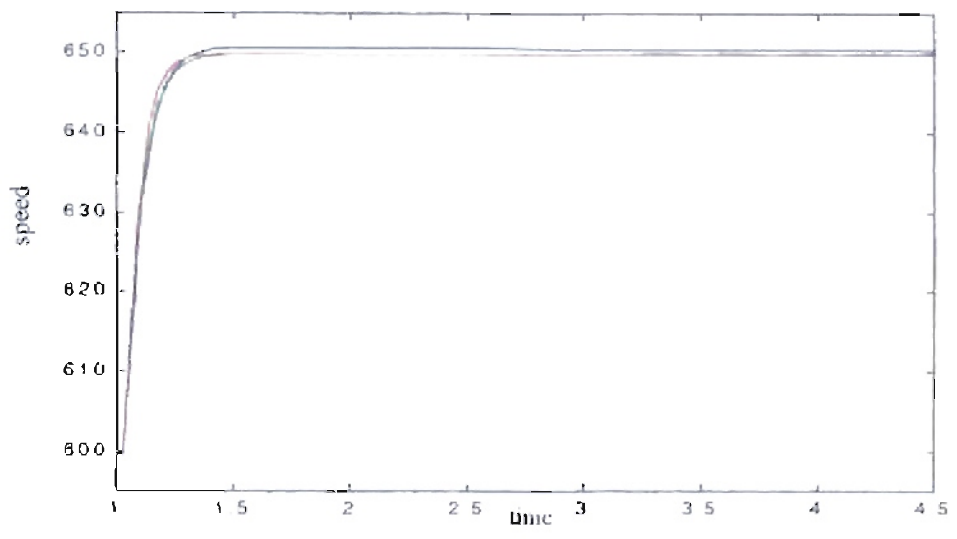


Figure 5.47: *Detail transition from 600 to 650 rpm. Blue — Original PID parameters, Green — Genitor optimized parameters (Population — 5), Red — Genitor optimized parameters (Population — 50). Epochs — 1500, Delay — 30 msec.*

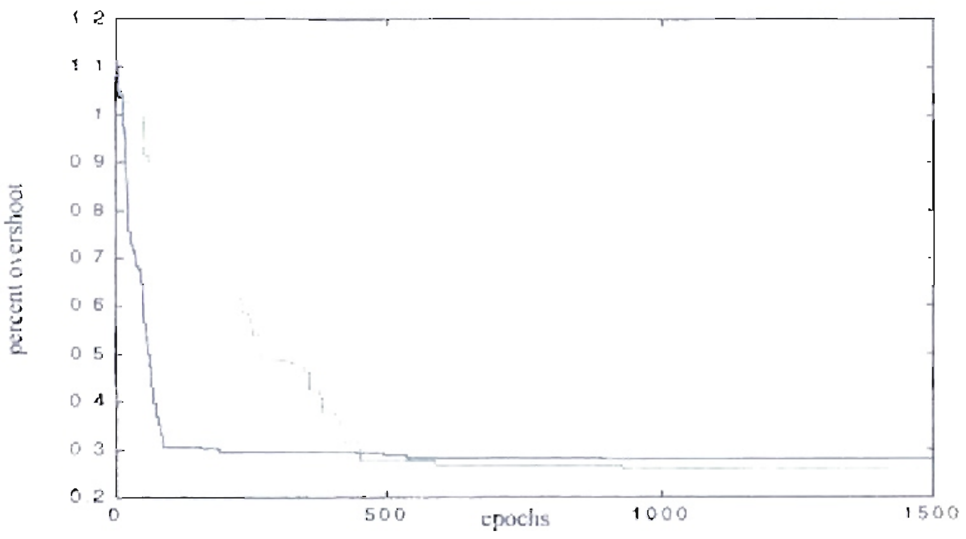


Figure 5.48: *Learning rate for engine delay — 30 msec. Blue: Population — 5, Green: Population — 50.*

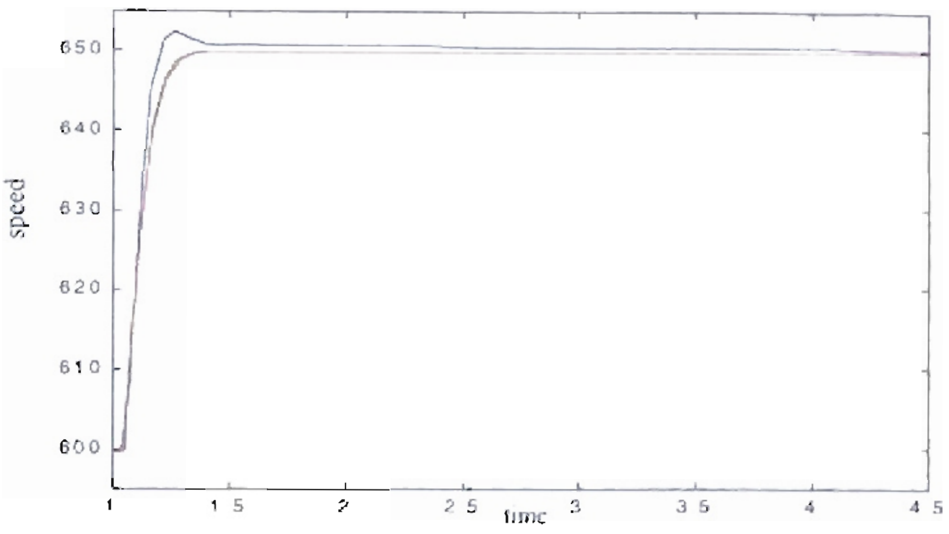


Figure 5.49: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 50 msec.*

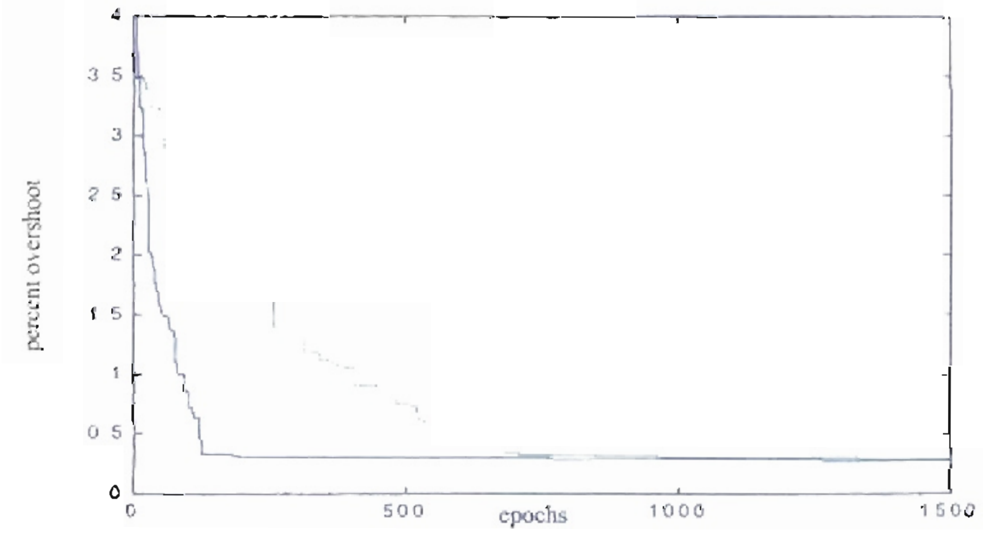


Figure 5.50: *Learning rate for engine delay 50 msec. Blue: Population 5, Green: Population 50.*

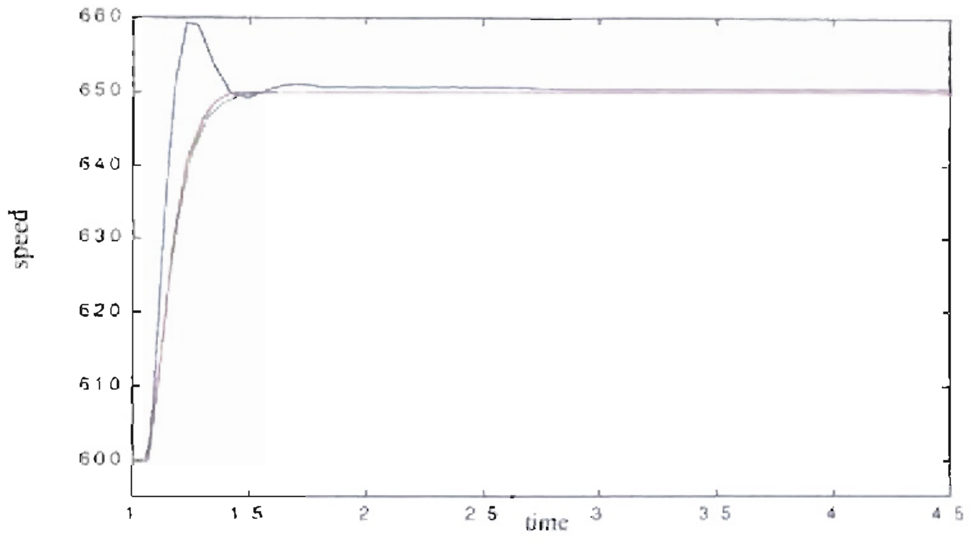


Figure 5.51: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 70 msec.*

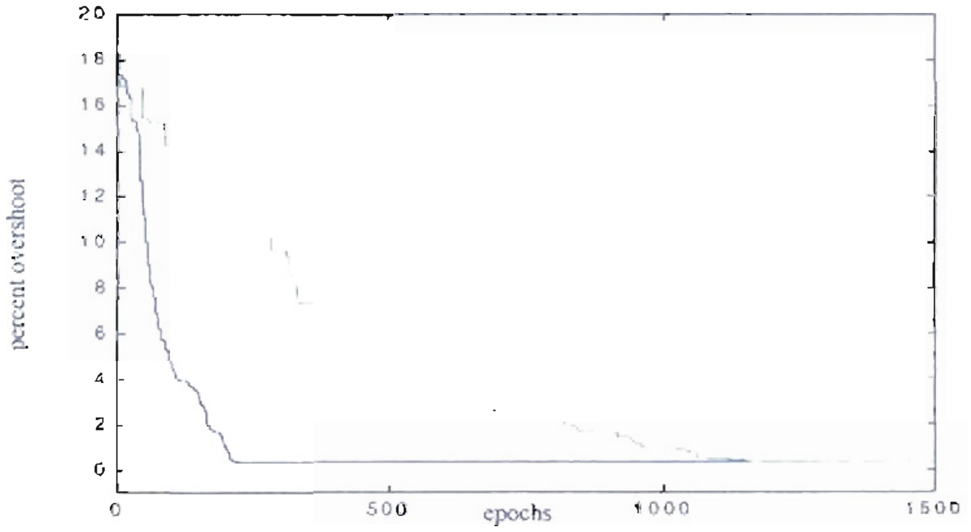


Figure 5.52: *Learning rate for engine delay = 70 msec. Blue: Population = 5, Green: Population = 50.*

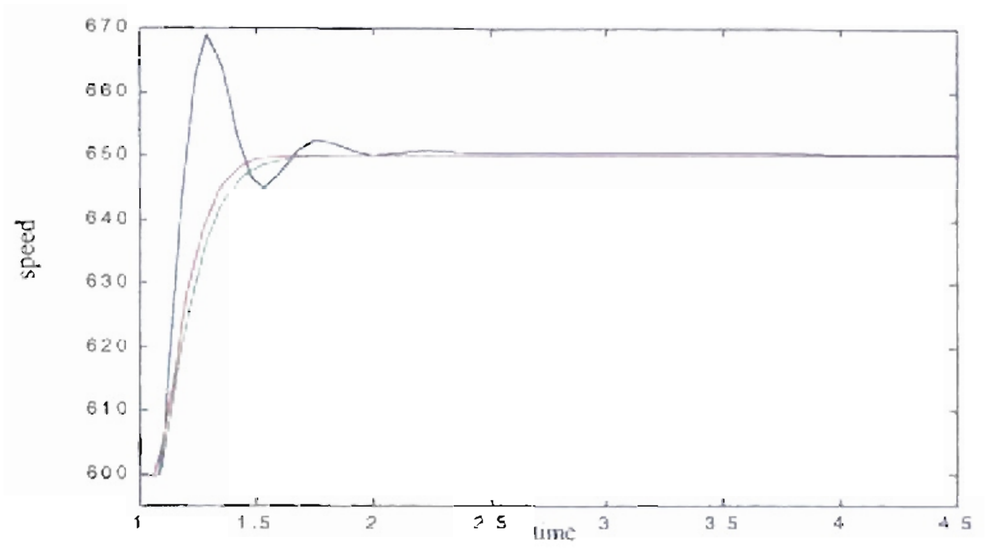


Figure 5.53: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 90 msec.*

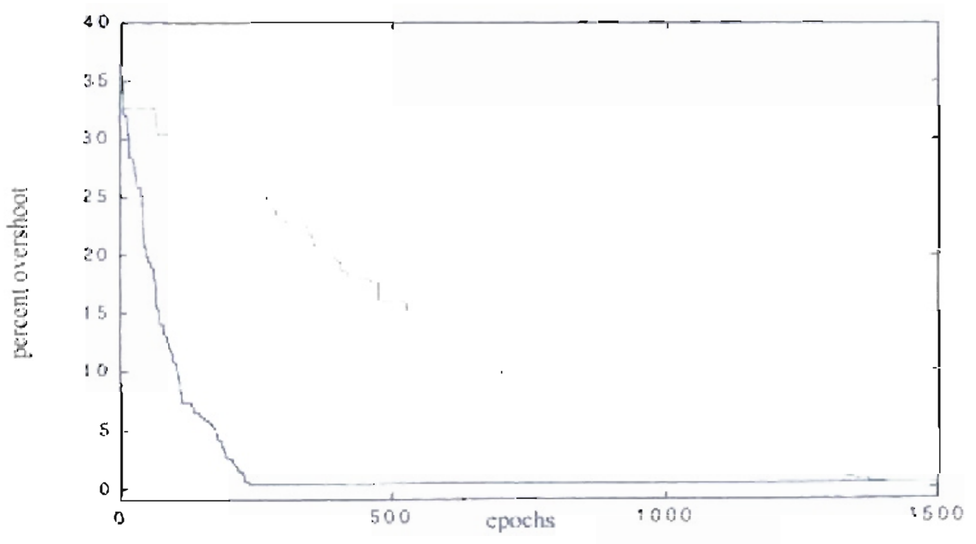


Figure 5.54: *Learning rate for engine delay 90 msec. Blue: Population 5. Green: Population 50.*

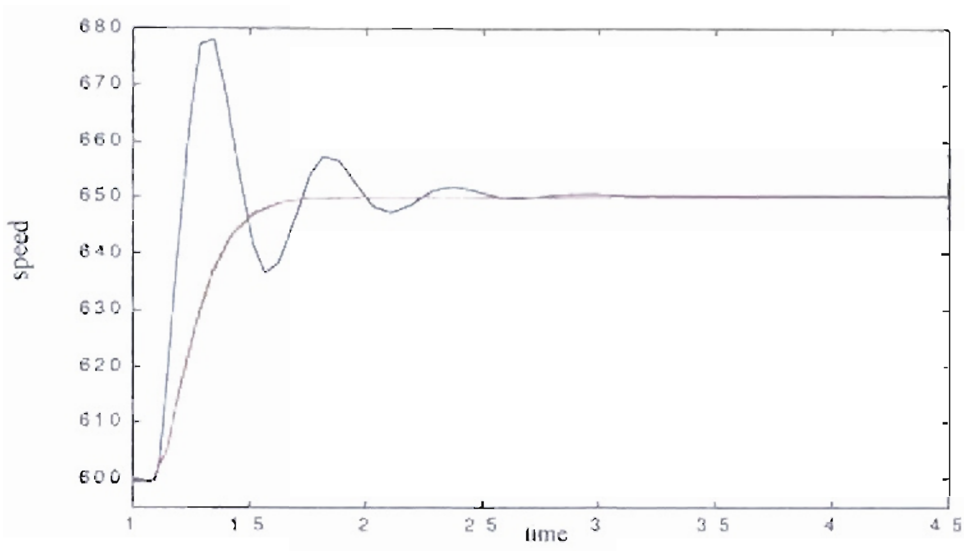


Figure 5.55: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 110 msec.*

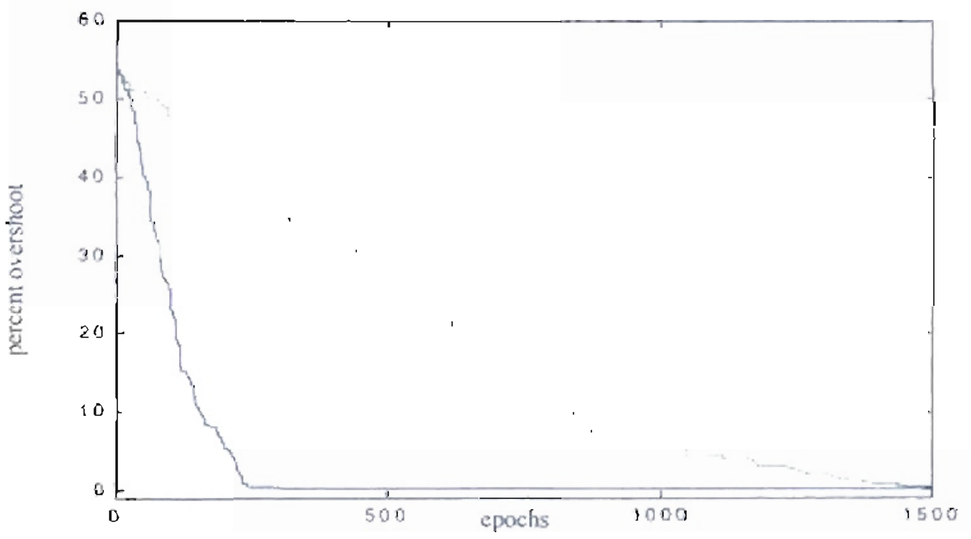


Figure 5.56: *Learning rate for engine delay 110 msec. Blue. Population 5. Green: Population 50.*

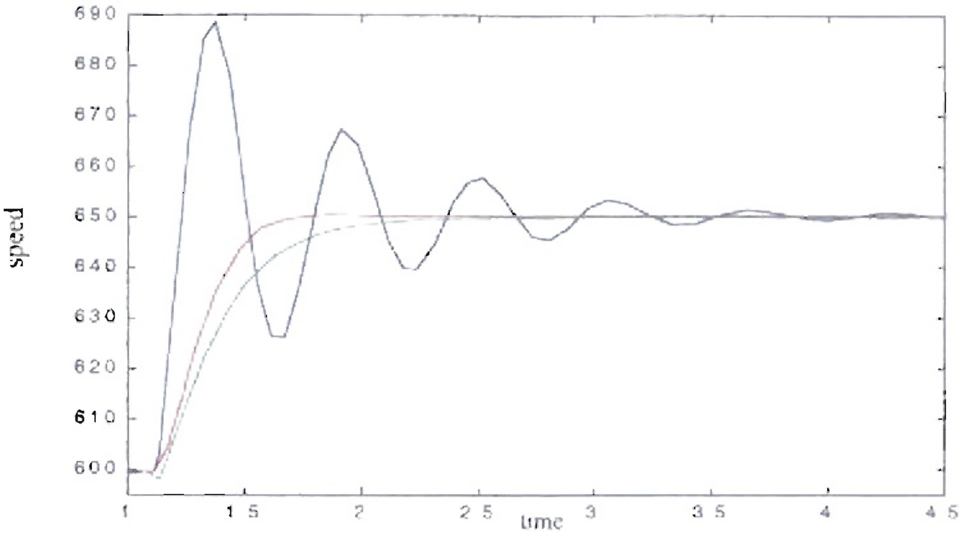


Figure 5.57: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 130 msec.*

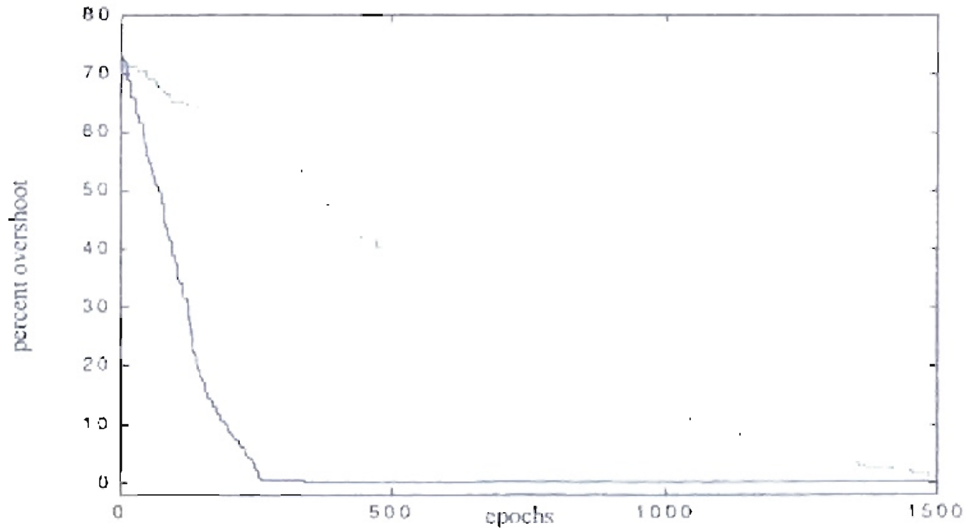


Figure 5.58: *Learning rate for engine delay 130 msec. Blue: Population 5, Green: Population 50.*

5.7. Genetic Algorithms applied to digital controllers.

In the next sections we will apply the GENITOR algorithm to the Digital version of a PID controller. We first converted the original analog PID controller to its digital version. To realize that conversion we took the PID original relation given by Eq. (3.1) then executed the conversion:

$$s \rightarrow \frac{z-1}{T} \tag{5.1}$$

where T is the sampling time. The transformation of Eq. (3.1) by using Eq. (5.1) is:

$$G(z) = \frac{\begin{array}{c}(Kp+Kd)z^2 + (-2(Kp+Kd) + T(Kp \cdot a + Ki))z \\ + (Kp + Kd - T(Kp \cdot a + Ki) + T^2(Ki \cdot a))\end{array}}{z^2 + (T \cdot a - 2)z + (1 - T \cdot a)} \tag{5.2}$$

For $Kp = 2$, $Ki = 0.5$, $Kd = 0.05$ and $a = 15$ (middle point between 10 and 20), and $T = 50\ ms$ we obtained the transfer function:

$$G(z) = \frac{2.05z^2 - 2.575z + 0.5437}{z^2 - 1.25z + 0.25} \tag{5.3}$$

Our first approach was to emulate the GENITOR training of the analog controller and execute the conversion from Eq. (5.1). This approach resulted in a slow training and generally the results were far from desired responses. Next, we parametrized the digital controller as:

$$G(z) = \frac{K(z-z_0)(z-z_1)}{(z-p_0)(z-p_1)} \tag{5.4}$$

With the implementation of Eq. (5.4) we can define the locations of the zeros and the poles without the restrictions of the PID controller. We defined the use of the controller from Eq. (5.4) where we have five parameters p_0, p_1, z_0, z_1 and K. The initial values of the previous parameters will be obtained from Eq. (5.3), resulting in $p_0 = 1$, $p_1 = 0.25$, $z_0 = 0.9875$, $z_1 = 0.2686$ and $K = 2.05$. Starting with those values we will execute the GENITOR algorithm defined in section 5.2 to optimize the controller parameters for different values of fueling delay and engine inertia.

5.8. GENITOR algorithm applied to digital controller and engine with friction b_f=0.1229.

5.8.1. Results for different engine inertia.

For the first digital controller case we executed the training for different engine inertia values. For each case the initial response changes according to the engine inertia value. Optimization is needed to adjust the controller values to match the engine inertia. We use the mean square error as our fitness value. As in the analog case, we changed the training scheme to two speed transitions to reduce the training time. The first transition is from 600 to 650 rpm at t_1 seconds. The second transition is from 650 to 600 at t_2 seconds. The engine and the controller were initialized with the engine conditions for 600 rpm under the basic controller. The simulation runs from 0 to t_1 seconds without error measurements, then the training is started. To ensure the correct values of the parameters for future training, we changed the mutation process in the GENITOR algorithm by accepting only poles and zeros inside the unit circle.

We executed three different training processes under different conditions:

1.- t_1 = 1 seconds, t_2 = 4.5 seconds. Searching step = 1/100.

2.- t_1 = 1 seconds, t_2 = 4.5 seconds. Searching step = 1/10.

3.- t_1 = 5 seconds, t_2 = 8.5 seconds. Searching step = 1/10.

Using the mean square error as fitness for each of the previous conditions we obtained the results shown in Table 5.6, Table 5.7 and Table 5.8. The training was made for six different inertia values and two different population sizes. If the search range is small the results are not too impressive, as we can see from Table 5.6 with Table 5.7 and Table 5.8. The variation in the t_1 value did not significantly affect the results, as we can see by comparing Table 5.7 and Table 5.8.

For very oscillatory engine responses we could reduce the mean square error to 6.72 % of its original value for inertia equal to 1 lb-ft-sec^2, as seen in first row of Table 5.8 and Figure 5.59. As inertia increases, the mean square error decreases to 42 % of its original value for inertia equal to 1.4 lb-ft-sec^2 (see second row of Table 5.7 and Figure 5.60). In the last four rows of Table 5.7 and Table 5.8 we can see reductions from 73 % to 94 % of the original mean square error for inertia between 1.8 lb-ft-sec^2 and 3 lb-ft-sec^2 (see also Figure 5.61, Figure 5.62, Figure 5.63 and Figure 5.64). If we compare the results related to the population size we noticed small differences in the results for 1500 epochs.

Table 5.6: *Engine with different inertia. Controller's results for mean square error based fitness. Searching step 1/100. Error calculation after t_l = 1 second.*

Inertia lb-ft-sec^2.	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 50
1	8169.89	5206.43 (63.73 %)	poles=1.0000 and 0.2506 zeros=0.9980 and 0.2802 gain = 2.0352	4453.27 (54.51 %)	poles=1.0000 and 0.2472 zeros=0.9889 and 0.2837 gain = 2.0224
1.4	1276.85	722.48 (56.58 %)	poles=1.0000 and 0.4441 zeros=0.9904 and 0.5377 gain = 1.7641	823.01 (64.46 %)	poles=1.0000 and 0.2690 zeros=1.0000 and 0.3892 gain = 1.9919
1.8	783.71	701.90 (89.56 %)	poles=1.0000 and 0.2497 zeros=0.9912 and 0.2994 gain = 1.8973	706.30 (90.12 %)	poles=1.0000 and 0.2477 zeros=0.9896 and 0.3193 gain = 2.0198
2.2	692.70	659.35 (95.19 %)	poles=0.9999 and 0.2419 zeros=0.9838 and 0.3091 gain = 1.9324	669.43 (96.64 %)	poles=1.0000 and 0.2458 zeros=0.9869 and 0.3038 gain = 2.0241
2.6	691.63	678.61 (98.12 %)	poles=0.9999 and 0.2490 zeros=0.9858 and 0.3202 gain = 2.0622	685.00 (99.04 %)	poles=1.0000 and 0.2435 zeros=0.9881 and 0.2863 gain = 2.0373
3	719.20	713.04 (99.14 %)	poles=0.9999 and 0.2511 zeros=0.9861 and 0.2980 gain = 2.1140	715.38 (99.47 %)	poles=1.0000 and 0.2467 zeros=0.9895 and 0.2892 gain = 2.0684

Table 5.7: *Engine with different inertia. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_f = 1 second.*

Inertia lb-ft-sec^2.	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 50
1	8169.89	655.75 (8.03 %)	poles = 1.0000 and 0.2892 zeros = 0.9885 and 0.5837 gain = 1.5860	719.90 (8.81 %)	poles = 1.0000 and 0.3234 zeros = 0.9972 and 0.5660 gain = 1.5558
1.4	1276.85	549.78 (43.06 %)	poles = 0.9998 and 0.3947 zeros = 0.9516 and 0.6942 gain = 1.5405	543.32 (42.55 %)	poles = 0.9998 and 0.2111 zeros = 0.9598 and 0.5755 gain = 1.7666
1.8	783.71	581.42 (74.19 %)	poles = 0.9999 and 0.4201 zeros = 0.9611 and 0.6877 gain = 1.9741	577.63 (73.70 %)	poles = 0.9999 and 0.1792 zeros = 0.9593 and 0.5028 gain = 2.0947
2.2	692.70	619.30 (89.40 %)	poles = 0.9999 and 0.4000 zeros = 0.9744 and 0.6337 gain = 2.1967	619.86 (89.48 %)	poles = 0.9999 and 0.2066 zeros = 0.9787 and 0.4449 gain = 2.2542
2.6	691.63	652.52 (94.35 %)	poles = 0.9998 and 0.4871 zeros = 0.9772 and 0.6694 gain = 2.5705	660.04 (95.43 %)	poles = 0.9999 and 0.2017 zeros = 0.9827 and 0.3854 gain = 2.2788
3	719.20	685.23 (95.28 %)	poles = 1.0000 and 0.2698 zeros = 0.9908 and 0.4782 gain = 2.8373	700.56 (97.41 %)	poles = 0.9999 and 0.2023 zeros = 0.9844 and 0.3182 gain = 2.3547

Table 5.8: *Engine with different inertia. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_f = 5 seconds.*

Inertia lb-ft-sec^2.	**Initial Fitness (error)^2.**	**Best Fitness (error)^2. Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (error)^2. Pop. = 50**	**Resulting Parameters Pop. = 50**
1	9850.95	549.08 (6.72 %)	poles = 1.0000 and 0.2170 zeros = 0.9113 and 0.6804 gain = 1.4068	593.91 (7.27 %)	poles = 1.0000 and 0.2087 zeros = 0.9699 and 0.5331 gain = 1.5162
1.4	1201.40	569.97 (44.64 %)	poles = 1.0000 and 0.3122 zeros = 0.9574 and 0.6051 gain = 1.6372	563.31 (44.12 %)	poles = 1.0000 and 0.1938 zeros = 0.9654 and 0.5511 gain = 1.8521
1.8	745.81	586.78 (74.87 %)	poles = 1.0000 and 0.2605 zeros = 0.9703 and 0.5720 gain = 2.1365	588.38 (75.08 %)	poles = 1.0000 and 0.2073 zeros = 0.9780 and 0.4921 gain = 2.0978
2.2	683.19	615.93 (88.92 %)	poles = 1.0000 and 0.2972 zeros = 0.9853 and 0.5661 gain = 2.4501	630.09 (90.96 %)	poles = 1.0000 and 0.2058 zeros = 0.9881 and 0.3851 gain = 2.0117
2.6	686.58	647.27 (93.59 %)	poles = 1.0000 and 0.2515 zeros = 0.9868 and 0.4625 gain = 2.5099	656.20 (94.88 %)	poles = 1.0000 and 0.2309 zeros = 0.9911 and 0.4162 gain = 2.2792
3	712.57	682.19 (94.85 %)	poles = 1.0000 and 0.2865 zeros = 0.9922 and 0.4758 gain = 2.6324	691.15 (96.10 %)	poles = 1.0000 and 0.2161 zeros = 0.9923 and 0.3453 gain = 2.4166

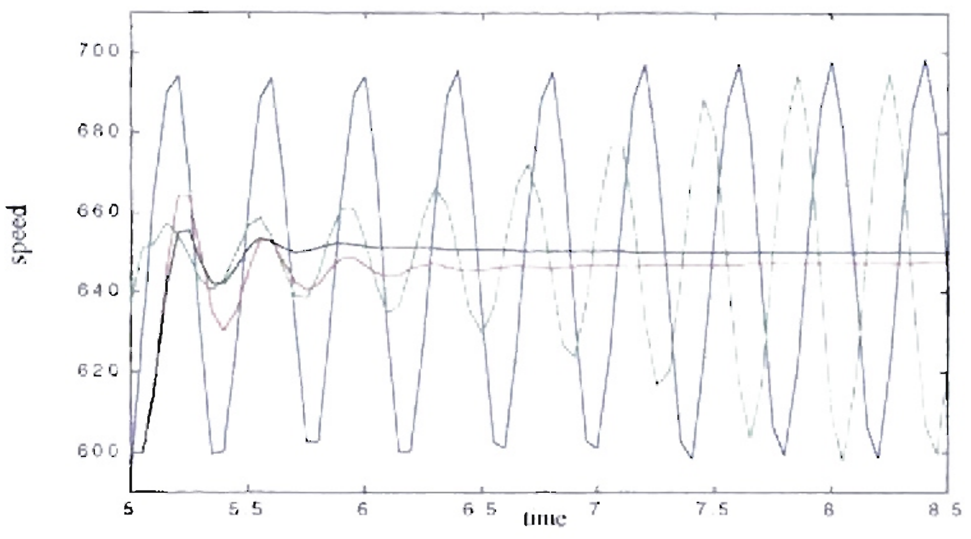


Figure 5.59: *Detail transition from 600 to 650 rpm. Epochs = 1500, Inertia = 1 lb-ft-sec^2.*[1]

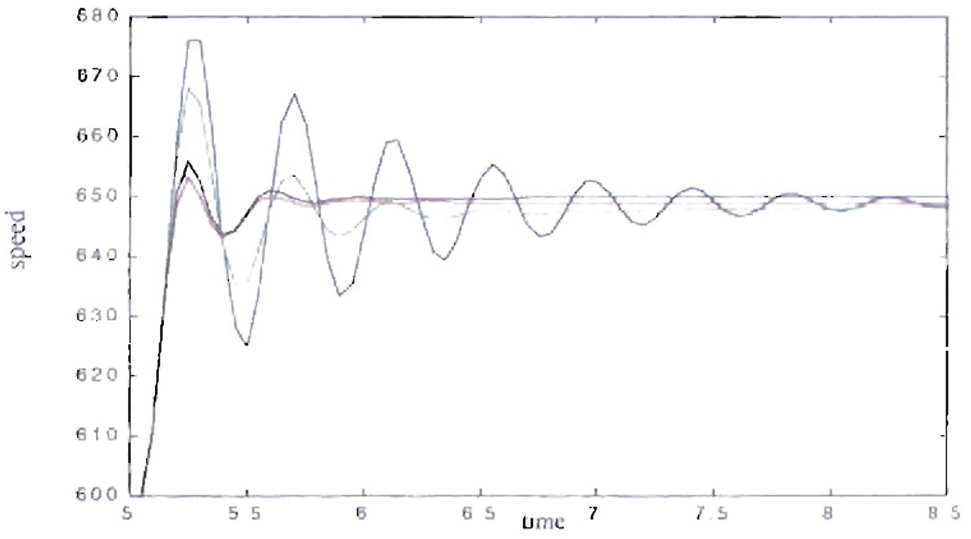


Figure 5.60: *Detail transition from 600 to 650 rpm. Inertia = 1.4 lb-ft-sec^2.*[2]

1. Color codes from Figure 5.59 to Figure 5.64: Blue = Original PID parameters, Green = GENITOR optimized parameters (searching step = 1/100, error calculation after 1 second), Red = GENITOR optimized parameters (searching step = 1/10, error calculation after 1 second), Black = GENITOR optimized parameters (searching step = 1/10, error calculation after 5 seconds).

2. See Note 1.

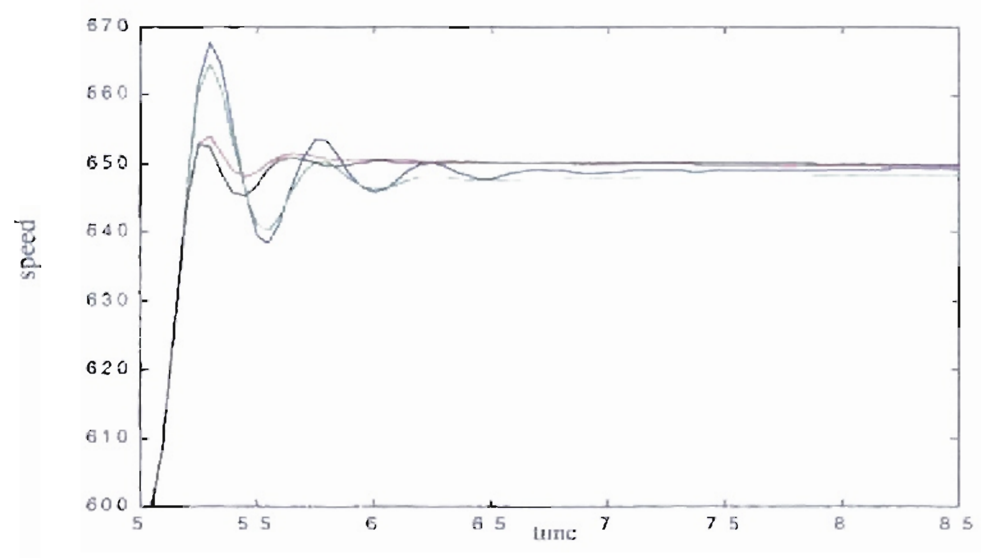


Figure 5.61: *Detail transition from 600 to 650 rpm. Inertia = 1.8 lb-ft-sec^2.*[1]

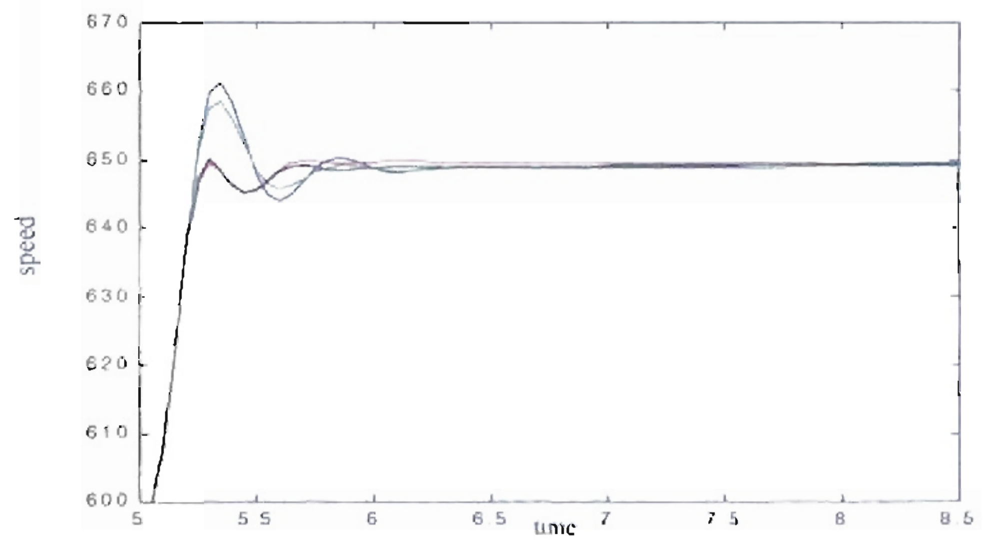


Figure 5.62: *Detail transition from 600 to 650 rpm. Inertia = 2.2 lb-ft-sec^2.*[2]

1 See note 1 on page 106.
2 See Note 1 on page 106

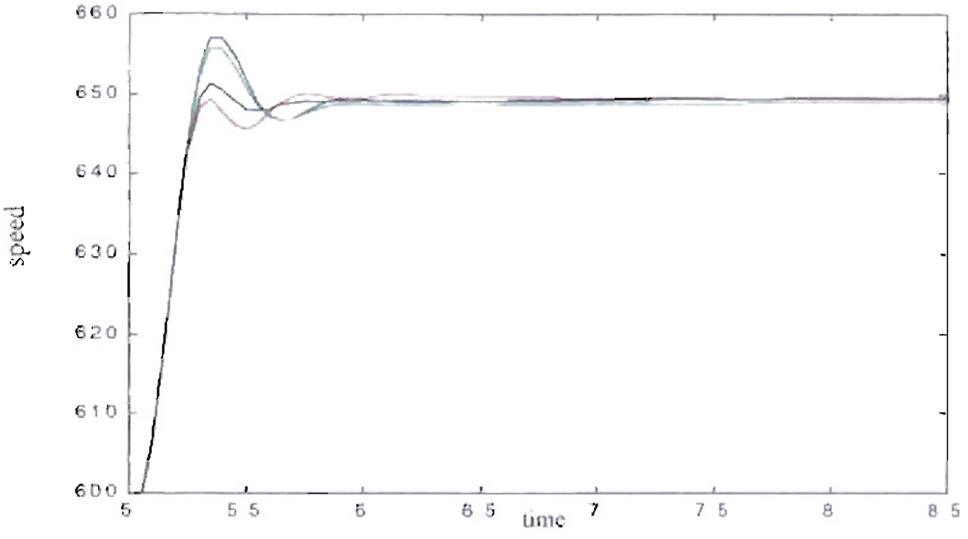


Figure 5.63: *Detail transition from 600 to 650 rpm. Inertia = 2.6 lb-ft-sec^2.*[1]

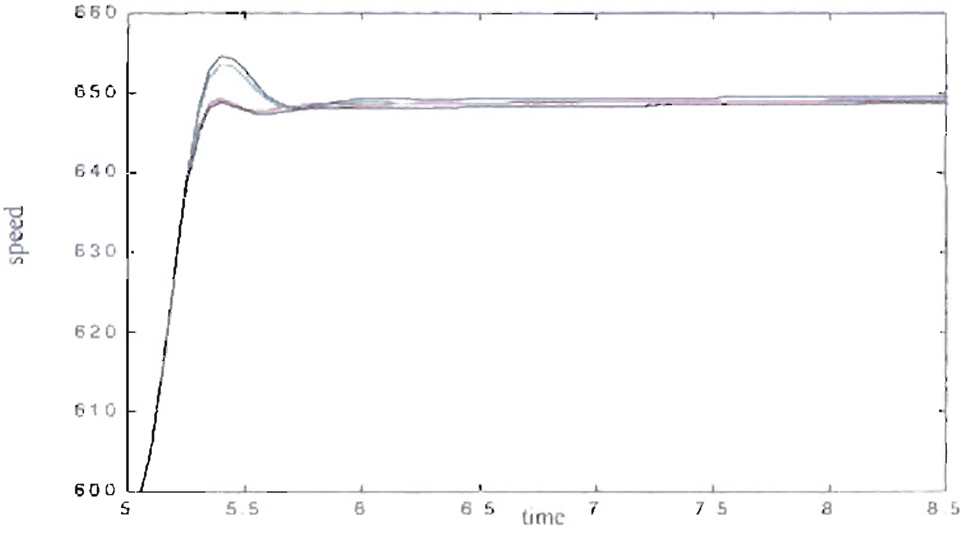


Figure 5.64: *Detail transition from 600 to 650 rpm. Inertia = 3 lb-ft-sec^2.*[2]

1 See Note 1 on page 106
2 See Note 1 on page 106

For the next case we repeated the training for different engine inertia I; changing the fitness function to percent overshoot. We used the same three training conditions described on page 102. For that fitness function we obtained the results shown in Table 5.9, Table 5.10 and Table 5.11. From those tables we can see that the percentage improvement depends on the initial overshoot. More impressive results were obtained with more initial overshoot and oscillatory responses. However, curious results were found for an inertia of 1 lb-ft-sec^2. For that inertia value we obtained very good results for two of the six possible training conditions, as seen in the first row of Table 5.9 and Table 5.10. Due to the selected fitness function, we can see from the Figure 5.65 to Figure 5.71 that the resulting responses tend to be flatter. If we compare the training processes, we obtained a better response for large populations versus small populations when the search range was 1/100 as seen in Table 5.9. As the search range increased, we cannot see a clear advantage for either population size. For this fitness function we can see that with smaller searching range we can obtain better results with larger populations because the recombination could be greater and the genetic algorithm could find controller combinations that stabilize the original system. As the searching range increases, the population size became a less important factor.

Table 5.9: *Engine with different inertia. Controller's results for percent overshoot based fitness. Searching step 1/100. Error calculation after t_1 = 1 second.*

Inertia *lb-ft-sec^2.*	**Initial Fitness (overshoot)**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 50**
1	162.3742	108.6194 (66.89 %)	poles=0.9983 and 0.2522 zeros=0.9840 and 0.2678 gain = 2.0321	0.9894 (0.61 %)	poles=1.0000 and 0.2557 zeros=0.9967 and 0.2820 gain = 2.0442
1.4	68.6588	53.3478 (77.70 %)	poles=1.0000 and 0.2555 zeros=0.9949 and 0.3385 gain = 1.9083	1.2924 (1.88 %)	poles=1.0000 and 0.2466 zeros=0.9895 and 0.3154 gain = 1.9951
1.8	43.3333	31.2830 (72.19 %)	poles=1.0000 and 0.2554 zeros=0.9943 and 0.3396 gain = 1.8816	0.4040 (0.92 %)	poles=1.0000 and 0.2464 poles=0.9886 and 0.3079 gain = 2.0192
2.2	29.7008	21.4776 (72.29 %)	poles=1.0000 and 0.2528 zeros=0.9942 and 0.3262 gain = 1.9461	0.4233 (1.41 %)	poles=1.0000 and 0.2460 zeros=0.9943 and 0.3009 gain = 1.9984
2.6	18.9070	9.8514 (52.12 %)	poles=1.0000 and 0.2521 zeros=0.9945 and 0.3334 gain = 1.8775	0.3329 (1.75 %)	poles=1.0000 and 0.2471 zeros=0.9924 and 0.3009 gain = 1.9936
3	12.4636	6.1016 (48.96 %)	poles=1.0000 and 0.2493 zeros=0.9909 and 0.3080 gain = 1.9183	0.3842 (3.05 %)	poles=1.0000 and 0.2452 zeros=0.9912 and 0.3016 gain = 2.0062

Table 5.10: *Engine with different inertia. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_l = 1 second.*

Inertia *lb-ft-sec^2.*	**Initial Fitness (overshoot)**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 50**
1	162.3742	0.5374 (0.33 %)	poles = 1.0000 and 0.6679 zeros = 0.9505 and 0.8812 gain = 0.9073	12.3415 (7.60 %)	poles = 1.0000 and 0.2246 zeros = 0.9747 and 0.8016 gain = 1.7781
1.4	68.6588	0.4853 (0.70 %)	poles = 1.0000 and 0.3170 zeros = 0.9683 and 0.7010 gain = 1.6805	18.7830 (27.36 %)	poles = 1.0000 and 0.2938 zeros = 1.0000 and 0.5915 gain = 1.6021
1.8	43.3333	0.4084 (0.92 %)	poles = 1.0000 and 0.3756 zeros = 0.9728 and 0.6239 gain = 1.6596	1.3871 (3.18 %)	poles = 1.0000 and 0.2092 zeros = 0.9793 and 0.5856 gain = 2.0012
2.2	29.7008	0.1698 (0.57 %)	poles = 1.0000 and 0.2003 zeros = 0.9752 and 0.4250 gain = 1.6103	0.7574 (2.53 %)	poles = 1.0000 and 0.2285 zeros = 0.9798 and 0.4822 gain = 1.9294
2.6	18.9070	0.3284 (1.69 %)	poles = 1.0000 and 0.3027 zeros = 0.9800 and 0.4822 gain = 1.9477	1.9389 (10.21 %)	poles = 1.0000 and 0.2216 zeros = 0.9859 and 0.4140 gain = 1.9364
3	12.4636	1.2683 (10.11 %)	poles = 1.0000 and 0.2542 zeros = 0.9856 and 0.4033 gain = 1.9950	0.0570 (0.46 %)	poles = 1.0000 and 0.2510 zeros = 0.9811 and 0.4175 gain = 2.0620

Table 5.11: *Engine with different inertia. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_f = 5 seconds.*

Inertia *lb-ft-sec²*.	**Initial Fitness (overshoot)**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 50**
1	139.49	65.9343 (40.60 %)	poles = 1.0000 and 0.3604 zeros = 1.0000 and 0.4277 gain = 1.8893	26.8802 (16.55 %)	poles = 1.0000 and 0.3191 zeros = 1.0000 and 0.8079 gain = 1.4954
1.4	67.21	1.8923 (2.75 %)	poles = 1.0000 and 0.5036 zeros = 0.9764 and 0.6747 gain = 1.2098	2.1943 (3.19 %)	poles = 1.0000 and 0.2372 zeros = 0.9801 and 0.5811 gain = 1.5248
1.8	43.11	5.7385 (13.22 %)	poles = 1.0000 and 0.5605 zeros = 0.9899 and 0.7116 gain = 1.5506	0.1427 (0.33 %)	poles = 1.0000 and 0.2151 zeros = 0.9698 and 0.5899 gain = 1.7173
2.2	29.78	1.5278 (5.12 %)	poles = 1.0000 and 0.4926 zeros = 0.9819 and 0.6117 gain = 1.5990	0.1213 (0.41 %)	poles = 1.0000 and 0.2394 zeros = 0.9758 and 0.4909 gain = 1.9150
2.6	19.03	0.6882 (3.60 %)	poles = 1.0000 and 0.2410 zeros = 0.9813 and 0.4053 gain = 1.8359	1.7825 (9.42 %)	poles = 1.0000 and 0.2115 zeros = 0.9855 and 0.3710 gain = 1.8451
3	12.75	0.4607 (3.69 %)	poles = 1.0000 and 0.2346 zeros = 0.9828 and 0.3142 gain = 1.6024	0.1711 (1.36 %)	poles = 1.0000 and 0.2122 zeros = 0.9819 and 0.3780 gain = 1.9775

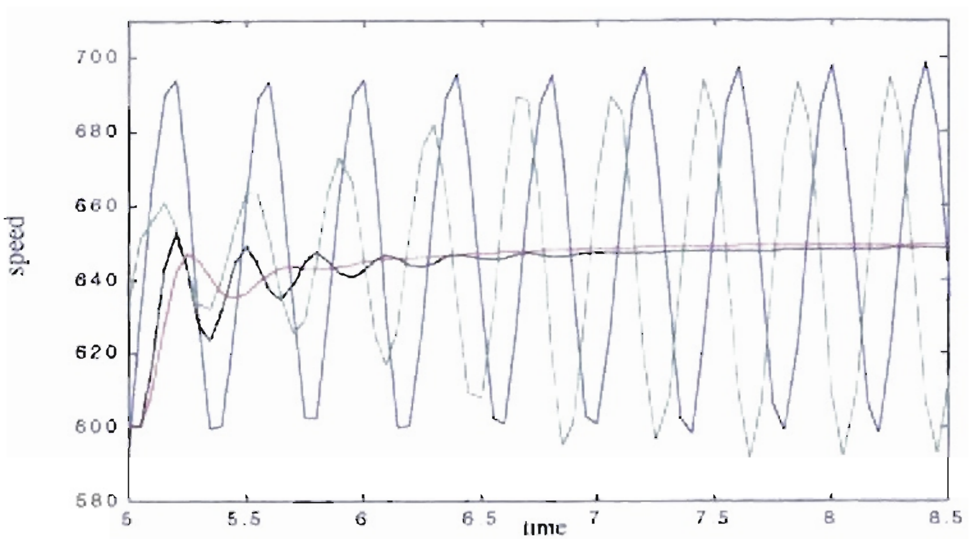


Figure 5.65: *Detail transition from 600 to 650 rpm. Inertia = 1 lb-ft-sec^2.*[1]

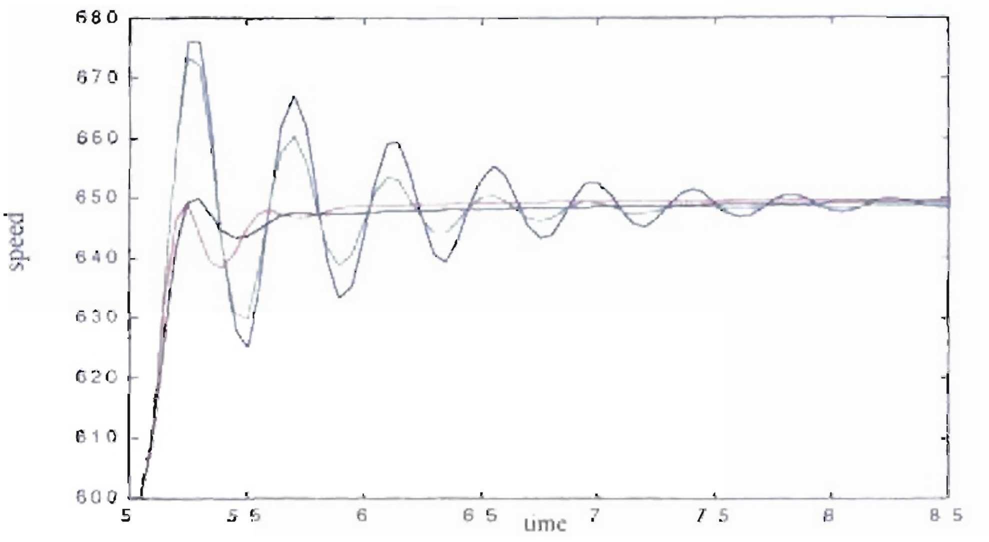


Figure 5.66: *Detail transition from 600 to 650 rpm. Inertia = 1.4 lb-ft-sec^2.*[2]

1. Color codes from Figure 5.65 to Figure 5.70. Blue = Original PID parameters, Green = GENITOR optimized parameters (searching step = 1/100, error calculation after 1 second), Red = GENITOR optimized parameters (searching step = 1/10, error calculation after 1 second), Black = GENITOR optimized parameters (searching step = 1/10, error calculation after 5 seconds)
2. See Note 1

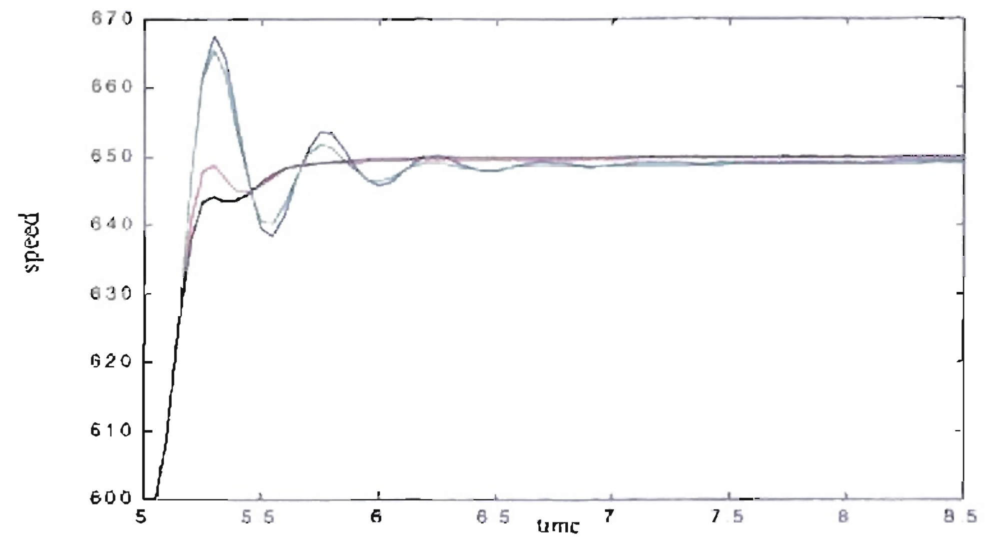


Figure 5.67: *Detail transition from 600 to 650 rpm. Inertia = 1.8 lb-ft-sec²*.[1]

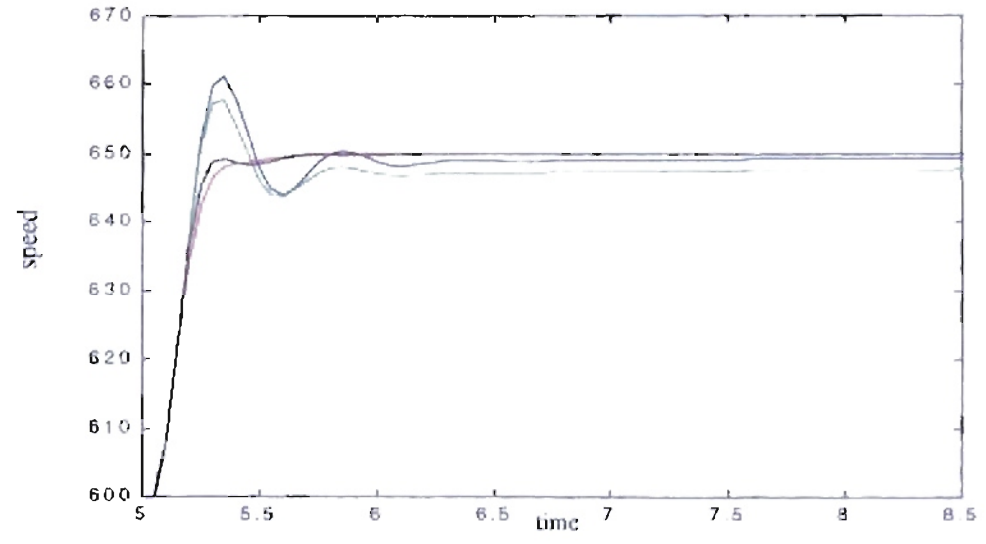


Figure 5.68: *Detail transition from 600 to 650 rpm. Inertia = 2.2 lb-ft-sec²*.[2]

1. See Note 1 on page 113
2. See Note 1 on page 113

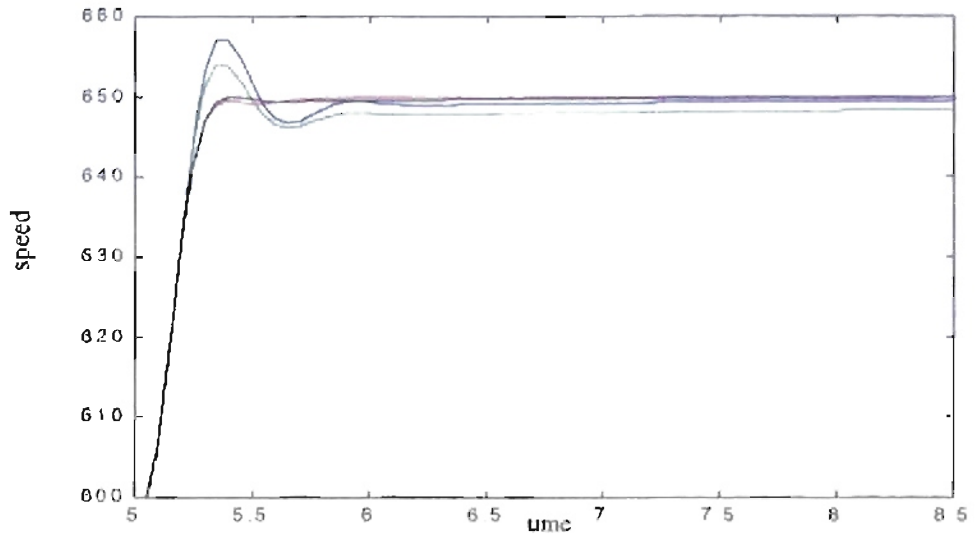


Figure 5.69: *Detail transition from 600 to 650 rpm. Inertia = 2.6 lb-ft-sec²*.[1]

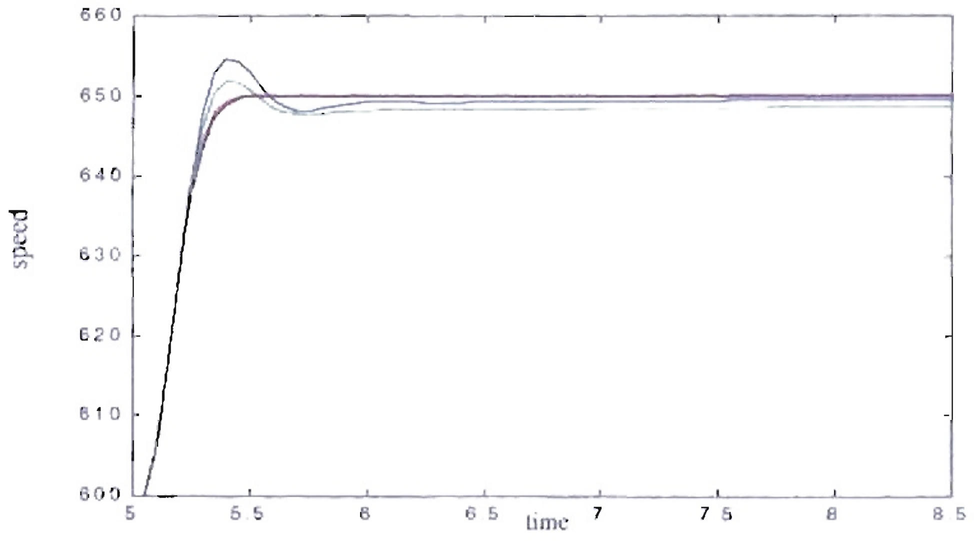


Figure 5.70: *Detail transition from 600 to 650 rpm. Inertia = 3 lb-ft-sec²*.[2]

1. See Note 1 on page 113.
2. See Note 1 on page 113.

5.8.2. Results for different delays.

For the next case we repeated the training for different engine fueling delays. For each case the initial response changed according to the fueling delay value with a fixed engine inertia of 2 lb-ft-sec^2. An optimization is needed to adjust the controller values to match the engine fueling delay. We use the mean square error as our fitness value. Like the previous case, we changed the training scheme to two speed transitions to reduce the training time. The first transition is from 600 rpm to 650 rpm at t_1 seconds. The second transition is from 650 rpm to 600 rpm at t_2 seconds. The engine and the controller are initialized with the engine conditions for 600 rpm. The simulation runs from 0 to t_1 seconds without error measurements, then the training is started. We used the same training conditions described on page 102.

Using percent overshoot as fitness, we obtained the results shown in Table 5.12, Table 5.13 and Table 5.14 for six different fueling delay values and two different population sizes. If we look at the previous tables and Figure 5.71 to Figure 5.76, we notice how the response improves as the fueling delay increases. These results differ from the analog case where the improvement was related to how close we are to the designed engine delay. For the digital case the improvement depends on how far the sampling time is from the engine delay.

Table 5.12: *Engines with different fueling delay. Controller's results for mean square error based fitness. Searching step 1/100. Error calculation after t_l = 1 second.*

Delay (msec)	Initial Fitness $(error)^2$.	Best Fitness $(error)^2$. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness $(error)^2$. Pop. = 50	Resulting Parameters Pop. = 5
30	366.47	356.75 (97.35 %)	poles = 0.9999 and 0.2541 zeros = 0.9766 and 0.2671 gain = 2.1214	358.72 (97.89 %)	poles = 0.9999 and 0.2437 zeros = 0.9787 and 0.2646 gain = 2.0911
50	469.38	461.02 (98.22 %)	poles = 0.9999 and 0.2435 zeros = 0.9760 and 0.2913 gain = 2.0700	464.40 (98.94 %)	poles = 1.0000 and 0.2464 zeros = 0.9824 and 0.2792 gain = 2.0631
70	620.16	582.52 (93.93 %)	poles = 1.0000 and 0.2417 zeros = 0.9862 and 0.3319 gain = 1.9743	595.76 (96.07 %)	poles = 1.0000 and 0.2452 zeros = 0.9866 and 0.3072 gain = 2.0248
90	852.29	753.27 (88.38 %)	poles = 1.0000 and 0.2487 zeros = 0.9914 and 0.3190 gain = 1.9314	788.65 (92.53 %)	poles = 1.0000 and 0.2448 zeros = 0.9906 and 0.2971 gain = 2.0059
110	1174.68	917.32 (78.09 %)	poles = 1.0000 and 0.2434 zeros = 0.9920 and 0.3193 gain = 1.8869	1012.52 (86.20 %)	poles = 1.0000 and 0.2473 zeros = 0.9948 and 0.3061 gain = 2.0059
130	1826.88	906.89 (49.64 %)	poles = 0.9999 and 0.5409 zeros = 0.9867 and 0.6698 gain = 1.6632	1145.11 (62.68 %)	poles = 1.0000 and 0.2601 zeros = 1.0000 and 0.3690 gain = 2.0070

Table 5.13: *Engines with different fueling delay. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_i = 1 second.*

Delay (msec)	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 5
30	366.47	330.92 (90.30 %)	poles = 0.9999 and 0.2928 zeros = 0.9744 and 0.4223 gain = 3.1151	346.39 (94.52 %)	poles = 0.9999 and 0.2632 zeros = 0.9695 and 0.3413 gain = 2.3733
50	469.38	440.23 (93.79 %)	poles = 0.9999 and 0.2248 zeros = 0.9667 and 0.4543 gain = 2.5945	446.18 (95.06 %)	poles = 0.9999 and 0.2111 zeros = 0.9703 and 0.4060 gain = 2.3017
70	620.16	549.66 (88.63 %)	poles = 0.9999 and 0.2903 zeros = 0.9763 and 0.5293 gain = 2.1874	546.97 (88.20 %)	poles = 0.9998 and 0.2038 zeros = 0.9725 and 0.4642 gain = 2.2211
90	852.29	649.03 (76.15 %)	poles = 0.9999 and 0.3609 zeros = 0.9604 and 0.6315 gain = 1.9741	654.83 (76.83 %)	poles = 1.0000 and 0.2067 zeros = 0.9700 and 0.5089 gain = 1.9925
110	1174.68	773.43 (65.84 %)	poles = 0.9998 and 0.6344 zeros = 0.9701 and 0.7757 gain = 1.5593	753.77 (64.17 %)	poles = 0.9999 and 0.2068 zeros = 0.9771 and 0.5476 gain = 1.9324
130	1826.88	848.56 (46.45 %)	poles = 0.9998 and 0.6606 zeros = 0.9510 and 0.8409 gain = 1.5581	904.30 (49.50 %)	poles = 1.0000 and 0.2939 zeros = 0.9966 and 0.5812 gain = 1.8480

Table 5.14: *Engines with different fueling delay. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_l = 5 seconds.*

Delay (msec)	Initial Fitness $(\text{error})^2$.	Best Fitness $(\text{error})^2$. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness $(\text{error})^2$. Pop. = 50	Resulting Parameters Pop. = 5
30	356.55	326.98 (89.22 %)	poles = 1.0000 and 0.3025 zeros = 0.9784 and 0.4026 gain = 2.9638	342.03 (93.33 %)	poles = 1.0000 and 0.2122 zeros = 0.9753 and 0.2732 gain = 2.3387
50	460.79	439.38 (93.61 %)	poles = 1.0000 and 0.3026 zeros = 0.9746 and 0.4981 gain = 2.6599	441.08 (93.97 %)	poles = 1.0000 and 0.2045 zeros = 0.9786 and 0.3824 gain = 2.4052
70	605.47	564.85 (91.08 %)	poles = 1.0000 and 0.6058 zeros = 0.9722 and 0.7341 gain = 2.0435	558.03 (89.98 %)	poles = 1.0000 and 0.2080 zeros = 0.9814 and 0.3780 gain = 2.0742
90	825.48	653.89 (76.72 %)	poles = 1.0000 and 0.2347 zeros = 0.9794 and 0.5629 gain = 2.2681	658.07 (77.21 %)	poles = 1.0000 and 0.2119 zeros = 0.9823 and 0.4970 gain = 2.0754
110	1255.27	784.63 (66.80 %)	poles = 1.0000 and 0.4066 zeros = 0.9891 and 0.6341 gain = 1.7960	767.37 (65.33 %)	poles = 1.0000 and 0.2652 zeros = 0.9743 and 0.6085 gain = 2.1472
130	2569.77	897.51 (49.13 %)	poles = 1.0000 and 0.4511 zeros = 0.9874 and 0.6877 gain = 1.6634	966.48 (52.90 %)	poles = 1.0000 and 0.2764 zeros = 1.0000 and 0.5938 gain = 1.9129

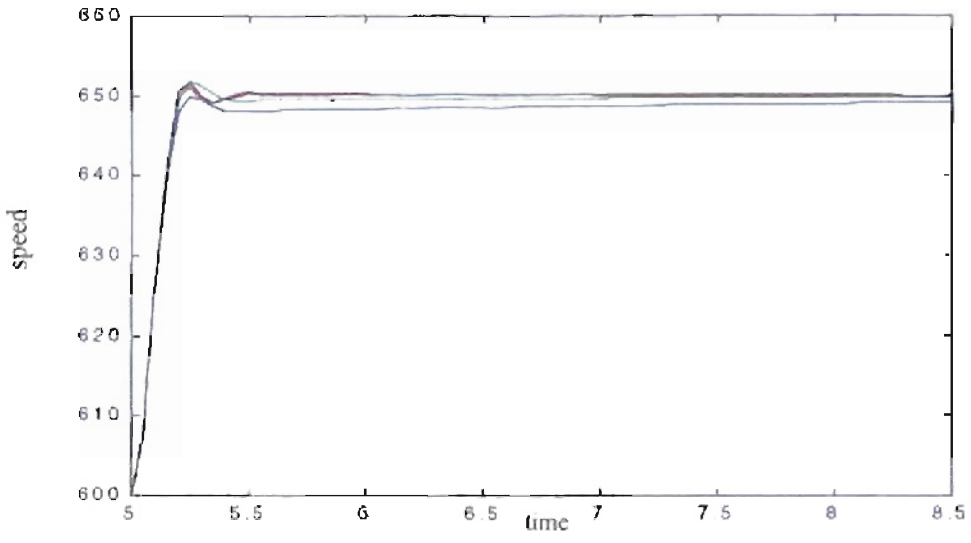


Figure 5.71: *Detail transition from 600 to 650 rpm. Fueling delay = 30 msec.* [1]

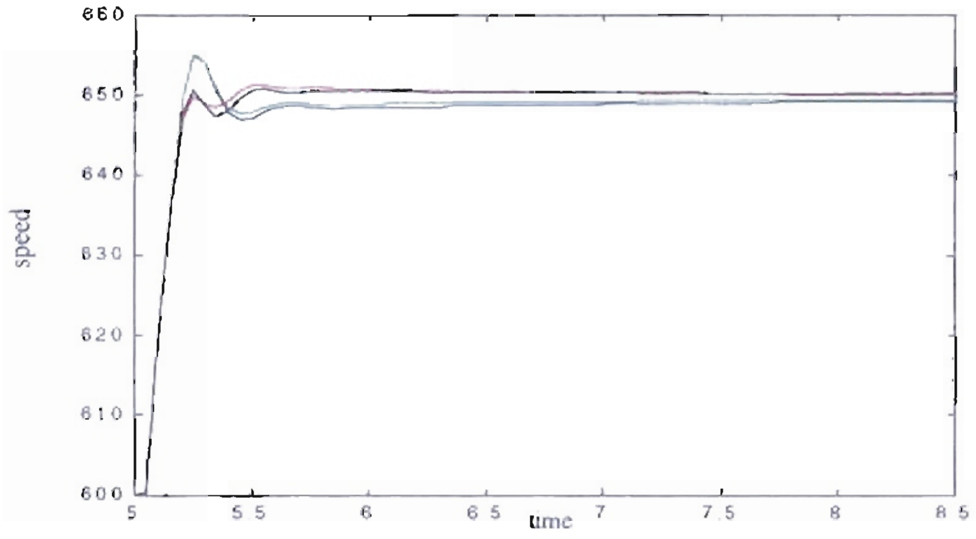


Figure 5.72: *Detail transition from 600 to 650 rpm. Fueling delay = 50 msec.* [2]

1. Color codes from Figure 5.71 to Figure 5.76: Blue = Original PID parameters, Green = GENITOR optimized parameters (searching step = 1/100, error calculation after 1 second), Red = GENITOR optimized parameters (searching step = 1/10, error calculation after 1 second), Black = GENITOR optimized parameters (searching step = 1/10, error calculation after 5 seconds)
2. See Note 1.

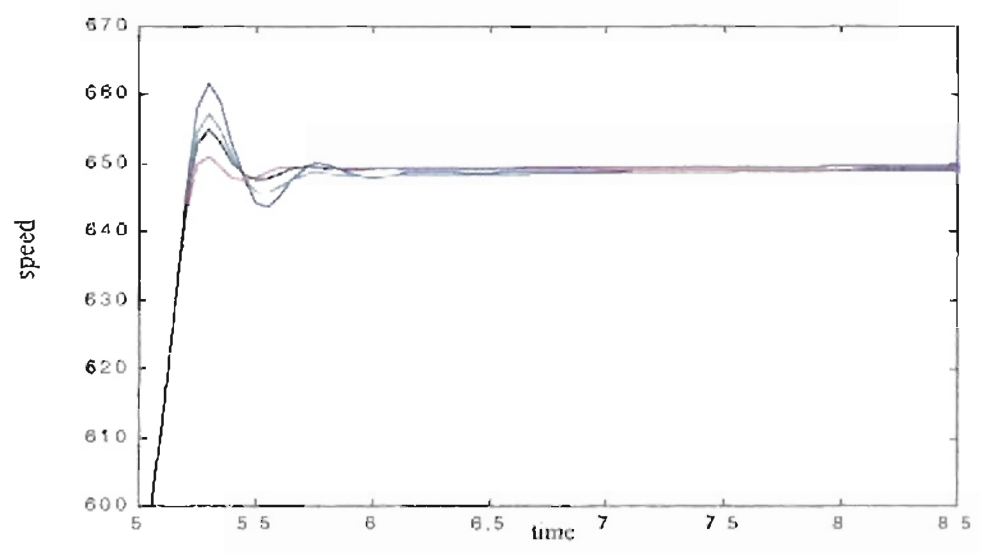


Figure 5.73: *Detail transition from 600 to 650 rpm. Fueling delay = 70 msec.* [1]

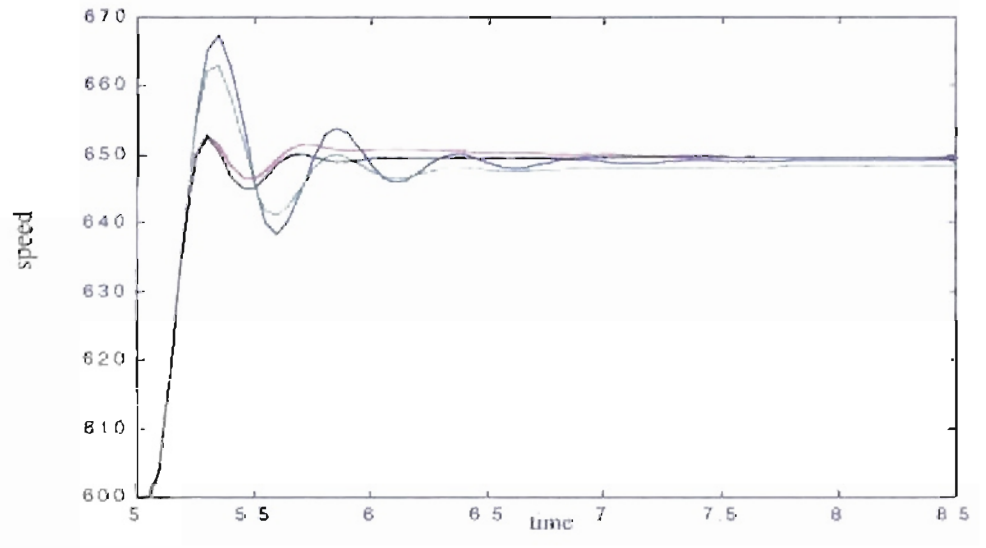


Figure 5.74: *Detail transition from 600 to 650 rpm. Fueling delay = 90 msec.* [2]

1 See Note 1 on page 120.
2 See Note 1 on page 120.

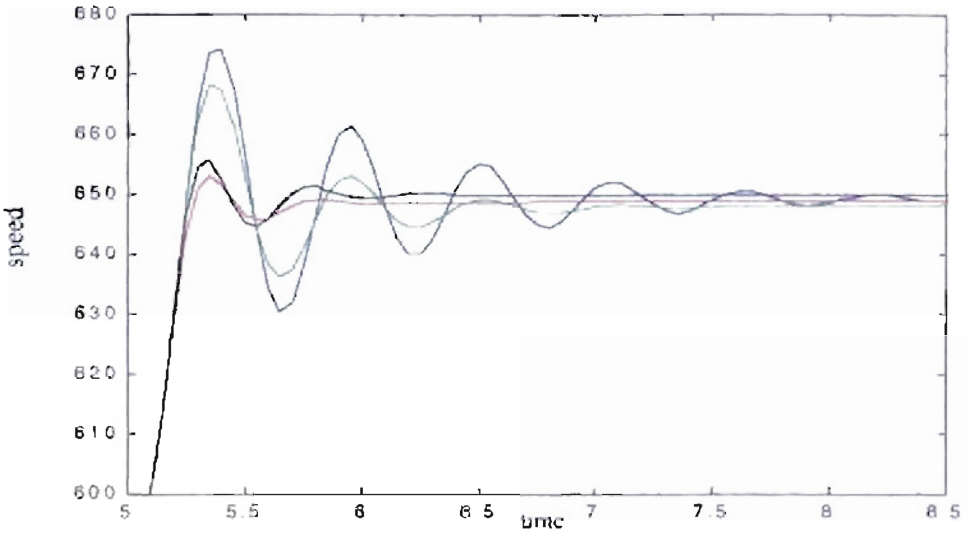


Figure 5.75: *Detail transition from 600 to 650 rpm. Fueling delay = 110 msec.* [1]

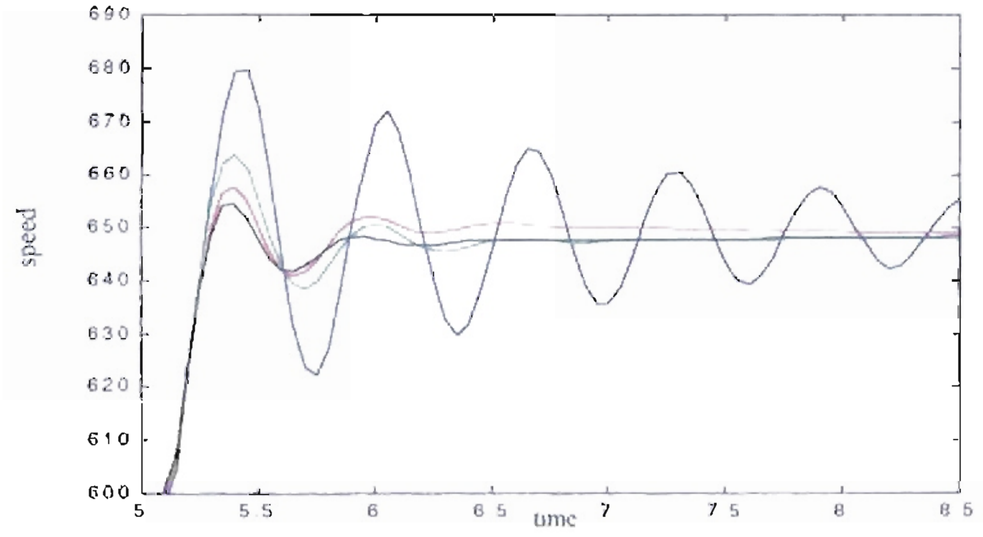


Figure 5.76: *Detail transition from 600 to 650 rpm. Fueling delay = 130 msec.* [2]

1 See Note 1 on page 120

2 See Note 1 on page 120

For the next case we repeated the training for different fueling delays, changing the fitness function to percent overshoot. For that fitness and the same training conditions on page 102 we obtained the results shown in Table 5.15, Table 5.16 and Table 5.17. We obtained better results for large searching steps. As we can see from Figure 5.77 to Figure 5.82 the resulting responses tend to be flatter.

Table 5.15: *Engines with different fueling delay. Controller's results for percent overshoot based fitness. Searching step 1/100. Error calculation after t_1 = 1 second.*

Delay (msec)	Initial Fitness (overshoot).	Best Fitness (overshoot). Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (overshoot). Pop. = 50	Resulting Parameters Pop. = 5
30	4.3475	2.3789 (54.61 %)	poles = 1.0000 and 0.2509 zeros = 0.9806 and 0.2724 gain = 2.0402	2.7069 (62.21 %)	poles = 1.0000 and 0.2507 zeros = 0.9822 and 0.2703 gain = 2.0444
50	14.9131	6.4058 (42.92 %)	poles = 1.0000 and 0.2510 zeros = 0.9899 and 0.3256 gain = 1.9209	10.0248 (67.20 %)	poles = 1.0000 and 0.2468 zeros = 0.9875 and 0.3005 gain = 2.0035
70	29.6319	21.3643 (72.09 %)	poles = 1.0000 and 0.2482 zeros = 0.9934 and 0.3104 gain = 1.9229	22.1842 (74.86 %)	poles = 1.0000 and 0.2512 zeros = 0.9901 and 0.3236 gain = 1.9987
90	44.8800	35.5064 (79.10 %)	poles = 1.0000 and 0.2583 zeros = 0.9930 and 0.3283 gain = 1.9630	37.0737 (82.60 %)	poles = 1.0000 and 0.2474 zeros = 0.9914 and 0.3141 gain = 2.0020
110	61.4867	51.0985 (83.10 %)	poles = 1.0000 and 0.2502 zeros = 0.9953 and 0.3135 gain = 1.9507	54.1569 (88.08 %)	poles = 1.0000 and 0.2459 zeros = 0.9948 and 0.3043 gain = 2.0215
130	76.6177	61.2543 (79.95 %)	poles = 1.0000 and 0.2565 zeros = 0.9941 and 0.3062 gain = 1.9624	63.2429 (82.55 %)	poles = 1.0000 and 0.2479 zeros = 0.9947 and 0.2959 gain = 2.0157

Table 5.16: *Engines with different fueling delay. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_f = 1 second.*

Delay (msec)	Initial Fitness (overshoot).	Best Fitness (overshoot). Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (overshoot). Pop. = 50	Resulting Parameters Pop. = 5
30	4.3475	0.2178 (4.84 %)	poles = 1.0000 and 0.3183 zeros = 0.9727 and 0.3663 gain = 1.8949	0.0143 (0.33 %)	poles = 1.0000 and 0.2447 zeros = 0.9710 and 0.3459 gain = 2.1505
50	14.9131	0.3430 (2.28 %)	poles = 1.0000 and 0.2397 zeros = 0.9746 and 0.4069 gain = 1.9280	0.0445 (0.30 %)	poles = 1.0000 and 0.2039 zeros = 0.9720 and 0.3828 gain = 1.8947
70	29.6319	1.2199 (4.08 %)	poles = 1.0000 and 0.2344 zeros = 0.9801 and 0.5061 gain = 1.9830	0.6890 (2.33 %)	poles = 1.0000 and 0.2333 zeros = 0.9766 and 0.4753 gain = 1.8643
90	44.8800	0.2175 (0.47 %)	poles = 1.0000 and 0.3508 zeros = 0.9754 and 0.6217 gain = 1.7957	2.3105 (5.15 %)	poles = 1.0000 and 0.2218 zeros = 0.9809 and 0.5246 gain = 1.8775
110	61.4867	5.4758 (8.90 %)	poles = 1.0000 and 0.6538 zeros = 0.9883 and 0.8104 gain = 1.4423	5.4389 (8.83 %)	poles = 1.0000 and 0.2414 zeros = 0.9893 and 0.6358 gain = 1.8963
130	76.6177	1.2039 (1.57 %)	poles = 1.0000 and 0.5512 zeros = 0.9776 and 0.7903 gain = 1.4636	18.6688 (24.36 %)	poles = 1.0000 and 0.3802 zeros = 1.0000 and 0.6496 gain = 1.6727

Table 5.17: *Engines with different fueling delay. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_f = 5 seconds.*

Delay (msec)	**Initial Fitness (overshoot).**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 5**
30	2.57	0.3038 (6.91 %)	poles = 1.0000 and 0.2458 zeros = 0.9769 and 0.3113 gain = 2.0756	0.1207 (2.76 %)	poles = 1.0000 and 0.2387 zeros = 0.9733 and 0.3391 gain = 2.0389
50	14.94	0.0955 (0.67 %)	poles = 1.0000 and 0.3265 zeros = 0.9740 and 0.4675 gain = 1.9196	0.3556 (2.41 %)	poles = 1.0000 and 0.2205 zeros = 0.9764 and 0.3774 gain = 1.9353
70	29.01	0.3626 (1.21 %)	poles = 1.0000 and 0.4230 zeros = 0.9744 and 0.5672 gain = 1.6055	1.3787 (4.66 %)	poles = 1.0000 and 0.1995 zeros = 0.9816 and 0.4363 gain = 1.8658
90	44.50	0.8586 (1.92 %)	poles = 1.0000 and 0.3579 zeros = 0.9772 and 0.6048 gain = 1.7043	2.2145 (4.92 %)	poles = 1.0000 and 0.2282 zeros = 0.9835 and 0.5459 gain = 1.8711
110	60.90	12.3427 (20.07 %)	poles = 1.0000 and 0.7187 zeros = 0.9954 and 0.8768 gain = 1.5469	17.1137 (27.83 %)	poles = 1.0000 and 0.2670 zeros = 1.0000 and 0.5511 gain = 1.7916
130	86.72	2.9630 (3.86 %)	poles = 1.0000 and 0.7746 zeros = 0.9792 and 0.9091 gain = 1.2471	1.5727 (2.05 %)	poles = 1.0000 and 0.2156 zeros = 0.9799 and 0.6306 gain = 1.7764

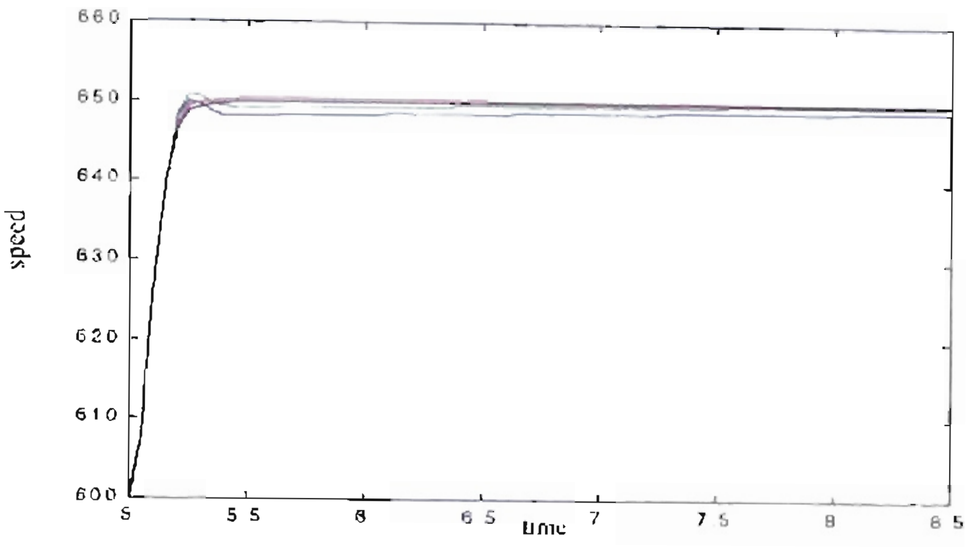


Figure 5.77: *Detail transition from 600 to 650 rpm. Fueling delay = 30 msec.* [1]

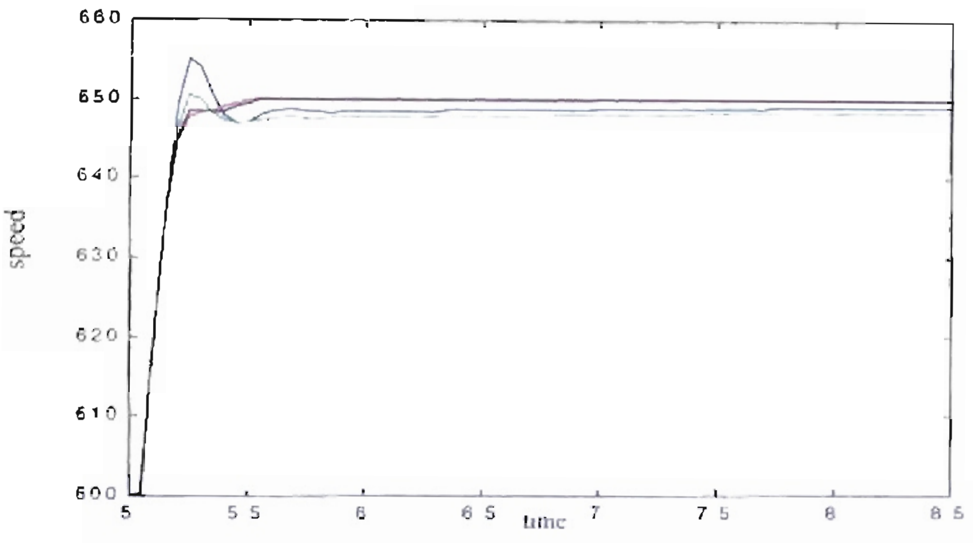


Figure 5.78: *Detail transition from 600 to 650 rpm. Fueling delay = 50 msec.* [2]

1. Color codes from Figure 5.77 to Figure 5.82. Blue = Original PID parameters, Green = GENITOR optimized parameters (searching step = 1/100, error calculation after 1 second), Red = GENITOR optimized parameters (searching step = 1/10, error calculation after 1 second), Black = GENITOR optimized parameters (searching step = 1/10, error calculation after 5 seconds)
2. See Note 1

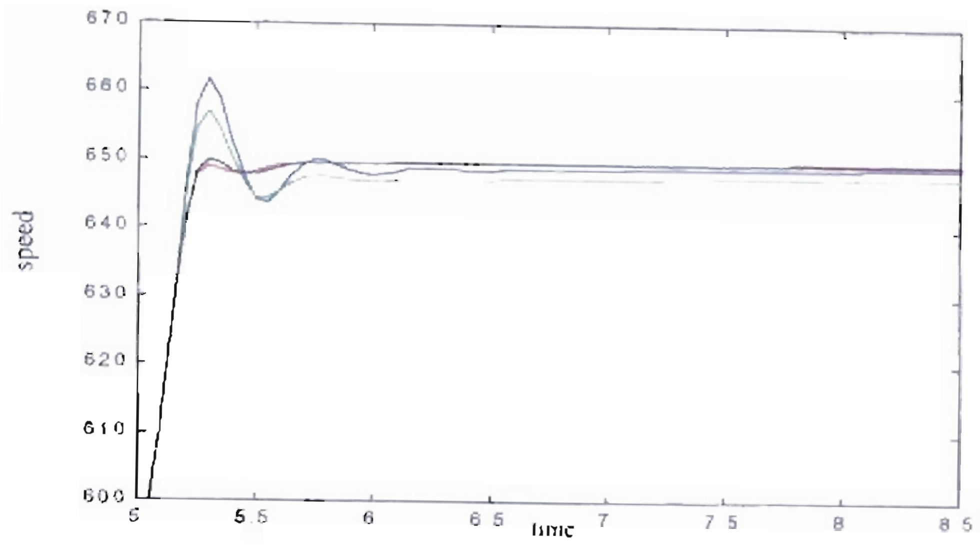


Figure 5.79: *Detail transition from 600 to 650 rpm. Fueling delay = 70 msec.* [1]

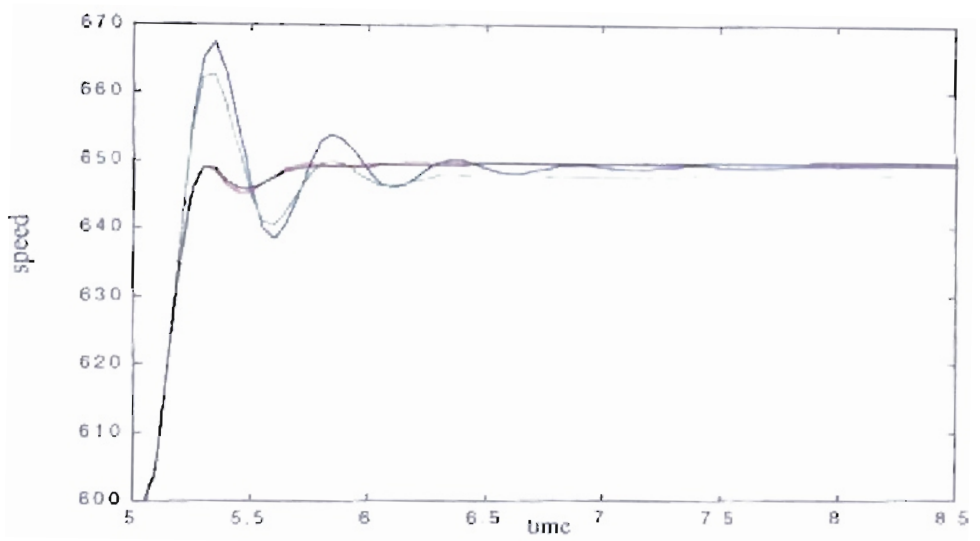


Figure 5.80: *Detail transition from 600 to 650 rpm. Fueling delay = 90 msec.* [2]

1 See Note 1 on page 127.
2 See Note 1 on page 127

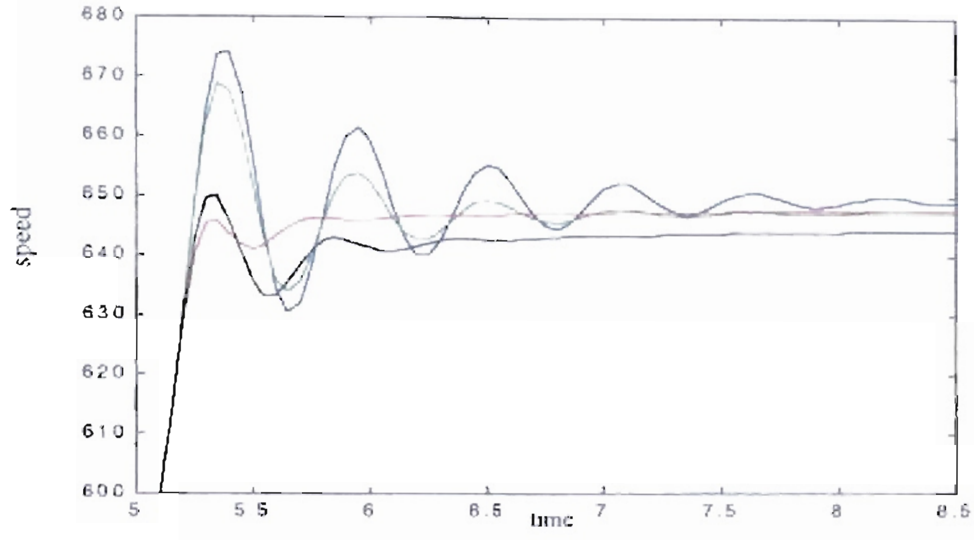


Figure 5.81: *Detail transition from 600 to 650 rpm. Fueling delay = 110 msec.*[1]

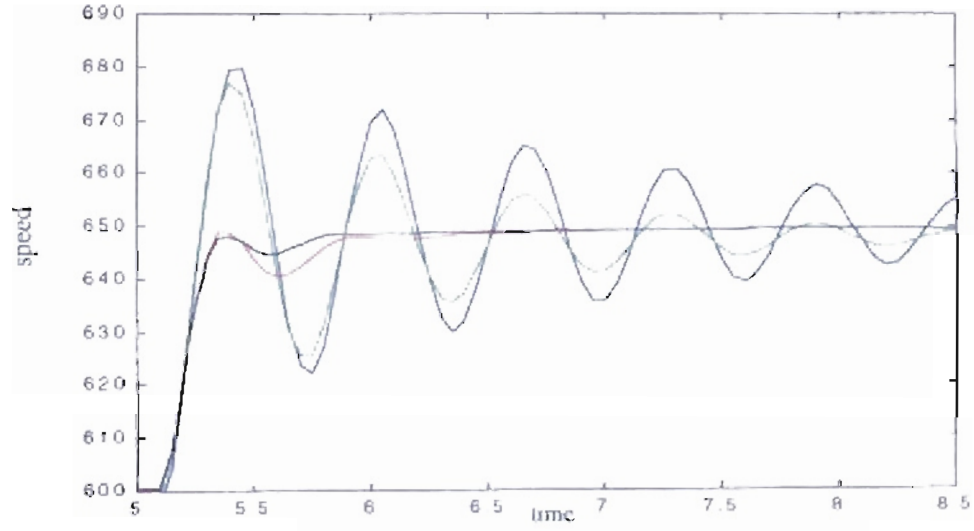


Figure 5.82: *Detail transition from 600 to 650 rpm. Fueling delay = 130 msec.*[2]

1 See Note 1 on page 127.
2 See Note 1 on page 127

5.9. GENITOR algorithm applied to digital controller and engine with friction b_l=0.01.

5.9.1. Results for different engine inertia.

For the next case we used the same $b_l = 0.01$ used in the analog training from section 5.6.1 and section 5.6.2. As in section 5.6.1, we executed the training for different engine inertias. For each case the initial response changed according to the engine inertia value. An optimization is needed to adjust the controller values to match the engine inertia. We use the mean square error as our fitness value. To reduce the training time we changed the training scheme to two speed transitions. The first transition is from 600 to 650 rpm at 5 seconds. The second transition is from 650 to 600 at 8.5 seconds. The engine and the controller are initialized with the engine conditions for 600 rpm. The simulation runs from 0 to 5 seconds without error measurements, then the training is started. To ensure the correct values of the parameters for future training, we changed the mutation process in the digital GENITOR algorithm by taking only poles or zeros with magnitudes inside the unit circle.

For the mean square error as fitness we obtained the results shown in Table 5.18 for six different engine inertia values and two different population sizes. For very oscillatory engine responses we could reduce the mean square error to 1.4 % of its original value for inertia equal to 1 lb-ft-sec^2, as seen in first row of Table 5.18 and Figure 5.83. As inertia increases, the mean square error decreases to 31 % of its initial value for inertia equal to 1.4 lb-ft-sec^2 (see second row of Table 5.18 and Figure 5.84). In the last four rows of

Table 5.18 we can see that the mean square error have values ranging from 70 % to 87 % of their original values for inertia between 1.8 lb-ft-sec^2 and 3 lb-ft-sec^2 (see also Figure 5.85, Figure 5.86, Figure 5.87 and Figure 5.88). As the analog case for $b_1 = 0.01$, if we compare the results related to the population size we note small differences in the results for 1500 epochs.

Table 5.18: *Engine with different inertia. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_i = 5 seconds.*

Inertia lb-ft-sec^2.	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 50
1	41362.21	595.87 (1.44 %)	poles = 1.0000 and 0.5765 zeros = 1.0000 and 0.7562 gain = 1.0489	583.25 (1.41 %)	poles = 1.0000 and 0.2718 zeros = 1.0000 and 0.7326 gain = 1.4687
1.4	1770.17	566.82 (32.02 %)	poles = 1.0000 and 0.3529 zeros = 1.0000 and 0.6580 gain = 1.6688	550.06 (31.07 %)	poles = 1.0000 and 0.2102 zeros = 1.0000 and 0.6024 gain = 1.8424
1.8	854.25	600.88 (70.34 %)	poles = 1.0000 and 0.6008 zeros = 1.0000 and 0.7601 gain = 1.7895	550.49 (64.44 %)	poles = 1.0000 and 0.2079 zeros = 1.0000 and 0.5991 gain = 2.3580
2.2	714.42	587.43 (82.22 %)	poles = 1.0000 and 0.5017 zeros = 1.0000 and 0.7209 gain = 2.3727	567.40 (79.42 %)	poles = 1.0000 and 0.3332 zeros = 1.0000 and 0.6384 gain = 2.5141
2.6	687.65	594.85 (86.50 %)	poles = 1.0000 and 0.5523 zeros = 1.0000 and 0.7423 gain = 2.6958	581.59 (84.58 %)	poles = 1.0000 and 0.3887 zeros = 1.0000 and 0.6211 gain = 2.6407
3	697.86	608.82 (87.24 %)	poles = 1.0000 and 0.6515 zeros = 1.0000 and 0.7936 gain = 2.8784	606.42 (86.90 %)	poles = 1.0000 and 0.5717 zeros = 1.0000 and 0.7093 gain = 2.6972

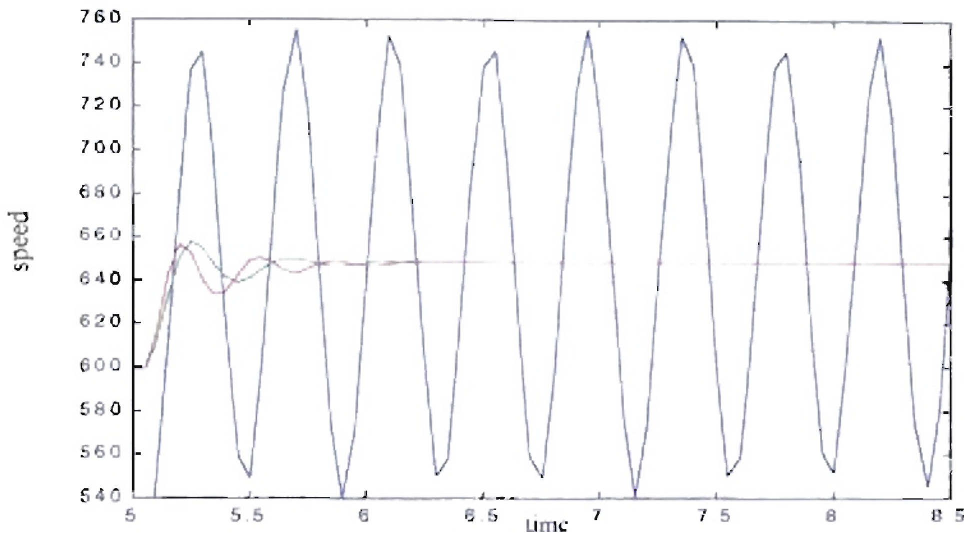


Figure 5.83: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters. Green = Genitor optimized parameters (Population = 5). Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 1 lb-ft-sec^2.*

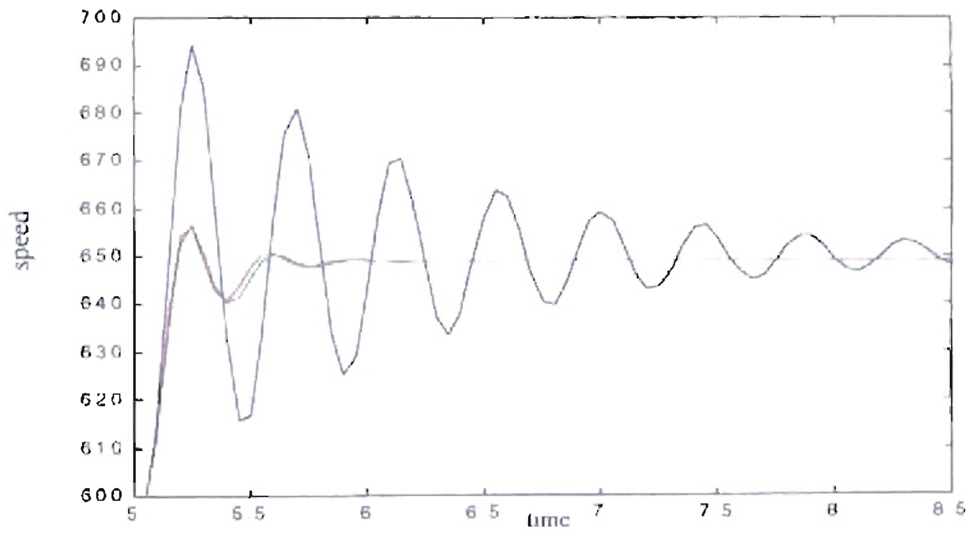


Figure 5.84: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters. Green = Genitor optimized parameters (Population = 5). Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 1.4 lb-ft-sec^2.*

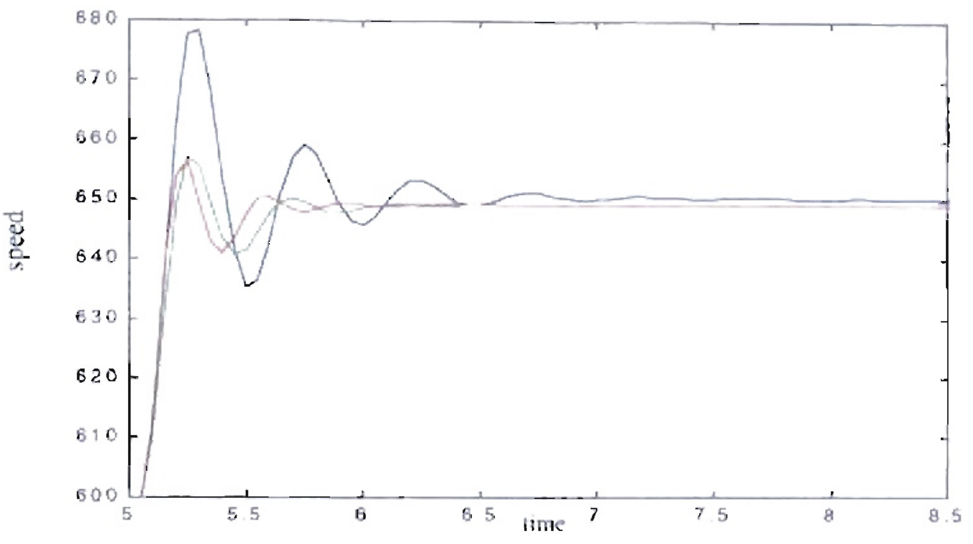


Figure 5.85: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 1.8 lb-ft-sec^2.*

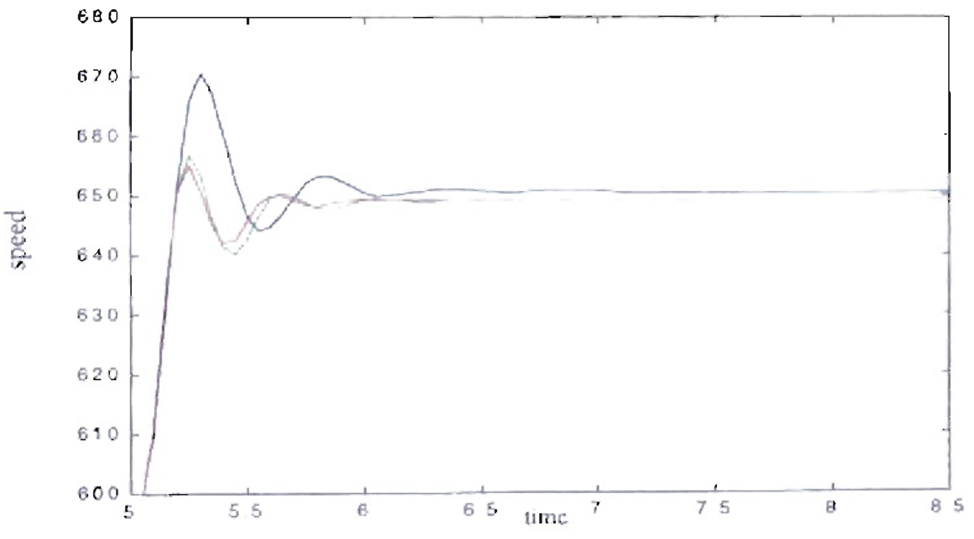


Figure 5.86: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 2.2 lb-ft-sec^2.*

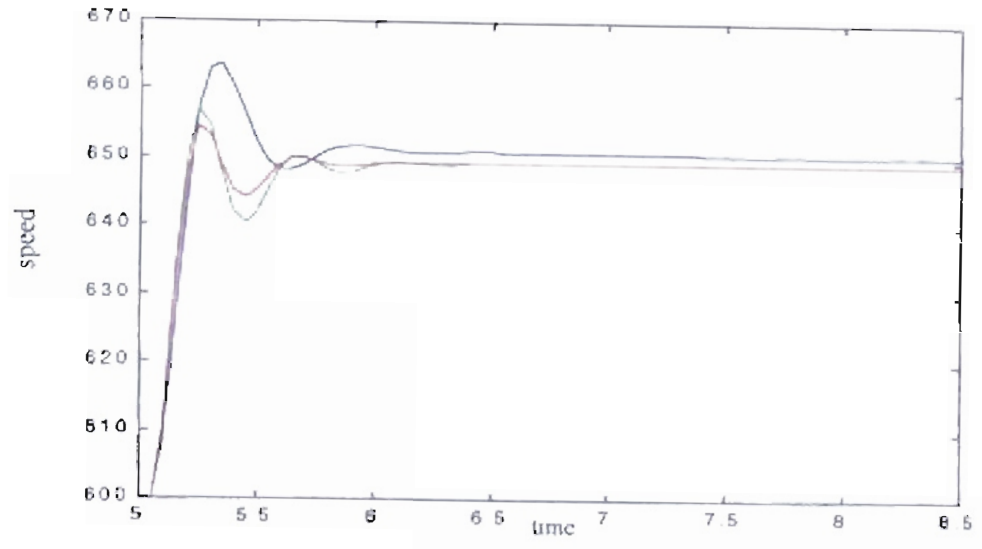


Figure 5.87: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 2.6 lb-ft-sec^2.*

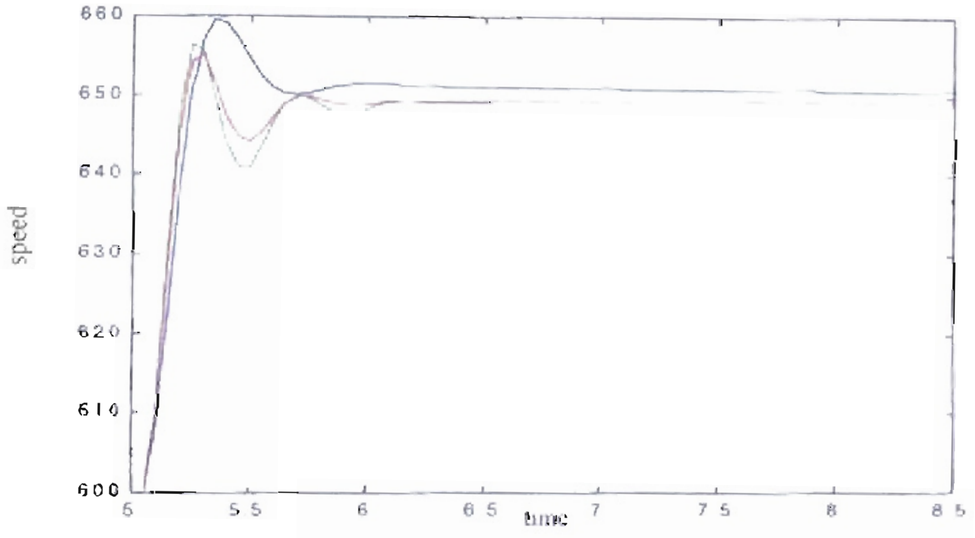


Figure 5.88: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 3 lb-ft-sec^2.*

For the next case we repeated the training for different engine inertias, changing the fitness function to percent overshoot. For that fitness we obtained the results shown in Table 5.19 and Figure 5.89 to Figure 5.94. Due to the selected fitness function, we can see that the resulting responses tend to be flatter. Also, the final response barely passes the required speed of 650 rpm. However an "undershoot" is generated below the required engine speed. In Figure 5.89 we can see an special case for inertia = 1 *lb-ft-sec*^{2}, where the training for a population of 5 was unable to obtain a stable response for the engine. For that case, the genetic algorithm was trapped in a local minimal. In Figure 5.92 and Figure 5.94 we can see a case where an overtraining problem occurred. Here the genetic algorithm reduced dramatically the overshoot, however the final response was too slow.

Table 5.19: *Engine with different inertia. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_f = 5 seconds.*

Inertia *lb-ft-sec^2*.	**Initial Fitness (overshoot)**	**Best Fitness (overshoot). Pop. = 5**	**Resulting Parameters Pop. = 5**	**Best Fitness (overshoot). Pop. = 50**	**Resulting Parameters Pop. = 50**
1	387.7173	203.4080 (52.46 %)	poles = 0.9956 and 0.2663 zeros = 0.9806 and 0.3060 gain = 1.8989	3.1312 (0.55 %)	poles = 1.0000 and 0.3968 zeros = 1.0000 and 0.8487 gain = 1.3042
1.4	87.7286	2.6900 (3.07 %)	poles = 1.0000 and 0.5893 zeros = 1.0000 and 0.8868 gain = 1.5138	2.0396 (2.32 %)	poles = 1.0000 and 0.3644 zeros = 1.0000 and 0.8134 gain = 1.7986
1.8	57.7658	2.3195 (4.02 %)	poles = 1.0000 and 0.7159 zeros = 1.0000 and 0.8798 gain = 1.5622	1.8719 (3.24 %)	poles = 1.0000 and 0.3654 zeros = 1.0000 and 0.7327 gain = 2.0497
2.2	41.5310	0.2195 (0.53 %)	poles = 1.0000 and 0.6332 zeros = 0.9971 and 0.8766 gain = 1.5167	1.8974 (4.57 %)	poles = 1.0000 and 0.4606 zeros = 1.0000 and 0.7373 gain = 2.1905
2.6	28.2821	1.6820 (5.95 %)	poles = 1.0000 and 0.8519 zeros = 0.9995 and 0.9287 gain = 1.7639	0.6776 (2.40 %)	poles = 1.0000 and 0.5199 zeros = 0.9973 and 0.7455 gain = 2.3595
3	19.9864	0.1737 (0.87 %)	poles = 1.0000 and 0.7620 zeros = 0.9973 and 0.9063 gain = 2.2832	0.4181 (2.09 %)	poles = 1.0000 and 0.2019 zeros = 0.9973 and 0.8231 gain = 2.3118

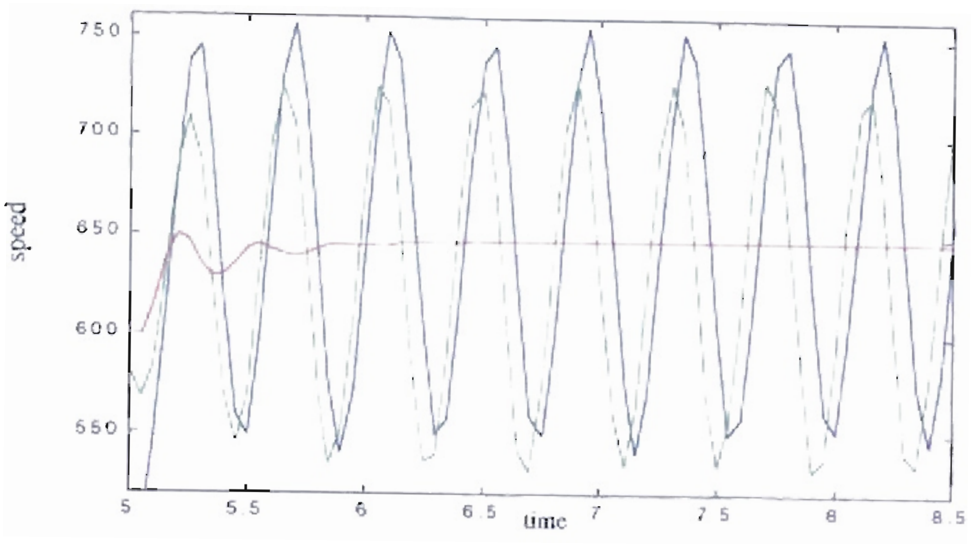


Figure 5.89: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 1 lb-ft-sec^2.*

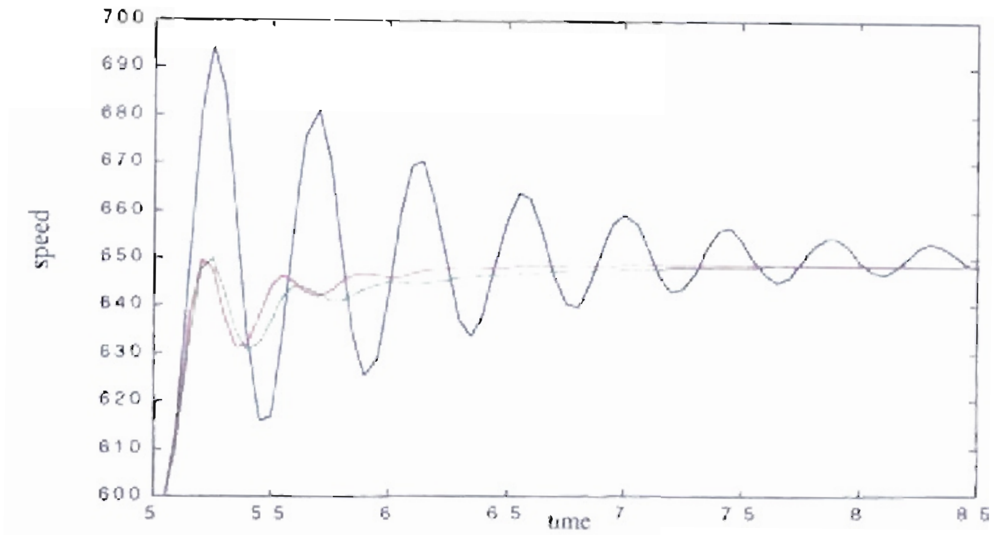


Figure 5.90: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Inertia 1.4 lb-ft-sec^2.*

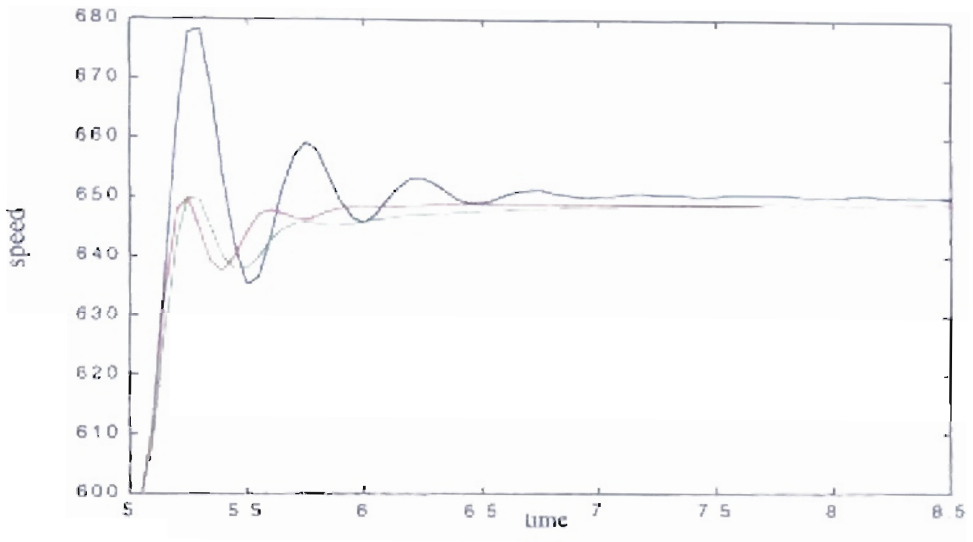


Figure 5.91: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 1.8 lb-ft-sec^2.*

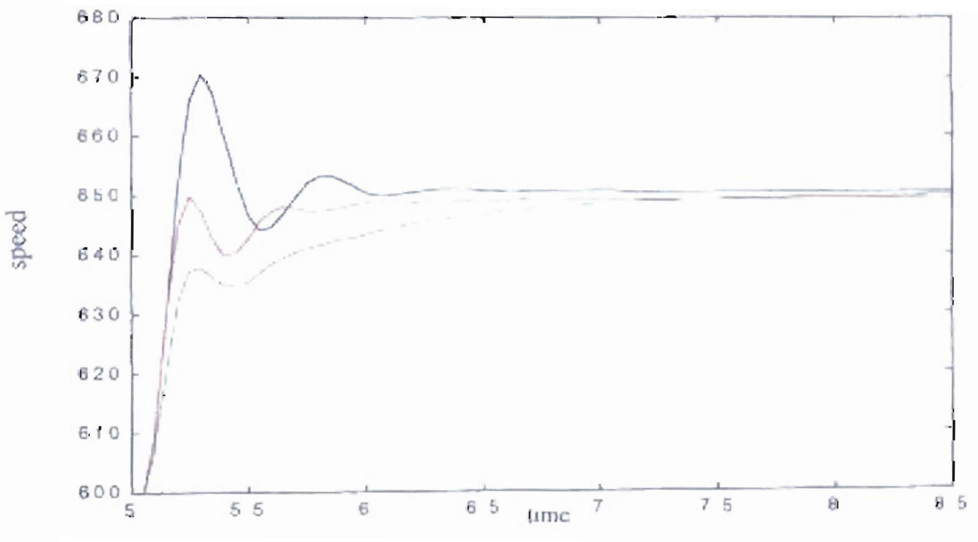


Figure 5.92: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 2.2 lb-ft-sec^2.*

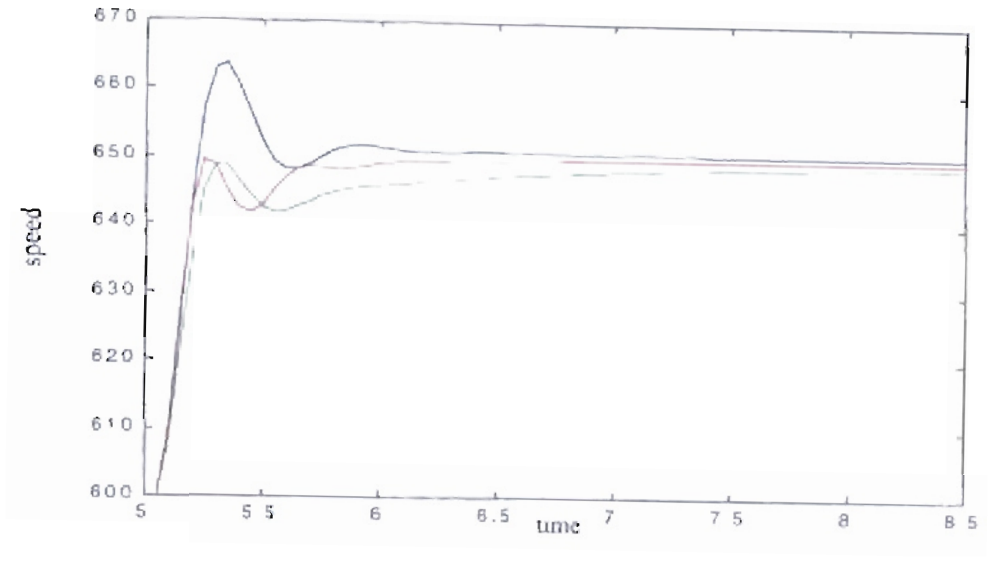


Figure 5.93: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 2.6 lb-ft-sec^2.*

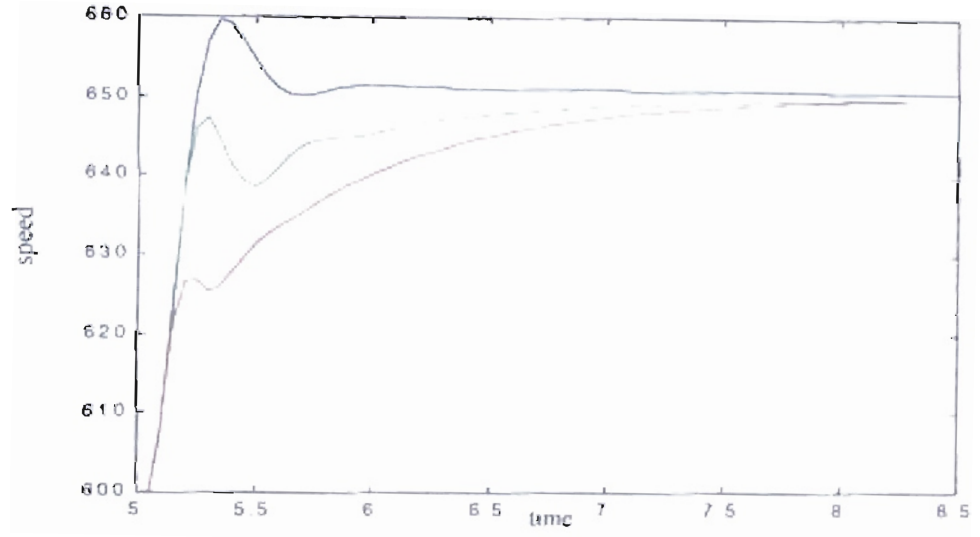


Figure 5.94: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Inertia = 3 lb-ft-sec^2.*

5.9.2. Results for different fueling delays.

For the next case we repeated the training for different fueling delays. For each case the initial response changed according to the fueling delay value with a fixed engine inertia of 2 lb-ft-sec^2. An optimization is needed to adjust the controller values to match the engine fueling delay. We use the mean square error as our fitness value. Like the previous case, we changed the training scheme to two speed transitions to reduce the training time. The first transition is from 600 rpm to 650 rpm at $t_1 = 5$ seconds. The second transition is from 650 rpm to 600 rpm at 4.5 seconds at $t_2 = 8.5$ seconds. The engine and the controller are initialized with the engine conditions for 600 rpm. The simulation runs from 0 to t_1 seconds without error measurements then the training is started.

Using percent overshoot as fitness, we obtained the results shown in Table 5.20 for six different fueling delay values and two different population sizes. Contrary to the analog case, Table 5.20 and Figure 5.95 to Figure 5.100, show that the percentage of improvement depends on how far the sampling rate is from the fueling delay. As the fueling delay increases we obtain a greater improvement. This is a logical response, because if the sampling time is close to the fueling delay, this implies less time for the controller to adjust to any change in the engine response.

Table 5.20: *Engines with different fueling delay. Controller's results for mean square error based fitness. Searching step 1/10. Error calculation after t_l = 5 seconds.*

Delay (msec)	Initial Fitness (error)^2.	Best Fitness (error)^2. Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (error)^2. Pop. = 50	Resulting Parameters Pop. = 5
30	329.53	280.17 (85.02 %)	poles = 1.0000 and 0.7650 zeros = 1.0000 and 0.8284 gain = 2.8845	276.04 (83.77 %)	poles = 1.0000 and 0.5088 zeros = 1.0000 and 0.6201 gain = 2.8112
50	446.33	403.69 (90.45 %)	poles = 1.0000 and 0.5122 zeros = 1.0000 and 0.6749 gain = 2.5230	389.01 (87.16 %)	poles = 1.0000 and 0.2809 zeros = 1.0000 and 0.5397 gain = 2.6709
70	603.08	536.37 (88.94 %)	poles = 1.0000 and 0.6025 zeros = 1.0000 and 0.7669 gain = 2.1766	502.76 (83.37 %)	poles = 1.0000 and 0.3015 zeros = 1.0000 and 0.6163 gain = 2.4664
90	943.02	656.13 (69.58 %)	poles = 1.0000 and 0.5346 zeros = 1.0000 and 0.7359 gain = 1.9619	617.26 (65.46 %)	poles = 1.0000 and 0.2556 zeros = 1.0000 and 0.6301 gain = 2.4043
110	1633.32	790.26 (48.38 %)	poles = 1.0000 and 0.5758 zeros = 1.0000 and 0.7820 gain = 1.7128	735.06 (45.00 %)	poles = 1.0000 and 0.2631 zeros = 1.0000 and 0.6583 gain = 2.1568
130	5615.07	905.05 (16.12 %)	poles = 1.0000 and 0.5579 zeros = 1.0000 and 0.7636 gain = 1.5999	841.78 (14.99 %)	poles = 1.0000 and 0.2481 zeros = 1.0000 and 0.6866 gain = 2.1734

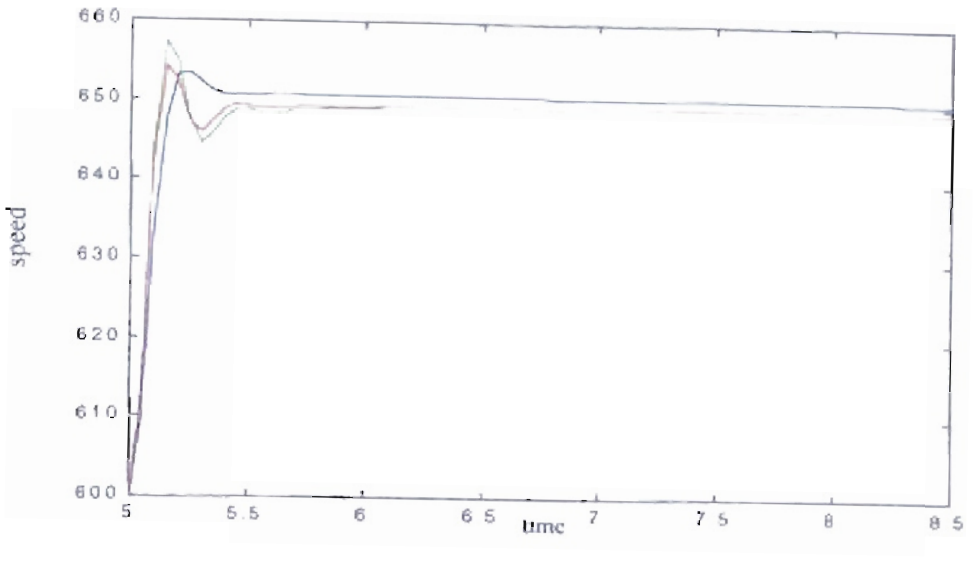


Figure 5.95: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 30 msec.*

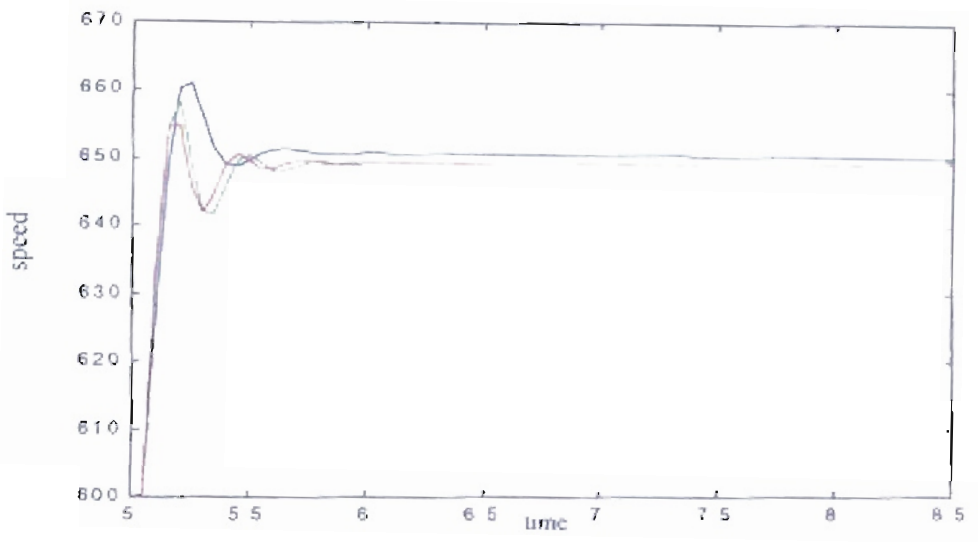


Figure 5.96: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 50 msec.*

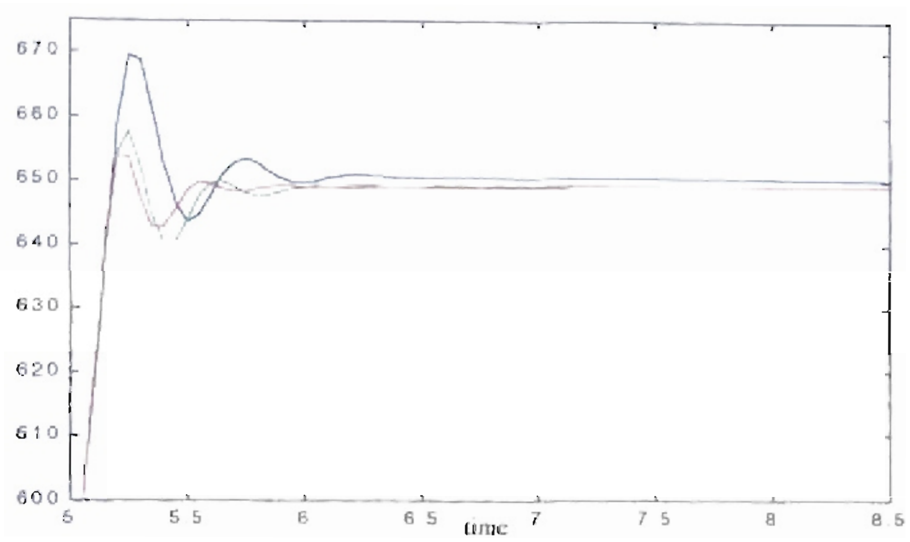


Figure 5.97: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 70 msec.*

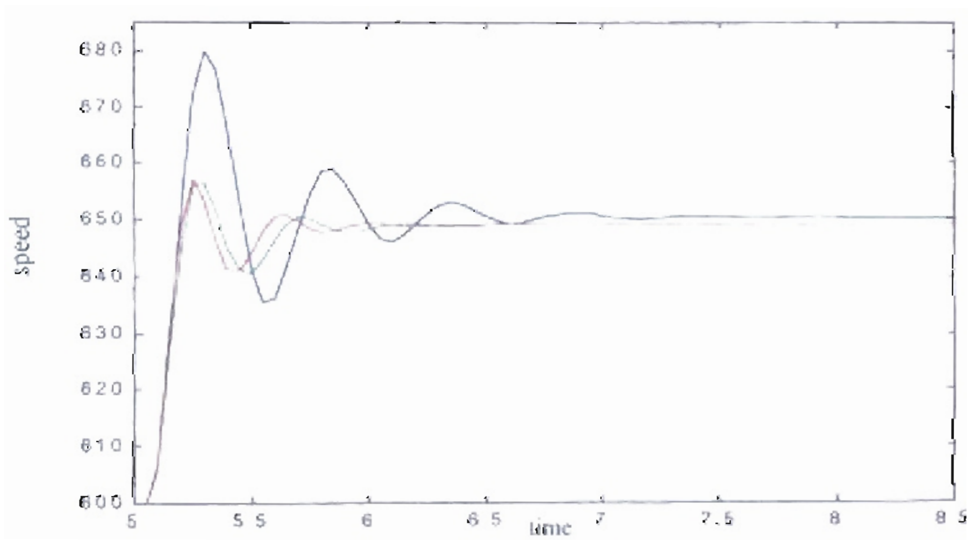


Figure 5.98: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 90 msec.*

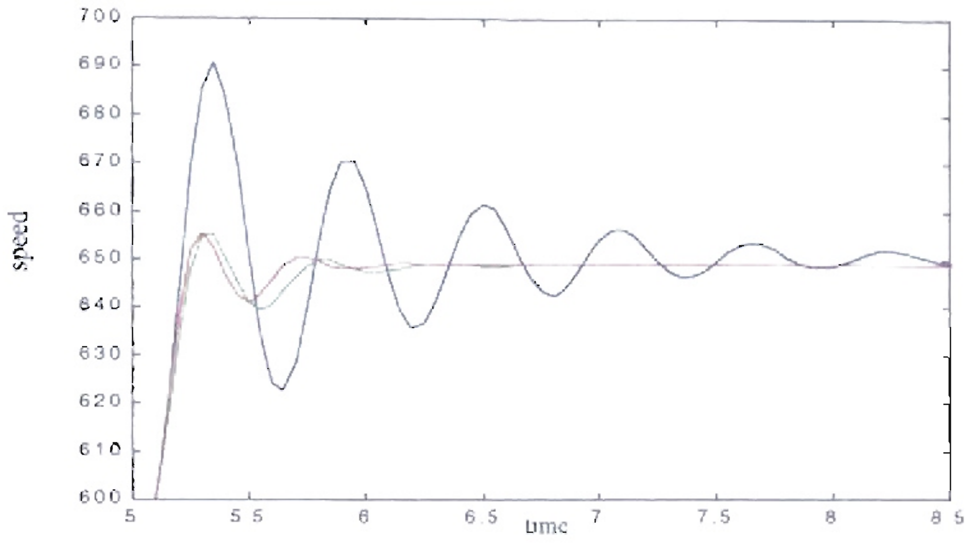


Figure 5.99: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 110 msec.*

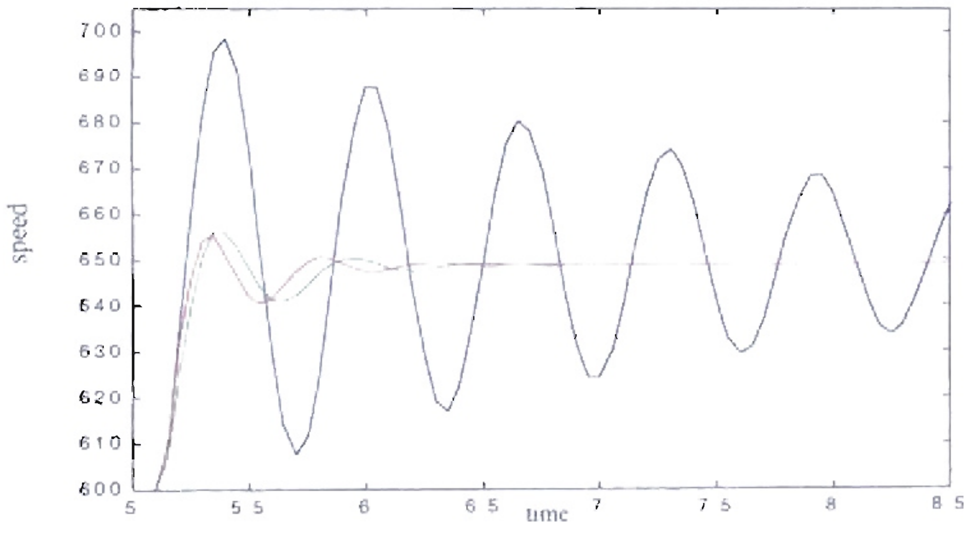


Figure 5.100: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 130 msec.*

For the next case we repeated the training for different fueling delays, changing the fitness function to percent overshoot. For that fitness we obtained the results shown in Table 5.21 and Figure 5.101 to Figure 5.106. We can see that the resulting responses tend to be flatter, with an "undershoot" response similar to that observed in last part of section 5.9.1. As in previous cases, impressive improvement is noticeable for large initial overshoots.

Table 5.21: *Engines with different fueling delay. Controller's results for percent overshoot based fitness. Searching step 1/10. Error calculation after t_l = 5 seconds.*

Delay (msec)	Initial Fitness (overshoot).	Best Fitness (overshoot). Pop. = 5	Resulting Parameters Pop. = 5	Best Fitness (overshoot). Pop. = 50	Resulting Parameters Pop. = 5
30	7.1765	0.4733 (6.60 %)	poles = 1.0000 and 0.2363 zeros = 0.9956 and 0.4176 gain = 1.9410	0.5744 (8.00 %)	poles = 1.0000 and 0.2408 zeros = 0.9961 and 0.3271 gain = 2.0563
50	22.8192	1.2809 (5.61 %)	poles = 1.0000 and 0.4893 zeros = 0.9982 and 0.7874 gain = 2.1714	0.6272 (2.75 %)	poles = 1.0000 and 0.3041 zeros = 0.9951 and 0.5068 gain = 2.0534
70	38.7529	0.5972 (1.54 %)	poles = 1.0000 and 0.5614 zeros = 0.9968 and 0.7693 gain = 1.7090	1.6407 (4.23 %)	poles = 1.0000 and 0.4146 zeros = 1.0000 and 0.6403 gain = 1.9027
90	60.2765	2.5053 (4.16 %)	poles = 1.0000 and 0.5819 zeros = 1.0000 and 0.7722 gain = 1.6324	2.4304 (4.03 %)	poles = 1.0000 and 0.3472 zeros = 1.0000 and 0.6324 gain = 1.7909
110	80.8133	1.7895 (2.21 %)	poles = 1.0000 and 0.8452 zeros = 0.9929 and 0.9575 gain = 1.2781	1.7445 (2.16 %)	poles = 1.0000 and 0.3725 zeros = 1.0000 and 0.7873 gain = 2.0911
130	123.5986	0.3775 (0.31 %)	poles = 1.0000 and 0.6679 zeros = 0.9960 and 0.9069 gain = 1.5411	2.8131 (2.28 %)	poles = 1.0000 and 0.2496 zeros = 1.0000 and 0.7104 gain = 1.9180

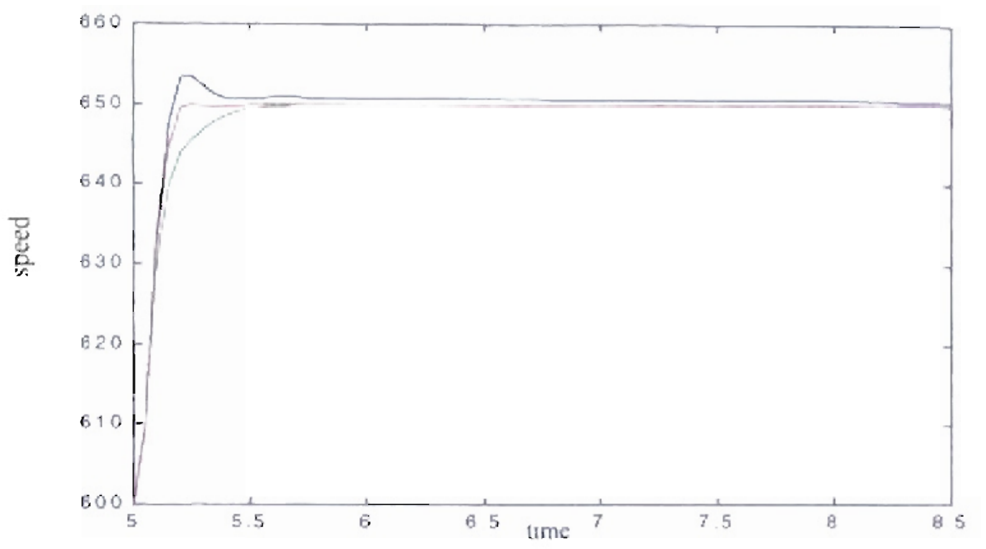


Figure 5.101: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 30 msec.*

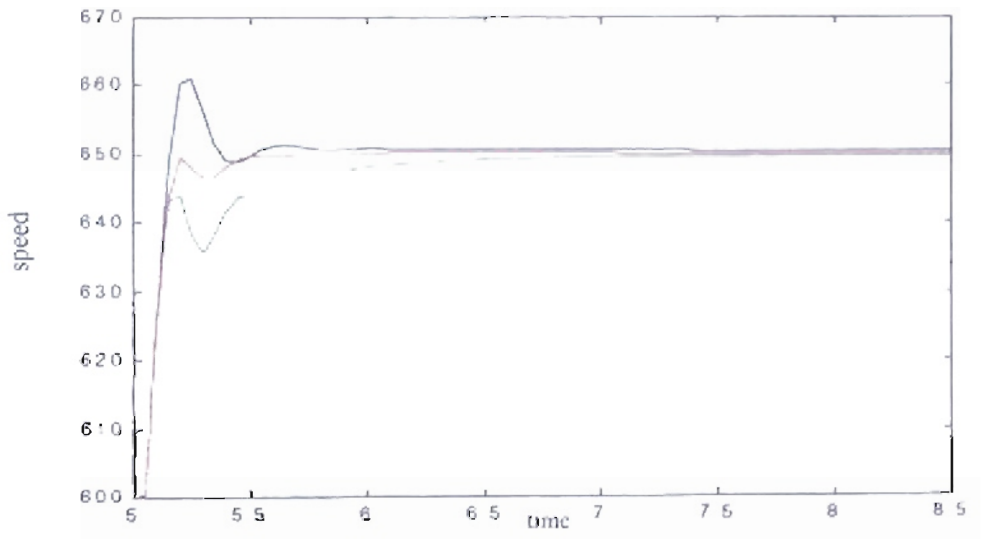


Figure 5.102: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 50 msec.*

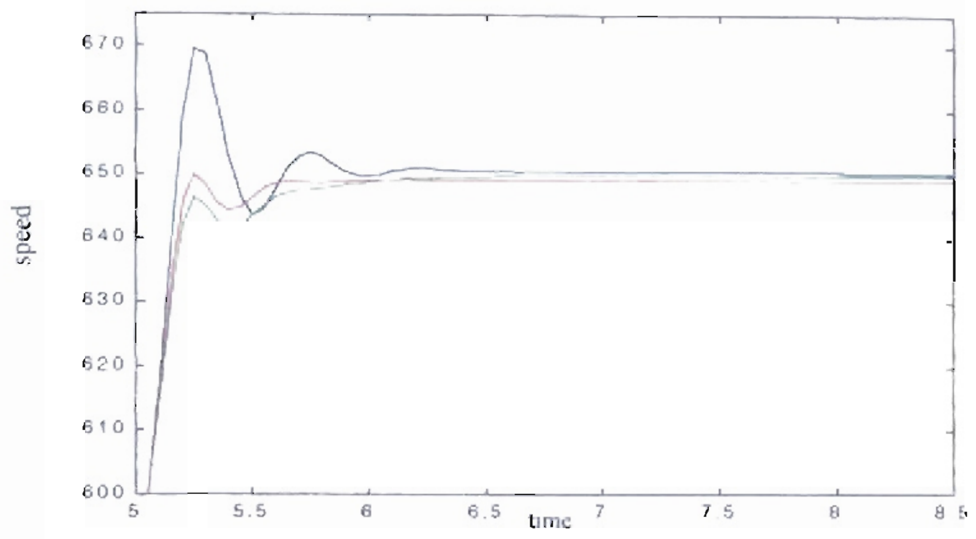


Figure 5.103: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 70 msec.*

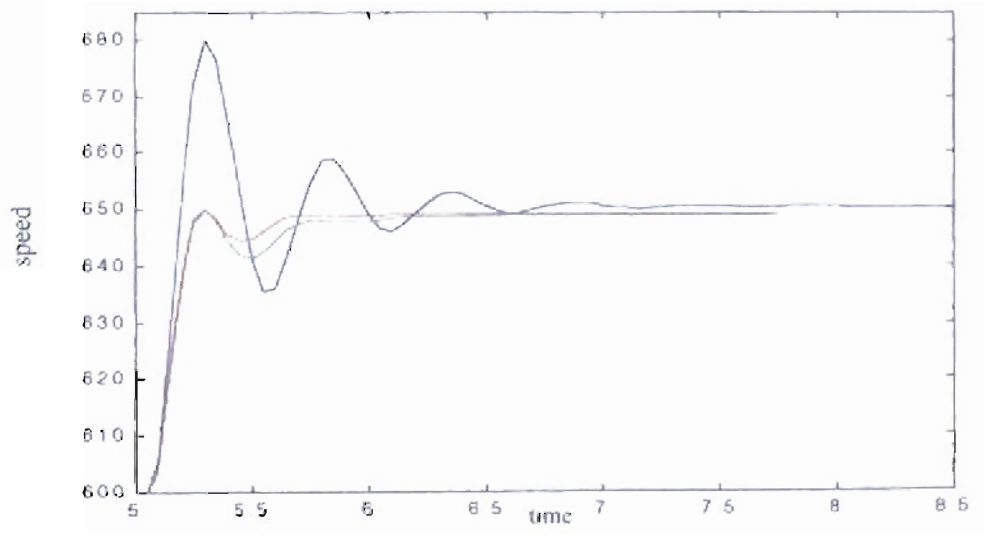


Figure 5.104: *Detail transition from 600 to 650 rpm. Blue Original PID parameters, Green Genitor optimized parameters (Population 5), Red Genitor optimized parameters (Population 50). Epochs 1500, Delay 90 msec.*

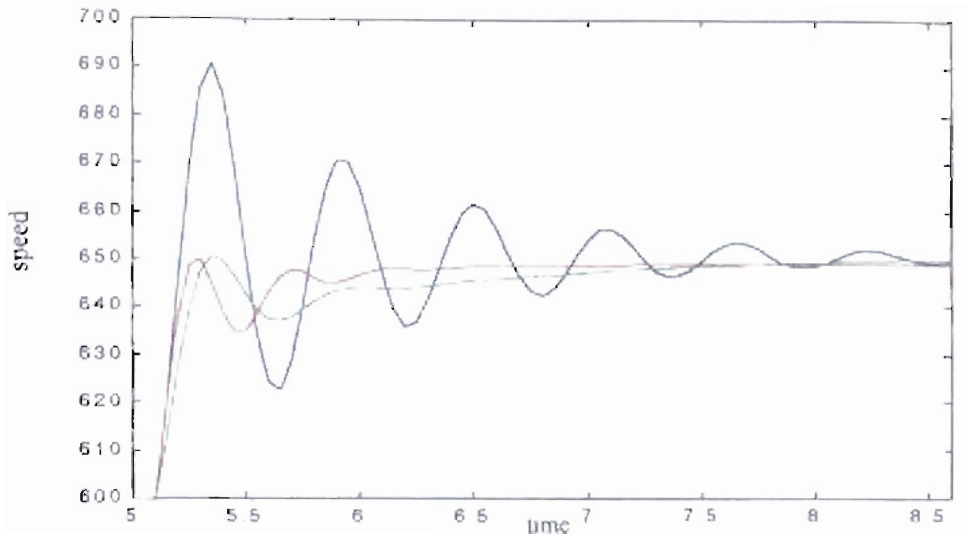


Figure 5.105: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 110 msec.*

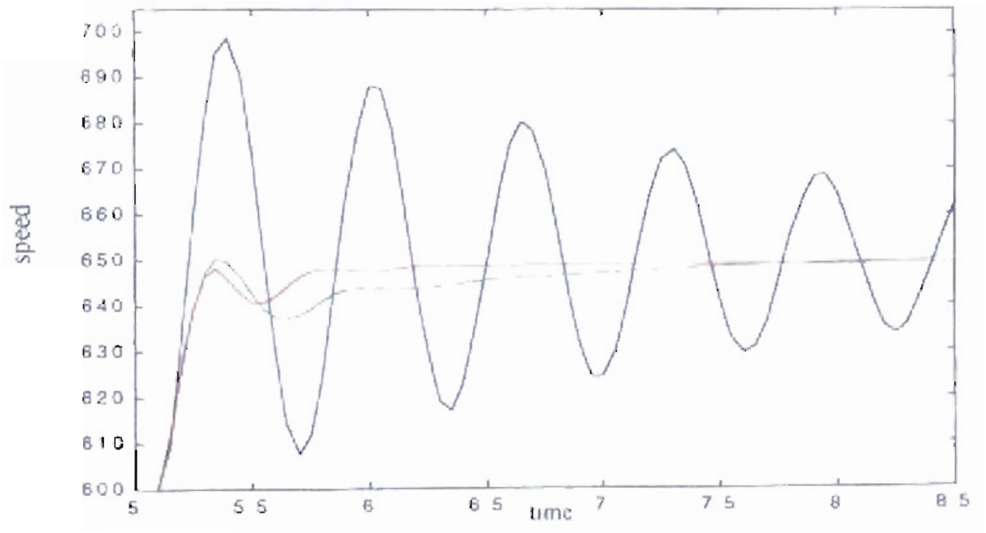


Figure 5.106: *Detail transition from 600 to 650 rpm. Blue = Original PID parameters, Green = Genitor optimized parameters (Population = 5), Red = Genitor optimized parameters (Population = 50). Epochs = 1500, Delay = 130 msec.*

5.9. Summary.

In this chapter we have seen the application of genetic reinforcement learning (GENITOR) for the parameter optimization of a diesel engine controller. It was first tested on an analog PID controller. The algorithm produced improvements in the engine response over the nominal PID controller provided by Cummins. We tested the algorithm with different diesel engines conditions by varying the inertia and the fueling delay. In each case the genetic algorithm provided improved performance over the base-line PID controller. The percentage improvement was greater when the engine response for the original PID controller was very oscillatory.

The fitness function (mean square error or percent overshoot) used has a large influence on the engine response. For the mean square error fitness, we normally obtained an overshoot, with the system following the reference speed very close. For the percent overshoot fitness we obtained a first-order like response.

The same experiments were repeated for a digital controller, to compare the results obtained in the analog implementation. The initial parameters were obtained by transforming the basic PID controller from the s-domain to the z-domain. The experiments were conducted for two values of engine friction $b_f = 0.1229$ and $b_f = 0.01$ and the same combinations of engine inertia J and fueling delay τ. The results obtained for the digital controller were similar to those obtained for the analog controller. Training performance was improved by varying the step size used in the mutation process of the Genitor algorithm. A large step resulted in better controllers.

CHAPTER 6

REINFORCEMENT LEARNING.

6.1. Introduction.

This chapter is based on the books Introduction to Reinforcement Learning (RL) by Sutton and Barto(12) and Neuro-Dynamic Programming by Bertsekas and Tsitsiklis(3). Related work by Jaakkola *et al.* (6), Singh and Sutton (10), Sutton (13 to 17), Watkins and Dayan (20), were considered to support the concepts and ideas of the previous books. We will discuss the different elements of Reinforcement Learning (RL) theory and we will show simulations for some techniques. We will describe the reinforcement learning framework, based on the relation between the environment and the agent, in section 6.2. In section 6.3 we will discuss the elements of Reinforcement Learning: discrete time dynamic system, cost or reward function, policy function, cost or reward accumulation function, and model of the environment. In section 6.4 we will define the types of possible actions that can be performed by the agent in a RL process. In the following section we will review the concept of rewards and the inclusion of a discount factor in case of cumulative rewards. In section 6.6 we will describe the Markov Property and its relation with RL. In the following section we will describe the relationship between RL and the Markov Decision Process (MDP).

In section 6.8 we will explain the types of expected cost functions and the difference between continual and episodic tasks. We will discuss the optimality of cost-to-go functions in the following section. The section 6.10 will discuss some elementary solution methods: Dynamic Programming, Monte Carlo and Temporal Difference Learning. We will present some variations of each method. We will also present some simulations. We will unify all the previous techniques in section 6.11. We will also include the concept of eligibility traces with its discount properties. We will discuss the gradient descent methods in section 6.12. In this section we will show the relationship between RL and neural networks. Finally, we will show two complete examples of simulations with RL: a mountain car task and the swing up of an Acrobot.

6.2. Reinforcement learning framework.

The reinforcement learning problem framework is shown in Figure 6.1. We have a system (or environment) which changes in stages according to discrete decisions. We cannot predict exactly each stage, but we know the statistics of the next outcome. After each action is executed we obtain an immediate cost or reward. Each decision affects the context where future actions will be made and the future costs or rewards we will receive. We want to minimize the total cost or maximize the total reward for all the stages. We want to combine immediate and future rewards or costs.

For a given time t, we have a state $s_t \in S$, where S is the set of the possible states. Based on that state we apply an action $a_t \in A(s_t)$ to the system or environment, where $A(s_t)$ is the set of possible actions in state s_t. That action generates a new state s_{t+1} with a probability $p_{s_t s_{t+1}}(a_t)$ and a reward $r_{t+1} \in \Re$ due to that action, where $\Re$ is the set of all possible costs or rewards. We can say that the cost or reward r_{t+1} is a function which depends on the states involved in the transition (s_t, s_{t+1}) and the action (a_t) executed: $r_{t+1} = g(s_t, a_t, s_{t+1})$. If we continue that sequence for each stage, at time t we have that (12, 3):

$$a_t \Rightarrow (s_{t+1}, r_{t+1}) \Rightarrow a_{t+1} \qquad \textbf{(6.1)}$$

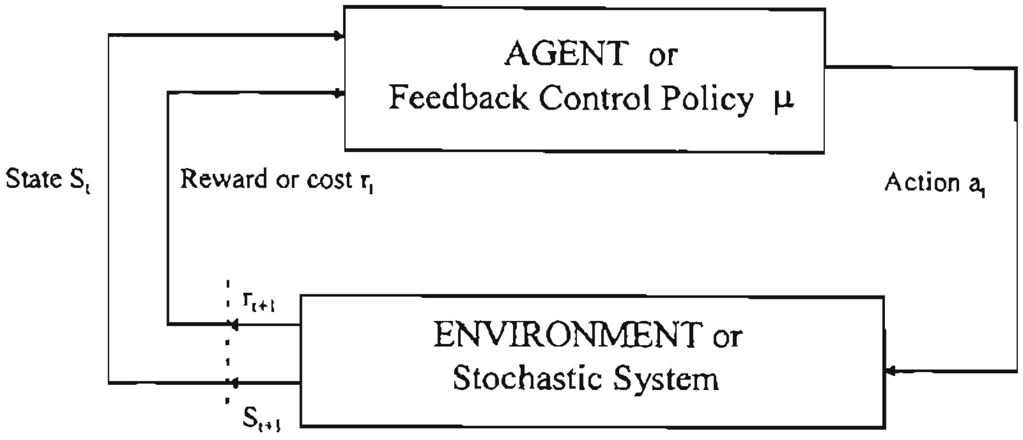


Figure 6.1: *The reinforcement learning framework* (12).

For each time t the Agent has a mapping that represents the probabilities of selecting the action a_t if the state is s_t. This mapping is called the agent's policy: $\pi_t(s, a)$ is equal to the probability of executing the action $a_t = a$ given that we are in state $s_t = s$. We want to balance not only the cost or reward r_{t+1} but also the desirability of the next state s_{t+1}. We can do that by ranking the optimal cost over the remaining states starting from the state s_{t+1}. This function is called the optimal *cost-to-go* of state s_{t+1} and is denoted by $J^*(s_{t+1})$. This relation must satisfy some form of *Bellman's equation*:

$$J^*(s_t) = \max_{a_t} E[g(s_t, a_t, s_{t+1}) + J^*(s_{t+1}) | s_t, a_t] \qquad \text{for all } s \in S \tag{6.2}$$

where $E[\cdot|s_t, a_t]$ denotes the expected value of the cost-to-go function with respect to s_{t+1} given s_t and a_t. From the relation expressed above we want to execute control actions that maximize (minimize) the expected reward (cost) of the current stage and the optimal expected cost of the future stages. One way to obtain an optimal solution J^* could be using dynamic programming (DP). This calculation is done off-line. We can obtain an optimal policy $\pi^*(s, a)$ from the off-line calculation of J^*, or we can obtain it on-line by maximizing the right-hand side of Eq. (6.2). The computational cost involved with the optimal solution is overwhelming, due to the large number of states and controls. Therefore, we need a suboptimal solution. An alternative is reinforcement learning, in which the agent's policy is modified during the execution of the process.

We can approximate the optimal cost-to-go function $J^*(s_{t+1})$ with an approximation $\tilde{J}(s_{t+1}, p)$, where p is a vector of parameters. We will use at the state s_t the suboptimal control $\tilde{a}_t(s_t)$ which maximizes the approximate right-hand side of Eq. (6.2):

$$\tilde{a}_t(s_t) = \arg\max_{a_t} E[g(s_t, a_t, s_{t+1}) + \tilde{J}(s_{t+1}, p) | s_t, a_t] \tag{6.3}$$

In practice, the calculation of $E[g(s_t, a_t, s_{t+1}) + \tilde{J}(s_{t+1}, p) | s_t, a_t]$ for each possible action a_t may be too complicated or too time-consuming. Then we can use an approximate expression of Bellman's equation:

$$Q^*(s_t, a_t) = E[g(s_t, a_t, s_{t+1}) + J^*(s_{t+1}) | s_t, a_t] \tag{6.4}$$

where this function is called the *Q-factor corresponding to* (s_t, a_t). We can replace $Q^*(s_t, a_t)$ with an approximation $\tilde{Q}(s_t, a_t, p)$:

$$\tilde{Q}(s_t, a_t, p) = E[g(s_t, a_t, s_{t+1}) + \tilde{J}(s_{t+1}, p) | s_t, a_t] \tag{6.5}$$

where p is a vector of parameters.We will use at the state s_t the suboptimal control $\tilde{a}_t(s_t)$ which maximizes the approximate right-hand side of Eq. (6.4):

$$\tilde{a}_t(s_t) = \underset{a_t}{\arg\max}\, \tilde{Q}(s_t, a_t, p) \tag{6.6}$$

6.3. Elements of Reinforcement Learning.

The elements which make up a typical Reinforcement Learning algorithm are:

- **A discrete time dynamic system.** The state transition depends on a control input a_t. We have n states denoted by $1, 2, \ldots, n$, with one additional termination state. For each state s_t we must choose the control action a_t from a finite set $A(s)$. The control action a_t specifies the transition probability $p_{s_t s_{t+1}}(a_t)$ from the state s_t to the state s_{t+1}

- **Cost or Reward function.** This function implies a cost or reward r_{t+1} given for the state transition from s_t to s_{t+1} with the action a_t. We can express the function cost as: $r_{t+1} = g(s_t, a_t, s_{t+1})$.

- **Policy function.** The policy consists of the rules which define how we want to operate in a given state. We can define a policy π as a mapping from states s into control actions a, or we can define a policy π as the probability that an specific action may be executed in a given state $\pi_t(s, a)$.

- **Cost or reward accumulation function**. The cost is accumulated over time and depends on the states visited and the actions executed. The cost function may be affected by a discount factor γ, that will be discussed in section 6.5.

- **Model of the environment** (optional). Generally we will use the models for planning. We can simulate and train the system off-line to obtain an initial coherent policy. Afterwards, we can improve our policy with on-line training on the real system.

We can explain the state sequence of a Reinforcement Learning problem using the game sequence shown in Figure 6.2. From the starting position, the opponent executes a move that changes the game state from **a** to **b**. In the state **b** we have different options. Our policy implies a move that changes our state from **b** to **c**. For example, we can define our policy as fixed rules or as random actions. The game continues with the opponent's move from **c** to **d**. In the state **d** if we execute the action **e** the opponent replies with **f** and consequently for future movements. We can see that both the action **f** from the opponent and the future decision **g** depend on the decision made in the state **e**. If we selected the state **e'**; we probably would finish with a different sequence **f'** and **g'**.

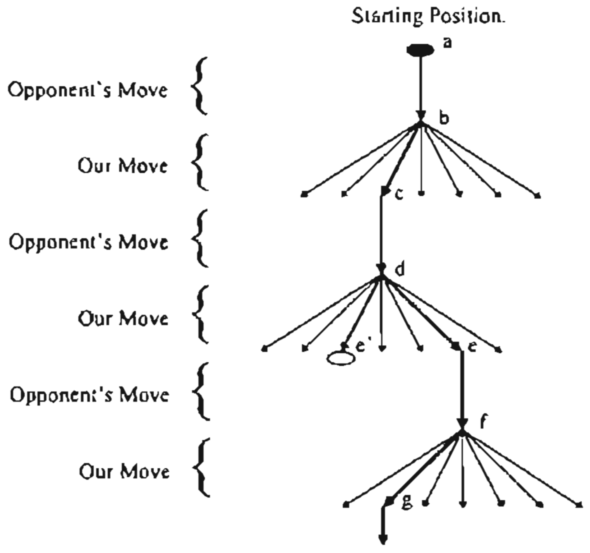


Figure 6.2: *Elements of Reinforcement Learning. Game sequence (12).*

6.4. Actions.

Sutton and Barto defined two types of actions(12):

6.4.1. Greedy Actions. For a given state, we execute the action whose estimated cumulative reward is greatest.

6.4.2. Exploring Actions. We do not necessarily follow the action whose estimated cumulative reward is greatest. For example, we can execute random actions. This type of action moves us to find new solutions.

We can define a procedure for action selection where we can normally execute greedy actions with a small percentage of random actions. We can define a probability ε

which is the probability of selecting an action at random. This number is generally small, for example 0.1 or 10 %.

6.5. Rewards and discount factor.

We can define the immediate rewards r_{t+1} as a numerical feedback generated by the environment and measured by the agent. The Agent's goal is to maximize (minimize) the total amount of future reward (cost), or the cumulative future reward (cost). If we execute our process one time we will obtain a total reward:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T = \sum_{k=0}^{T} r_{t+k+1} \qquad \textbf{(6.7)}$$

$$= \sum_{k=0}^{T} g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1})$$

where T is the final time step.

The variable r represents the reward for each step and R_t represents the undiscounted accumulated reward received after the time t. We can discount the present value of the future rewards:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^T r_T = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \qquad \textbf{(6.8)}$$

$$= \sum_{k=0}^{T} \gamma^k g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1})$$

where γ is a discount rate or discount factor used to reinforce the importance of present rewards over future rewards. Generally γ is an scalar and is limited to: $0 \le \gamma \le 1$, so that J_t is bounded. If $\gamma \approx 0$, then we maximize the immediate rewards. If $\gamma \approx 1$, then we maximize the future rewards.

6.6. Markov property.

Define a system where the new states and rewards depend on all previous states and rewards. Then the probability distribution of that system is:

$$Pr\{(s_{t+1} = s_F);(r_{t+1} = r_F)|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_1, a_1, r_1, s_0, a_0\} \quad \textbf{(6.9)}$$

We can say that a system has the Markov property if its response at time $t+1$ depends only on the conditions at time t. Then its probability distribution is:

$$Pr\{(s_{t+1} = s_F);(r_{t+1} = r_F)|s_t, a_t\} \quad \textbf{(6.10)}$$

We can conclude that systems with the Markov property have dynamics based on one step. "Markov states provide the best possible basis for choosing actions."(12)

6.7. Markov decision process (MDP).

MDP refers to any reinforcement learning process that satisfies the Markov property described previously. When the state and action spaces are finite, the process is called a Finite Markov Decision Process. A finite MDP is defined by its state and action sets and by the one-step dynamics of the next state s_F, given the state s and the action a:

$$P^a_{ss_F} = Pr\{s_{t+1} = s_F|(s_t = s), (a_t = a)\} \quad \textbf{(6.11)}$$

and the expected next reward is:

$$R^{a}_{ss_F} = E[r_{t+1} | (s_t = s), (a_t = a), (s_{t+1} = s_F)] \tag{6.12}$$

6.8. Types of expected cost functions.

6.8.1. Cost-to-go functions. We can define the finite horizon problem, where we have a process with final state s_F and we accumulate the cost over that finite period of time T. For this type of problem, the expected cost-to-go following a policy π and starting from an initial state s is:

$$J^{\pi}_{T}(s) = E_{\pi}\left[\gamma^{T} G(s_F) + \sum_{k=0}^{T-1} \gamma^{k} g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1}) \middle| s_t = s\right] \tag{6.13}$$

where $\gamma^{T} G(s_F)$ is the terminal cost or reward related for arriving to the final state s_F.

The optimal cost-to-go function for finite horizon problems is denoted by the relation:

$$J_T^{*}(s) = \max_{\pi} J^{\pi}_{T}(s) \tag{6.14}$$

We call an episode the transition from an initial state to the final state, at time T, of a finite horizon problem. We call episodic tasks the process with repeated episodes. We can also start the new episode in a fixed or a random initial state.

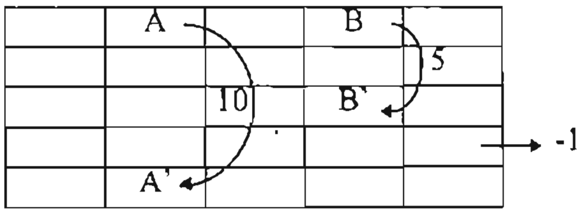
We also have infinite horizon problems, where we accumulate the cost indefinitely. We have that the expected cost following a policy π starting from an initial state s is:

$$J^{\pi}(s) = \lim_{T \to \infty} E_{\pi}\left[\sum_{k=0}^{T} \gamma^{k} g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1}) \middle| s_t = s \right] \tag{6.15}$$

The optimal cost-to-go function is:

$$J^{*}(s) = \max_{\pi} J^{\pi}(s) \tag{6.16}$$

6.8.2. Gridworld example. This is one example to estimate the cost-to-go functions in a given state. The Gridworld is a 5 x 5 two dimensional cell space where the initial policy π specifies that we can move in four directions with the same probability, as shown in Figure 6.3. We have a penalty of -1 each time we move outside the board. If we are in the state A the only possible movement is to A' and we receive a reward of 10 units. Similarly, at position B we can only move to B" and the reward is 5 units.



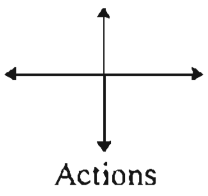


Figure 6.3: *Gridworld example. Original movement rules (12).*

We initilialized the cost-to-go values to zero. Then we execute this process under the policy π as an infinite horizon problem, we will find a cost-to-go function J^{π} for each state based in Eq. (6.15), as shown in Figure 6.4. We note that the cost-to-go value at position A is lower than 10, because we obtain an immediate reward of 10 from A to A', but after that we would move outside the board obtaining an immediate negative reward. We also note that the cost-to-go value at position B is greater than 5, because after the immediate reward of 5 we would move to the A position for an immediate reward of 10.

3.3090	8.7893	4.4276	5.3224	1.4922
1.5216	2.9923	2.2501	1.9076	0.5474
0.0508	0.7382	0.6731	0.3582	-0.4031
-0.9736	-0.4355	-0.3549	-0.5856	-1.1831
-1.8577	-1.3452	-1.2293	-1.4229	-1.9752

Figure 6.4: *Gridworld example. Cost-to-go values from original policy.*

*6.8.3. **Q-factor functions.*** Generally the reinforcement learning algorithms are based on the estimation of "how good" a given state or a given state-action pair is. The estimated cost-to-go function is defined by Eq. (6.13) and Eq. (6.15). The previous relations only provide us information about the state. If we want information about the combination of the state and the action we will use a relation based on the Q-factor function defined by Eq. (6.4). The Q-factor value of a finite horizon problem is the expected return starting from s, taking action a, and thereafter following the policy π (12, 3):

$$Q^{\pi}(s, a) = E_{\pi}\left[\gamma^{T} G(s_F) + \sum_{k=0}^{T-1} \gamma^{k} g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1}) \middle| (s_t = s), (a_t = a)\right] \quad (6.17)$$

where $\gamma^{T} G(s_F)$ is the terminal cost or reward related for arriving to the final state s_F.

For infinite horizon problems, the Q-factor starting from an initial state s_t, taking action a_t, and following a policy π is:

$$Q^{\pi}(s, a) = \lim_{T \to \infty} E_{\pi}\left[\sum_{k=0}^{T} \gamma^k g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1}) \middle| (s_t = s),(a_t = a)\right] \quad (6.18)$$

6.9. Optimal cost-to-go functions.

If we want to solve a problem of reinforcement learning, we want to obtain the maximum reward (or the minimum cost) for each state or state-action. If we talk about the optimal cost-to-go function starting from one state we must select the policy π that guarantees an optimal cost-to-go function:

$$J_T{}^*(s) = \max_{\pi} J_T^{\pi}(s) \qquad \forall s \in S \quad (6.19)$$

If we talk about the state-action relation or Q-factor function, we must find an optimal Q-factor function:

$$Q_T{}^*(s, a) = \max_{\pi} Q_T^{\pi}(s, a) \qquad \forall s \in S, \forall a \in A(s)$$

$$= E[g(s_t, a_t, s_{t+1}) + \gamma J^*(s_{t+1}) | s_t, a_t] \quad (6.20)$$

If we want to relate the cost-to-go function of an state s under an optimal policy π with the expected return for the best action of that state, we can use the Bellman Optimality equation for J^*:

$$J^*(s) = \max_{a} Q^{\pi^*}(s, a)$$

$$= \max_{a} E_{\pi^*}\{R_t | (s_t = s), (a_t = a)\}$$

$$= \max_{a \in A(s)} E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | (s_t = s), (a_t = a)\right\}$$

$$= \max_{a \in A(s)} E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | (s_t = s), (a_t = a)\right\}$$

$$= \max_{a \in A(s)} E\{r_{t+1} + \gamma J^*(s_{t+1}) | (s_t = s), (a_t = a)\}$$

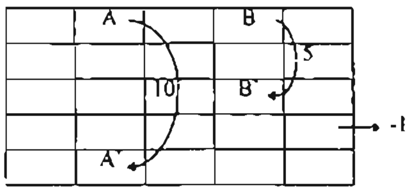
$$= \max_{a \in A(s)} \sum_{s_t} P^a_{ss_F}[R^a_{ss_t} + \gamma J^*(s_F)] \qquad \textbf{(6.21)}$$

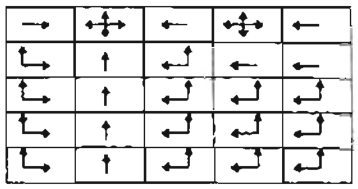
If we want to relate the action-value (or Q-factor function) of an state under an optimal policy with the expected return for the best action of that state, we can use the Bellman Optimality equation for Q^*:

$$Q^*(s, a) = E\{r_{t+1} + \gamma \max Q^*(s_{t+1}, a_F)/(s_t = s), (a_t = a)\}$$

$$= \sum_{s_F} P^a_{ss_F}\left[R^a_{ss_F} + \gamma \max_{a_F} Q^*(s_F, a_F)\right] \qquad \textbf{(6.22)}$$

We can solve the Gridworld example of Figure 6.3 by solving the Bellman Optimality equation for J^* (Eq. (6.21)). That method is viable if we have a low number of states. As that state number increases we can implement exhaustive search, looking for solutions by implementing different policies. We solved the Gridworld by both ways looking for the optimal cost-to-go value for each state. The results shown in Figure 6.5. Also we can see the policy π^* that maximizes our cost-to-go function J^*.





Optimal policy

21.9775	24.4194	21.9775	19.4194	17.4775
19.7797	21.9775	19.7797	17.8018	16.0216
17.8018	19.7797	17.8018	16.0216	14.4194
16.0216	17.8018	16.0216	14.4194	12.9775
14.4194	16.0216	14.4194	12.9775	11.6797

Optimal cost-to-go function.

Figure 6.5: *Solving the Gridworld* (12).

6.10. Elementary Solution Methods.

*6.10.1. **Dynamic Programming.*** We want to use dynamic programming to obtain good policies. We must follow the following steps in the dynamic programming evaluation:

*6.10.1.1. **Policy evaluation:*** We want to compute the cost-to-go function J^{π} for an arbitrary policy π, based on the relation:

$$J_{k+1}(s) = E_{\pi}\{r_{t+1} + \gamma J_k(s_{t+1}) | s_t = s\}$$

$$= \sum_{a} \pi(s, a) \sum_{s_F} P^{a}_{ss_F}[R^{a}_{ss_F} + \gamma J^{*}(s_F)] \quad (6.23)$$

We can make that evaluation iteratively, knowing that $\{J_k\}$ generally converges to J^{π} as $k \to \infty$. The iterative policy evaluation algorithm is shown in Figure 6.6.

```
Iterative policy evaluation.
Input:
    $\pi$, the policy to be evaluated
    $P^a_{ss_F}$, the probability of finish on state $s_F$ if we start on state $s$
        and execute the action $a$
    $R^a_{ss_F}$, the reward received after finish on state $s_F$ if we start on
        state $s$ and execute the action $a$
Initialize $J(s) = 0$, for all $s \in S^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$ for all possible $s_F \in S$ and $a \in A(s)$:
        $v \leftarrow J(s)$
        $J(s) \leftarrow \sum_a \pi(s, a) \sum_{s_F} P^a_{ss_F}[R^a_{ss_F} + \gamma J^*(s_F)]$
        $\Delta \leftarrow \max(\Delta, |v - J(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $J \approx J^{\pi}$
```

Figure 6.6: *Iterative Policy Evaluation* (12).

6.10.1.2. Policy improvement. We want to know if an action a different from that suggested by the current policy could produce a better Q-factor function. Therefore, we must maintain a structure with all of the expected returns starting from the state s and following the action a:

$$\begin{aligned} Q^{\pi}(s, a) &= E\{r_{t+1} + \gamma J^{\pi}(s_{t+1}) | (s_t = s), (a_t = a)\} \\ &= \sum_{s_F} P^a_{ss_F}[R^a_{ss_F} + \gamma J^{\pi}(s_F)] \end{aligned} \tag{6.24}$$

We want to see if after selecting the action a in the state s and following π we can find a new and better policy π_F. We can define the Policy improvement theorem as:

$$\text{If } Q^{\pi}(s, \pi_F(s)) \geq J^{\pi}(s) \qquad \forall s \in S \Rightarrow \left\{ \begin{array}{l} \text{policy } \pi_F \text{ must be as good as,} \\ \text{or better than } \pi \ (J^{\pi_F}(s) \geq J^{\pi}(s)) \end{array} \right\}$$

6.10.1.3. ***Policy iteration.*** When a policy π has been improved using J^{π} resulting in a better policy π_F we can compute J^{π_F}. If we improve the new policy, we could have π_{FF} and so on. Ideally, we can see a sequence of policy evaluation and policy improvement as the sequence:

$$\pi_0 \xrightarrow{E} J^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} J^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} J^{\pi_2} \xrightarrow{I} \ldots \pi^* \xrightarrow{E} J^{\pi^*}$$

The algorithm for policy iteration is shown in Figure 6.7.

```
Policy Iteration.
1. Initialization.
   $J(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
   $P^a_{ss_F}$, the probability of finish on state $s_F$ if we start on state $s$ and
        execute the action $a$
   $R^a_{ss_F}$, the reward received after finish on state $s_F$ if we start on state
        $s$ and execute the action $a$
2. Policy evaluation.
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in S$
         $v \leftarrow J(s)$
         $J(s) \leftarrow \sum_a \pi(s, a) \sum_{s_F} P^a_{ss_F} [R^a_{ss_F} + \gamma J^*(s_F)]$
         $\Delta \leftarrow \max(\Delta, |v - J(s)|)$
   until $\Delta < \theta$ (a small positive number)
   Output $J \approx J^\pi$
3. Policy improvement.
   policy_stable $\leftarrow$ true
   For each $s \in S$:
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s_F} P^{\pi(s)}_{ss_F} [R^{\pi(s)}_{ss_F} + \gamma J(s_F)]$
      If $b \neq \pi(s)$ then olicy_stable $\leftarrow$ false
   If policy_stable, then stop; else go to 2.
```

Figure 6.7: *Policy iteration* (12).

6.10.1.4. Value iteration. This is the name given to the DP iteration that executes Eq. (6.21) starting from some J. With that iteration we could find the optimal cost-to-go function J^*. Also, we can see that value iteration is a combination of policy improvement and truncated policy-evaluation steps, as seen in Figure 6.8.

```
Value Iteration.
Initialize $J$ arbitrarily, e.g., $J(s) = 0$, for all $s \in S^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$
        $v \leftarrow J(s)$
        $J(s) \leftarrow \max_a \sum_{s_F} P^a_{ss_F}[R^a_{ss_F} + \gamma J(s_F)]$
        $\Delta \leftarrow \max(\Delta, |v - J(s)|)$
until $\Delta < \theta$ (a small positive number)
Output a deterministic policy, $\pi$, such that:
    $\pi(s) = \arg\max_a \sum_{s_F} P^a_{ss_F}[R^a_{ss_F} + \gamma J(s_F)]$
```

Figure 6.8: *Value Iteration* (12).

6.10.1.5. Generalized policy iteration. GPI occurs when we have an interaction between the policy evaluation process and the policy improvement process, as seen in Figure 6.9. In policy evaluation we execute the actual policy to obtain the current cost-to-go function. With policy improvement we define the policy according to the current cost-to-go function. After many iterations we will find the optimal policy and cost-to-go function.

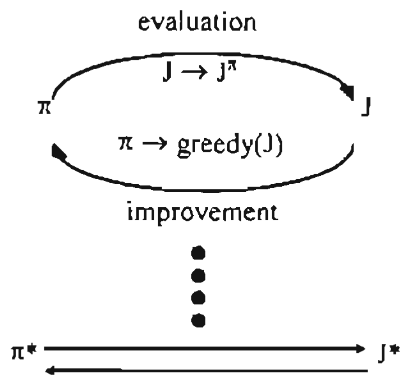


Figure 6.9: *Generalized Policy Iteration* (12).

6.10.2. Monte Carlo methods. Monte Carlo methods only require information about states, actions and rewards that originate from a real or simulated system. These are algorithms that learn from experience. Generally we don't need detailed information from the process. We can differentiate between every-visit and first-visit MC methods for estimating J^π. The every-visit MC method executes the average of all the returns after all the visits to the state s. The first-visit MC method executes the average of all the returns after the first visit to the state s. We can see the algorithm for the first-visit MC method in Figure 6.10.

```
First visit Monte Carlo for estimating $J^\pi$.
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $J \leftarrow$ an arbitrary cost-to-go function
    $Returns(s) \leftarrow$ an empty list, for all $s \in S$
Repeat forever:
    (a) Generate an episode using $\pi$.
    (b) For each $s$ appearing in the episode:
        $R \leftarrow$ return following the first occurrence of $s$.
        Append $R$ to $Returns(s)$.
        $J(s) \leftarrow$ average($Returns(s)$)
```

Figure 6.10: *Algorithm for first-visit Monte Carlo method for estimating J^π (12).*

We not only need the estimation of the cost-to-go values for a given state. With Monte Carlo Methods we could obtain an estimate of the Q-factors Q^π of each action to obtain an optimal policy.

As we described in section 6.10.1.5 about generalized policy iteration, we evaluate a policy and a cost-to-go function until we obtain the optimal configuration of both functions. If we apply Monte Carlo methods, we can start with an initial policy π_0 ending with an optimal policy and optimal Q-factor. The policy improvement is made by taking the action that maximizes the Q-factor function:

$$\pi(s) \leftarrow \arg\max_{a} (Q(s,a)) \qquad \textbf{(6.25)}$$

In policy evaluation for Monte Carlo methods, we evaluate the Q-factor function of the states during the episodes and improve the policy at the end of each episode. An example of this method, called Monte Carlo with Exploring Starts (or Monte Carlo ES), is shown in Figure 6.11.

```
Monte Carlo with Exploring Starts.
Initialize, for all $s \in S, a \in A(s)$:
    $Q(s, a) \leftarrow$ arbitrary.
    $\pi(s) \leftarrow$ arbitrary.
    $Returns(s, a) \leftarrow$ an empty list.
Repeat forever:
    (a) Generate an episode using $\pi$.
    (b) For each pair $s, a$ appearing in the episode:
        $R \leftarrow$ return following the first occurrence of $s, a$.
        Append $R$ to $Returns(s, a)$.
        $Q(s, a) \leftarrow$ average $(Returns(s, a))$
    (c) For each pair $s$ in the episode:
        $\pi(s) \leftarrow \arg\max_a (Q(s, a))$
```

Figure 6.11: *Algorithm for Monte Carlo with exploring starts* (12).

6.10.3. Temporal difference learning. Temporal difference (TD) methods are a combination of Monte Carlo and Dynamic Programming methods. TD combines learning from experience and updating the estimates without waiting for the end of the episode. For example, we can define a simple every-visit Monte Carlo method as:

$$J(s_t) \leftarrow J(s_t) + \alpha[R_t - J(s_t)] \tag{6.26}$$

That is called constant-α MC, where α is a constant step-size training parameter and R_t is the actual return after the time t when the episode finished, as shown in Eq. (6.8). We can update the cost-to-go function $J(s_t)$ only at the end of the episode, because we need R_t. Temporal difference learning methods can update $J(s_t)$ as they know the observed reward r_t. If we update each time step, we obtain the TD(0) method:

$$J(s_t) \leftarrow J(s_t) + \alpha[r_{t+1} + \gamma J(s_{t+1}) - J(s_t)] \tag{6.27}$$

where $r_{t+1} + \gamma J(s_{t+1}) - J(s_t) = \delta_t$ is defined as the TD error for one backup step. This method is called TD(0) because the eligibility trace parameter λ is equal to zero as shown in section 6.11.

If we compare targets, we note that Monte Carlo uses the total reward of one episode R_t and Temporal Difference uses $r_{t+1} + \gamma J(s_{t+1})$. If we rewrite Eq. (6.15) as:

$$J^{\pi}(s) = E_{\pi}[R_t | s_t = s] \tag{6.28}$$

$$= E_{\pi}\left[\sum_{k=0}^{T} \gamma^k g(s_{t+k}, a_{t+k}(s_{t+k}), s_{t+k+1}) \middle| s_t = s\right]$$

$$= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k g(s_{t+k+1}, a_{t+k+1}(s_{t+k+1}), s_{t+k+2}) \middle| s_t = s\right]$$

$$= E_{\pi}[r_{t+1} + \gamma J^{\pi}(s_{t+1}) | s_t = s] \tag{6.29}$$

where we can see that Monte Carlo methods use Eq. (6.28) for their estimates and temporal difference methods use Eq. (6.29) for their estimates.

Figure 6.12 shows an algorithm for estimating J^{π} using TD(0).

```
TD(0) algorithm for estimating $J^\pi$.
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $J \leftarrow$ an arbitrary state-cost-to-go function.
Repeat (for each episode):
    Initialize $s$.
    Repeat (for each step of episode):
        $a \leftarrow$ action given by $\pi$ for $s$.
        Take action $a$; observe reward $r$, and next state $s_F$.
        $J(s) \leftarrow J(s) + \alpha[r + \gamma J(s_F) - J(s)]$
        $s \leftarrow s_F$.
    until $s$ is terminal.
```

Figure 6.12: *TD(0) algorithm for estimating* J^π *(12)*.

6.10.3.1. Random walk example. We want to compare Monte Carlo and Temporal Difference methods. For our example we implemented a random walk as seen in Figure 6.13.

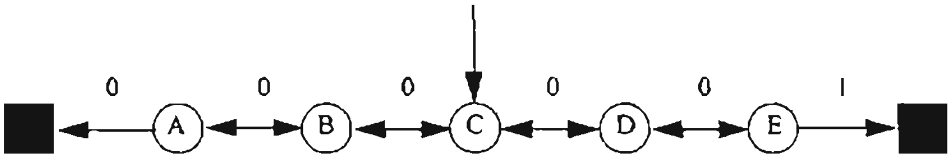


Figure 6.13: *Random Walk (12)*.

All episodes start in the center (C) and move left or right with equal probability. We finish each episode at the left or right box. The reward is always 0 in all positions except when we finish at right with reward 1. For this example, the true cost-to-go function for each state from A to E is $\{1/6, 1/3, 1/2, 2/3, 5/6\}$.

After running the random walk for both methods, we compare TD(0) and MC in Figure 6.14 and Figure 6.15 with respect the real cost-to-go function. We noted a better approximation if we use the TD(0) method compared with the Monte Carlo Method. If we see the learning curve for TD(0) (Figure 6.16) we can see that TD(0) has a more stable learning curve for $\alpha = 0.05$. If we increase α we obtain an initial faster learning, but after some steps the TD(0) algorithm oscillates. In the learning curves for MC (Figure 6.17) we noted that the step training constant must be around $\alpha = 0.01$, which is lower than TD(0). If we increase the constant α, the MC algorithm oscillates at different training times

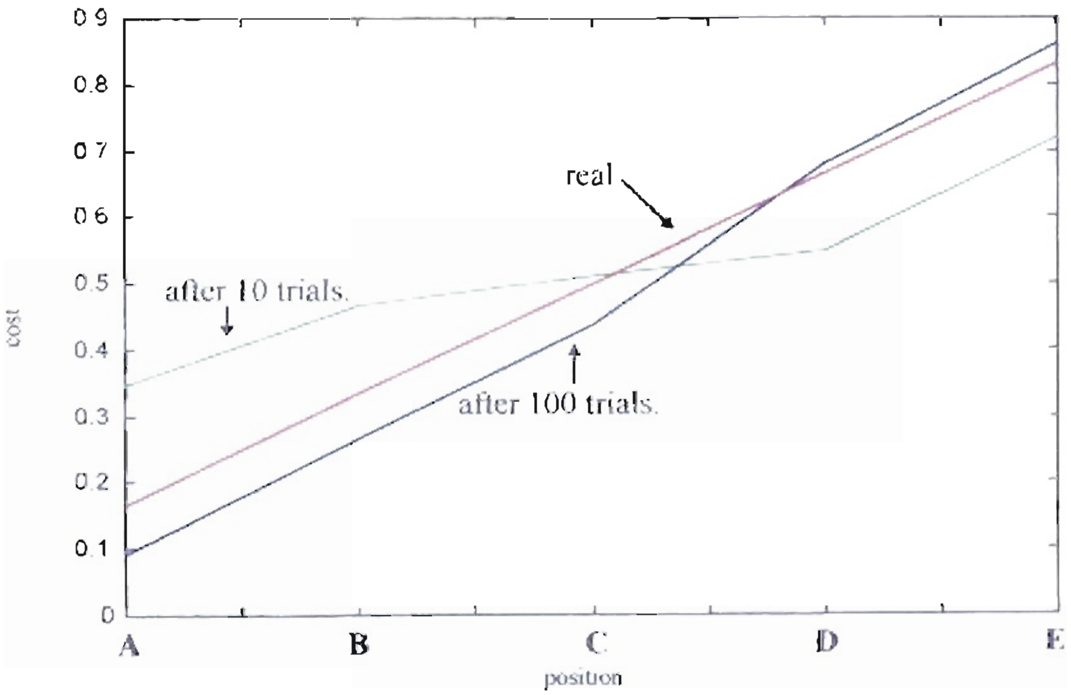


Figure 6.14: *Cost-to-go values learned by TD(0).*

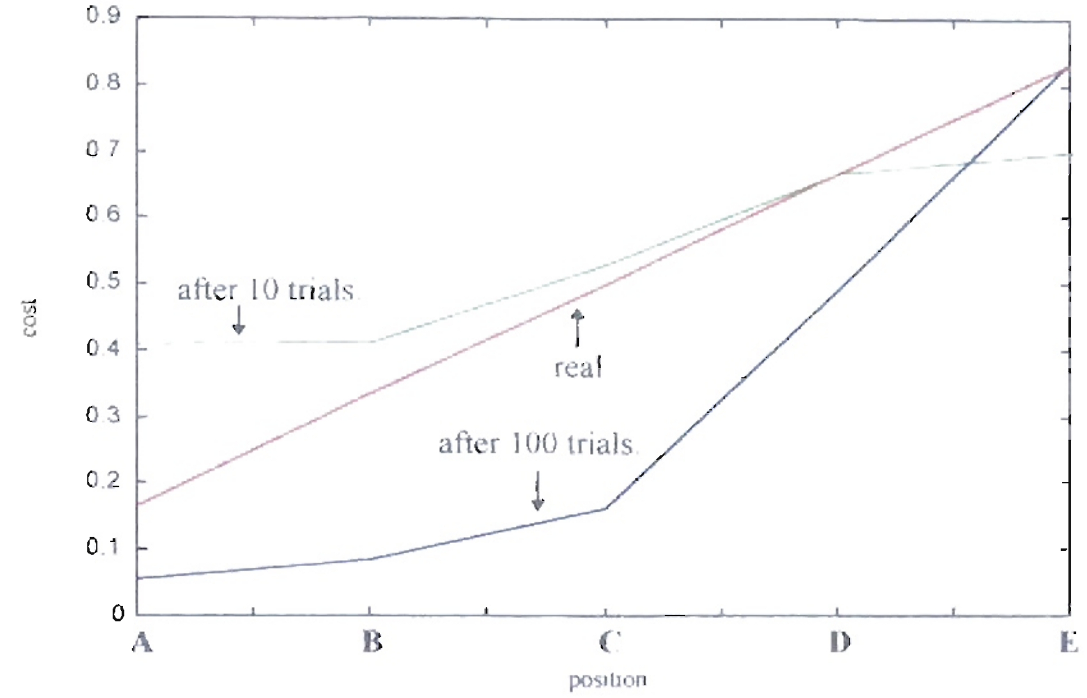


Figure 6.15: *Cost-to-go values learned by MC.*

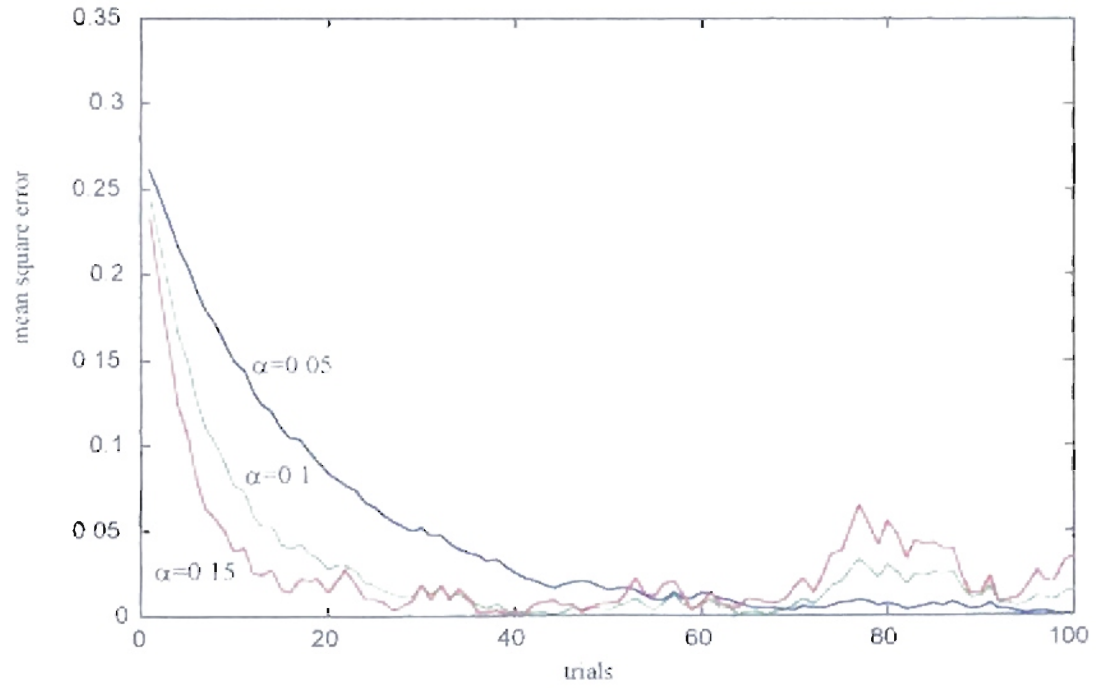


Figure 6.16: *Learning curves for TD(0).*

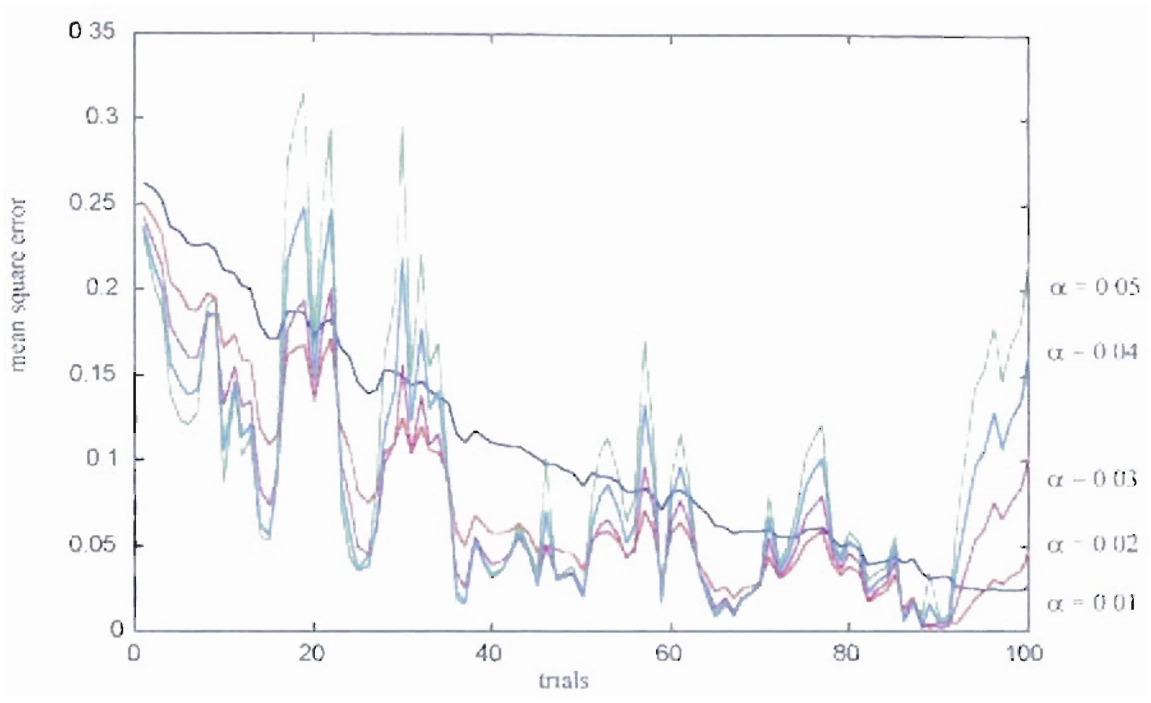


Figure 6.17: *Learning curves for MC.*

6.10.4. n-step TD prediction.

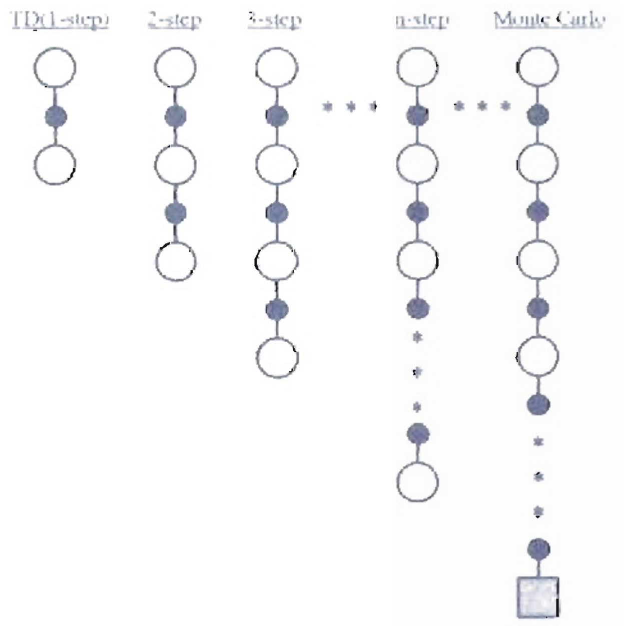


Figure 6.18: *n-step TD prediction* (12).

Figure 6.18 shows different processes where the white circles represent the states s_t and the black circles represent the rewards r_t. For different For Monte Carlo methods the estimate $J_t(s_t)$ of the cost-to-go function of the current policy $J^{\pi}(s_t)$ is updated in the direction of the complete return:

$$R_t = r_{t+1} + \gamma r_{t+1} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$$

From 1 to n steps backup the reward is:

$$R_t^{(1)} = r_{t+1} + \gamma J_t(s_{t+1})$$

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 J_t(s_{t+2})$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+1} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n J_t(s_{t+n}) \qquad \textbf{(6.30)}$$

The estimated value of $J^{\pi}(s_t)$ at the time t is $J_t(s_t)$ due to an n-step backup of s_t is:

$$\Delta J_t(s_t) = \alpha[R_t^n - J_t(s_t)] \qquad \textbf{(6.31)}$$

If we use On-line updating, we update each step with the relation:

$$J_{t+1}(s) = J_t(s) + \Delta J_t(s) \qquad \textbf{(6.32)}$$

If we use Off-line updating, we update at the end of the episode with the relation:

$$J(s) = J(s) + \sum_{t=0}^{T-1} \Delta J(s) \qquad \textbf{(6.33)}$$

6.10.5. Sarsa: On policy TD Control. If we want to execute the training of a learning process based on generalized policy iteration (GPI), we must combine exploration and exploitation. Sutton and Barto(12) define two classes of methods: On-policy and Off-policy.

The On-policy TD method tries to estimate the function $Q^{\pi}(s, a)$ for all the states s and the actions a. Then we can apply the same relation used for the cost-to-go function to the Q-factor function as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad \textbf{(6.34)}$$

If s_{t+1} is terminal, then we define the last Q-factor function $Q(s_{t+1}, a_{t+1})$ to be zero. The name for the Sarsa algorithm comes from the use of the five parameters $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. An implementation of the Sarsa algorithm is given in Figure 6.19.

```
Sarsa algorithm: On policy TD Control.
Initialize $Q(s, a)$ arbitrarily.
Repeat (for each episode):
    Initialize $s$.
    Choose $a$ from $s$ using policy derived from $Q$ ($\varepsilon$-greedy).
    Repeat (for each step of episode):
        Take action $a$; observe reward $r$, and next state $s_F$.
        Choose $a_F$ from $s_F$ using policy derived from $Q$ ($\varepsilon$-greedy).
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s_F, a_F) - Q(s, a)]$
        $s \leftarrow s_F, a \leftarrow a_F$
    until $s$ is terminal.
```

Figure 6.19: *Sarsa Algorithm (12).*

6.10.5.1. Windy Gridworld example. We can apply the Sarsa algorithm to the Windy Gridworld example seen in Figure 6.20. For this example we have one start and one goal point, called S and G respectively. Each episode starts at the point S and finishes at the point G. We have four possible movements shown in Figure 6.20. If we try to move outside the Gridworld we remain in the same position. For each movement there is a penalty of -1 until we arrive at the goal G. Our movements are complicated by a wind that moves from the bottom of the grid with a force described in Figure 6.20. The optimal path is described by the 15 steps at the lower part of the same figure.

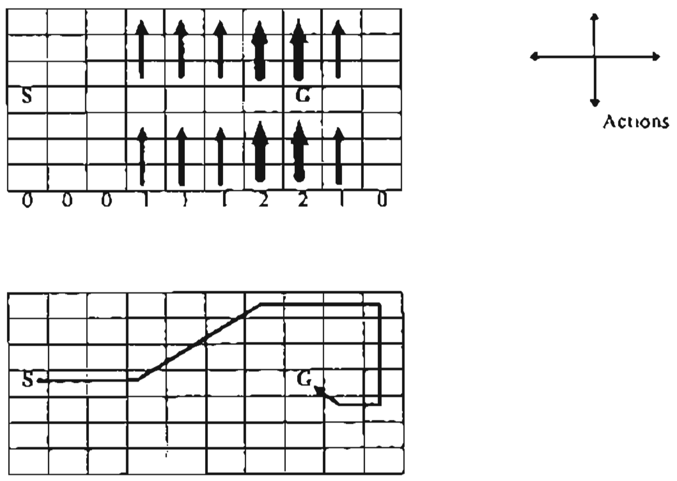


Figure 6.20: *Gridworld. Basic operation and real optimal path* $_{(12)}$.

We executed the Sarsa algorithm for the Gridworld with a learning rate $\alpha = 0.1$. The movements in the Gridworld were greedy (taking the action with the maximum action-value). If two or more actions have the same action-value we select one at random. To search new solutions, we also included a probability $\varepsilon = 0.1$ of random actions. We found the solution shown in Figure 6.21, where we drew the optimal movement for each position. If we follow the suggested movements from the start S we will follow the optimal path shown in Figure 6.20. We note that some positions do not have suggestions because they were not visited, due to the wind effect.

Optimal path:

↓	→	→	→	→	→	→	→	→	↓
→	↓	→	→	→	→	→	→	↑	↓
↓	→	↓	→	→	→	→	↓	→	↓
S→	→	→	→	→	→	→	G	→	↓
↓	↓	↓	→	→	→		↓	←	←
→	→	→	→	→				→	↑
→	→	↑	↑					↑	←
0	0	0	1	1	1	2	2	1	0

Figure 6.21: *Gridworld. Calculated optimal path.*

In Figure 6.22 we plotted the number of episodes completed versus the total number of steps. We can see at the beginning it took many steps to complete each episode. As the training process continues, the episodes concluded in fewer steps.

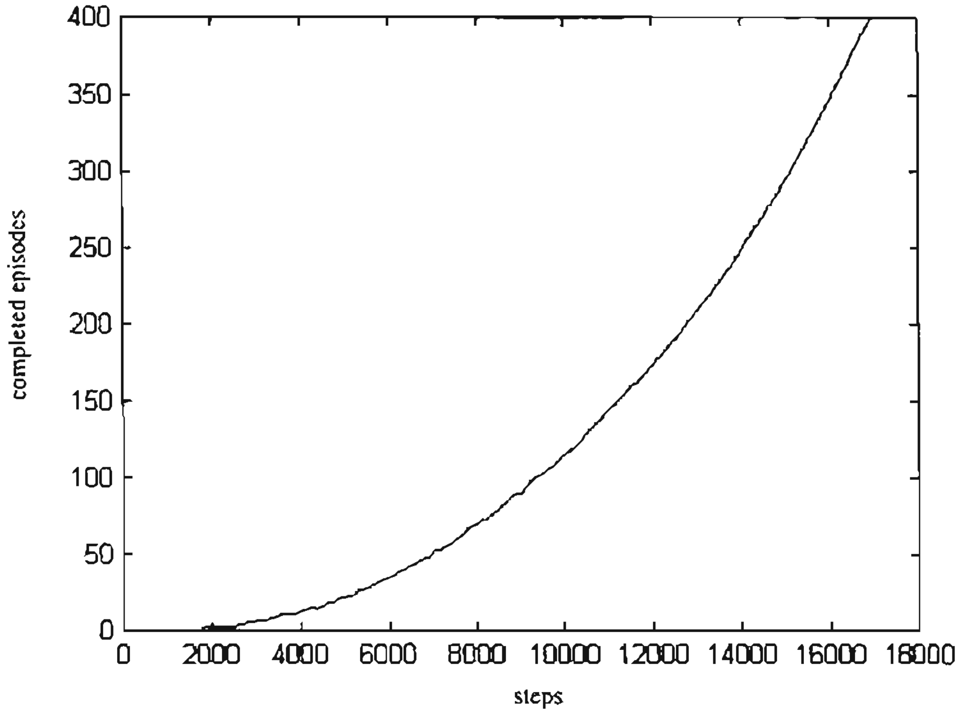


Figure 6.22: *Number of completed episodes versus number of steps.*

6.10.6. Q-learning: Off policy TD control. This TD method is Off policy because is independent of the current policy. The method takes the current Q-factor function Q as the approximate real function Q^*. For example, for 1-step Q-learning we have:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{6.35}$$

An implementation of the Q-learning algorithm is shown in Figure 6.23.

```
*Q-learning algorithm: Off policy TD Control.*
Initialize $Q(s, a)$ arbitrarily.
Repeat (for each episode):
    Initialize $s$.
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ ($\varepsilon$-greedy).
        Take action $a$; observe reward $r$, and next state $s_F$.
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a} Q(s_F, a) - Q(s, a)\right]$
        $s \leftarrow s_F$
    until $s$ is terminal.
```

Figure 6.23: *Q-learning: Off policy TD control* (12).

6.10.6.1. Cliff Walking. We execute the Cliff Walking example of Figure 6.24 to compare Q-learning and Sarsa algorithms. We want to move from the Start to the Goal position with the movements shown in the same figure. An episode concludes when we arrive at the Goal position. For each movement we have a penalty of -1, except at the Cliff zone where we receive a punishment of -100 and return to the Start position. We can consider two possible trajectories: an optimal path running very close to the Cliff zone and a safe path running along the safe path far from the Cliff zone. If we compare both algorithms, we can see that the Q-learning policy tends to move closer to the Cliff zone than the Sarsa method. Those results are due to the implementation of the Q-learning method that is based on the current optimal Q-factor function (action where $Q(s_{t+1}, a)$ is maximum). The Sarsa method is based on the next action executed and, due to the penalty at the Cliff zone, the algorithm tends to move toward the safest zone.

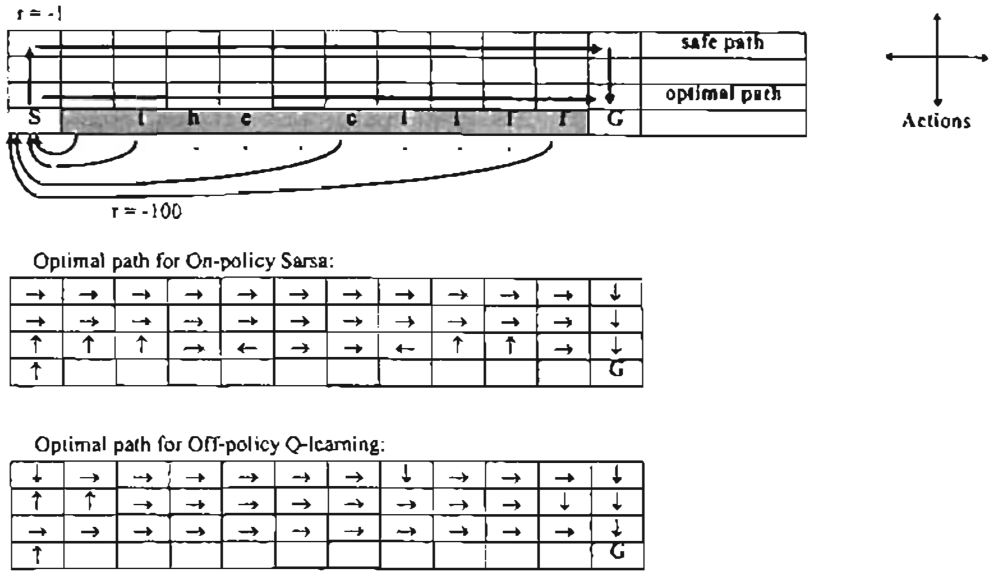

r = -1
safe path
optimal path
S
the cliff
G
r = -100
Actions
Optimal path for On-policy Sarsa:
Optimal path for Off-policy Q-learning:

Figure 6.24: *Cliff walking* (12).

6.10.7. Actor-Critic Methods. This method separates in two independent variables the policy (known as the actor) and the cost-to-go function (known as the critic) (see Figure 6.25). The critic learns about the process and critiques the current policy. The learning is always On-policy. The critic is generally a cost-to-go function, because after each state transition the critic must compare the results from all possible actions looking for any improvement. The critique takes the form of a TD error (see Eq. (6.27)) :

$$\delta_t = r_{t+1} + \gamma J(s_{t+1}) - J(s_t) \tag{6.36}$$

where J is the cost-to-go function evaluated by the critic.

With this relation we evaluate the selected action a_t at the state s_t. A Positive TD error means that the critic will support that action at the future. A negative TD error reduces the support to execute that action.

For example, we can define an actor's policy $\pi_t(s, a)$ with modifiable policy parameters $p(s, a)$, then we can update those parameters with the relation:

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t \tag{6.37}$$

where β is a positive step-size parameter. Another update relation could be:

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t[1 - \pi_t(s_t, a_t)] \tag{6.38}$$

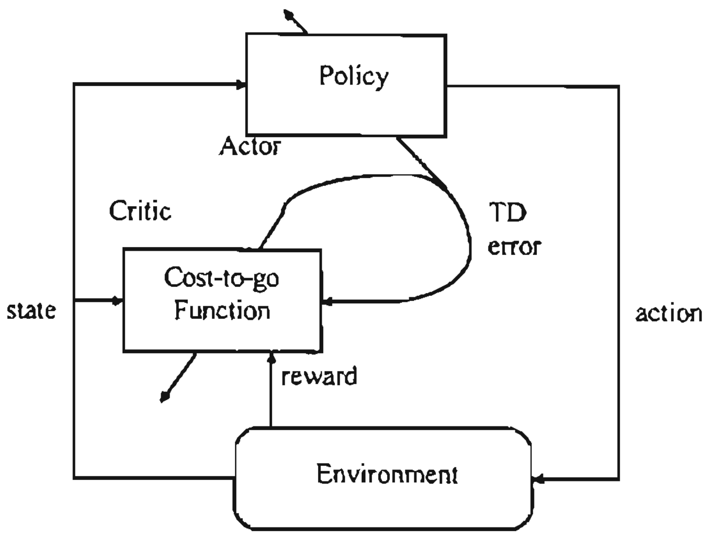


Figure 6.25: *Actor-critic method* (12).

6.11. Unified algorithms.

In this section we want to combine concepts from dynamic programming, Monte Carlo and temporal difference methods.

6.11.1. Eligibility traces. The idea of eligibility traces is related to the update of the cost-to-go function, defining how each visited state influences that update process. For example, we want states which have not been visited to have little or no influence on the calculation of the cost-to-go function. On the other hand, frequently visited states must have an important role in that calculation.

We can use the idea of eligibility traces to define the transition from the TD method to the Monte Carlo method. If we use the TD method, as defined in section 6.10.4, we could increase the number of steps n in the time horizon as we move in the future to explore a solution, arriving at the final time to the Monte Carlo method. This view is called the forward view, as seen in Figure 6.18, because we are going in the same direction as the process. We known which states are visited and therefore are used at the cost-to-go function update.

Another form of describing Eligibility Traces is called the backward view. Here we can see the Eligibility Traces as "a temporary record of the occurrence of an event."(12) When an event or state occurs, that event is marked with one flag, which is later discounted in time. When a TD occurs, the error is charged to the visited events according to their discount value.

"The more theoretical view of eligibility traces is called the forward view, and the more mechanistic view is called the backward view. The forward view is most useful for understanding what is computed by methods using eligibility traces, whereas the backward view is more appropriate for developing intuition about the algorithms themselves."(12)

A temporal difference algorithm that uses eligibility traces is called TD(λ).

6.11.2. The forward view of TD( ). This a theoretical point of view of TD( ). We do not have to update our cost-to-go functions with the n-step return. We can update with an average of n-step returns with the relation:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{6.39}$$

For a finite process of length T:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t \tag{6.40}$$

where the increment of the cost-to-go function is:

$$\Delta J_t(s_t) = \alpha[R_t^{\lambda} - J_t(s)] \tag{6.41}$$

Since we need future results of the cost-to-go function, we can see that this implementation is not causal.

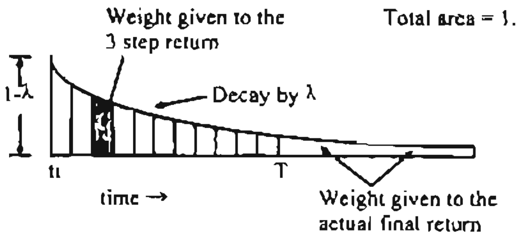

Weight given to the 3 step return
Total area = 1.
1-λ
Decay by λ
t
T
time →
Weight given to the actual final return

Figure 6.26: *Weighting given in the λ-return to each of the n-step return* (12).

6.11.3. The backward view of TD( ). This a mechanistic point of view of the eligibility traces. This implementation depends on past values, therefore it is causal. We have a value associated with each state, called the eligibility trace. "On each step, the eligibility traces for all states decay by $\gamma\lambda$, and the eligibility trace for the one state visited on the step is incremented by 1"(12), as shown in Figure 6.27.

$$e(t) = \begin{cases} \gamma\, \lambda\, e_{t-1}(s) & \text{If } s \neq s_t \\ \gamma\, \lambda\, e_{t-1}(s) + 1 & \text{If } s = s_t \end{cases} \tag{6.42}$$

where: $\gamma \rightarrow$ Discount rate.

$\lambda \rightarrow$ Parameter used by the forward view (previous section).

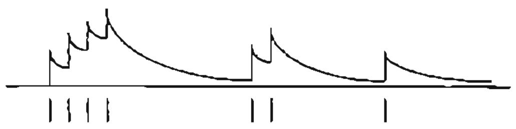


Figure 6.27: *Graphical representation of backward view of eligibility traces (12).*

"The traces are said to indicate the degree to which each state is eligible for undergoing learning changes should a reinforcement event occur."(12)

$$\Delta J_t(s) = \alpha\, \delta_t\, e_t(s) \qquad \text{for all } s \in S \tag{6.43}$$

where:

$$\delta_t = r_{t+1} + \gamma J_t(s_{t+1}) - J_t(s_t) \tag{6.44}$$

For example in the case of 1-step TD error we have:

If:	$\lambda = 0$	$\Rightarrow$ TD(0)
	$\lambda = 1$	$\Rightarrow$ The credit error falls by γ per step.
	$\lambda = 1, \gamma = 1$	$\Rightarrow$ TD(1). The eligibility traces do not decay at all with time. Work as Monte Carlo undiscounted episodic task.
	λ large, but still $\lambda < 1$	$\Rightarrow$ More of the preceding states are changed, but the more precedent is changed less.

An example of the application of this concept is shown in Figure 6.28, where we use eligibility traces to estimate J^{π} for a given policy π.

```
On line Tabular TD(λ) for estimating J^π.
Initialize:
    $J(s) \leftarrow$ arbitrarily.
    $e(s) = 0$ for all $s \in S$.
Repeat (for each episode):
    Initialize $s$.
    Repeat (for each step of episode):
        $a \leftarrow$ action given by $\pi$ for $s$.
        Take action $a$; observe reward $r$, and next state $s_F$.
        $\delta \leftarrow r + \gamma J(s_F) - J(s)$
        $e(s) \leftarrow e(s) + 1$
        For all $s$:
            $J(s) \leftarrow J(s) + \alpha \delta e(s)$
            $e(s) \leftarrow \gamma \lambda e(s)$
        $s \leftarrow s_F$.
    until $s$ is terminal.
```

Figure 6.28: *On-line Tabular TD(λ) for estimating J^{π} (12).*

6.12. Gradient-Descent Methods.

This section describes how we can construct approximate representations of the cost-to-go function using neural networks or similar structures. The same results will apply to the Q-factor functions. Bertsekas and Tsitsiklis$_{(3)}$ stated that we must define an approximation architecture powerful enough to approximate the desired function, and we must establish effective algorithms to execute the training process. Generally, these objectives are conflicting, because powerful architectures generally imply large numbers of parameters and nonlinear internal relations.

A general approximation process with neural networks is based on data pairs (x_i, y_i), where we want to find the function $y = f(x)$ that is the best approximation to that data set, as shown in Figure 6.29.

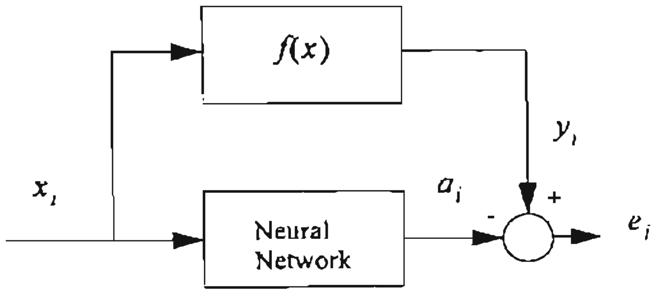


Figure 6.29: *General neural network training.*

In the reinforcement learning context, we want to obtain the optimal cost-to-go function J^*, based in the data pairs $(s_i, J^*(s_i))$, where s_i is contained in a subset of the state space. However, since the function J^* is unknown or not measurable, the training pairs are unavailable. In that case, we need a training algorithm which also tries to compute J^*. One idea is to obtain the approximate cost-to-go function $\hat{J}^{\pi}(s)$ during one episode, starting from the state s_0 and following the policy π. For example, this approximate cost-to-go function can be obtained using Monte Carlo simulation (as in Eq. (6.26)) or by temporal difference method (as in Eq. (6.27)). As a result of this process we will obtain training pairs for the neural network $(s_t, \hat{J}^{\pi}(s_t))$ as shown in Figure 6.30.

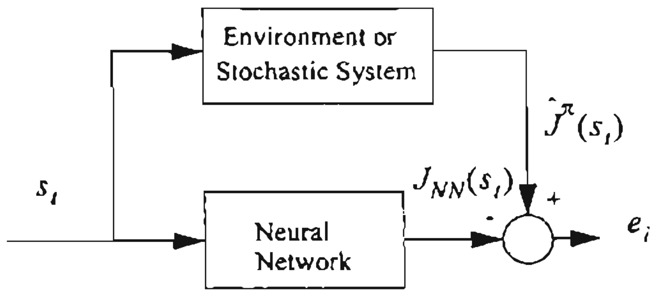


Figure 6.30: *Neural network training for a Reinforcement Learning problem.*

For the neural network training the inputs are the states visited during the episode. The target will be the approximate cost-to-go function $\hat{J}^{\pi}(s_t)$. The network will be trained to minimize the mean square error:

$$\mathrm{MSE}(\vec{\theta}_t) = P(s_t) \sum_{s \in S} (\hat{J}^{\pi}(s_t) - J_{NN}(s_t))^2 \tag{6.45}$$

where $P(s_t)$ is the state probability mass function. If the states appear with the same distribution P, we can minimize the error on the observed examples by the following steepest descent algorithm:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}(\hat{J}^{\pi}(s_t) - J_{NN}(s_t))^2$$

$$= \vec{\theta}_t - \alpha(\hat{J}^{\pi}(s_t) - J_{NN}(s_t))\nabla_{\vec{\theta}_t}(J_{NN}(s_t)) \qquad \textbf{(6.46)}$$

where α is a positive step-size parameter and $\vec{\theta}$ is the vector of neural network weights and biases.

After the neural network has been trained it provides a new updated policy because we can then determine the action that minimizes the cost-to-go function. We must then continue to update the cost-to-go function.

If TD methods are used to estimate the cost-to-go values, we have $\hat{J}^{\pi}(s_t) = R_t^{\lambda}$ for the forward view update (Eq. (6.39) and Eq. (6.40)) and we can update the weights and biases with the relation:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha(R_t^{\lambda} - J_{NN}(s_t))\nabla_{\vec{\theta}_t}(J_{NN}(s_t)) \qquad \textbf{(6.47)}$$

but, for $\lambda < 1$, R_t^{λ} is a non causal approximation to $J^{\pi}(s_t)$, and this is not a practical implementation. For the backward view TD update we have:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha\delta_t\vec{e}_t \qquad \textbf{(6.48)}$$

where δ_t is the TD error:

$$\delta_t = r_{t+1} + \gamma J_{NN}(s_{t+1}) - J_{NN}(s_t) \qquad \textbf{(6.49)}$$

and $\vec{e}_t$ is a vector of eligibility traces defined by:

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t}(J_{NN}(s_t)) \qquad \textbf{(6.50)}$$

with $\vec{e}_o = \vec{0}$.

*6.12.1. **Linear Methods.*** This is the case when the cost-to-go function J_{NN} is a linear function of the parameter vector $\vec{\theta}_t$:

$$J_{NN}(s_t) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \vec{\theta}_t^T(i)\ \vec{\phi}_s(i) \qquad \Rightarrow \nabla_{\vec{\theta}_t}(J_{NN}(s_t)) = \vec{\phi}_s \tag{6.51}$$

where $\vec{\phi}_s(i) = (\phi_s(1), \phi_s(2), \ldots, \phi_s(n))^T$ is a vector of features which correspond to each state s. Sutton and Barto(12) defined $\vec{\phi}_s(i)$ as a vector of features and Bertsekas and Tsitsiklis(3) defined $\vec{\phi}_s(i)$ as basis functions. We can rewrite the mean square error equation as:

$$\text{MSE}(\vec{\theta}_t) = P(s_t) \sum_{s \in S} \left(J(s_t) - \sum_{i=1}^{n} \vec{\theta}_t^T(i)\ \vec{\phi}_s(i) \right)^2 \tag{6.52}$$

We can use different types of basis functions to obtain the cost-to-go function. For example, Sutton and Barto mention coarse coding, tile coding, radial basis functions and Kanerva coding(12). We will explain the tile coding that will be used in the final simulations.

6.12.2. Tile Coding (CMAC NN).

For tile coding (or CMAC neural networks), we divide the state space into m subspaces as shown in Figure 6.31 for a two dimensional space. The division could be with the same spacing or with arbitrary spacing. Arbitrary shapes are also allowed.

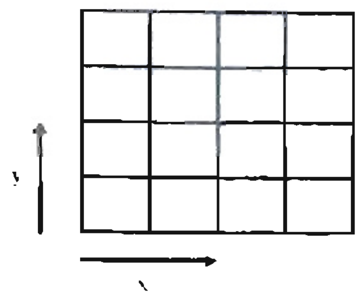


Figure 6.31: *Two-dimensional Tile Coding (12).*

For each state-value only one of the tiles will be active at a time. Therefore, the output of the network will be the weight associated with that tile corresponding to the current state-value. Therefore, the gradient $\nabla_{\vec{\theta}_t}(J_{NN}(x_t))$ will usually be equal to 1 for the active tile and equal to 0 for all the others.

To obtain a better resolution in the implementation of the tiles, we can include an additional tile for each dimension, and the tilings can be shifted by a random number, as shown for the two dimensional case in Figure 6.32. In that figure we have the original space in black with two displaced tilings in blue and green.

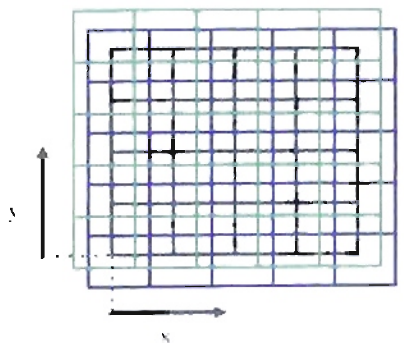


Figure 6.32: *Two-dimensional Tile Coding with two tiles (green and blue) (12).*

*6.12.3. **Control with function approximation.*** We can extend the previous concept of cost-to-go function prediction to Q-factor function prediction. With the Q-factor function we can obtain the cost-to-go derived from each possible action for each state. We can define Q^{π} as a function of $\vec{\theta}_t$. We can say that the Q-factor function will generate an output of the form $s_t, a_t \rightarrow \hat{Q}^{\pi}$, where $\hat{Q}^{\pi}$ could be any approximation of $Q^{\pi}(s_t, a_t)$, as the Monte Carlo return R_t or the 1-step Sarsa-style return $r_{t+1} + \gamma Q_{NN}(s_{t+1}, a_{t+1})$. Then we can write the gradient-descent update for the Q-factor function as:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(\hat{Q}^{\pi}(s_t, a_t) - Q_{NN}(s_t, a_t))\nabla_{\vec{\theta}_t}(Q_{NN}(s_t, a_t)) \qquad (6.53)$$

For the backward view of TD(λ), using the CMAC NN, we have that:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \qquad (6.54)$$

where:

$$\delta_t = r_{t+1} + \gamma Q_{NN}(s_{t+1}, a_{t+1}) - Q_{NN}(s_t, a_t) \qquad (6.55)$$

and:

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1}(s) + \nabla_{\vec{\theta}_t}(Q_{NN}(s_t, a_t)) \qquad (6.56)$$

We can apply the update algorithm for a reinforcement learning problem following the process described in Figure 6.33.

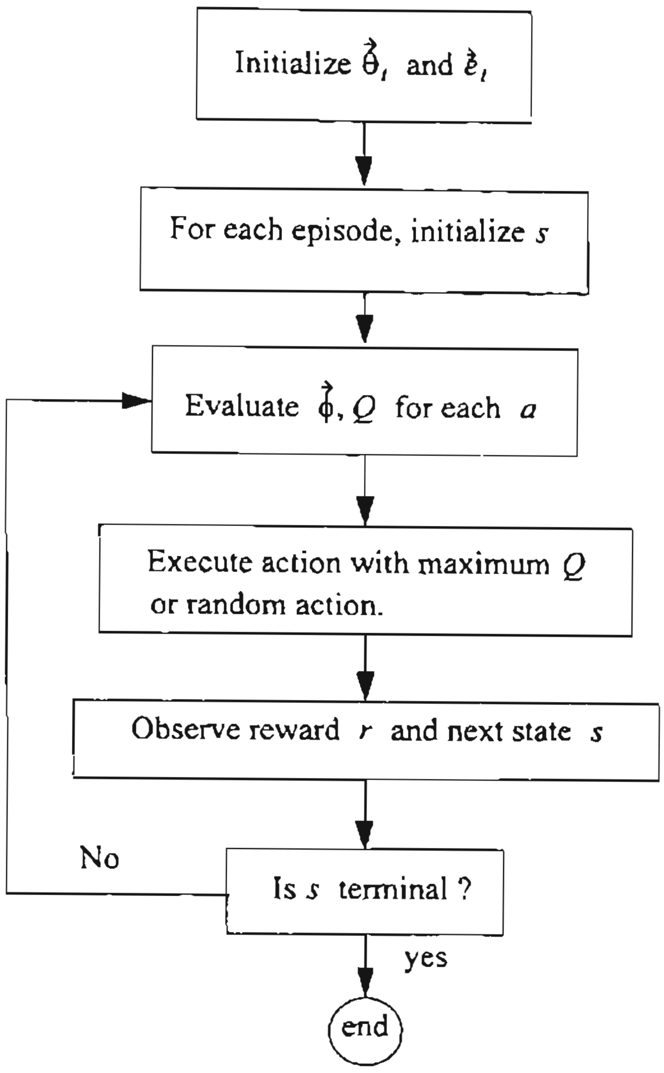

Initialize $\vec{\theta}_t$ and $\vec{e}_t$
For each episode, initialize s
Evaluate $\vec{\phi}$, Q for each a
Execute action with maximum Q or random action.
Observe reward r and next state s
Is s terminal ?
No
yes
end

Figure 6.33: *General linear update algorithm*

An example of On-policy approximation is the linear, gradient-descent Sarsa(λ) algorithm shown in Figure 6.34. An Off-policy implementation is the linear, gradient-descent version of Watkins's Q(λ) as shown in Figure 6.35.

```
Linear, gradient-descent Sarsa ($\lambda$).

Initialize $W$ arbitrarily (preferable zero). Initialize $\vec{e} = \vec{0}$.
Repeat (for each episode):
    Initialize $s$ (random or fixed initial condition).
    For all $a \in A(s)$:
        $F_a \leftarrow$ set of features present in $s, a$.
        $Q_a \leftarrow \sum_{i \in F_a} W(i)$
    $a \leftarrow \arg\max_a Q_a$
    With probability $\varepsilon$: $a \leftarrow$ a ramdom action $\in A(s)$.
    Repeat (for each step of episode):
        $\vec{e} \leftarrow \gamma\lambda\vec{e}$
        For all $\bar{a} \neq a$:            (optional block for replacing traces)
            For all $i \in F_{\bar{a}}$:
                $e(i) \leftarrow 0$
        For all $i \in F_a$:
            $e(i) \leftarrow e(i) + 1$          (accumulating traces)
            or $e(i) \leftarrow 1$              (replacing traces)
        Take action $a$; observe reward $r$, and next state $s_F$.
        $\delta \leftarrow r - Q_a$
        If $s_F$ is not terminal:
            For all $a \in A(s)$:
                $F_a \leftarrow$ set of features present in $s, a$.
                $Q_a \leftarrow \sum_{i \in F_a} W(i)$
            $a_F \leftarrow \arg\max_a Q_a$.
            With probability $\varepsilon$: $a_F \leftarrow$ a ramdom action $\in A(s)$.
        If $s^f$ is terminal then: $Q_{a_F} = 0$
        $\delta \leftarrow \delta + \gamma Q_{a_F}$
        $W \leftarrow W + \alpha\delta e$
        $s \leftarrow s_F$, $a \leftarrow a_F$
        Recalculate: $Q_a \leftarrow \sum_{i \in F_a} W(i)$
    until $s$ is terminal.
```

Figure 6.34: *Linear, gradient-descent Sarsa (λ)* (12).

```
Linear, gradient-descent Q(λ).
Initialize $W$ arbitrarily (preferable zero).
Initialize $\vec{e} = \vec{0}$.
Repeat (for each episode):
    Initialize $s$ (random or fixed initial condition).
    For all $a \in A(s)$:
        $F_a \leftarrow$ set of features present in $s$, $a$.
        $Q_a \leftarrow \sum_{i \in F_a} W(i)$
Repeat (for each step of episode):
    With probability $1 - \varepsilon$:
        $a \leftarrow \arg\max_a Q_a$.
        $e \leftarrow \gamma\lambda e$
    else
        $a \leftarrow$ a random action $\in A(s)$.
        $e \leftarrow 0$
    For all $i \in F_a$: $e(i) \leftarrow e(i) + 1$
    Take action $a$; observe reward $r$, and next state $s_F$.
    $\delta \leftarrow r - Q_a$
    For all $a \in A(s)$:
        $F_a \leftarrow$ set of features present in $s_F$, $a$.
        $Q_a \leftarrow \sum_{i \in F_a} W(i)$
    $a_F \leftarrow \arg\max_a Q_a$.
    $\delta \leftarrow \delta + \gamma Q_{a_F}$
    $W \leftarrow W + \alpha\delta e$
    until $s$ is terminal.
```

Figure 6.35: *Linear, gradient-descent Q(λ)* (12).

6.12.4. Mountain-Car example. As an example, we can consider the task of driving a Mountain-Car task as suggested in Figure 6.36. Since the car does not have enough power to climb the hill directly, the intuitive solution implies that we must first go backwards and then accelerate forward. We want to minimize the time to climb that hill starting from a random initial position.

The simplified equations of the car are:

$$x_{t+1} = \text{bound}[x_t + \dot{x}_{t+1}] \tag{6.57}$$

$$\dot{x}_{t+1} = \text{bound}[\dot{x}_t + 0.001 a_t - 0.0025 \cos(3x_t)] \tag{6.58}$$

where the bound function enforces a limit in the state variables: $-1.2 \le x_{t+1} \le 0.5$ and $-0.07 \le \dot{x}_{t+1} \le 0.07$.

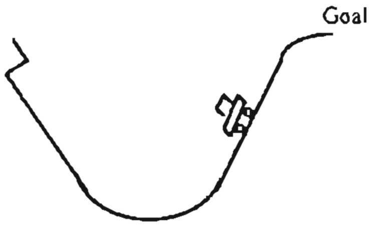


Figure 6.36: *Mountain Car task* (12).

The control input to the system has three options: full throttle forward ($a_t = 1$), full throttle reverse ($a_t = -1$), and zero throttle ($a_t = 0$). The reward function is -1 until the car passes Goal position when the episode ends. We use the Sarsa learning method with parameters $\lambda = 0.9$, $\varepsilon = 0$, $\alpha = 0.05(0.1/m)$.

With all the initial actions set to zero we will select randomly between equal cost-to-go functions. Thus we have an extensive exploration, even though $\varepsilon = 0$. We used the same tiling scheme proposed by Sutton and Barto(12). We divided the state variables into 10 tilings divided into 9 x 9 equally spaced segments.

We trained the system for 9000 episodes. Approximate solutions after 104 episodes and after 9000 episodes are shown in Figure 6.37 and Figure 6.38 respectively. We can see that after 104 episodes we have an approximate representation of the optimal cost-to-go function shown in Figure 6.39.

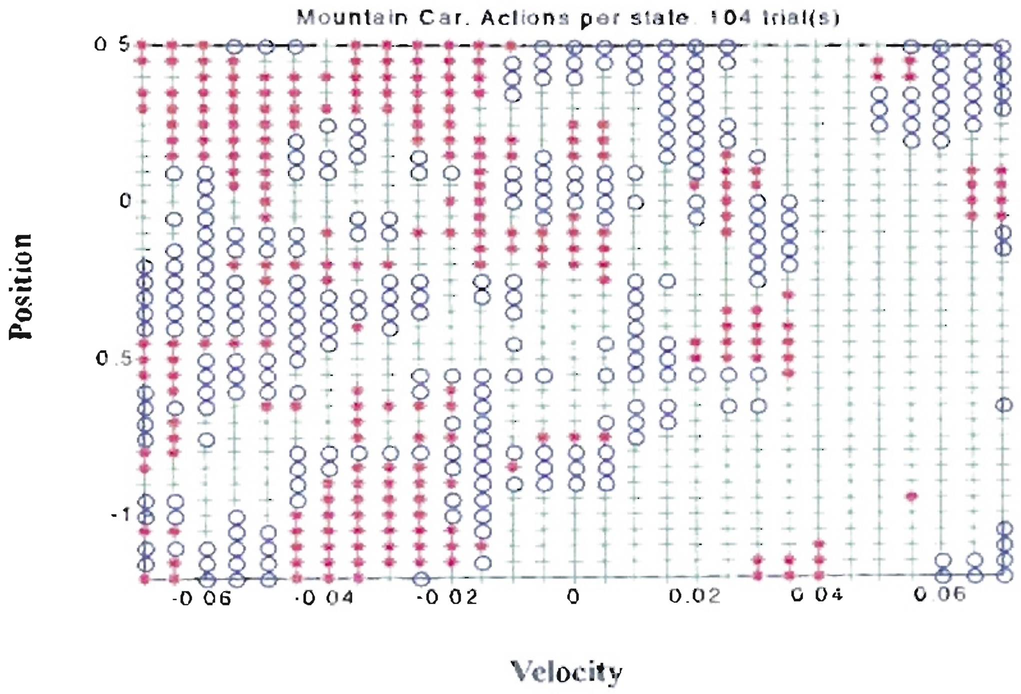


Figure 6.37: *Approximate Action per state after 104 episodes ((lft), *(rt), O(zr)).*

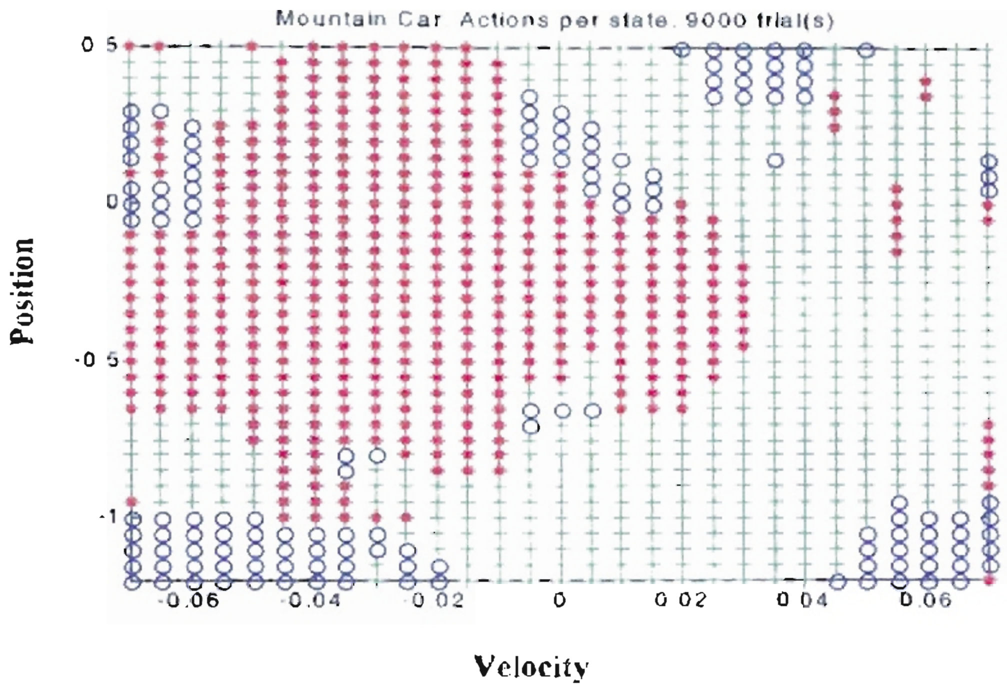


Figure 6.38: *Approximate Action per state after 9000 episodes ((fl), *(fr), O(zr)).*

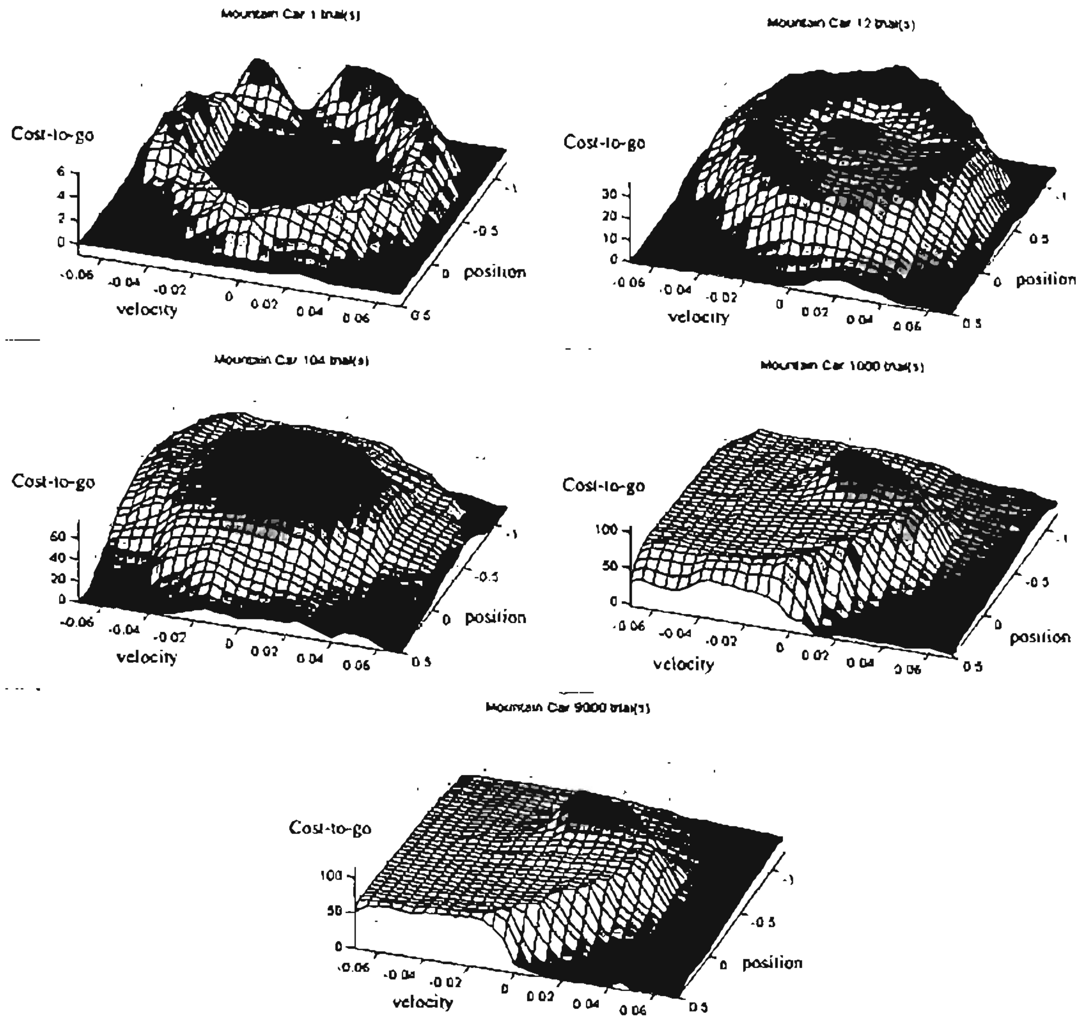


Figure 6.39: *Cost-to-go function ($-\max_a Q_t(s,a)$) learned during one run.*

In Figure 6.39 we can see the cost-to-go function ($-\max_a Q_t(s_t, a_t)$) learned during one run. We can see that after one episode we have a circular cost representation of the back and forth movement of the car. As the number of training episodes increases, the cost function takes shape. For 1000 episodes the cost function shape is so similar to the cost function for 9000 episodes that we can consider is to be a close approximation to the real cost function.

6.12.5. Acrobot example. As a second example, we can consider a reinforcement learning problem applied to an Acrobot. "The Acrobot is an underactuated two-link planar robot that mimics the human acrobat who hangs from a bar and tries to swing up to a perfectly balanced upside-down position with his/her hands still on the bar"(4) as seen in Figure 6.40.

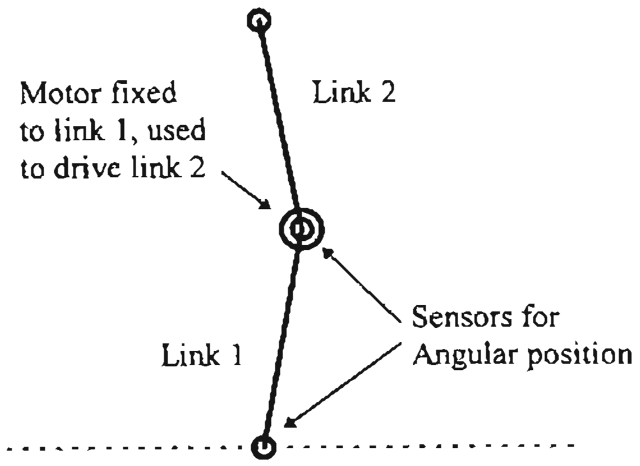

Motor fixed to link 1, used to drive link 2
Link 2
Sensors for Angular position
Link 1

Figure 6.40: *Diagram of an Acrobot (4).*

The dynamic equations for the Acrobot are (4, 11):

$$d_{11}\ddot{q}_1 + d_{12}\ddot{q}_2 + h_1 + \phi_1 = 0 \tag{6.59}$$

$$d_{12}\ddot{q}_1 + d_{22}\ddot{q}_2 + h_2 + \phi_2 = \tau \tag{6.60}$$

Where the coefficients of the relations (6.59) and (6.60) are:

$$d_{11} = m_1 l_{c1}^2 + m_2[l_1^2 + l_{c2}^2 + 2l_1 l_{c2}\cos(q_2)] + I_1 + I_2 \quad (6.61)$$

$$d_{22} = m_2 l_{c2}^2 + I_2 \quad (6.62)$$

$$d_{12} = m_2[l_{c2}^2 + l_1 l_{c2}\cos(q_2)] + I_2 \quad (6.63)$$

$$h_1 = -m_2 l_1 l_{c2}\sin(q_2)\dot{q}_2^2 - 2m_2 l_1 l_{c2}\sin(q_2)\dot{q}_1\dot{q}_2 \quad (6.64)$$

$$h_2 = m_2 l_1 l_{c2}\sin(q_2)\dot{q}_1^2 \quad (6.65)$$

$$\phi_1 = (m_1 l_{c1} + m_2 l_1)g\cos(q_1) + m_2 l_{c2} g\cos(q_1 + q_2) \quad (6.66)$$

$$\phi_2 = m_2 l_{c2} g\cos(q_1 + q_2) \quad (6.67)$$

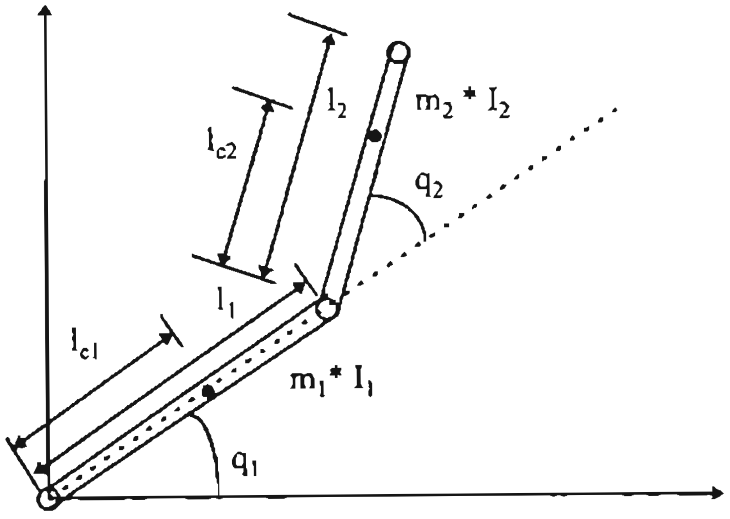


Figure 6.41: *Simple Acrobot notation (4, 11)*.

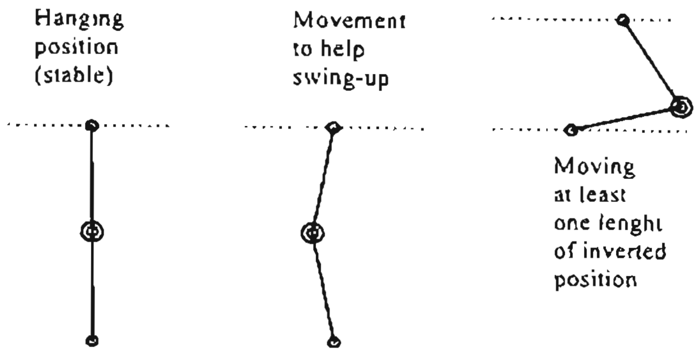

Hanging position (stable)
Movement to help swing-up
Moving at least one lenght of inverted position

Figure 6.42: *Swing-up the Acrobot* (4).

The goal is to swing up in minimum time the lower end of the Acrobot at least one length over the vertical position as shown in Figure 6.42. We always will start in the stable hanging position.

The control input to the system has three options: full positive torque ($\tau_t = 1$), full negative torque ($\tau_t = -1$), and zero torque ($\tau_t = 0$). The reward function is -1 for each step until the Acrobot reaches one length above the inverted position. We use the Sarsa learning method with the parameters $\lambda = 0.9$, $\varepsilon = 0$, $\alpha = 0.2/48$. The angular velocities were limited to $\dot{q}_1 \in [-4\pi, 4\pi]$ and $\dot{q}_2 \in [-9\pi, 9\pi]$ with no limits in the angular values q_1 and q_2. We allowed multiple rotations of the Acrobot links. From the figures that Sutton and Barto presented in their book (12) we noted that those movements were not allowed. For our example we used real parameters taken from the Acrobot of Brown and Passino (4):

$m_1 = 1.9008$;

$m_2 = 0.7175$;

$l_1 = 0.2;$

$l_2 = 0.2;$

$l_{c1} = 0.18522;$

$l_{c2} = 0.062052;$

$I_1 = 0.0043399;$

$I_2 = 0.0052285;$

$g = 9.8;$

We used the same tiling scheme proposed by Sutton and Barto (12). We divided the angle state variables q_1 and q_2 into six equally spaced intervals and the velocity state variables $\dot{q}_1$ and $\dot{q}_2$ into seven equally spaced intervals. We created 12 tilings with four dimensions as discussed before. We created a second group of 12 tilings by taking three of the dimensions for each tiling. We created a third group of 12 tilings with a combination of two dimensions and a final group of 12 tilings with one dimension each one. We offset each tiling by a random fraction of a tile.

We made some simulations with the Acrobot. Initially, we set the initial action values to low random numbers. We set the algorithm for no exploring actions ($\varepsilon = 0$). With those parameters we obtained a system training curve shown in Figure 6.43. From an initial episode of 2206 steps, the system optimizes with a faster execution of 172 steps. We note that the system stops the training after the episode 77, when the algorithm establishes 200 steps as its optimal path. That condition was due to the greedy actions we were taking, and no futures exploration actions were generated. To avoid that problem, we made a change in the algorithm to include exploring actions ($\varepsilon = 0.1$) after two consecutive episodes with the same step number. As shown in Figure 6.44, we avoid the straight line of the previous figure, obtaining a minimal response of 148 steps for one episode.

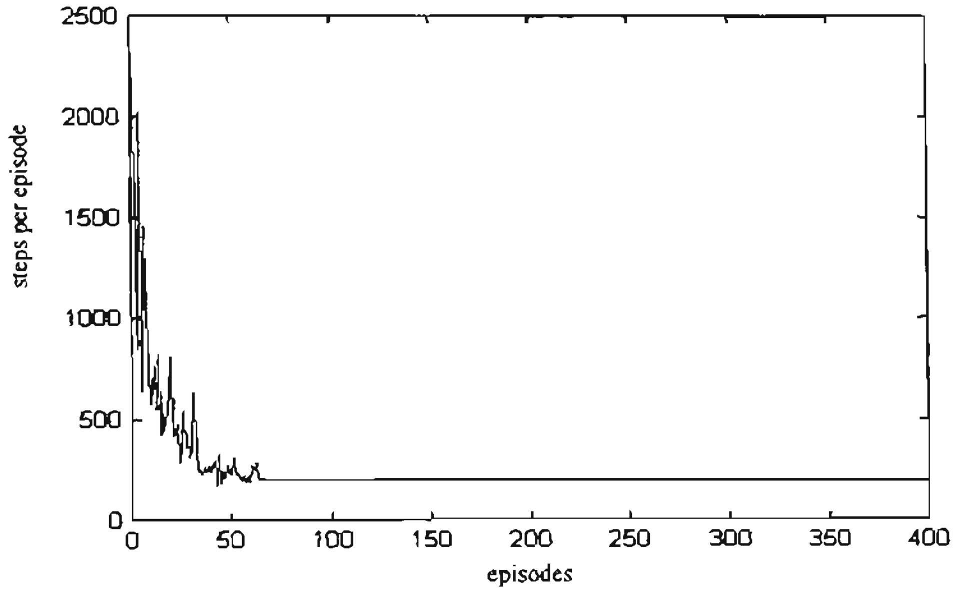


Figure 6.43: *Acrobot. Steps per Trial. Initial random weights. No exploring actions. Minimum steps 172 (one time).*

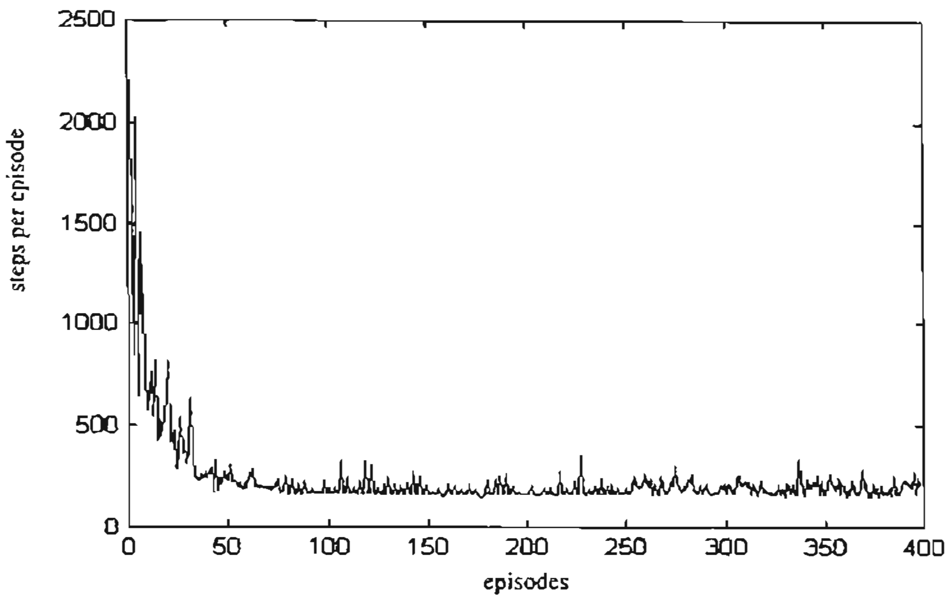


Figure 6.44: *Acrobot. Steps per Trial. Initial random weights. Exploring actions after 2 consecutive episodes with the same steps. Minimum 148 steps (1 time).*

We found a new minimum. However, the system oscillates and the error is not decreasing. For a third experiment (Figure 6.45) we included a decreasing factor in the exploring actions ε. That decreasing factor was taken as the same $\lambda = 0.9$. After two consecutive episodes with the same step number we set $\varepsilon = 0.1$, and that number was decreased by λ. For this change we noted that the peaks after 100 episodes decreased, but our new minimum was 155 steps for one episode.

For a fourth experiment we repeated the first case with the initial action-value set at zero, where we will select randomly between equal Q-factor function values. Theoretically we will have an extensive exploration, even though $\varepsilon = 0$. We note that the exploration was faster than the first case, but in the first case after the episode 53 the training stopped with an optimal path of 177 steps per trial. We see the benefits of the training with initial zero Q-factor function values, but we want to include more exploration as the training continues.

In our final experiment we trained with the initial Q-factor function set at zero, with exploring actions ($\varepsilon = 0.1$) after two consecutive episodes with the same step number. Here we obtained the lower number of 147 steps per episode.

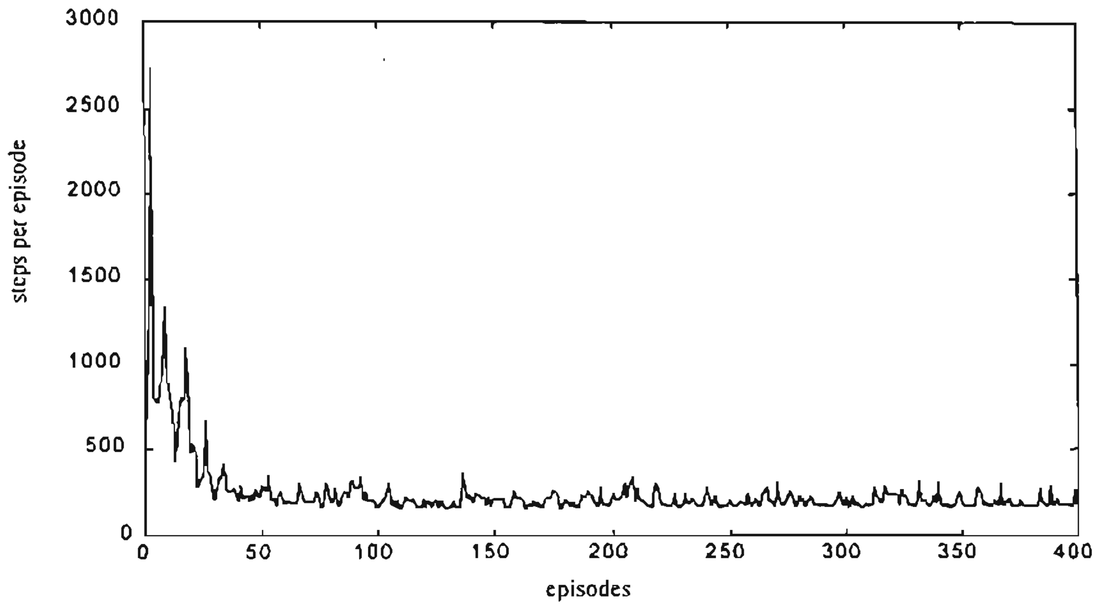


Figure 6.45: *Acrobot. Steps per Trial. Initial random weights. Exploring actions after two consecutive episodes with same steps. decreased by λ. Minimum steps 155 (one time).*

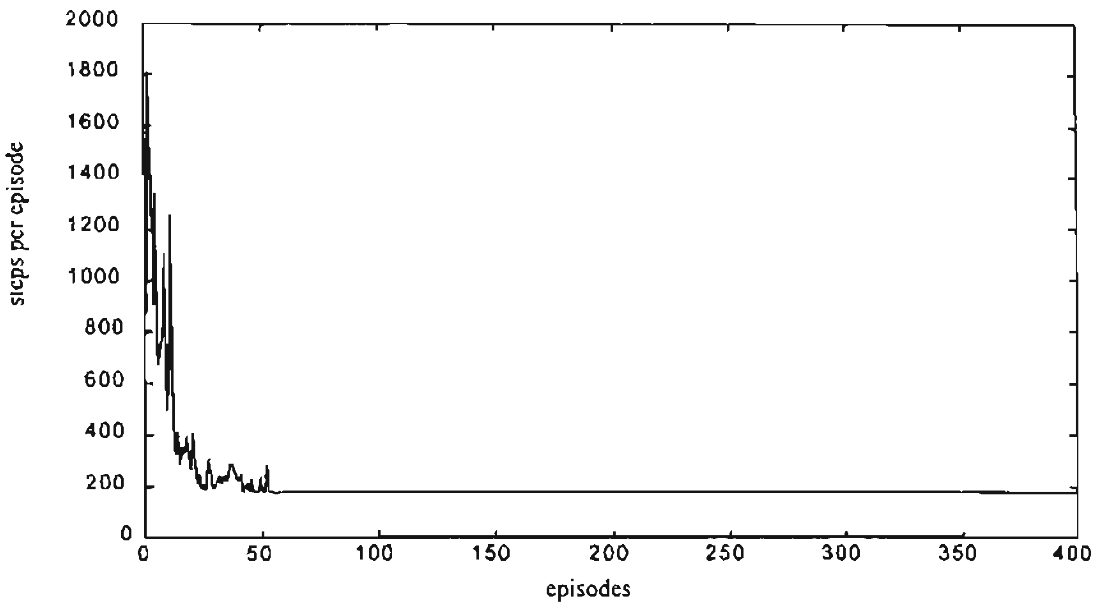


Figure 6.46: *Acrobot. Steps per Trial. Initial zero weights. No exploring actions. Minimum steps 175 (one time).*

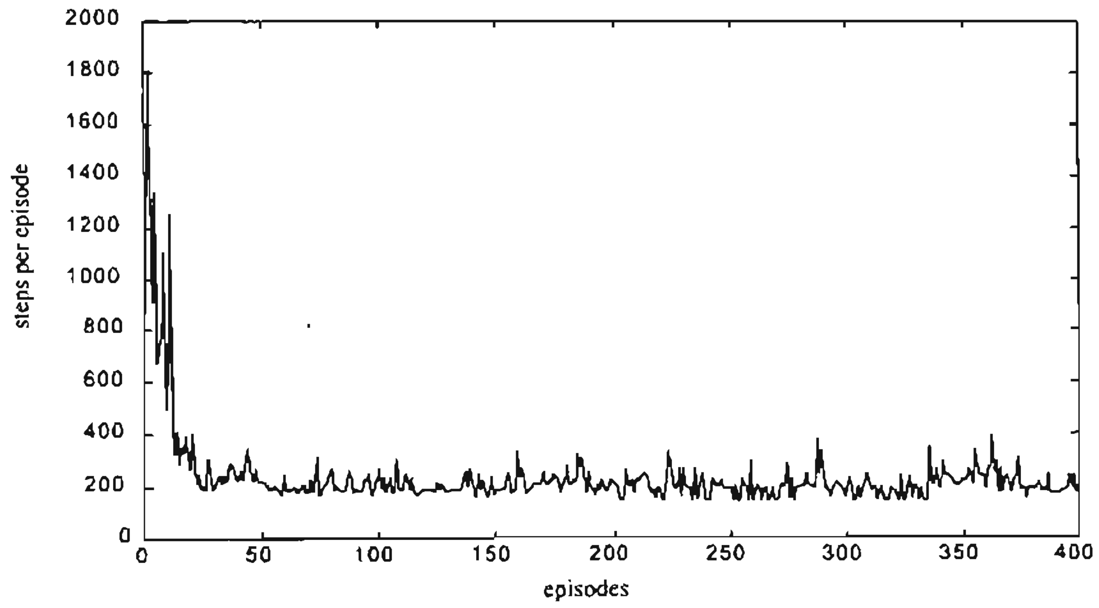
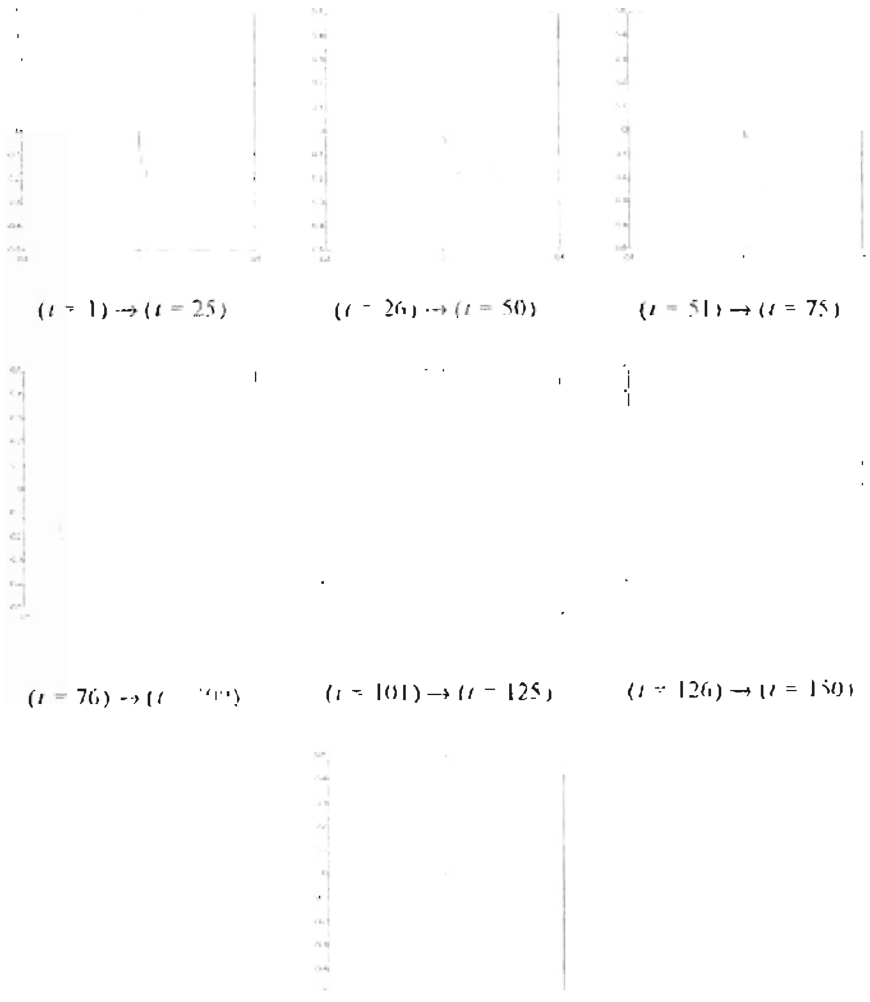


Figure 6.47: *Acrobot. Steps per Trial. Initial zero weights. Exploring actions after two consecutive episodes with same steps, decreased by λ. Minimum steps 147 (37 times).*

We can see in Figure 6.48 and Figure 6.49 the movement of the Acrobot for 172 and 147 step episodes. We can see how the movement of the second link generates an oscillation of the first link until the goal is reached.



$(t = 1) \rightarrow (t = 25)$
$(t = 26) \rightarrow (t = 50)$
$(t = 51) \rightarrow (t = 75)$
$(t = 76) \rightarrow (t$
$(t = 101) \rightarrow (t = 125)$
$(t = 126) \rightarrow (t = 150)$
$(t = 151) \rightarrow (t = 172)$

Figure 6.48: *Acrobot movement for 172 steps (green first link, red second link, blue trajectory of the Acrobot end).*

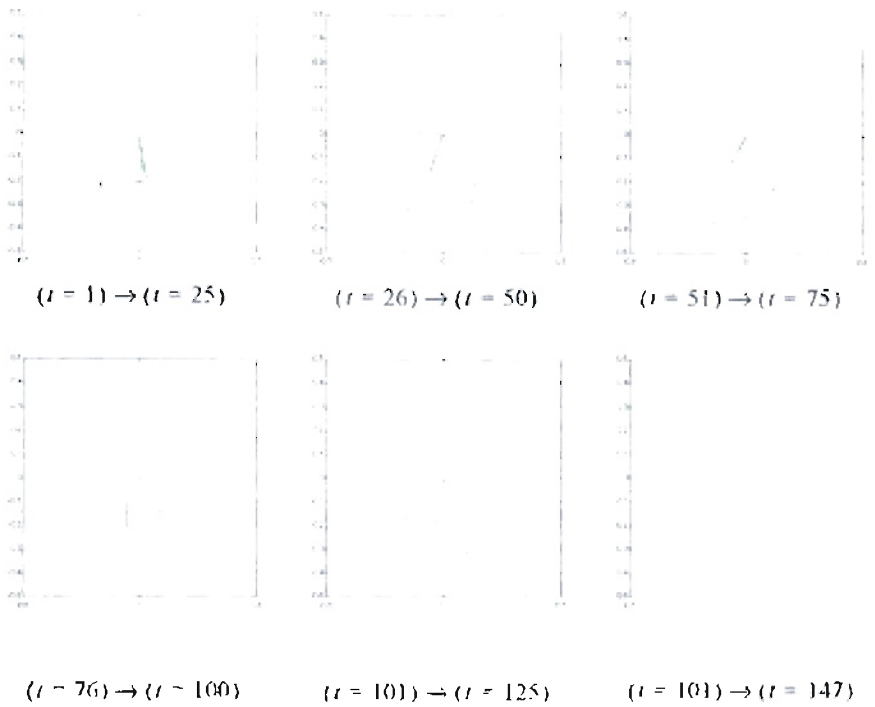


Figure 6.49: *Acrobot movement for 147 steps (green – first link, red – second link, blue – trajectory of the Acrobot end).*

6.13. Conclusions.

This chapter has described the general reinforcement learning framework. Reinforcement learning is an approximate form of dynamic programming, in which an appropriate control policy is chosen to optimize future performance. There are two steps involved in reinforcement learning. The first step is the development of a model for predicting future performance, and the second step is determining the appropriate control action to optimize that performance.

With the reinforcement learning framework there are many different learning strategies for model development which have been proposed. In this chapter we have discussed Monte Carlo and Temporal Difference Learning procedures for model development. For Monte Carlo methods, a number of trials are made and averaging techniques are used to estimate performance functions. In temporal difference learning, estimates are updated at each step of the process. This chapter has described the relationship between Monte Carlo methods and the various forms of temporal difference learning. and has illustrated the convergence characteristics of each method. We also saw the importance of balancing exploitation (maximum reward) and exploration (looking for new solutions). This was important in the Acrobot example. Contrary to other training methods for neural networks, we noted the importance of zero initial weights (or Q-factor functions) for the reinforcement learning problems. Zero initial Q-factor values causes an initial exploration for new solutions. However, after some training we will feel the necessity of increase the exploring actions to maintain the learning process. We can do that by changing the exploration factor ε from zero to some small value when we see a repeated solution. We illustrated this process for the Acrobot example.

We can implement the concepts of Reinforcement Learning using different tools, such as decision trees and neural networks. For neural networks we demonstrated the implementation of CMAC Neural Networks, called tilings by Sutton and Barto (12). Of interest was the tiling implementation with an extra tile and random displacement, allowing a better interaction throughout the state space.

This chapter has shown the feasibility of using reinforcement learning to train controllers for dynamic systems. This technique may be suitable for developing controllers for diesel engines. This will be proposed in the following chapter.

CHAPTER 7

APPLYING REINFORCEMENT LEARNING TO THE DIESEL ENGINE.

We will apply the reinforcement learning techniques discussed in Chapter 6 for the diesel engine control. The five initial experiments will use the engine speed and fueling as the state variables. The first experiment will is intended to obtain a control scheme that learn how to change the engine speed from a fixed initial condition to an specified speed. The second case includes a lower border penalty. The idea is to teach the algorithm how to avoid low speeds. The third experiment includes a higher speed border penalty. The fourth experiment was designed to execute the training from random initial speeds. The final episode is executed with the same initial conditions of the three previous cases. The fifth experiment was executed with random initial speeds and random initial fueling. As we move in our experiments we will explore more conditions of the state space.

After the previous experiments we will execute reinforcement learning experiments for tracking a reference engine speed. We will include the engine acceleration as a third state variable. For the sixth experiment, we will try to control the engine speed using the absolute value error as our fitness with higher penalties for the lower and higher engine speed bounds. For the seventh experiment we will use a time scheme reward equivalent to the used on the five initial experiments. The last experiment is intended to test configurations with two and three CMAC neural networks, and compare the results with one CMAC neural network.

For the experiments we will use two types of tile coding. We called the first as equidistant tile coding. For this coding all the state space for the variable is divided in tiles of the same size. We called the second as log sigmoid tile coding. This coding is described in section 7.6.

7.1. Learning a speed transition.

In this section, we will execute different experiments applying reinforcement learning ideas from Chapter 6 to the two engine models detailed in Chapter 3. For both models the algorithm must learn how to change the engine speed from given initial conditions for speed and fueling to a new speed setpoint.

The reader will notice how the objectives increase in complexity as new experiments are introduced.

7.1.1. Basic engine model.

For the basic model of Figure 3.4 we selected a fueling delay of $80\ ms$ and a fixed load of $150\ lb\text{-}ft$. The reinforcement learning updates were made every $80\ ms$.

We execute the reinforcement learning algorithm until the engine speed reached 650 ± 10 rpm starting from 576 rpm and a very low fueling of $0.0558\ mm^3/stroke$. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 650 ± 10 rpm or the episode lasts 100 *seconds* without arriving at the desired speed, the episode is concluded. Then we start a new episode in the same conditions described previously. The reinforcement learning algorithm has three possible actions: increase the fueling by 10 $mm^3/stroke$, decrease the fueling by the same quantity or maintain the same fueling.

After 100 episodes we obtained the learning curve shown in Figure 7.1. The longest episode was 1251 steps long corresponding to the conditions when the engine runs for 100 *seconds* without reaching the desired speed. The shortest episode was 43 steps and was obtained at the 82th episode where the learning stopped.

Figure 7.2 shows the speed transition for the optimal episode. Due to the low initial fueling and the conditions for increasing and decreasing fueling, the speed reduces to about 190 rpm and after that is increased until 650.65 rpm. The simplest solution for this problem is to increase the fueling by the specified step of 10 $mm^3/stroke$ at each update. That solution produces the green line in Figure 7.2 with 40 steps and a final speed of 648.08 rpm.We notice how the reinforcement algorithm found a suboptimal solution with little knowledge of the physical system. Figure 7.3 shows the fueling for both solutions. We notice how the reinforcement learning solution increases the fueling, and near the required speed it adjusts the fueling to obtain the desired speed.

Figure 7.4 shows the cost-to-go function from this experiment. We notice how the combination of low speed and low fueling has the highest cost. For that combination we need a higher effort to move the engine speed to our desired goal. We also note a high cost for high speed and fueling over 250 $mm^3/stroke$. Due to the problem conditions, if we arrive at the full speed we will decrease the fueling in the given increments to arrive to the solution.

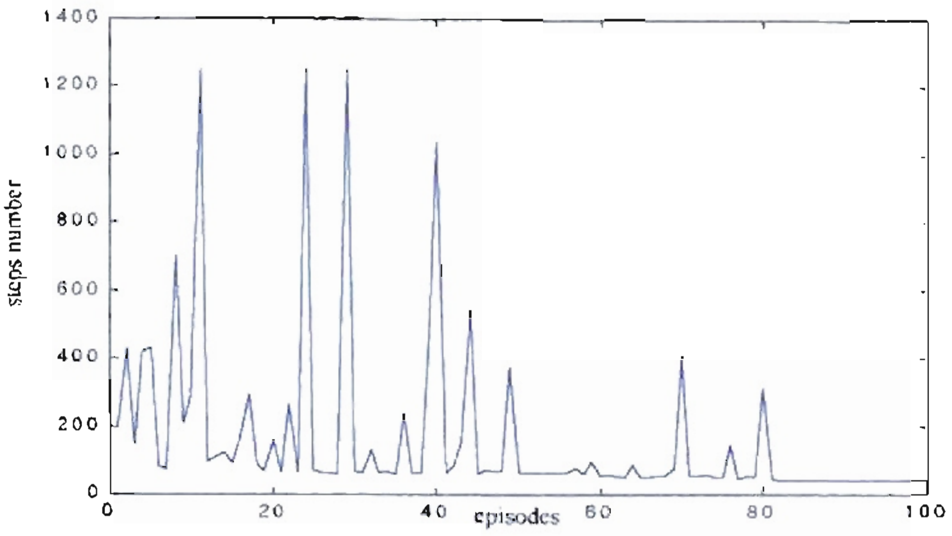


Figure 7.1: *Learning algorithm for speed transition basic engine model.*

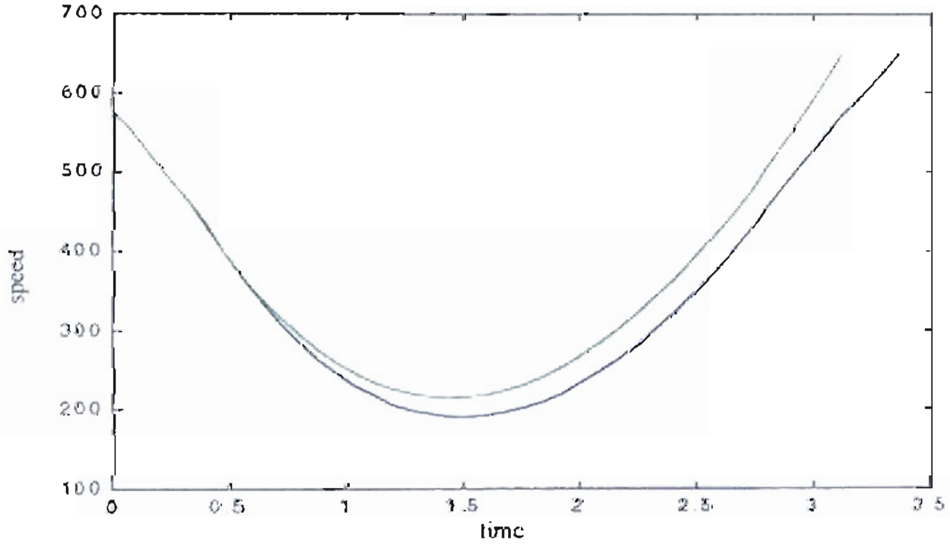


Figure 7.2: *Engine speed response for the basic model. Blue — Suboptimal solution from the reinforcement learning algorithm. Green — Optimal solution.*

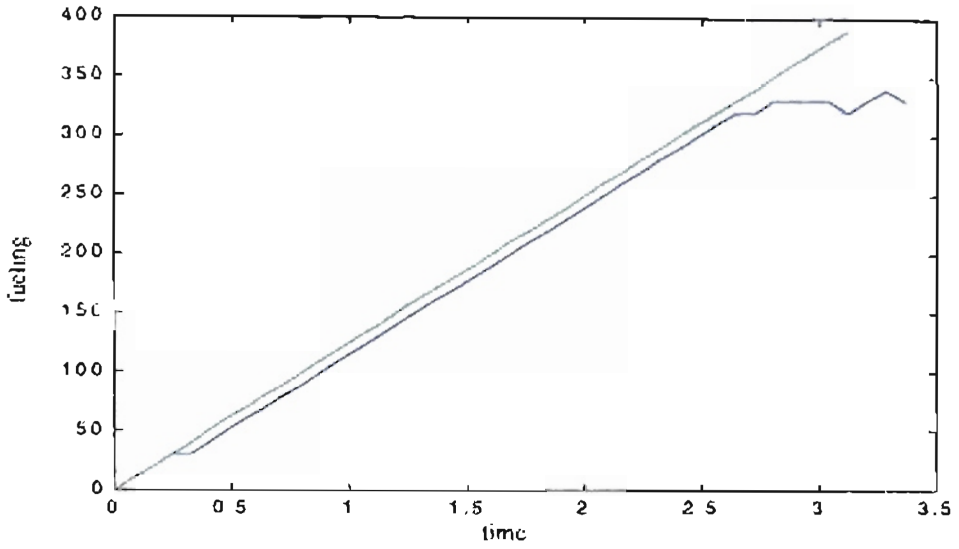


Figure 7.3: *Engine fueling for the basic engine model. Blue — Suboptimal solution from the reinforcement learning algorithm. Green — Optimal solution.*

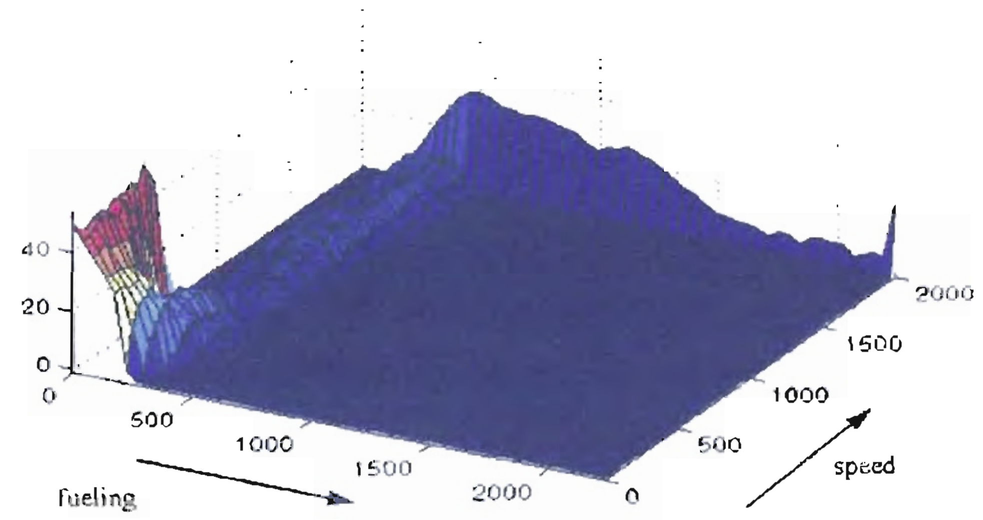


Figure 7.4: *Cost-to-go function for the speed transition experiment basic engine model*

7.1.2. Neural Network model.

For the Neural Network model shown in Figure 3.7 we used the same fueling delay of 80 *ms* and the reinforcement learning update were made every 80 *ms*.We applied a variable load shown in Figure 7.5.

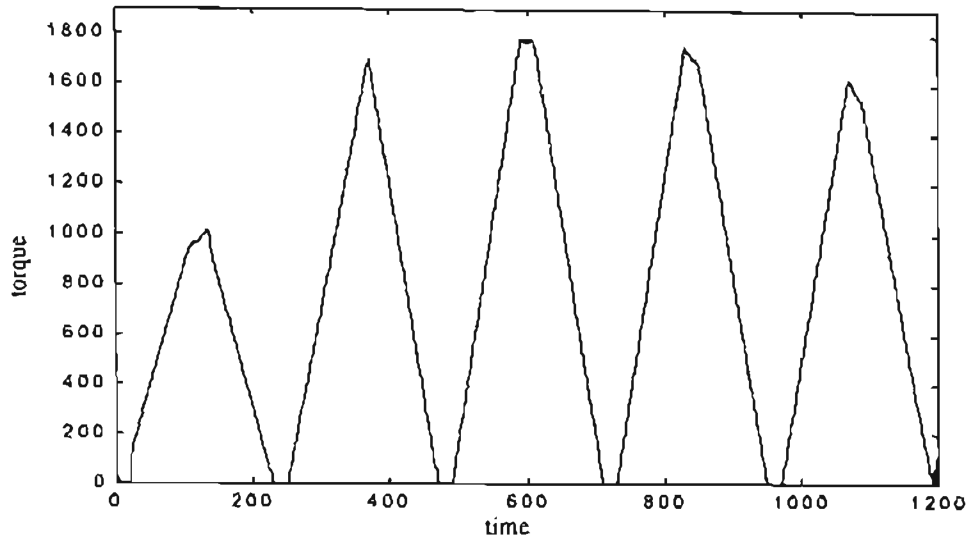


Figure 7.5: *External load applied to the Neural Network model engine.*

We execute the reinforcement learning algorithm until the engine speed reaches 1500 ±10 rpm starting from 800 rpm and a very low fueling of 0.0558 *lb/min*. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 1500 ±10 rpm or the episode lasts 1200 *seconds* without arriving at the desired speed, the episode is concluded. Then we start a new episode with the same conditions described previously. The reinforcement learning algorithm has three possible actions: increase the fueling by 0.01 *lb/min*, decrease the fueling by the same quantity or maintain the same fueling.

After 100 episodes we obtained the learning curve shown in Figure 7.6. The longest episode was 388 steps long. For this model the engine always reached the goal speed. The shortest episode has 34 steps at the 9th episode. However, the algorithm learned after the 70th episode a suboptimal path with 42 steps.

Figure 7.7 shows the speed transition for the suboptimal episode. For the conditions of this experiment the algorithm learned to increase the engine speed until the desired goal. Figure 7.8 shows the fuel mass and the air mass for the suboptimal solution. We notice how the reinforcement learning solution increases the fueling, and the air mass is increased by the effect of the fueling and the engine speed.

Figure 7.9 shows the cost-to-go function for this experiment. Due to the characteristics of this experiment, where we are moving from low to high speed with low fueling, we notice that the highest cost is near low speed and low fueling. For this experiment the engine never arrived at the maximum speed of 2000 rpm.

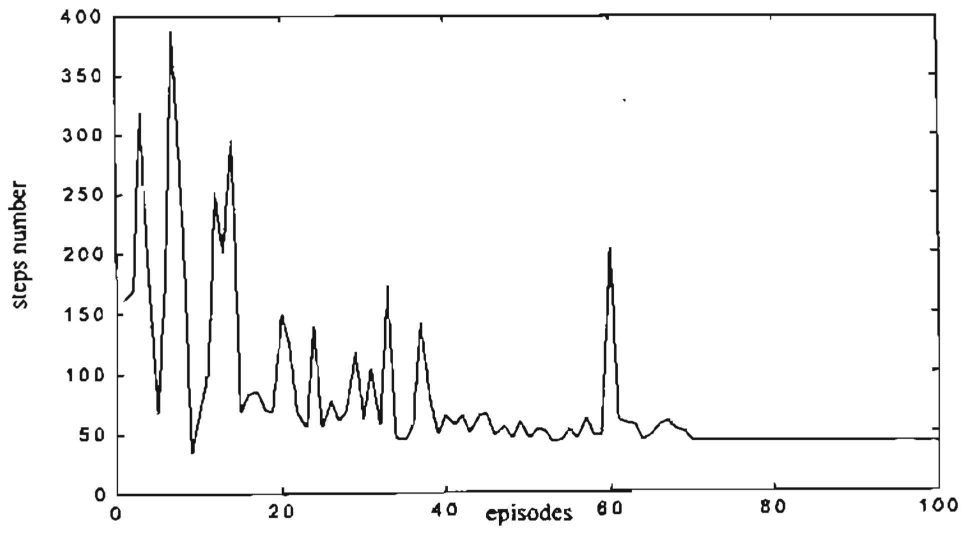


Figure 7.6: *Learning algorithm for speed transition neural network model.*

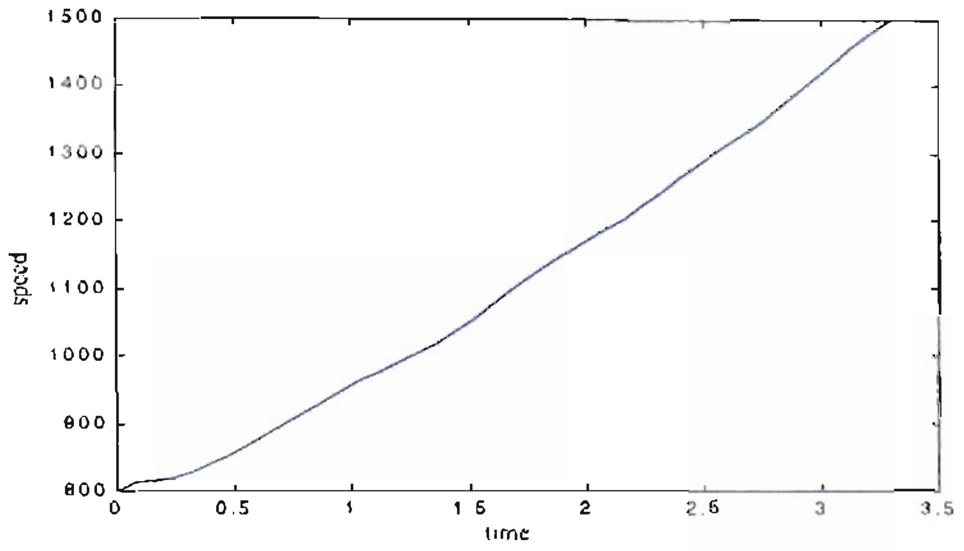


Figure 7.7: *Engine speed for the 42 steps episode.*

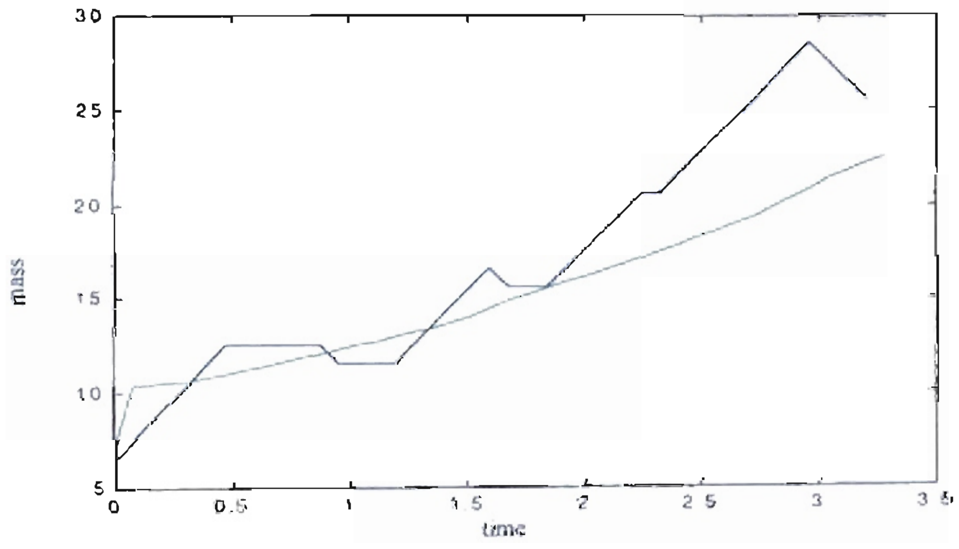


Figure 7.8: *Air mass (green) and fuel mass (x 100) (blue) for the 42 steps episode.*

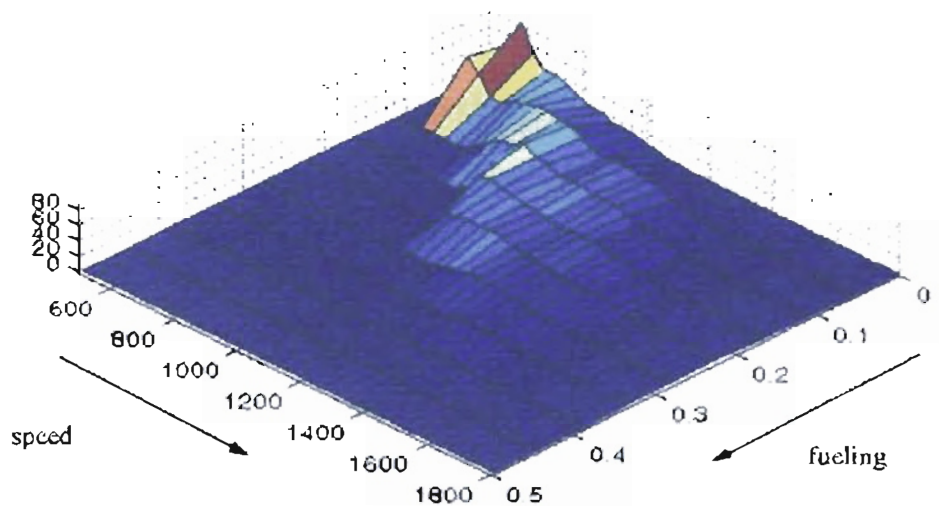


Figure 7.9: *Cost-to-go function for speed transition experiment neural network model.*

7.2. Learning a speed transition with lower border penalty.

For this experiment we applied some of the previous conditions, but we added more restrictions. First, the episode will conclude if the engine speed reaches a lower limit. In that case the penalty is more severe with $r = -100$. Second, the speed band error was reduced to ± 1 rpm.

7.2.1. Basic engine model.

We used the same conditions of delay equal to 80 *ms* and a fixed load of 150 *lb-ft*. The reinforcement learning updates were also made every 80 *ms*.

We executed the reinforcement learning algorithm until the engine speed reached 650 ± 1 rpm starting from 576 rpm and a very low fueling of 0.0558 $mm^3/stroke$. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. The episode is concluded by three conditions: the engine speed arrives at 650 ± 1 rpm, the episode lasts 100 *seconds* without arriving at the desired speed or the engine speed is under 100 rpm. In the last case the penalty is $r = -100$. Each new episode is started at the same conditions described previously. The reinforcement learning algorithm has the same three possible actions: increase the fueling by 10 $mm^3/stroke$, decrease the fueling by the same quantity or maintain the same fueling.

After 200 episodes we obtained the learning curve shown in Figure 7.10. The longest episode was 1251 steps long corresponding to the conditions where the engine runs for 100 *seconds* without reaching the desired speed. The shortest episode with final speed

650 ± 1 rpm was 46 steps, that was obtained at the 123th episode where the learning stopped.

Figure 7.11 shows the speed transition for the optimal episode. Due to the low initial fueling and the conditions for increasing and decreasing fueling, the speed reduces to about 192 rpm and after that is increased until 649.65 rpm. The simple solution applied in the previous section was not possible for this problem. If we continuously increase the fueling, the engine speed will pass over the range of ± 1 rpm. Figure 7.12 shows the fueling for this solution. We notice how the reinforcement learning solution increases the fueling as the solution shown in Figure 7.3, with extra steps to obtain the desired speed.

Figure 7.13 shows the cost-to-go function from this experiment. We notice how the combination of low speed and low fueling has the highest cost. This cost is higher in comparison with the cost shown in Figure 7.4. Also a break is shown near 100 rpm. For that combination we still need a higher effort to move the engine speed to our desired goal, but we receive a higher penalty for crossing the 100 rpm border. As the cost shown in Figure 7.4, we also note a higher cost for maximum speed and fueling over 250 $mm^3/stroke$. Due to the problem conditions, if we arrive at full speed we will decrease the fueling in the given increments to arrive at the solution.

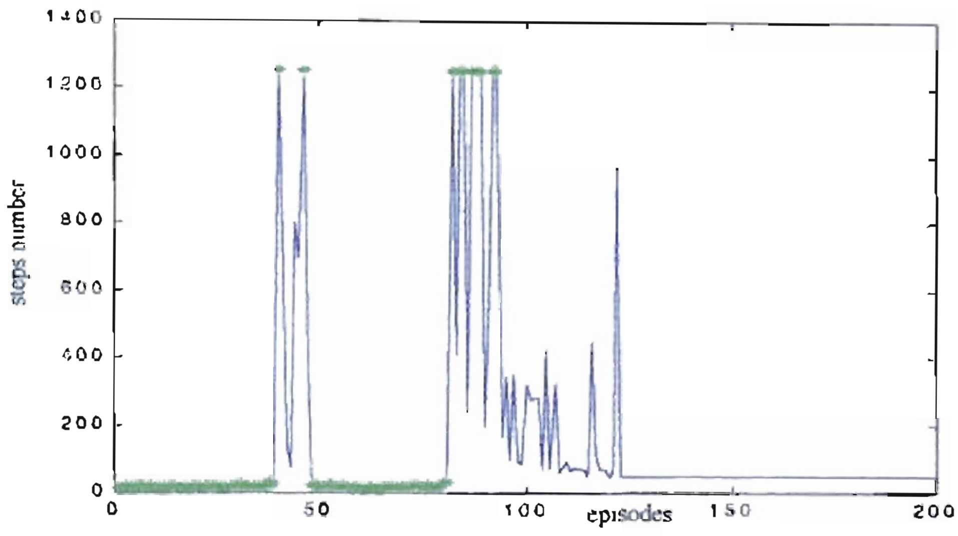


Figure 7.10: *Learning algorithm for speed transition with lower border penalty experiment using basic engine model.( ) - episodes where* 650 ±1 rpm *was not reached*

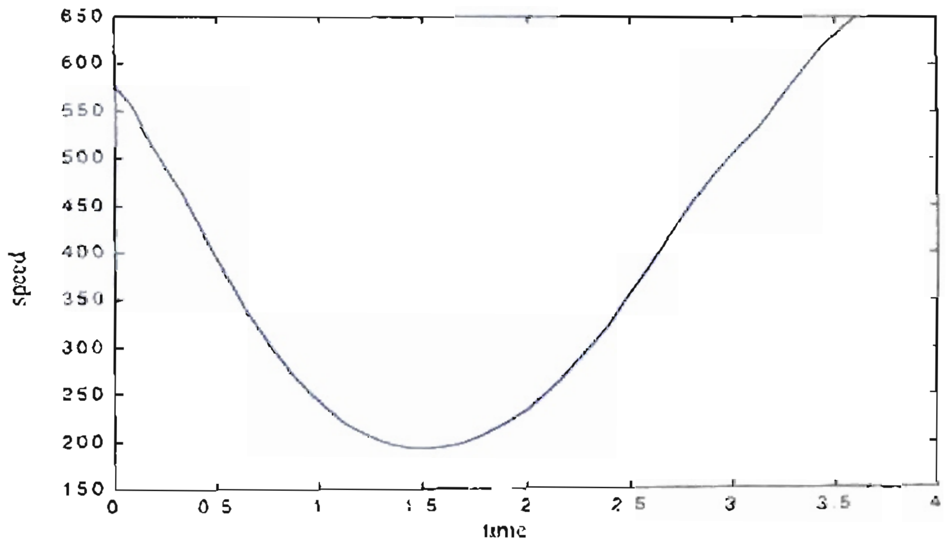


Figure 7.11: *Engine speed response for the basic model.*

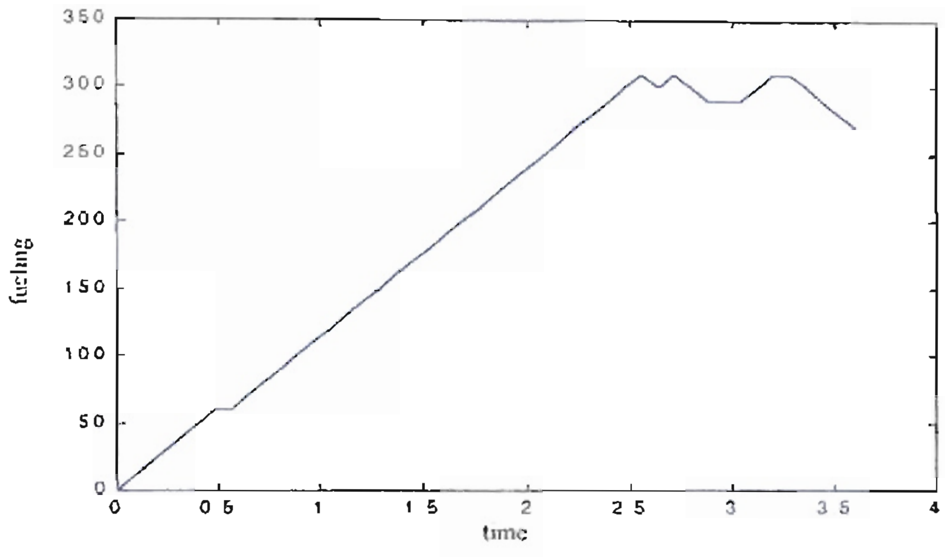


Figure 7.12: *Engine fueling for the basic engine model. Blue — Suboptimal solution from the reinforcement learning algorithm. Green — Optimal solution.*

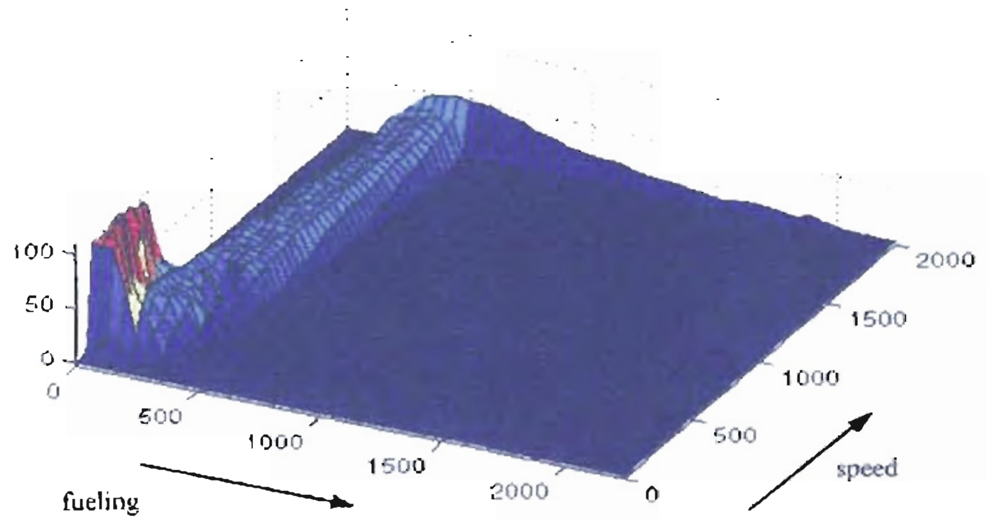


Figure 7.13: *Cost-to-go function for speed transition with lower border penalty experiment using basic engine model.*

7.2.2. Neural Network model.

We execute the reinforcement learning algorithm until the engine speed reaches 1500 ± 1 rpm starting from 800 rpm and a very low fueling of 0.0558 lb/min. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 1500 ± 1 rpm or the episode lasts 1200 *seconds* without arriving at the desired speed, the episode is concluded. Then we start a new episode with the same initial conditions described previously. The episode also concludes if the engine speed falls to 570 rpm with a penalty $r = -100$. The reinforcement learning algorithm has three possible actions: increase the fueling by 0.01 lb/min, decrease the fueling by the same quantity or maintain the same fueling.

After 100 episodes we obtained the learning curve shown in Figure 7.14. The longest episode was 4267 steps long and the desired speed was not reached. The shortest episode has 43 steps at the 77th episode. However, the algorithm learned after that episode a suboptimal path with 48 steps.

Figure 7.15 shows the speed transition for the suboptimal episode. For the conditions of this experiment the algorithm learned to increase the engine speed until the desired goal. Figure 7.16 shows the fuel mass and the air mass for the suboptimal solution. We notice how the reinforcement learning solution increases and reduces the fueling to obtain a smooth speed transition.

Figure 7.17 shows the cost-to-go function from this experiment. Due to the characteristics of this experiment, where we are moving from low to high speed with low fueling, we notice that the highest cost is near low speed and low fueling. We also note a higher cost for the maximum speed of 2000 rpm.

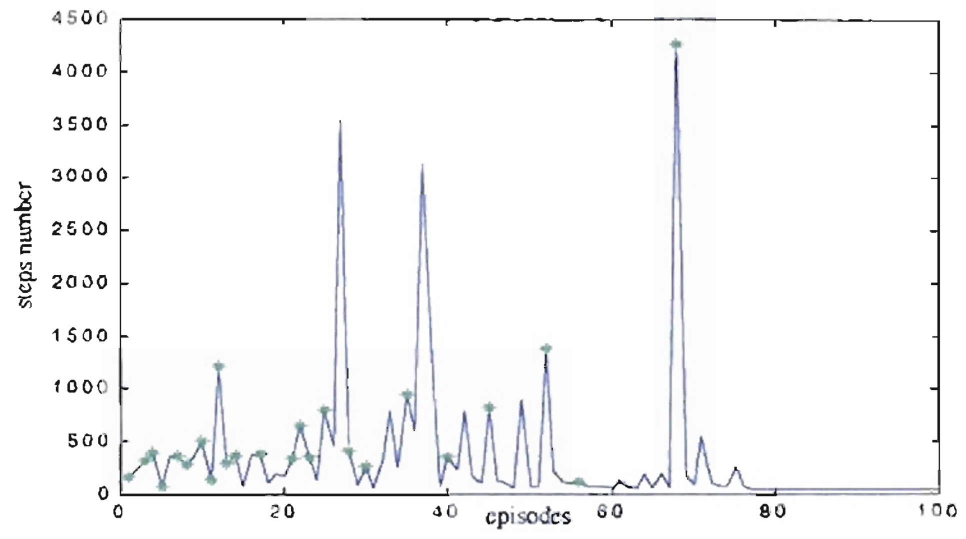


Figure 7.14: *Learning curve for speed transition with lower border penalty experiment using neural network model. ( ) episodes where* 1500 ±1 rpm *was not reached.*

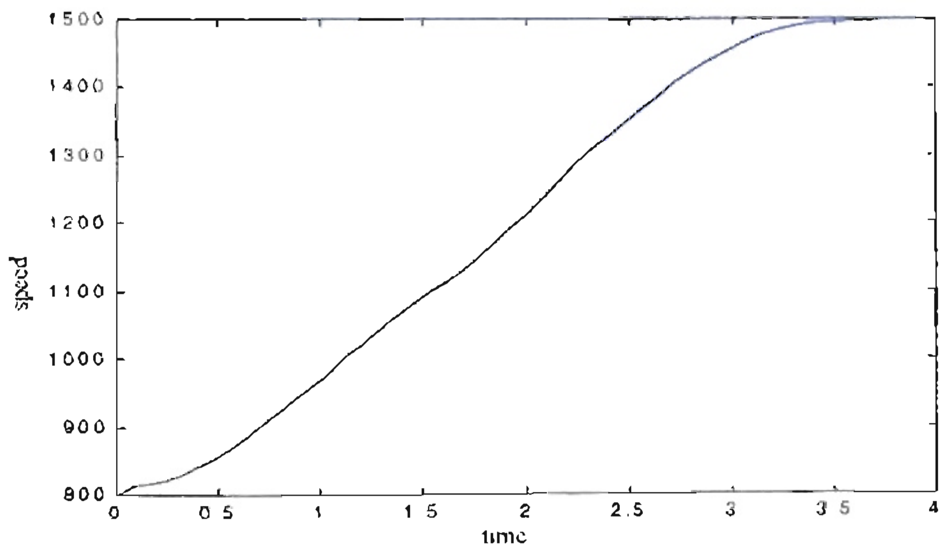


Figure 7.15: *Engine speed for the 48 steps episode.*

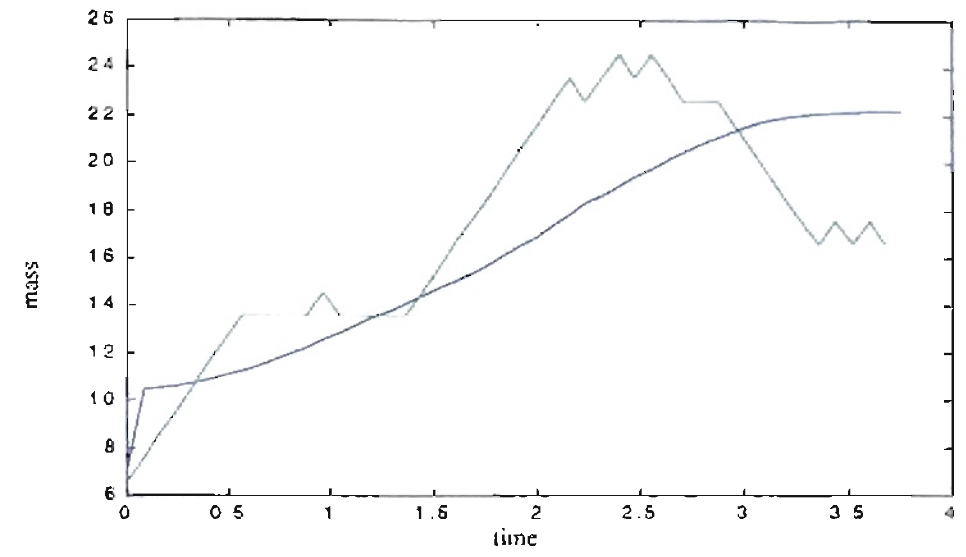


Figure 7.16: *Air mass (green) and fuel mass (x 100) (blue) for the 48 steps episode.*

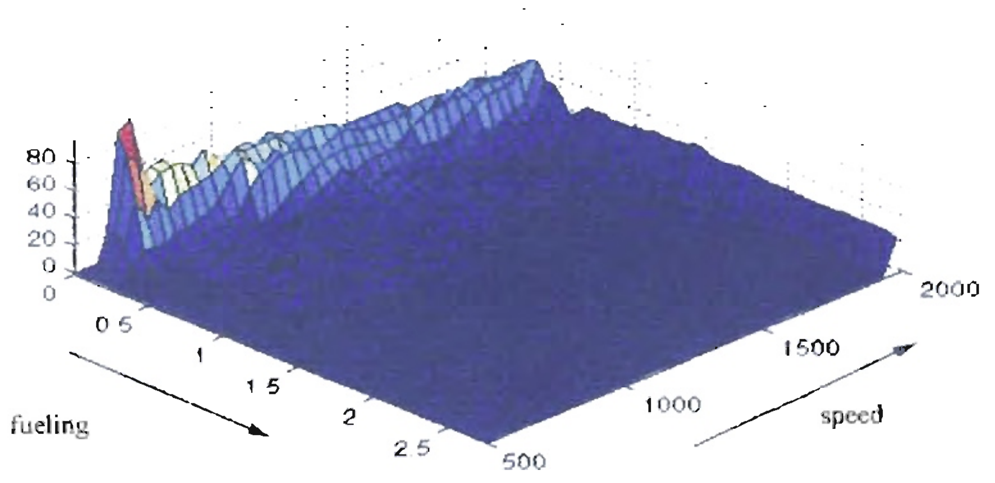


Figure 7.17: *Cost-to-go function for speed transition with lower border penalty experiment with neural network model.*

7.3. Learning a speed transition with lower and higher border penalty.

For this experiment we included a restriction for higher speed. Therefore, the episode will conclude if the engine speed reaches a upper limit. We used the same severe penalty $r = -100$ as in the case of lower speed violations. We maintained the same speed band error of ± 1 rpm.

7.3.1. Basic engine model.

We used the same conditions of delay equal to 80 ms and a fixed load of 150 $lb\text{-}ft$. The reinforcement learning updates were also made every 80 ms.

We executed the reinforcement learning algorithm until the engine speed reached 650 ± 1 rpm starting from 576 rpm and a very low fueling of 0.0558 $mm^3/stroke$. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. The episode is concluded by four conditions: the engine speed arrives at 650 ± 1 rpm, the episode lasts 100 *seconds* without arriving at the desired speed, the engine speed is under 100 rpm or over 2000 rpm. In the last two cases the penalty is $r = -100$. Each new episode is started with the same initial conditions described previously. The reinforcement learning algorithm has the same three possible actions: increase the fueling by 10 $mm^3/stroke$, decrease the fueling by the same quantity or maintain the same fueling.

After 200 episodes we obtained the learning curve shown in Figure 7.18. The longest episode was 1038. That episode corresponds to one where the engine avoids the limit speeds and arrives at the desired speed. The shortest episode, with final speed

650 ± 1 rpm, has 44 steps. That episode was obtained at the 138th episode where the learning stopped.

Figure 7.19 shows the speed transition for the optimal episode. Due to the low initial fueling and the conditions for increasing and decreasing fueling, the speed reduces to about 167 rpm and after that is increased until 650.09 rpm. Figure 7.20 shows the fueling for this solution. We notice how the reinforcement learning solution increases the fueling with some variations in the middle of the trajectory.

Figure 7.21 shows the cost-to-go function from this experiment. We still notice how the combination of low speed and low fueling has the highest cost. This cost is higher in comparison with the cost shown in Figure 7.4 but similar to the cost plotted showed in Figure 7.13. The break near 100 rpm is also in Figure 7.21. However the cost near 2000 rpm differs with the two previous experiments. Due to the end of the episode and the penalty at the top speed, we only see a high cost near 2000 rpm from 250 to 500 $mm^3/stroke$. For other fueling values the cost is zero because those regions were not explored for the conditions of this experiment where we only move from one speed to another.

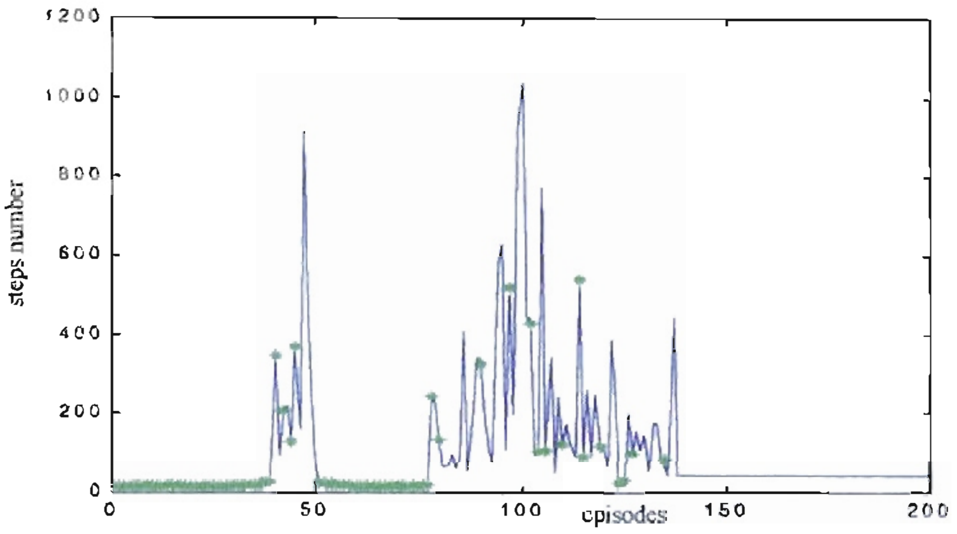


Figure 7.18: *Learning algorithm for speed transition with lower and upper border penalty experiment using basic engine model.( ) episodes where* 650 ±1 rpm *was not reached.*

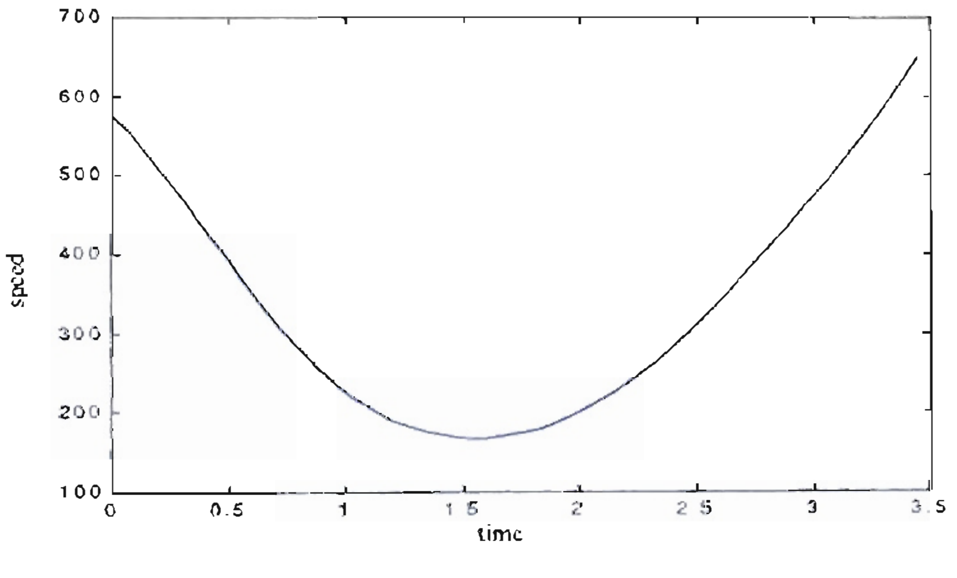


Figure 7.19: *Engine speed response for the basic model.*

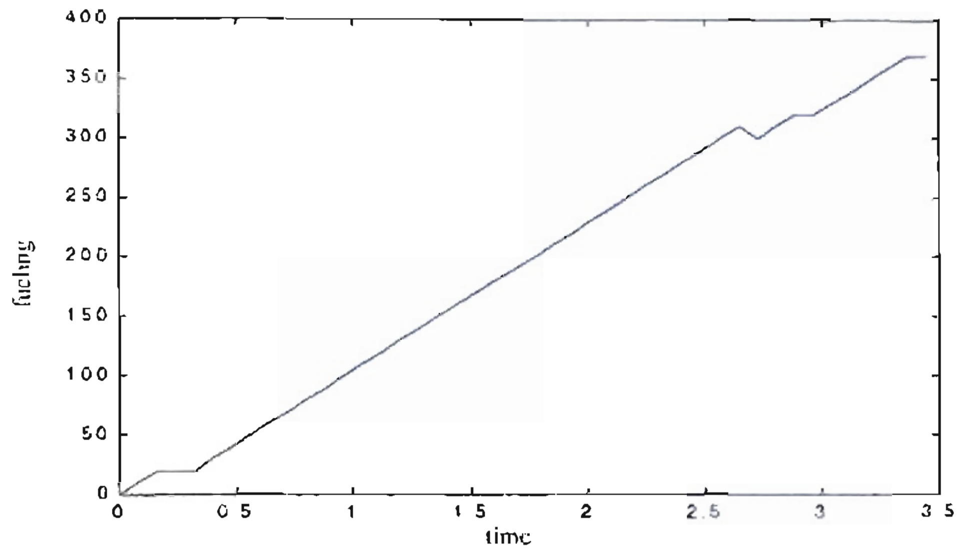


Figure 7.20: *Engine fueling for the basic engine model. Blue — Suboptimal solution from the reinforcement learning algorithm. Green — Optimal solution.*

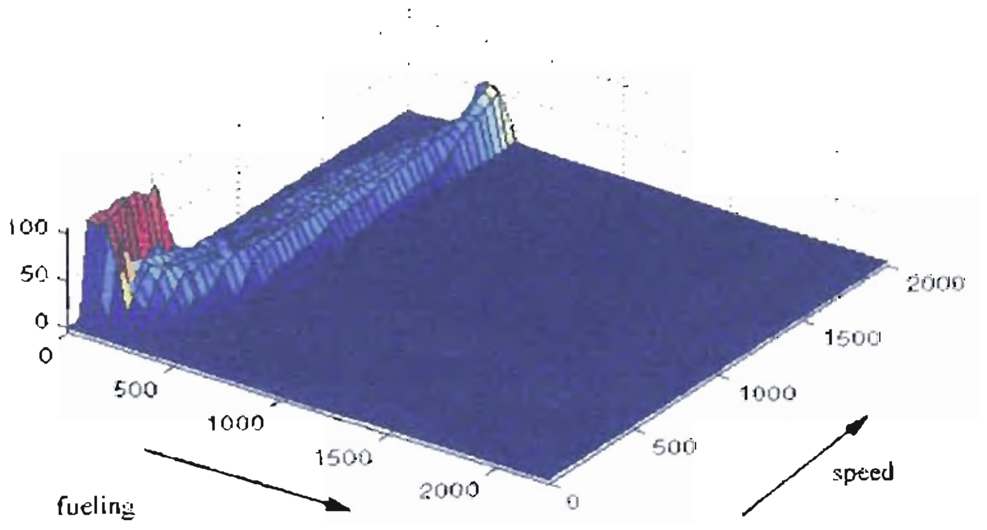


Figure 7.21: *Cost-to-go function for speed transition with lower and upper border penalty experiment using basic engine model.*

7.3.2. Neural Network model.

We execute the reinforcement learning algorithm until the engine speed reached 1500 ± 1 rpm starting from 800 rpm and a very low fueling of 0.0558 lb/min. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 1500 ± 1 rpm or the episode lasts 1200 *seconds* without arriving to the desired speed, the episode is concluded. Then we start a new episode with the same initial conditions described previously. The episode also concludes if the engine speed reduces to 570 rpm or increases to 2000 rpm with a penalty $r = -100$ for each case. The reinforcement learning algorithm has three possible actions: increase the fueling by 0.01 lb/min, decrease the fueling by the same quantity or maintain the same fueling.

After 100 episodes we obtained the learning curve shown in Figure 7.22. The longest episode was 946 steps long and the desired speed was not reached. The shortest episode has 46 steps at the 38th episode. However, after the 85th episode the algorithm learned a suboptimal path with 51 steps.

Figure 7.23 shows the speed transition for the suboptimal episode. For the conditions of this experiment the algorithm learned to increase the engine speed with a very small overshoot until the desired goal. Figure 7.24 shows the fuel mass and the air mass for the suboptimal solution. As in the previous experiment, we notice how the reinforcement learning solution increases and reduces the fueling to obtain the smooth speed transition.

Figure 7.25 shows the cost-to-go function for this experiment. Due to the characteristics of this experiment, where we are moving from low to high speed with low fueling, we notice that the highest cost is near low speed and low fueling. We also note a higher cost for the maximum speed of 2000 *rpm* only for the fueling explored by the algorithm.

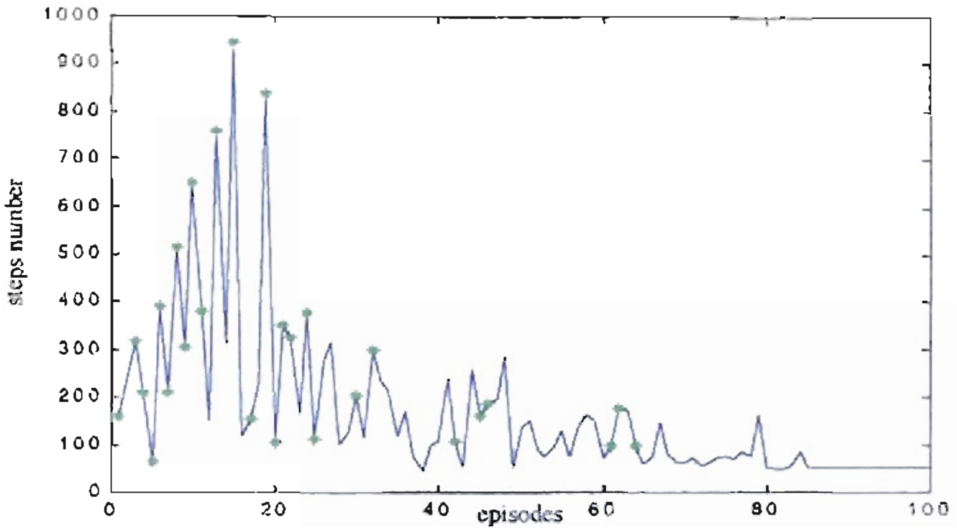


Figure 7.22: *Learning curve for speed transition with lower and upper border penalty experiment using neural network model. ( ) = episodes where* 1500 ±1 rpm *was not reached.*

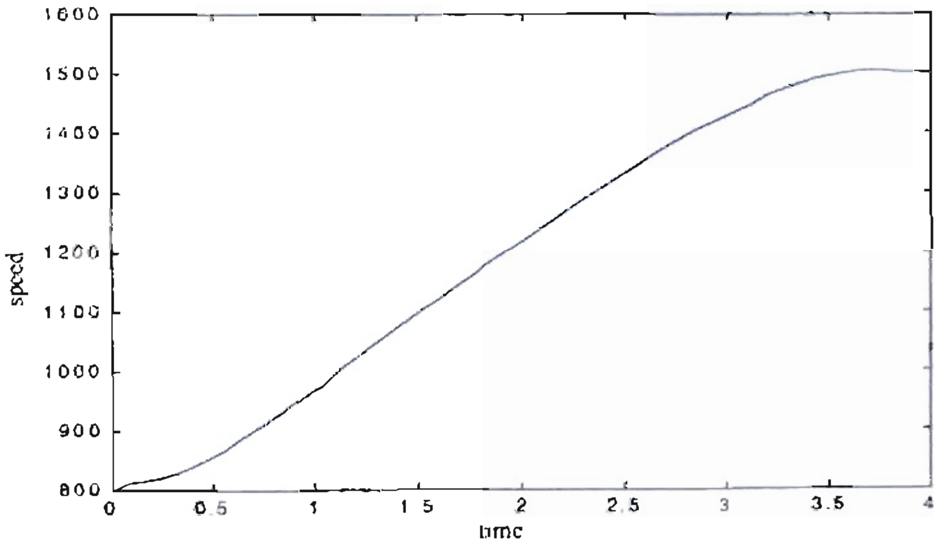


Figure 7.23: *Engine speed for the 51 steps episode.*

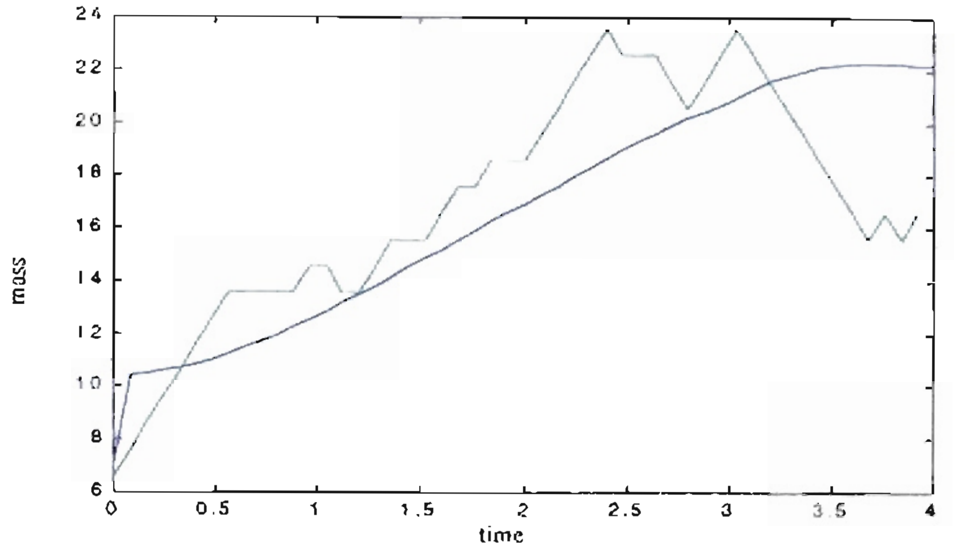


Figure 7.24: *Air mass (green) and fuel mass (x 100) (blue) for the 51 steps episode.*

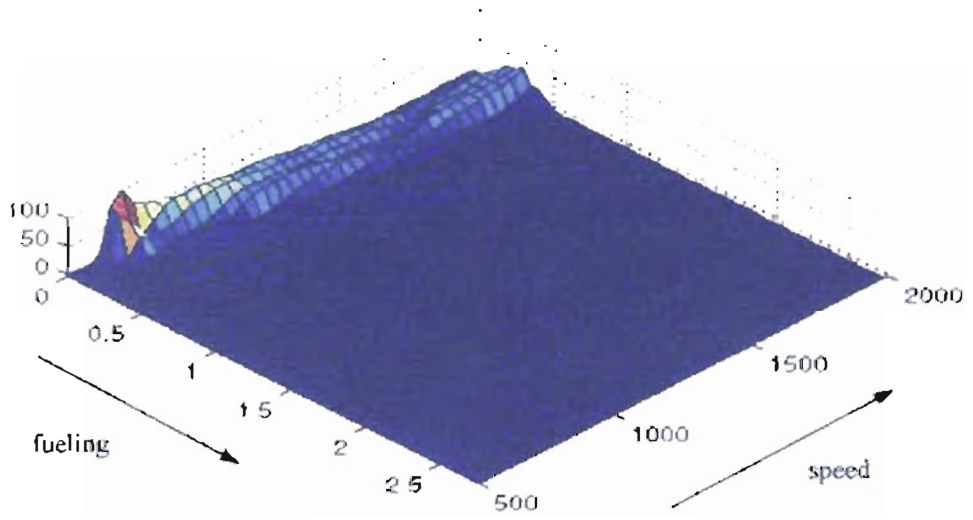


Figure 7.25: *Cost-to-go function for speed transition with lower and upper border penalty experiment using the neural network model.*

7.4. Learning a speed transition with random initial speed.

For this experiment we maintained the restrictions for lower and higher speed. The training was made with random initial speed between 300 rpm and 1700 rpm. Each episode will conclude if the engine speed reaches the required speed ± 1 rpm, or the lower or upper speed limit. We used the same severe penalty $r = -100$ as in the case of lower or upper speed violations.

7.4.1. Basic engine model.

We used the same conditions of delay equal to 80 *ms* and a fixed load of 150 *lb-ft*. The reinforcement learning update were also made every 80 *ms*.

We executed the reinforcement learning algorithm until the engine speed reached 650 ± 1 rpm, starting from random speed between 300 rpm and 1700 rpm with a very low fueling of 0.0558 $mm^3/stroke$. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. The episode is concluded by four conditions: the engine speed arrives at 650 ± 1 rpm, the episode lasts 100 *seconds* without arriving at the desired speed, the engine speed is under 100 rpm or over 2000 rpm. In the last two cases the penalty is $r = -100$. Each new episode is started with the same initial conditions or random speed and low fueling described previously. The reinforcement learning algorithm has the same three possible actions: increase the fueling by 10 $mm^3/stroke$, decrease the fueling by the same quantity or maintain the same fueling.

After 501 episodes we obtained the learning curve shown in Figure 7.26. The longest episode was 1251 steps long corresponding to the conditions where the engine runs for 100 *seconds* without reaching the desired speed. The final episode, starting with speed 576 rpm and fueling of 0.0558 $mm^3/stroke$ and reaching a final speed of 650 ± 1 rpm, has 46 steps. Other episodes for different initial speeds also reached the specified speed.

Figure 7.27 shows the speed transition for the last episode. As in the previous experiments, due to the low initial fueling and the conditions for increasing and decreasing fueling, the speed reduces to about 233 rpm and after that is increased until 650.69 rpm. Figure 7.28 shows the fueling for this solution. We notice how the reinforcement learning solution increases the fueling until one point where the fueling is decreased to obtain the required speed.

Figure 7.29 shows the cost-to-go function from this experiment. This cost is similar to the cost function shown in Figure 7.21 where the combination of low speed and low fueling has the highest cost and a higher cost near 2000 rpm from 250 to 500 $mm^3/stroke$.

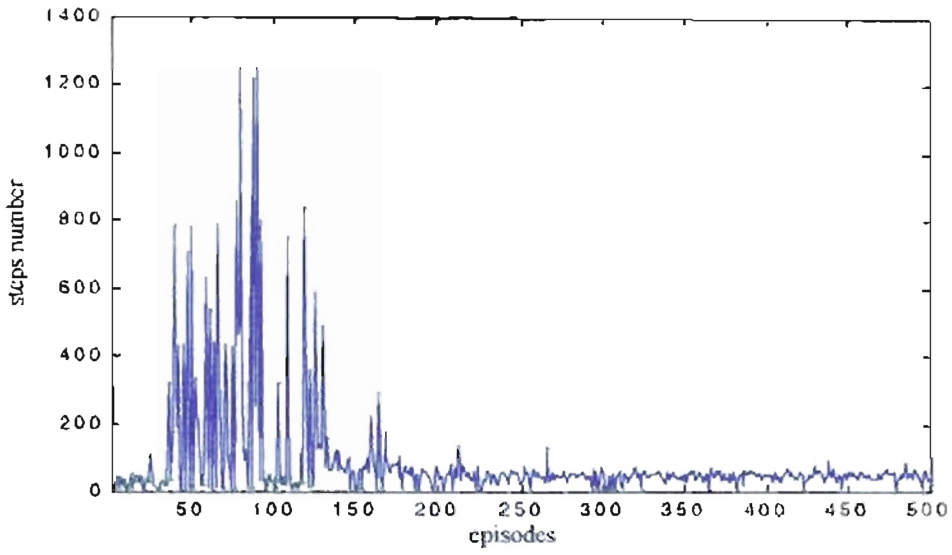


Figure 7.26: *Learning algorithm for speed transition with random initial speed and low fueling using the basic engine model.(·) episodes where* 650 ±1 rpm *was not reached*

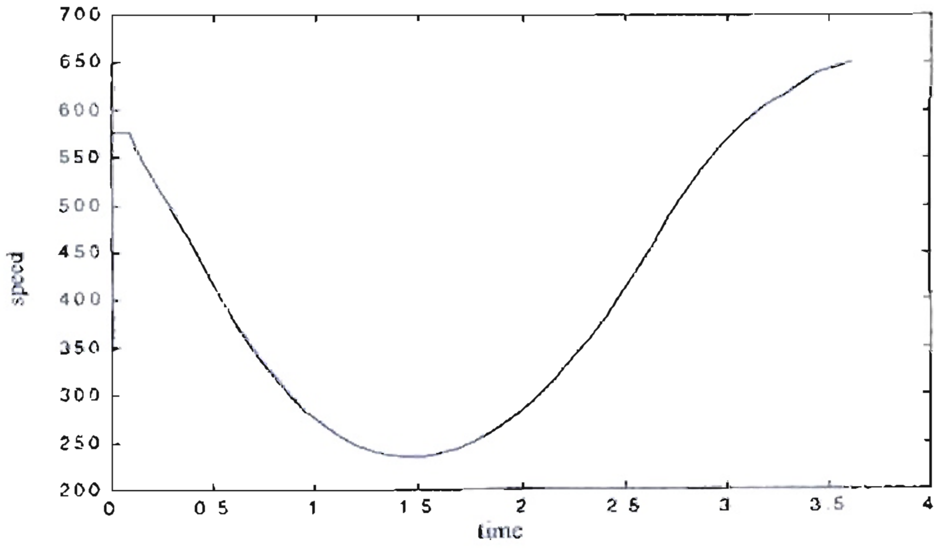


Figure 7.27: *Engine speed response for the basic model.*

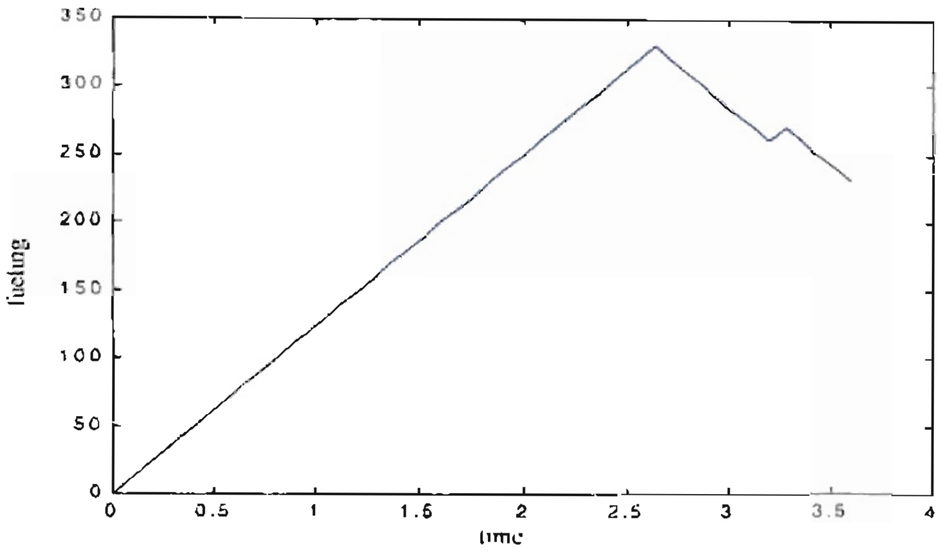


Figure 7.28: *Engine fueling for the basic engine model. Blue — Suboptimal solution from the reinforcement learning algorithm. Green — Optimal solution.*

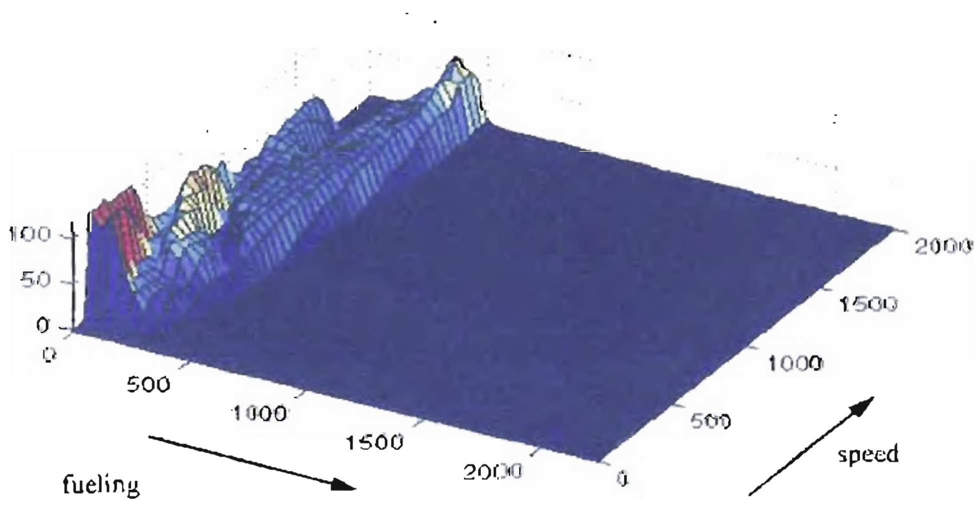


Figure 7.29: *Cost-to-go function for speed transition with random initial speed and low fueling using the basic engine model.*

7.4.2. Neural Network model.

We execute the reinforcement learning algorithm until the engine speed reaches 1500 ± 1 rpm starting from a random speed between 800 and 1700 rpm and a very low fueling of 0.0558 lb/min. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 1500 ± 1 rpm or the episode lasted 1200 *seconds* without arriving at the desired speed, the episode is concluded. Then we start a new episode in the same conditions described previously. The episode also concludes if the engine speed reduces to 570 rpm or increases to 2000 rpm with a penalty $r = -100$ for each case. The reinforcement learning algorithm has three possible actions: increase the fueling by 0.01 lb/min, decrease the fueling by the same quantity or maintain the same fueling.

After 100 episodes we obtained the learning curve shown in Figure 7.30. The longest episode was 1188 steps long and the desired speed was not reached. The shortest episode has 9 steps at the 57th episode. However, this episode started with 1689.91 rpm, and that, combined with the lower fueling, permitted a short and successful episode. We note that after 40 episodes the learning curve is improved and the frequency of episodes that reach the goal is increased, as shown in Figure 7.30.

After training we executed an episode starting at 800 rpm and the same fueling used for the training. Figure 7.31 shows the speed transition for that episode in 67 steps. For the conditions of this experiment the algorithm increased the engine speed slowly in comparison with the previous experiments. Figure 7.32 shows the fuel mass and the air mass for the last episode. Here the algorithm increased the fueling with intermediate stages where the fueling remain constant.

Figure 7.33 shows the cost-to-go function from this experiment. We observe similar characteristics to the cost function plotted in Figure 7.25. We notice that the highest cost is near low speed and low fueling. We also note a higher cost for the maximum speed of 2000 rpm only for the fueling explored by the algorithm.

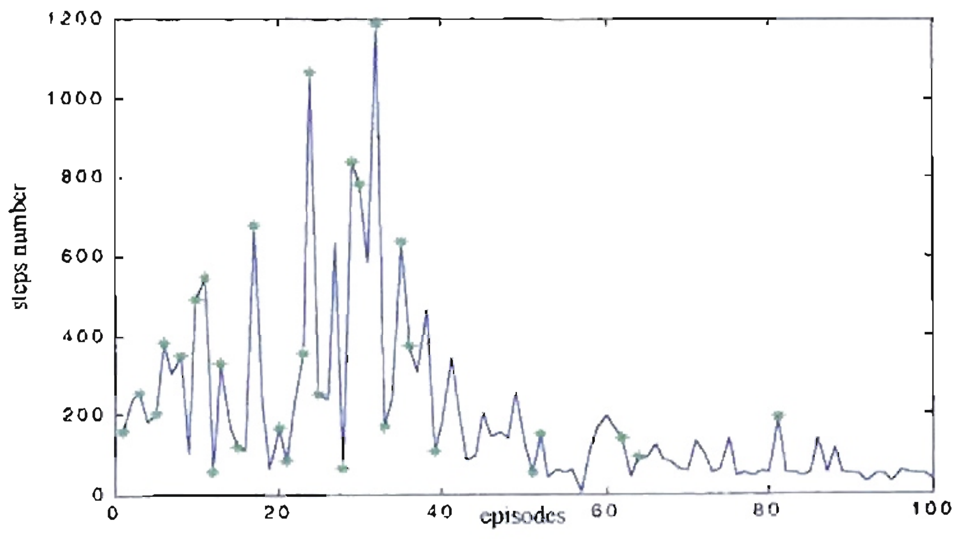


Figure 7.30: *Learning algorithm for speed transition with random initial speed and low fueling using the neural network model. (*) : episodes where* 1500 ±1 rpm *was not reached.*

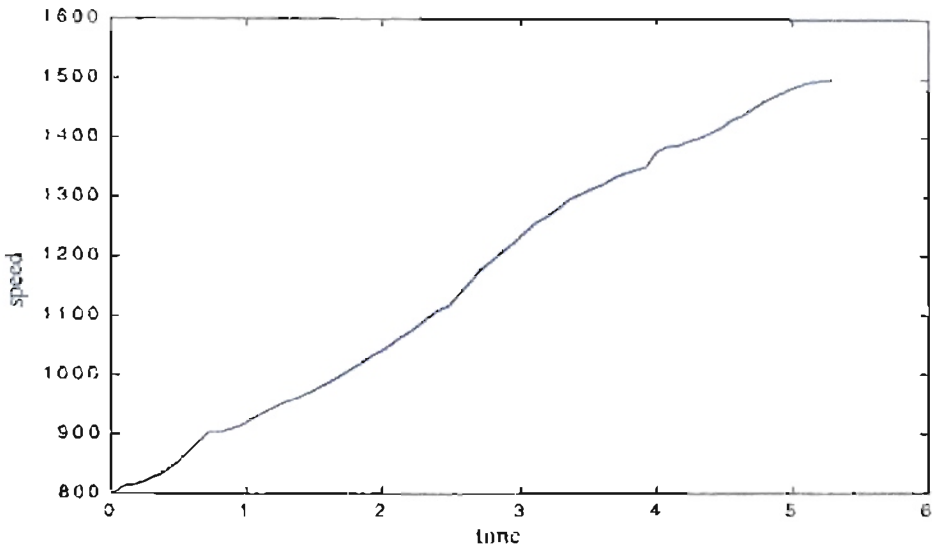


Figure 7.31: *Engine speed for the 67 steps episode starting with 800 rpm.*

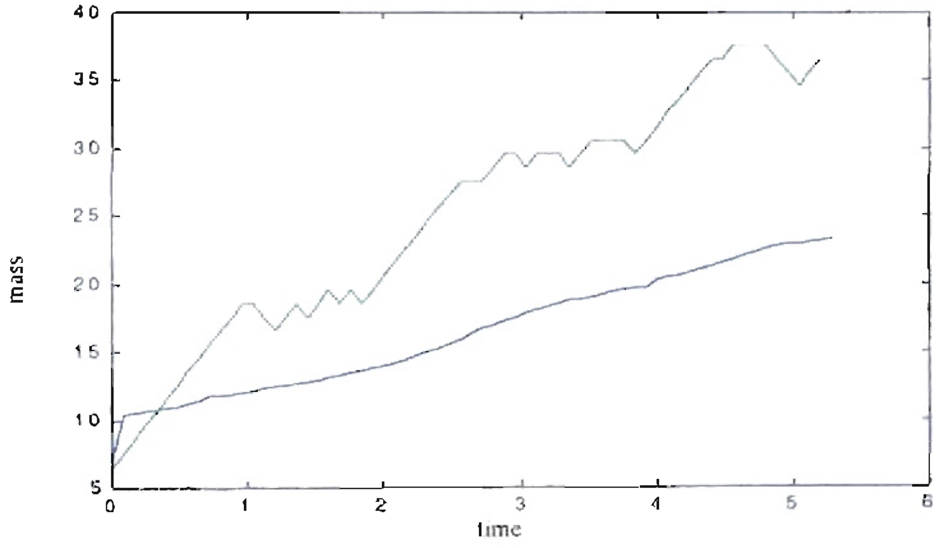


Figure 7.32: *Air mass (green) and fuel mass (x 100) (blue) for the 67 steps episode starting with 800 rpm.*

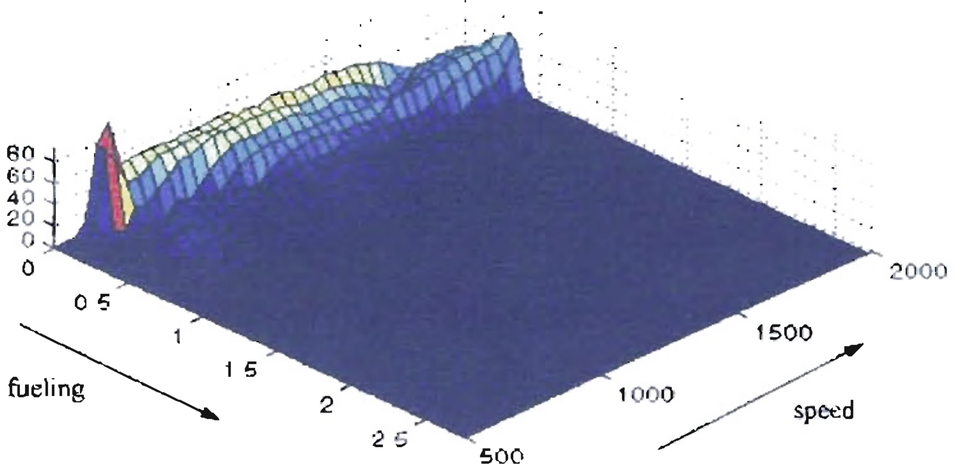


Figure 7.33: *Cost-to-go function for speed transition with random initial speed and low fueling using the neural network model.*

7.5. Learning a speed transition with random initial speed and fueling.

For this experiment we maintained the restrictions for lower and higher speed. The training was made with random initial speed between 300 rpm and 1700 rpm and with random initial fueling between 0.0558 $mm^3/stroke$ and 560 $mm^3/stroke$. Each episode will conclude if the engine speed reaches the required speed ± 1 rpm, or the lower or upper speed limit. We used the same severe penalty $r = -100$ as in the case of lower or upper speed violations.

7.5.1. Basic engine model.

We used the same conditions of delay equal to 80 ms and a fixed load of 150 $lb\text{-}ft$. The reinforcement learning update were also made every 80 ms.

We executed the reinforcement learning algorithm until the engine speed reached 650 ± 1 rpm, starting from a random speed between 300 rpm and 1700 rpm with random initial fueling between 0.0558 $mm^3/stroke$ and 560 $mm^3/stroke$. The algorithm receives a penalty of $r = -1$ for each step where the specified speed is not reached. The episode is concluded by four conditions: the engine speed arrives at 650 ± 1 rpm, the episode lasted 100 *seconds* without arriving at the desired speed, the engine speed is under 100 rpm or over 2000 rpm. In the last two cases the penalty is $r = -100$. Each new episode is started with the same initial conditions or random speed and low fueling described previously. The reinforcement learning algorithm has the same three possible actions: increase the fueling by 10 $mm^3/stroke$, decrease the fueling by the same quantity or maintain the same fueling.

After 501 episodes we obtained the learning curve shown in Figure 7.34. The longest episode was 1251 steps long corresponding to the conditions where the engine runs for 100 *seconds* without reaching the desired speed. We notice that the initial episodes took more steps to reach the final objective and failed more frequently than the later episodes that required fewer steps to reach the objective. If we start the last episode with a speed of 576 rpm and fueling of 0.0558 $mm^3/stroke$ the specified speed was not reached. That result was due to the characteristics of the training. The algorithm learned how to reach the objective speed from different initial random fueling levels. The low initial fueling may not have been tested in the training. If we change the initial fueling to 20 $mm^3/stroke$, we will obtain the final speed of 650 ± 1 rpm in 50 steps.

Figure 7.35 shows the speed transition for the last episode under two different initial fueling levels. For an initial fueling of 0.0558 $mm^3/stroke$ we can see how the engine speed falls to less than 100 rpm. For an initial fueling of 20 $mm^3/stroke$ we can see that the speed reduces to about 278 rpm and after that is increased with an small overshoot until reaching 649.13 rpm. Figure 7.36 shows the fueling for both initial fueling levels. We notice how the reinforcement learning solution for 20 $mm^3/stroke$ increases the fueling until one point where the fueling is decreased to obtain the required speed. For an fueling of 0.0558 $mm^3/stroke$ the reinforcement learning algorithm maintained the fueling near zero because the algorithm is still learning about the process.

Figure 7.37 shows the cost-to-go function for this experiment. This cost is similar to the cost function shown in Figure 7.21 and Figure 7.29. The major difference is that the region close to high speed and low fueling were explored and therefore the cost in that zone is different from zero.

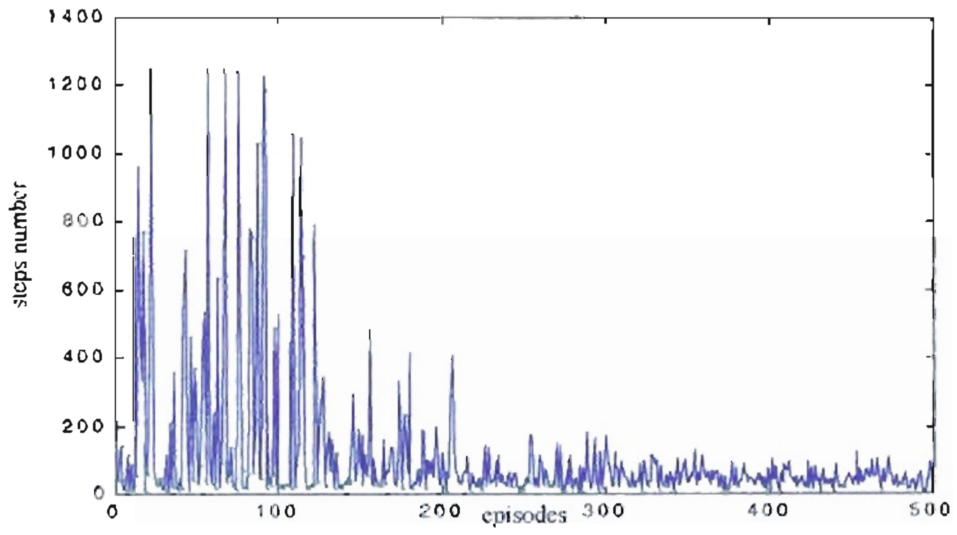


Figure 7.34: *Learning algorithm for speed transition with random initial speed and random initial fueling using the basic engine model.( ) episodes where* 650 ±1 rpm *was not reached*

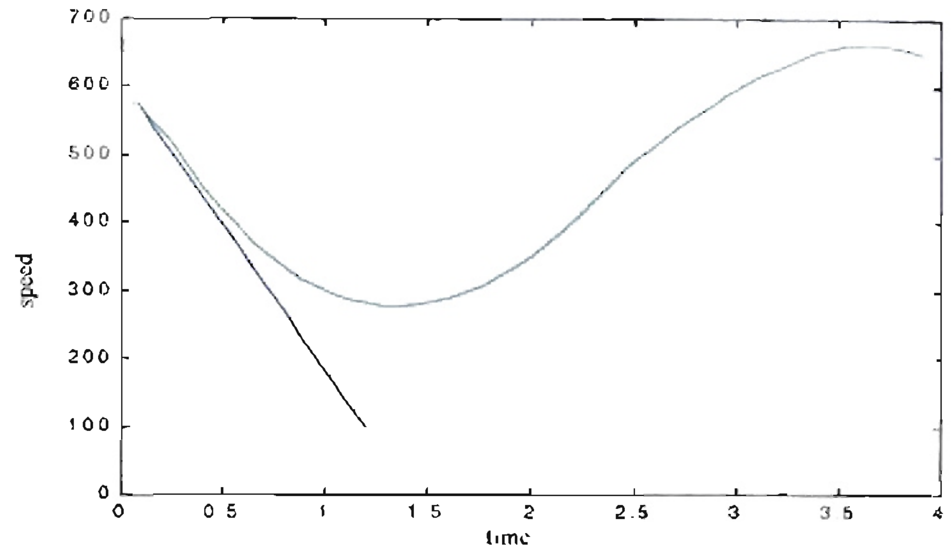


Figure 7.35: *Engine speed response for the basic model. Blue: Initial fueling*

0.0558 mm^3 */ stroke. Green: Initial fueling 20* mm^3 *stroke.*

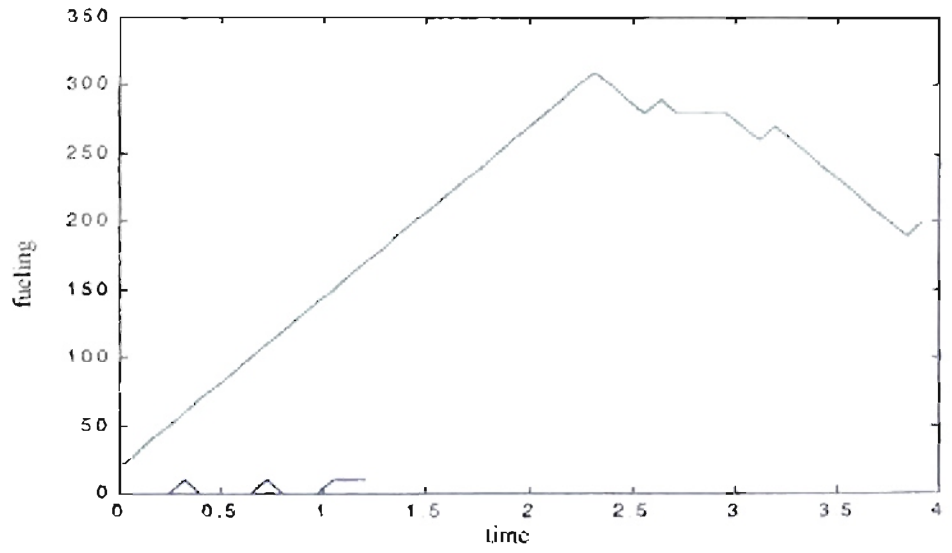


Figure 7.36: *Engine fueling for the basic engine model. Blue: Initial fueling*

0.0558 mm^3 */ stroke. Green: Initial fueling 20* mm^3 *stroke.*

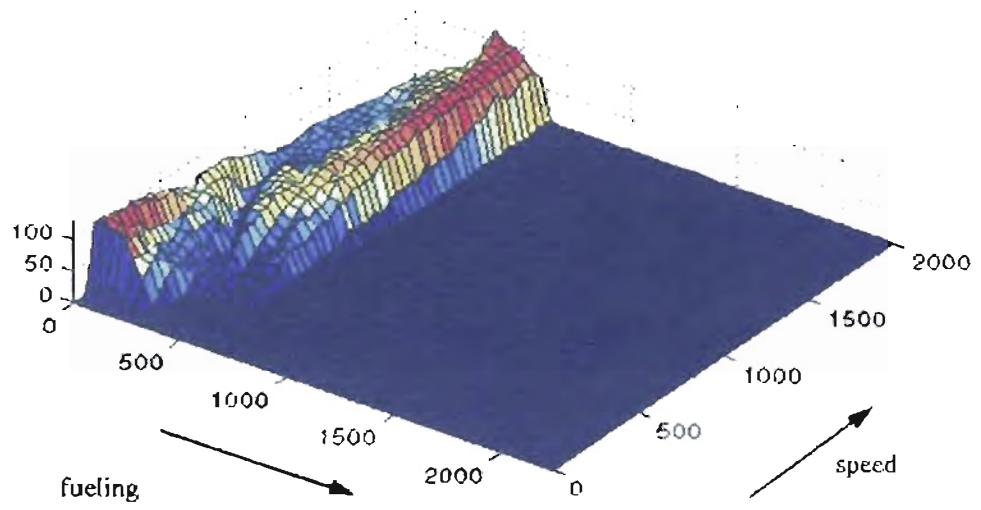


Figure 7.37: *Cost-to-go function for speed transition with random initial speed and random initial fueling using the basic engine model.*

7.5.2. Neural Network model.

We execute the reinforcement learning algorithm until the engine speed reaches 1500 ± 1 rpm starting from a random speed between 800 and 1700 rpm and a random fueling between 0.0558 and 0.7433 lb/min. The algorithm receives a penalty of $r = -1$ for each step that the specified speed is not reached. When the engine speed arrives at 1500 ± 1 rpm or the episode lasts 1200 *seconds* without arriving at the desired speed, the episode is concluded. Then we start a new episode with the same initial conditions described previously. The episode also concludes if the engine speed reduces to 570 rpm or increases to 2000 rpm with a penalty $r = -100$ for each case. The reinforcement learning algorithm has three possible actions: increase the fueling by 0.01 lb/min, decrease the fueling by the same quantity or maintain the same fueling.

After 500 episodes we obtained the learning curve shown in Figure 7.38. The longest episode was 1900 steps long. However, the desired speed was reached for that episode. We noted that after 250 episodes the learning curve is improved and the frequency of episodes that reach the goal is increased as shown in Figure 7.38.

After training we executed an episode starting at 800 rpm and a low fueling of 0.0558 lb/min. Figure 7.39 shows the speed transition for that episode in 78 steps. For the conditions of this experiment the algorithm increased the engine speed slower than the previous experiments. The solution is less optimal as we search with more initial conditions. However, the goal is reached for the final experiment. Figure 7.40 shows the fuel mass and the air mass for the last episode. Here the algorithm increased and decreased the fueling with some intermediate stages where the fueling remain constant.

Figure 7.41 shows a detail of the cost-to-go function from this experiment. We notice how the cost has a flatter surface due to the extended initial conditions and increased number of episodes used for the training.

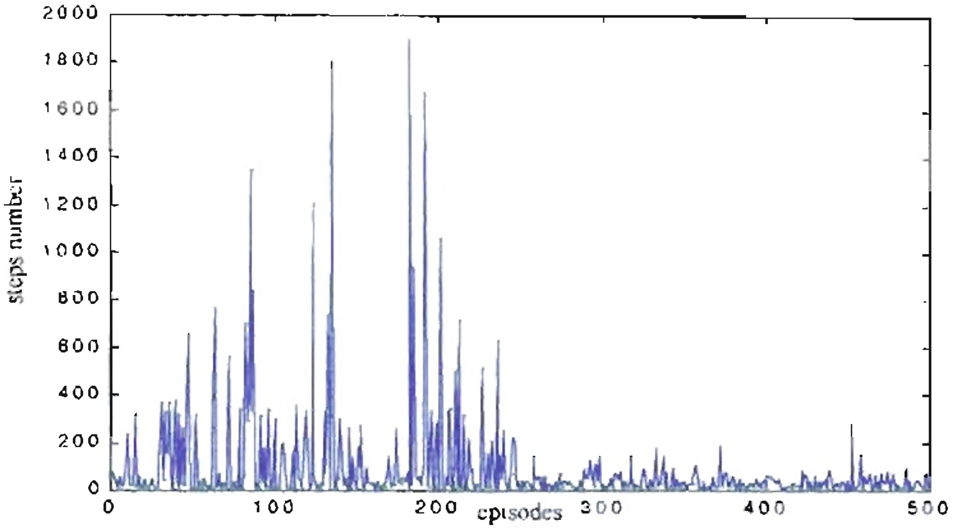


Figure 7.38: *Learning curve for speed transition with random initial speed and random initial fueling using the neural network model. ( ) - episodes where* 1500 ±1 rpm *was not reached.*

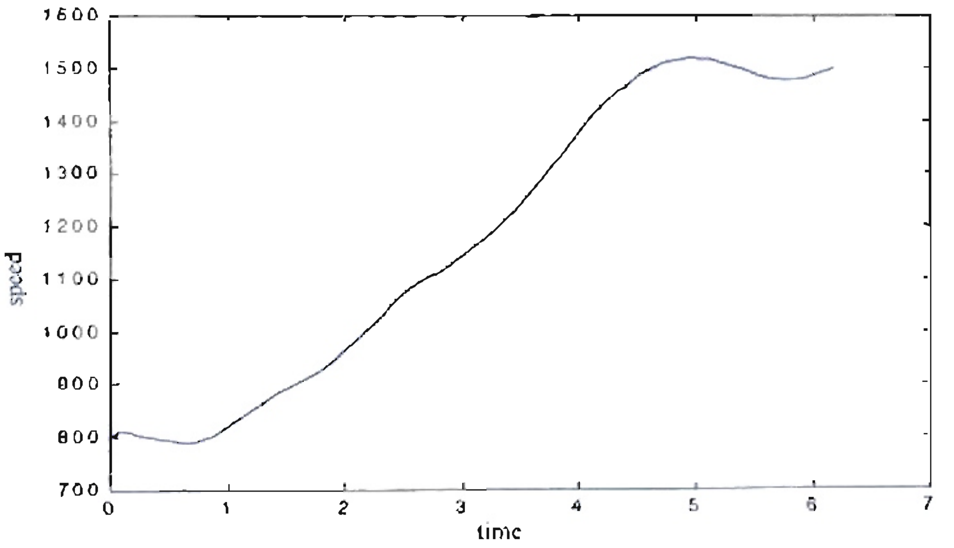


Figure 7.39: *Engine speed for the 78 steps episode starting with 800 rpm.*

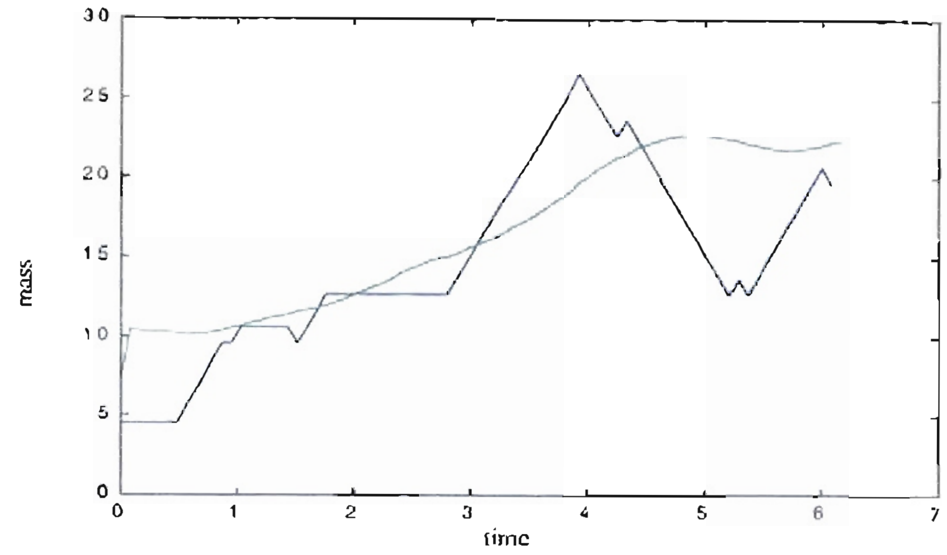


Figure 7.40: *Air mass (green) and fuel mass (x 100) (blue) for the 78 steps episode starting at 800 rpm.*

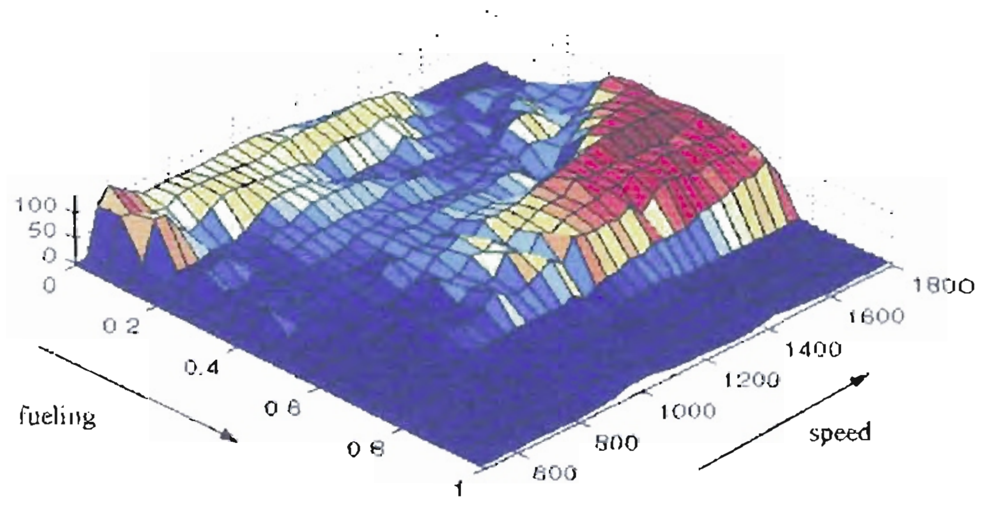


Figure 7.41: *Cost-to-go function for speed transition with random initial speed and random initial fueling using the neural network model.*

We repeated the experiment with the same initial speed of 800 rpm and fueling of 0.0558 *lb/min*, but we will continue with the engine operation until we arrive at 570 rpm or 2000 rpm. We note that the engine speed oscillates around 1500 rpm after 4 seconds of operation and maintains that oscillation until 23 seconds, as shown in Figure 7.42. The engine speed decays after that time due to the load torque which changes to a positive value as shown in Figure 7.43, where we see an initial portion of the torque used for the simulations that is shown in Figure 7.5.

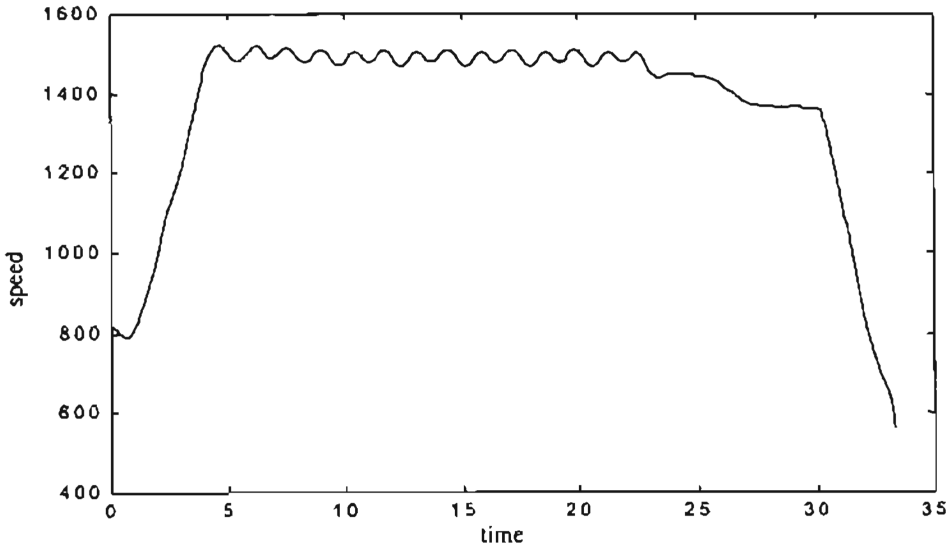


Figure 7.42: *Engine speed for continual operation.*

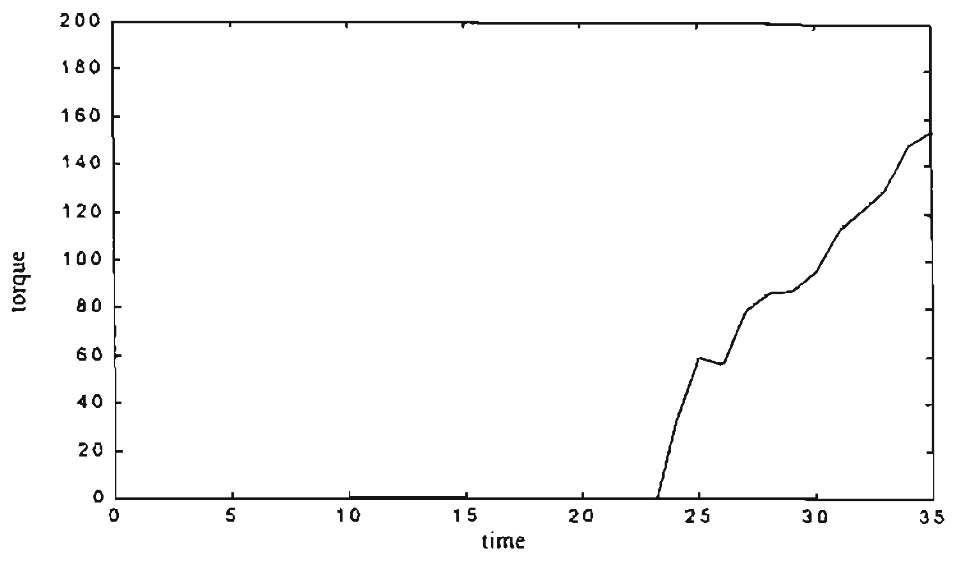


Figure 7.43: *Load torque for continual operation.*

For the next case, we set the load torque to zero and execute the continual operation for 20 minutes. Here the engine operated around the 1500 rpm as shown in Figure 7.44. Figure 7.45 details the engine speed from 400 to 460 rpm. We note an abrupt change in the frequency of oscillation, then an small oscillation is generated, given a new oscillatory scheme with lower amplitude. A similar behavior is observed in Figure 7.46 from 940 to 1100 rpm.

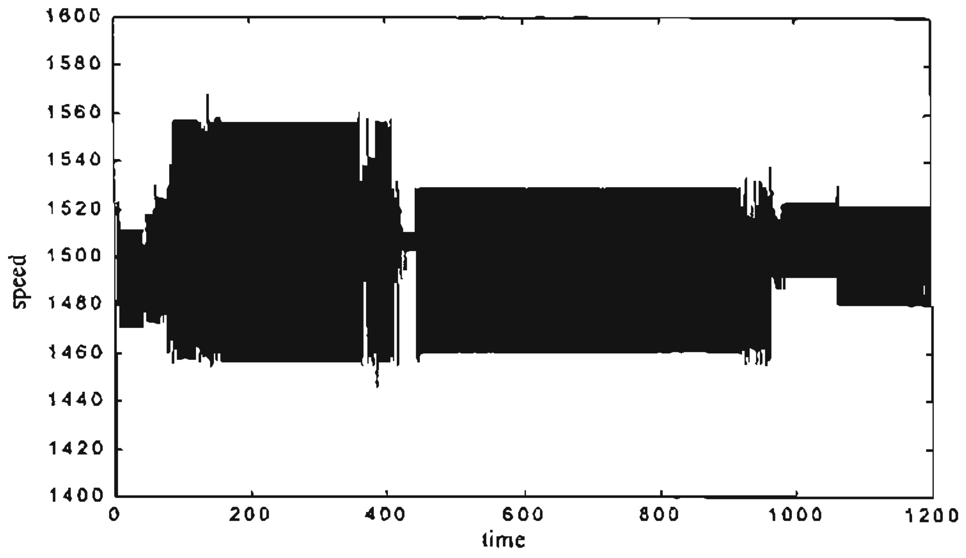


Figure 7.44: *Engine speed for continual operation and zero load torque.*

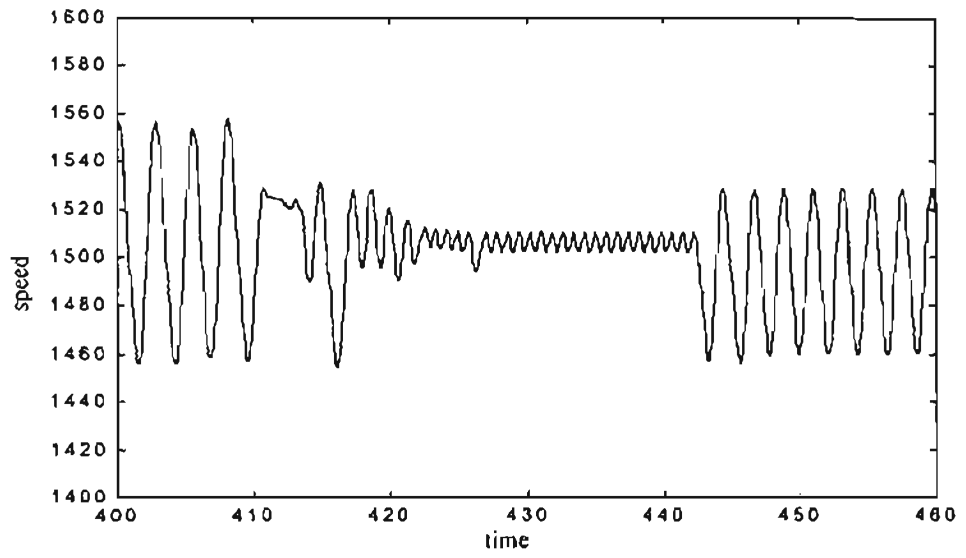


Figure 7.45: *Detail of the engine speed for continual operation and zero load torque.*

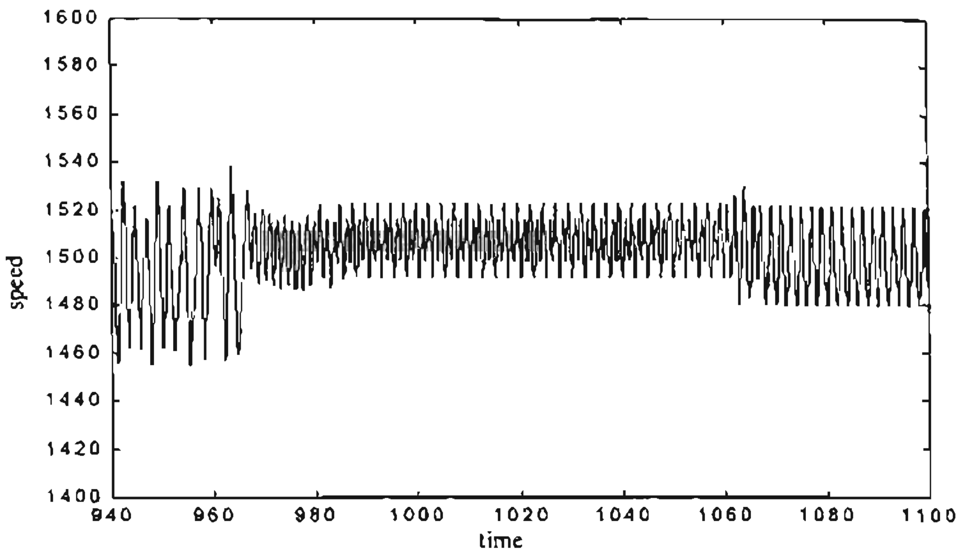


Figure 7.46: *Detail of the engine speed for continual operation and zero load torque.*

7.6. Tracking a reference engine speed.

After testing how the reinforcement learning algorithm could learn how to change the engine speed from one level to another, we now will test how the controller tracks a desired engine speed. A change was to include the engine acceleration as a third variable to analyze the engine behavior and update the reinforcement learning algorithm. That state variable will be used in all the subsequent experiments.

The previous experiments worked with a equally divided tiling for fueling and engine speed. We noted that the tile coding resolution near the desired speed (650 or 1500 rpm) and near zero acceleration generated oscillations in the control response. We changed the tilings resolution by applying a log sigmoid function to both state variables centered at the desired speed or zero acceleration, as shown in Figure 7.47 and Figure 7.48. With this new tiling, we obtained a better resolution near the important points. We will use the initial equidistant tile coding and the log sigmoid tile coding in the experiments of the subsequent sections.

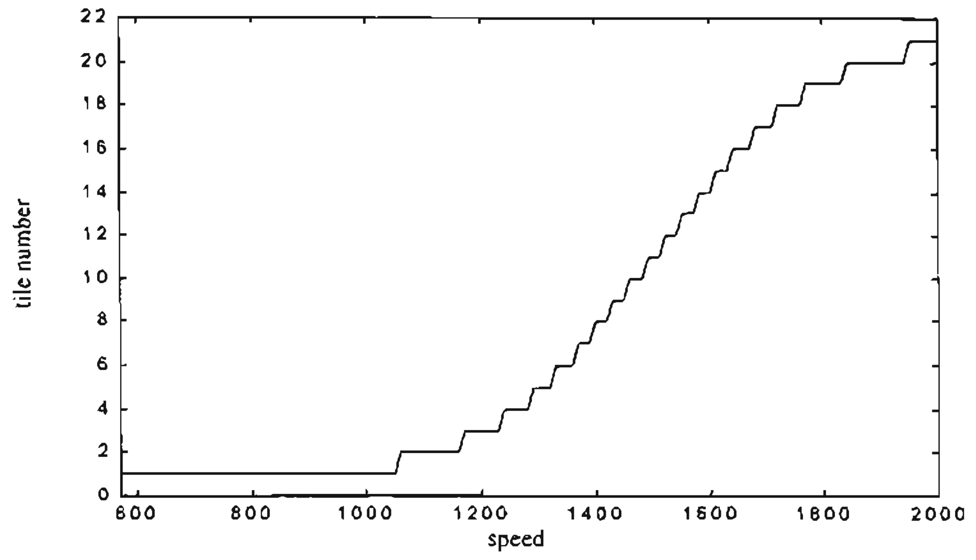


Figure 7.47: *Tile description for engine speed. Case 1500 rpm.*

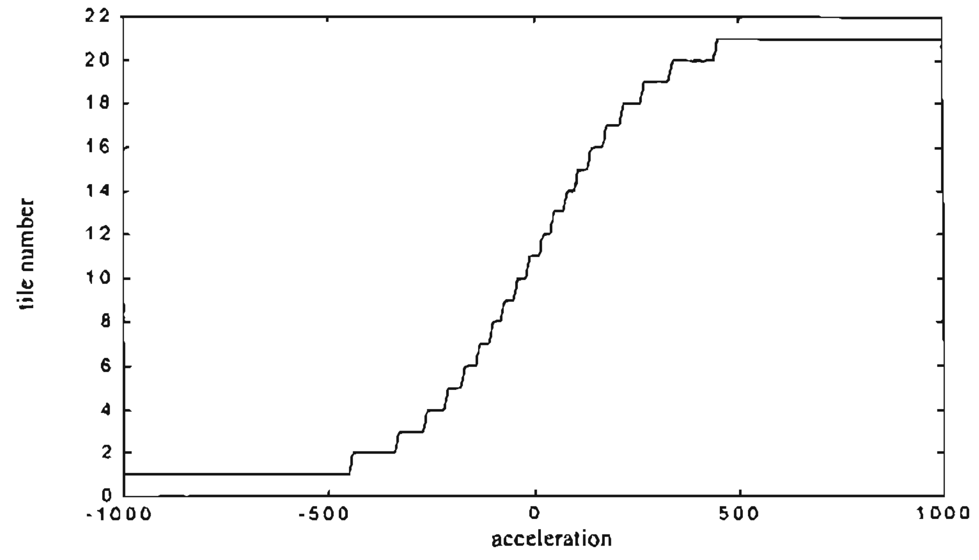


Figure 7.48: *Tile description for engine acceleration.*

7.6.1. Basic engine model.

We executed the reinforcement learning SARSA algorithm to maintain the engine speed near 650 rpm starting from a random initial speed between 300 and 1700 rpm and random initial fueling between 0.0558 $mm^3/stroke$ and 560 $mm^3/stroke$. We applied the following: a reward equal to the absolute value of the error between the desired speed and the actual speed and a large final reward if the engine arrives at 2000 rpm or 100 rpm. For these experiments we used -13938840 as the final reward. For this experiment, we use the log sigmoid tile coding.

After training, if we start the engine with an initial speed of 576 rpm and fueling of 60 $mm^3/stroke$, we obtain the response shown in Figure 7.49. We observe that after an initial overshoot the engine response is maintained close to the desired speed. However, before the episode concludes, an additional overshoot is generated and the engine does not reach the required speed. That behavior could be due to a saturation effect. The neural network, after working near a given speed, increases the cost function for the actions near that condition. The engine then jumps to a non explored zone.

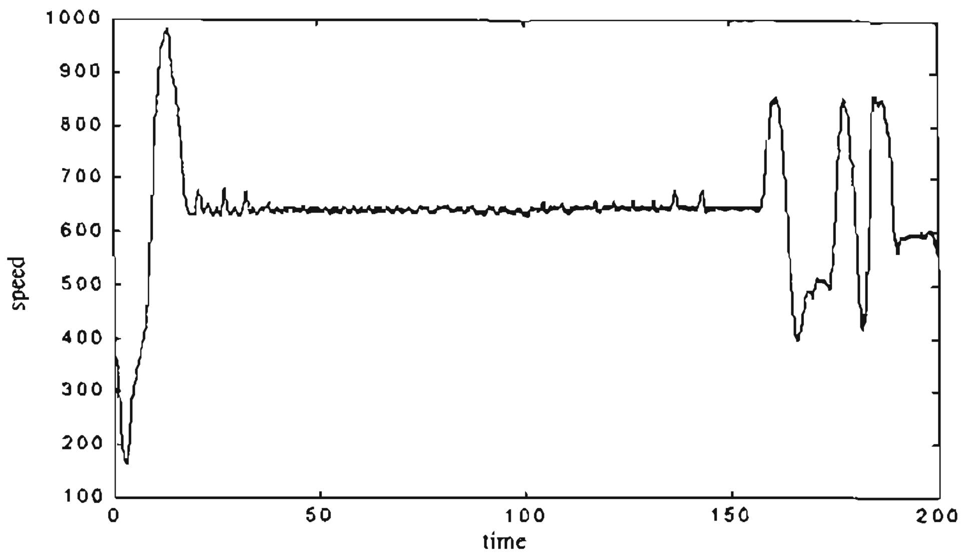


Figure 7.49: *Engine speed for the 2500 steps episode starting with 576 rpm.*

7.6.2. Neural Network model.

We execute the reinforcement learning SARSA algorithm to maintain the engine speed near 1500 rpm starting from a random initial speed between 800 and 1700 rpm and random initial fueling between 0.0558 and 0.7433 lb/min. We applied the same reward of the absolute value of the error between the desired speed and the actual speed and a large reward of -13938840 if the engine arrives at 2000 rpm or 570 rpm.

If we execute this experiment we obtain the learning curve shown in Figure 7.50. After training, if we start the engine with an initial speed of 800 rpm and fueling of 0.0558 lb/min, we obtain the response shown in Figure 7.51. We observe a noisy response in the engine speed, but the engine speed is maintained inside the range from 570 to 2000 rpm.

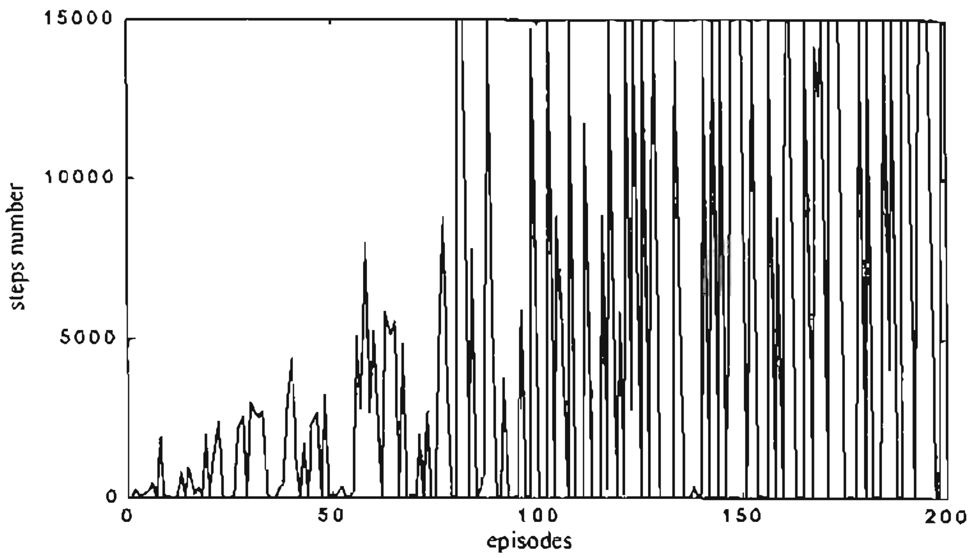


Figure 7.50: *Learning curve for tracking a reference speed with error based reward. Number of steps per episode.*

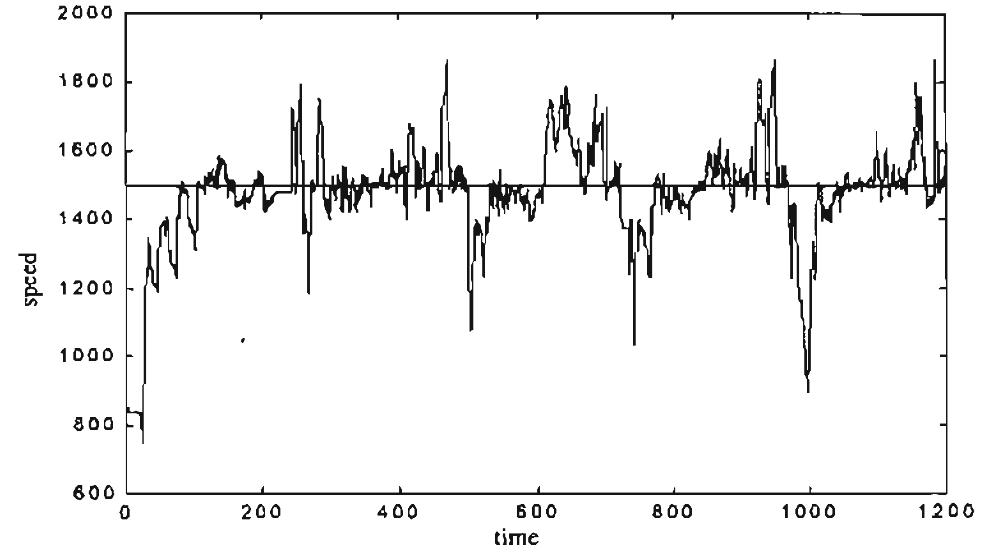


Figure 7.51: *Engine speed for the 14976 steps episode starting with 800 rpm.*

7.7. Tracking a reference engine speed time reward scheme including positive rewards.

In this section we will test the tracking ability of the reinforcement learning algorithm if positive rewards are added. In the previous cases only negative rewards were used.

7.7.1. Basic engine model.

We executed the reinforcement learning SARSA algorithm to maintain the engine speed at 650 ± 20 rpm starting from a random initial speed between 300 and 1700 rpm and random initial fueling between 0.0558 and 560 $mm^3/stroke$. We included the following rewards:

- 15000 if we arrive at 2000 rpm or 100 rpm. The episode also finishes.

+1 if we arrive at 650 ± 20 rpm. The episode continues.

-1 if we arrive at any different state. The episode continues.

The acceleration was calculated as the difference between the actual and previous engine speed divided by the algorithm update time of 0.08 seconds. For this experiment we used the equidistant tile coding defined at the beginning of this chapter.

If we execute this experiment we obtain the learning curve shown in Figure 7.52. That curve shows the combination of the total number of steps in the episode plus the total number of steps where the engine is inside the region from 630 to 670 rpm. After 200 training episodes, if we start the engine with an initial speed of 800 rpm and fueling of 0.0558 lb/min, we obtained the response shown in Figure 7.53. Here the engine speed is maintained inside the 100 to 2000 rpm range for 2500 steps or 200 seconds, but the engine stayed inside the 650 ±20 rpm interval for only 103 steps or 8.24 seconds, which represents a 4 % of the total time, as seen in Figure 7.53.

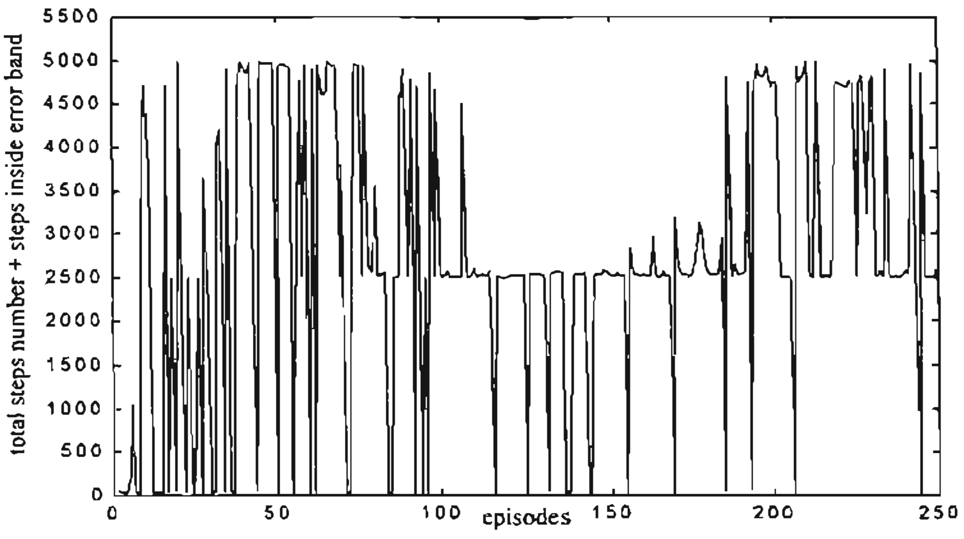


Figure 7.52: *Learning curve for tracking a reference speed. Number of steps per episode plus steps where the engine is inside* 650 ±20 rpm .

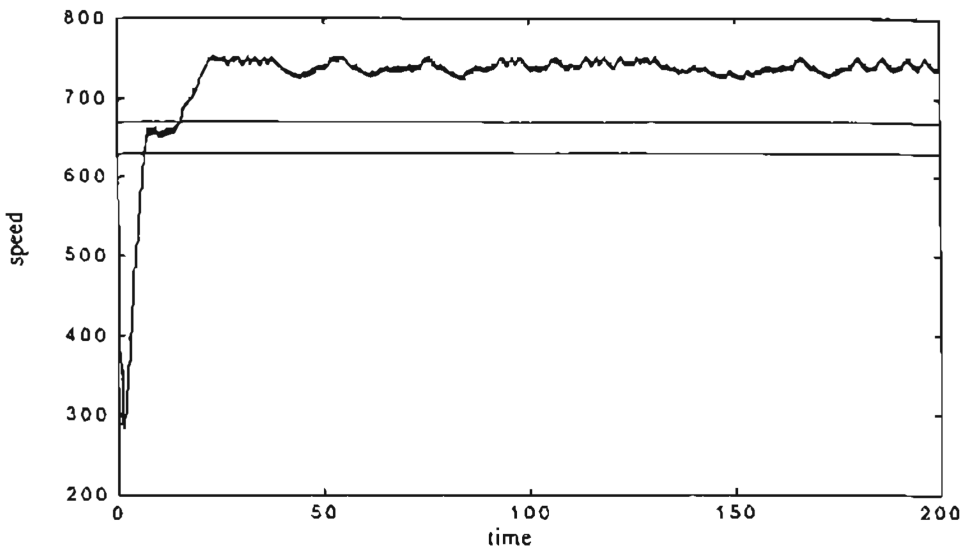


Figure 7.53: *Engine speed for the 2500 steps episode starting with 576 rpm.*

We tried to improve the response of Figure 7.53 by using the log sigmoid tile coding shown in Figure 7.47 and Figure 7.48. With that tile coding we obtained the learning curve shown in Figure 7.54 where we have the combination of the total number of steps in the episode plus the total number of steps where the engine is inside the region from $630 \text{ to } 670 \text{ rpm}$. That curve shows an improvement with respect to the previous learning curve of Figure 7.52, with more episodes where the combination of the total number of steps in each episode plus the desired range is higher. Also, improvement in the engine response is shown in Figure 7.55. We notice that the engine speed is maintained inside the range $630 \text{ to } 670 \text{ rpm}$.

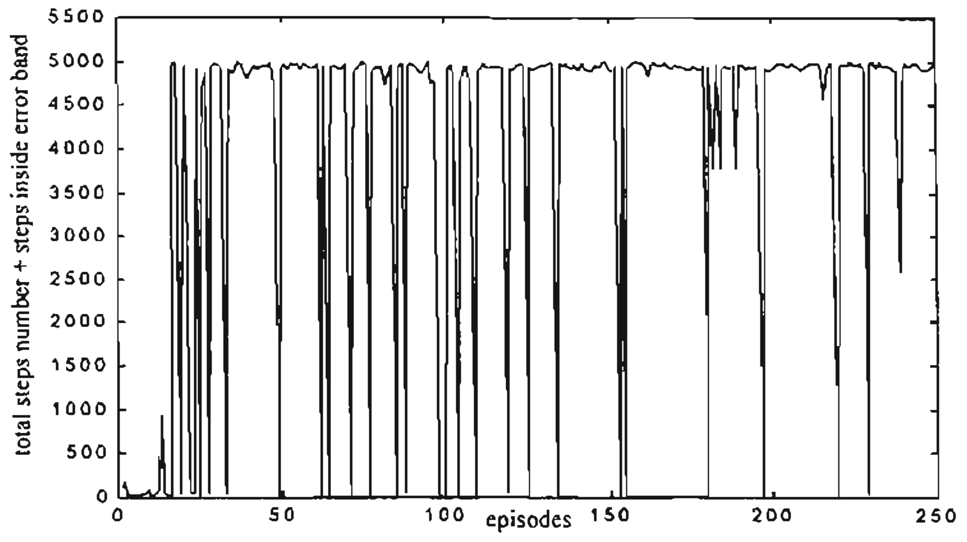


Figure 7.54: *Learning curve for tracking a reference speed with log sigmoid state scaling for the engine speed. Number of steps per episode plus number of steps where the engine is inside* 650 ±20 rpm.

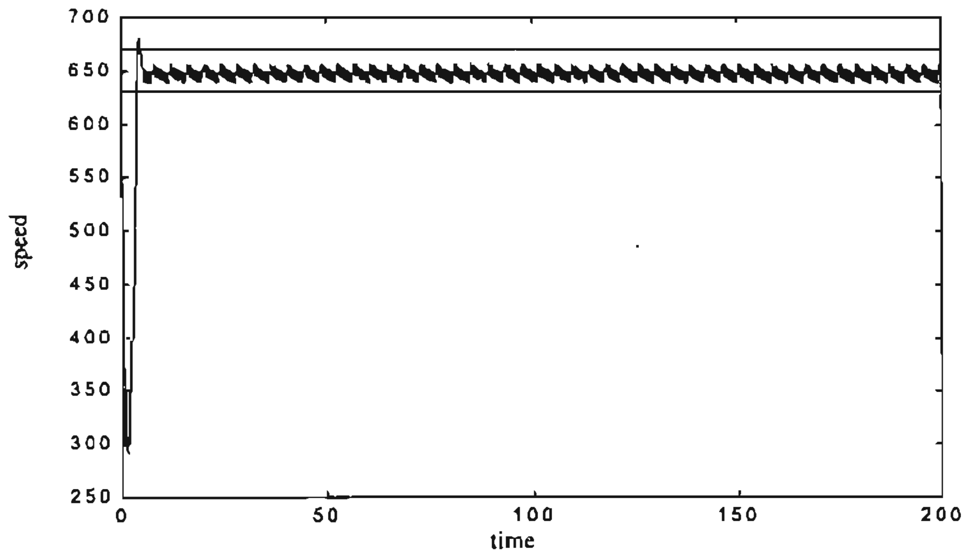


Figure 7.55: *Engine speed for the 2501 steps episode starting with 560 rpm.*

7.7.2. Neural Network model.

We execute the reinforcement learning SARSA algorithm to maintain the engine speed at 1500 ± 20 rpm starting from a random initial speed between 800 and 1700 rpm and random initial fueling between 0.0558 and 0.7433 lb/min. We included the following rewards:

- 15000 if we arrive at 2000 rpm or 570 rpm. The episode finishes.

+1 if we arrive at 1500 ± 20 rpm. The episode continues.

-1 if we arrive at any different state. The episode continues.

We used the equidistant tile coding described at the beginning of the chapter.

If we execute this experiment we obtain the learning curve shown in Figure 7.56. We note that we can maintain the engine speed inside the region from 570 to 2000 rpm in 28 of the simulations. After 200 training episodes, if we start the engine with an initial speed of 800 rpm and fueling of 0.0558 lb/min, we obtain the response shown in Figure 7.57. Here the engine speed is maintained inside the 570 to 2000 rpm range for 6223 steps or 497.84 seconds, but the engine stayed inside the 1500 ± 20 rpm for only 505 steps or 40.40 seconds, which represents a 8 % of the total time as seen in Figure 7.57.

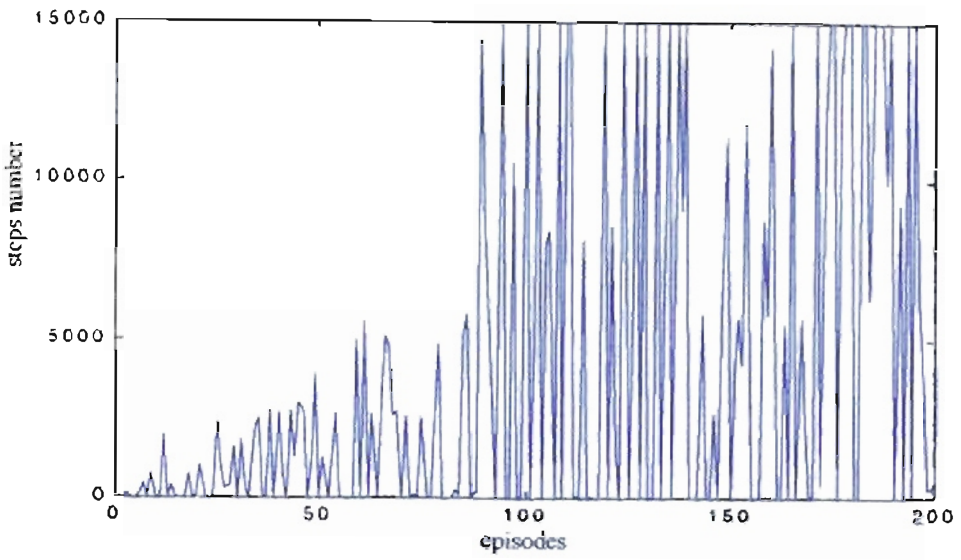


Figure 7.56: *Learning curve for experiment 10. (.) episodes where* 1500 ±20 rpm *was not reached.*

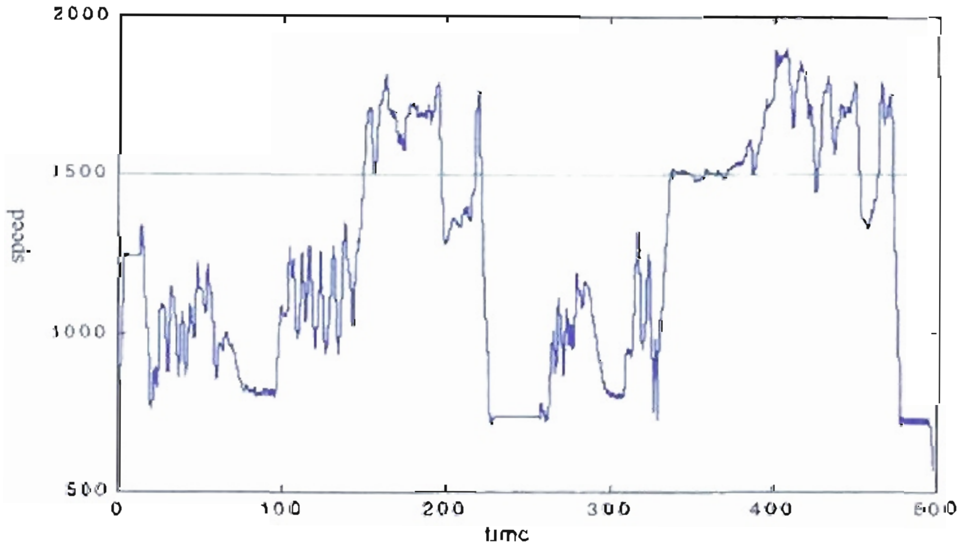


Figure 7.57: *Engine speed for the 1715 steps episode starting with 800 rpm*

We tried to improve the response of Figure 7.57 by using the log sigmoid tile coding shown in Figure 7 47 and Figure 7.48. With that change we obtained the learning curve shown in Figure 7.58. That curve shows an improvement with respect to the previous learning curve of Figure 7.56. Also, improvement in the engine response is shown in Figure 7.59.

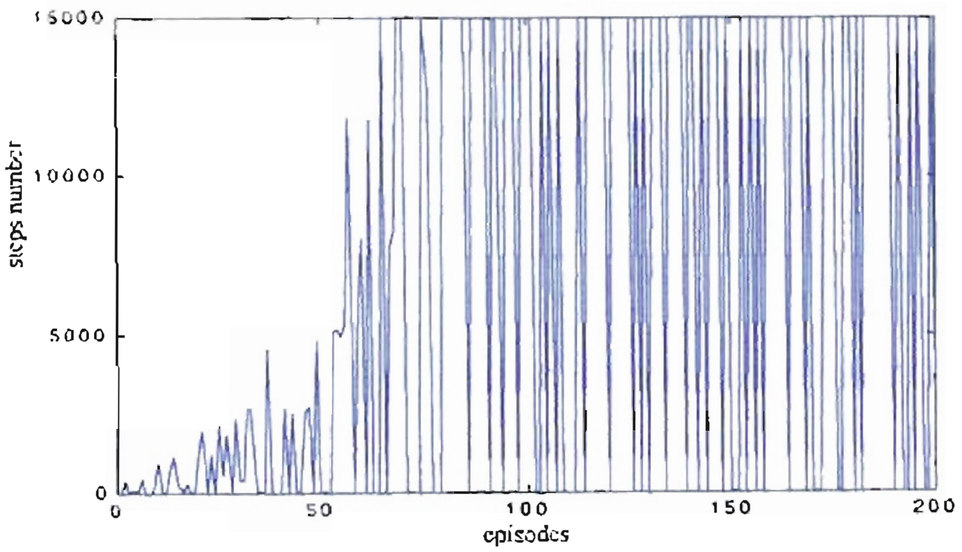


Figure 7.58: *Learning curve for experiment 10 with log sigmoid state scaling. ( ) episodes where* 1500 ±20 rpm *was not reached.*

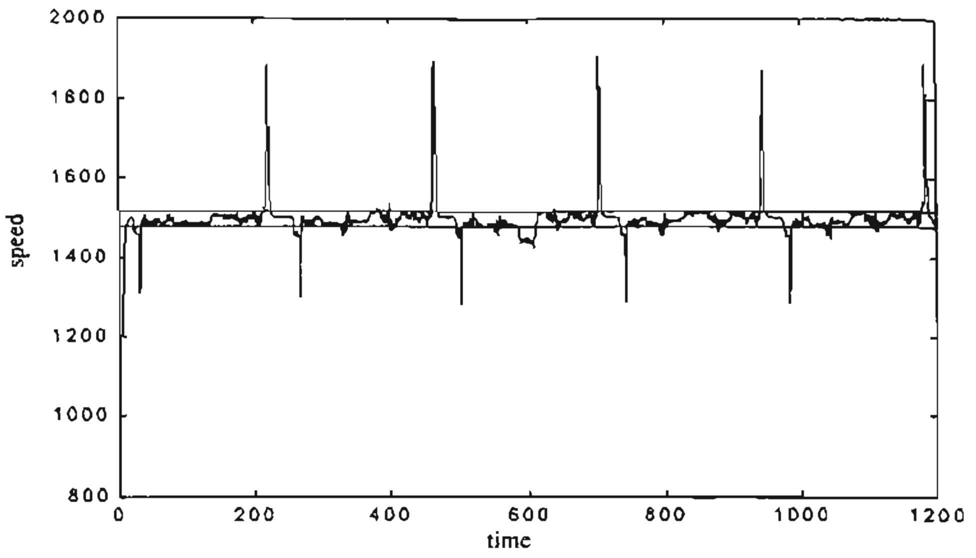


Figure 7.59: *Engine speed for the 14976 steps episode starting with 800 rpm.*

7.8. Tracking a reference engine speed with multiple neural networks.

Another reinforcement learning approach that we will try consists in factoring the state space into different regions as suggested by Dean and Lin $_{(5)}$. We divided the state space into three regions. We define a region where the engine speed will converge. That first region, which we called the middle region, is defined as the zone $desiredspeed \pm errorband$. The second region, which we called the lower region, is below the middle region $(desiredspeed - errorband)$. The third region, or the upper region, will be above $desiredspeed + errorband$. The last two regions will have extreme limits where the episode will conclude with failure.

For each region we have a CMAC neural network. Only one of the networks will be active at a given time. When the engine speed crosses from region A to region B, this will imply the pseudo-end of an episode for the neural network of the region A. For region B it will be like a new episode, starting at the conditions given at the transition moment. Another important factor is the way in which the rewards are given. For the region inside the desired speed, the rewards are always positive to maintain the engine speed close to our objective. Otherwise, the rewards at the outside regions are negative, with high negative reward at maximum and minimum speed. With that scheme we will try to force the engine speed to move toward our objective region.

We used the equidistant tile coding defined at the beginning of the chapter for all the experiments.

7.8.1. Basic engine model.

The middle region was defined between 640 and 660 rpm. The lower and upper region are from 100 to 640 rpm, and from 660 to 2000 rpm respectively. We defined the control actions in the middle region to be: increment fueling by 1 $mm^3/stroke$, decrement fueling by the same amount or maintain fueling constant. In the other two regions the actions were: increment fueling by 1 $mm^3/stroke$, decrement fueling by the same amount or maintain fueling constant. The rewards were also different. Inside the middle region the reward is +1 each time we stay inside that region. We use that scheme of rewards to reinforce the algorithm to stay inside 640 to 660 rpm. The other two regions have a reward of -1 each time we are inside the region. With that penalty we want to reinforce the algorithm to move out those regions. We also include a penalty of -15000 each time we arrive at 100 rpm or 2000 rpm to avoid those borders. We started each training with a random initial speed between 300 and 1700 rpm and random initial fueling between 0.0558 and 560 $mm^3/stroke$.

After training, we started the engine with an initial speed of 567 rpm and fueling of 100 $mm^3/stroke$, we obtained the response shown in Figure 7.60. We observe a noisy response in the engine speed, with the speed changing between the zones after an initial overshoot.

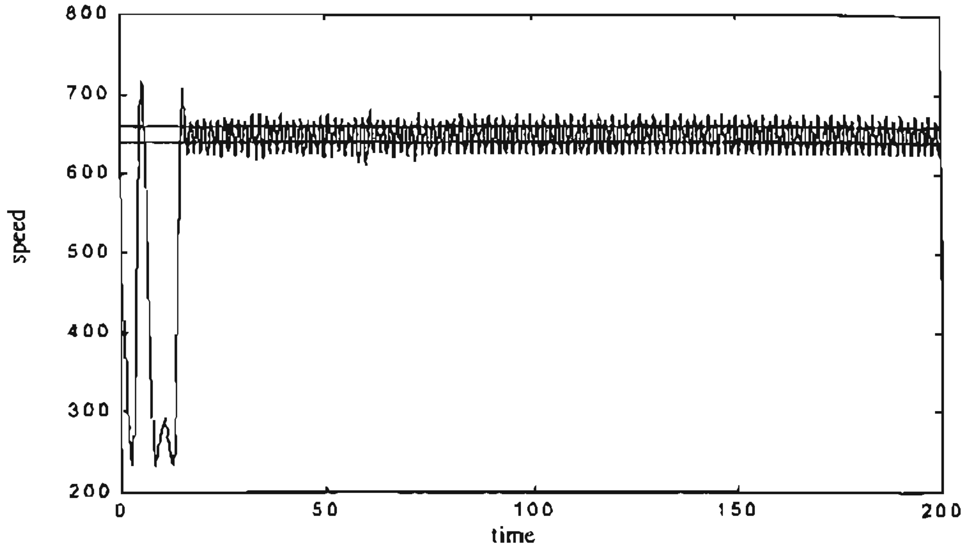


Figure 7.60: *Engine speed for the 2500 steps episode starting with 576 rpm and three CMAC neural networks.*

A variation for this technique could be to eliminate the middle zone. In this case we will obtain two neural networks operating in opposite directions. The lower region will try to move the engine speed toward the desired speed and avoid low speeds. The upper region will avoid faster speeds and will move the engine speed toward our goal. With the crossing zone defined at 600 rpm, we trained the controller. After training, we started the engine with an initial speed of 567 rpm and fueling of 60 $mm^3/stroke$, we obtained the response shown in Figure 7.61. If we compare this with the previous case, we notice a reduced transition period, and the oscillation amplitude is also reduced.

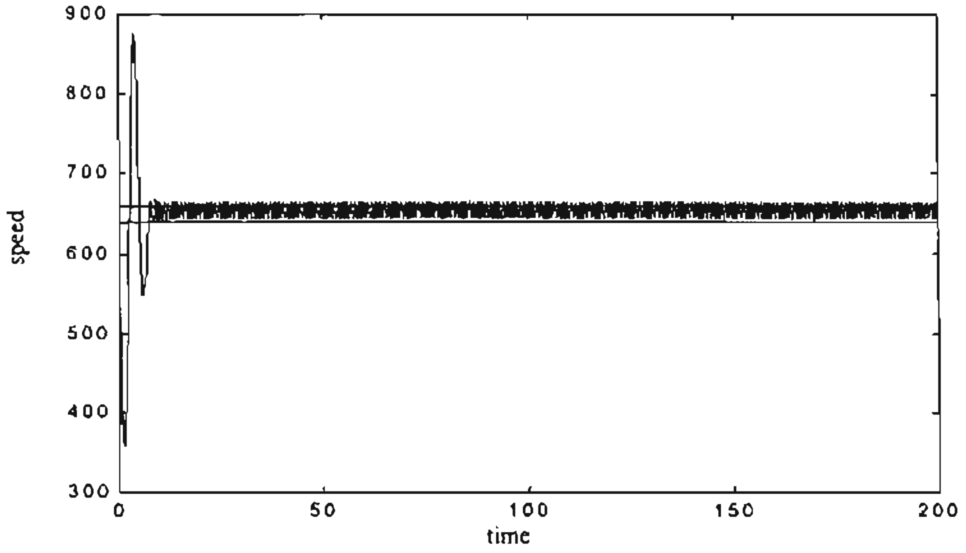


Figure 7.61: *Engine speed for the 2500 steps episode starting with 576 rpm and two CMAC neural networks.*

7.8.2. Neural Network model.

We defined the middle region to be between 1480 and 1520 rpm. The other two regions were from 570 to 1480 rpm, and from 1520 to 2000 rpm. We defined the control actions inside the middle region to be: increment fueling by 0.001 lb/min, decrement fueling by 0.001 lb/min or maintain fueling constant. In the other two regions the actions were: increment fueling by 0.01 lb/min, decrement fueling by 0.01 lb/min or maintain fueling constant. The rewards were also different. Inside the desired region the reward is +1 each time we stay inside that region, to reinforce the algorithm to stay inside the 1480 to 1520 rpm region. The other two regions have a reward of -1 each time we were inside the region. With that penalty we want to reinforce the algorithm to move out from those regions. We also include a penalty of -15000 each time the engine arrived at 570 rpm or

2000 rpm, avoiding those borders. We started each experiment with a random initial speed between 800 and 1700 rpm and random initial fueling between 0.0558 and 0.7433 lb/min .For this case we change the probability for random actions to 0.1 after the 110th step.

If we execute this experiment we obtain the learning curve shown in Figure 7.62. We notice how the number of episodes that are completed increases with the training. After training, if we start the engine with an initial speed of 800 rpm and fueling of 0.0558 lb/min, we obtain the response shown in Figure 7.63. We observe a noisy response in the engine speed, but the reinforcement algorithm maintained the speed inside the range from 1480 to 1520 rpm the majority of the time. Some of the peaks outside the middle region could be explained as exploratory actions from the reinforcement learning algorithm.

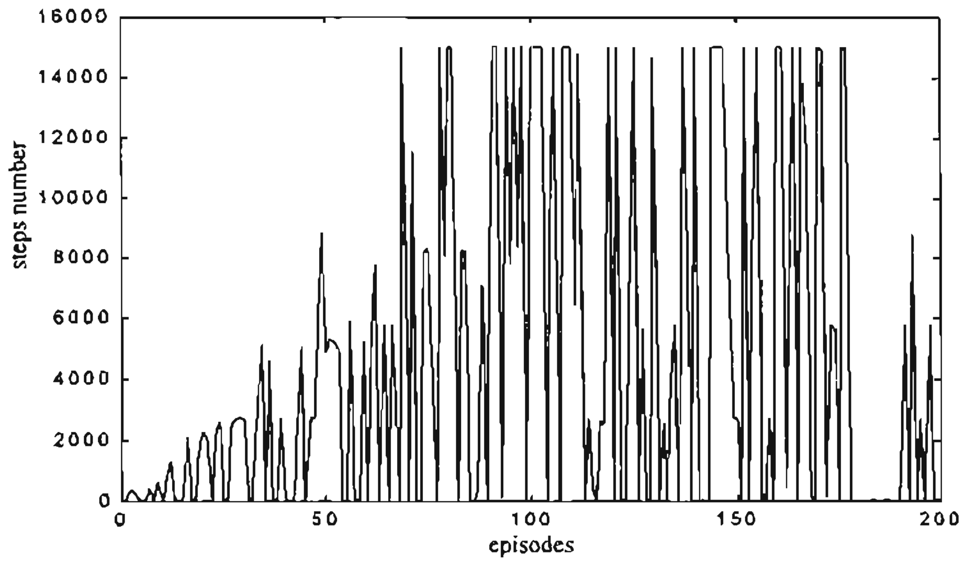


Figure 7.62: *Learning algorithm for tracking a reference engine speed with multiple neural networks.*

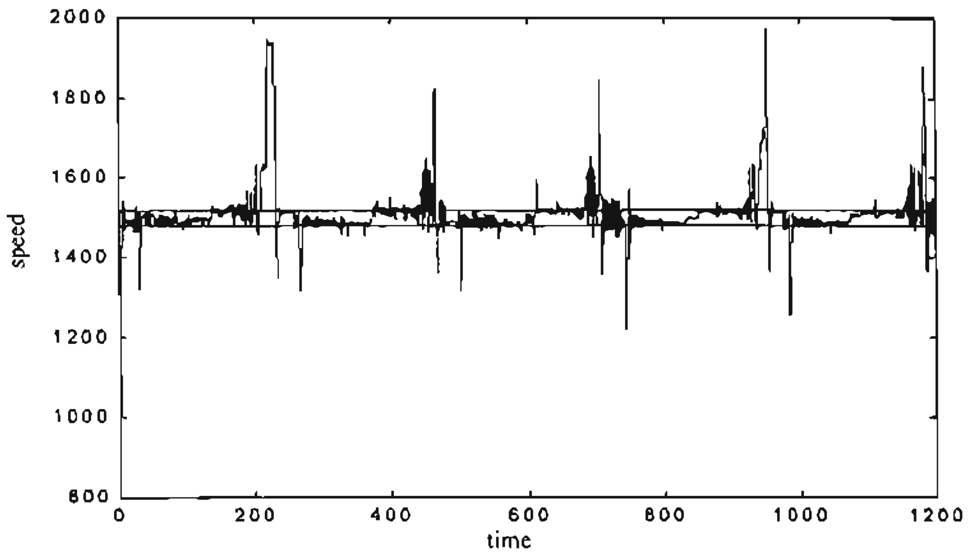


Figure 7.63: *Engine speed for the 14976 steps episode starting with 800 rpm.*

From the experiments made in this sections, we can see that the reinforcement learning algorithm is capable to learn how to change the engine speed from an initial conditions to a given set-point. For the conditions of the experiment we notice that the combination of positive and negative rewards resulted in a better engine response. The experience with two and three neural networks did not result in better responses. We estimated that better results will be obtained with the inclusion of more networks. An additional improvement was notice when we used the log sigmoid tile coding. This tile coding implies a better exploration of the state space and therefore better responses will be allowed.

CHAPTER 8

CONCLUSIONS

In this chapter we present a brief summary of results. This is followed by recommendations for future work.

Summary of Results

We have discussed several procedures which might be useful in the speed control of diesel engines. In Chapter 2 we presented a discussion of the diesel engine operation. We reviewed the direct injection (DI) and indirect injection (IDI) engines. A second division is based on how the gas exchange process is performed. Here we could divide the engines between two-stroke and four-stroke models.

In Chapter 3 we presented two engine models that were later used for the simulations of the reinforcement learning algorithms. The first model was based on a proposed pseudo-linear model, which considered fueling delay and engine inertia and friction. The second model was based on data collected from a real engine. With that data we developed a neural network model of the engine.

In Chapter 4 we applied the self-tuning regulator, with adaptive pole placement, to the diesel engine control. We found that the best approach is to start with the closed-loop locations of the base-line PID controller. When the system is identified, we can optimize the final pole locations by reducing their magnitude.

In Chapter 5 we applied genetic reinforcement learning to the optimization of PID controller parameters. The genetic algorithm provided improved performance over the base-line PID controller. The engine response changed with the selected fitness function (mean square error or percent overshoot). The results were valid for the analog PID controllers as well as digital versions of the controller.

In Chapter 6 we presented a general framework for general reinforcement learning. Reinforcement learning is an approximate dynamic programming framework. This framework is most appropriate when developing controllers for complex nonlinear systems that are difficult to model in closed form, but that can be simulated.

There are two stages in the reinforcement learning process. The first step is to develop a prediction of future system performance. The second step is to determine the appropriate controller to optimize future performance. There are many different implementations of reinforcement learning. Chapter 6 discussed Monte Carlo methods and temporal difference methods.

Several different simulation studies were discussed in Chapter 6. These simulations demonstrate the feasibility of using reinforcement learning for training neural network controllers for nonlinear systems.

Chapter 7 showed some implementations of reinforcement learning for the speed control of diesel engines. We demonstrated that the algorithm will easily learn how to move the engine speed from an initial condition to a desired speed. For speed tracking we included the speed, acceleration and fueling as our state variables. Reward schemes based on absolute error and time inside the error zone were tested. Absolute error rewards produced a less oscillatory response and good tracking. However, in some cases they generated peaks outside our desired engine speed. Rewards based on penalty per time outside the desired speed region generated oscillatory responses near the reference speed. Additional improvement was obtained by concentrating the tiling distribution of the CMAC neural networks around the reference speed or zero acceleration. Also, state space partition, with the implementation of multiple neural networks, was tested, showing improvements in the controller response and training time. We tested configurations with two and three CMAC neural networks, and good results were also obtained.

If we compare the genetic reinforcement learning approach implemented in this thesis with the adaptive control experiments we made, we find that GENITOR allowed us to obtain controllers that improve the engine response. The training process of the GENITOR algorithm requires more time than the adaptive control algorithm. However the responses were better as seen in Table 8.1. The mean square error of the engine responses were better in 4 of the 6 configurations tested. The percent overshoot were reduced in all the GENITOR cases to less than 10 percent of the original engine response.

		Fueling delay 50 ms		Fueling delay 110 ms		Fueling delay 130 ms	
		$b_1 = 0.1229$	$b_1 = 0.01$	$b_1 = 0.1229$	$b_1 = 0.01$	$b_1 = 0.1229$	$b_1 = 0.01$
Mean square error	**Self-tuning**	111	112	60	56	22	19
	Genitor	93	87	64	45	46	15
Percent overshoot	**Self-tuning**	68	82	30	39	27	39
	Genitor	1	3	9	2	2	1

Table 8.1: *Percentage of the original engine response according to the control technique and fitness function.*

The reinforcement learning algorithm is intended for systems with large nonlinearities. Due to the characteristics of the algorithm, the engine response contains an oscillatory component that is missing from the adaptive controller and the genetic adaptive controller. However, its importance is based on its ability to learn with little or no information of the system. Reinforcement learning algorithms are the most time consuming training algorithms of all we tested.

Recommendations for Future Work

Additional work could be made by using genetic reinforcement learning applied to a neural network controller. We made some initial experiments with no promising results. However additional research could be done with different neural networks configurations and possible improvements could be obtained.

Additional research in the application of reinforcement learning algorithms could be done using additional neural network architectures, such as radial basis function or backpropagation neural networks. Also, the relation of reinforcement learning with other algorithms, like fuzzy logic could be explored. Experiments with other non-linear systems could be of interest to compare results and experience.

Additional research could be made with reinforcement learning algorithms that start with some information about the system. We could train a neural network with some input-output data from different operational conditions and with those initial values we could reduce the training time.

With respect to the implementation process, we found that porting the Matlab code to C language improved our training time. Future developments with large training time will be first written in C language.

The reinforcement learning algorithms will benefit from faster computers. The training process is highly time consuming. Therefore, reliable applications could be seen as the processing power increases.

REFERENCES

1. Astrom, K.J., Wittenmark, B., *Adaptive Control,* Reading, MA: Addison Weslay, 1995.
2. Benson, R.S., Whitehouse, N.D., *Internal Combustion Engines,* Oxford, England: Pergamon Press Ltd., 1979.
3. Bertsekas, D., Tsitsiklis, J.N., *Neuro-Dynamic Programming,* Belmont, MA: Athena Scientific, 1996.
4. Brown, S.C., Passino, K.M., "Intelligent control for an Acrobot," *Journal of Intelligent and Robotic Systems,* Vol. 18, 1997, pp. 209-248.
5. Dean, T, Lin, S.H., "Decomposition techniques for Planning in Stochastic Domains," In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence,* Menlo Park, Calif., 1995, pp. 1121-1127.
6. Jaakkola, T., Jordan, M.I., Singh, S.P., "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms," *Neural Computation,* Vol. 6, 1995, pp. 1185-1201.
7. Kao, M., Moskwa, J.J., "Turbocharged Diesel Engine Modeling for Nonlinear Engine Control and State Estimation," *Transactions of the ASME,* Vol. 117, 1995, pp. 20-30.
8. Lilly, L.C.R. *Diesel Engine Reference Book.* London, UK: Butterworth and Co, 1984.
9. Mathias, K., Whitley, D., "Remaping Hyperspace During Genetic Search: Canonical Delta Folding," *Foundations of Genetic Algorithms,* Vol. 2, 1993, pp. 167-186.
10. Singh, S.P., Sutton, R.S., "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning,* Vol. 22, 1996, pp. 123-158.
11. Spong, M.W., "Swing up control of the Acrobot using Partial Feedback Linearization," *Robot Control*, 1994, pp. 739-743.
12. Sutton, R.S., Barto, A.G., *Introduction to Reinforcement Learning.* Cambridge,Mass.: MIT Press, 1997. (Available from web site http://www-anw.cs.umass.edu/~rich/ sutton.html).
13. Sutton, R.S., "Integrated Architectures for Learning, Planning, and Reacting Based on

Approximating Dynamic Programming," *Proceedings of the Seventh Int. Conf. On Machine Learning,* pp. 216-224, 1973. Morgan Kaufmann: MIT Press, 1997.

14. Sutton, R.S., "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding," *Advances in Neural Information Processing Systems,* Vol. 8, Proceedings of the 1995 Conference, 1996, pp. 1038-1044.
15. Sutton, R.S., "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," *Working Notes of the 1991 AAAI Spring Symposium,* 1991, pp. 151-155.
16. Sutton, R.S., "Learning to Predict by the Methods of temporal Differences," *Machine Learning,* Vol. 3, 1988, pp. 9-44.
17. Sutton, R.S., "Implementation Details of the TD(λ) Procedure for the Case of Vector Predictions and Backpropagation," GTE Laboratories Technical Note TN87-509.1 May 1987.
18. Tuken, T., Fullmer, R.R., VanGerpen, J., "Modeling, Identification, and Torque of a Diesel Engine for Transient Test Cycles," SAE Technical Paper 900235, 1990.
19. Various Authors. *Webster's Encyclopedic Unabridged Dictionary of the English language.* New York, NY: Gramercy Books, 1996.
20. Watkins, C.J., Dayan, P., "Q-learning," *Machine Learning,* Vol. 18, 1992, pp. 279-292.
21. Watson, N., Janota, M.S., *Turbocharging the Internal Combustion Engine.* New York, NY: John Wiley & Sons, 1982.
22. Watson, N., "Dynamic Turbocharged Diesel Engine Simulator for Electronic Control System Development," *Journal of Dynamic Systems, Measurement, and Control,* Vol. 106, 1984, pp. 27-45.
23. Whitley, D., Beveridge, R., Graves, C., Mathias, K., "Test Driving Three 1995 Genetic Algorithms: New Test Functions and Geometric Matching," *Journal of Heuristics,* Vol. 1, 1995, pp. 77-104.
24. Whitley, D. *A.* "Genetic Algorithm Tutorial," *Statistics and Computing,* Vol. 4, 1994, pp. 65-85.
25. Whitley, D., Dominic, S., Das, R., Anderson, C.W., "Genetic Reinforcement Learning for Neurocontrol Problems," *Machine Learning,* Vol. 13, 1993, pp. 259-284.
26. Whitley, D., "An Executable Model of a Simple Genetic Algorithm," *Foundations of Genetic Algorithms,* Vol. 2, 1993, pp. 45-62.
27. Whitley, D., "Fundamental Principles of Deception in Genetic Search," *Foundations of Genetic Algorithms,* Vol. 1, 1992, pp. 221-241.
28. Whitley, D., Dominic, S., Das, R., "Genetic Reinforcement Learning with Multilayer

Neural Networks," *Proceedings of the International Conference on Genetic Algorithms,* 1991, pp. 562-569.

29. Whitley, D., Starkweater, T., Bogart, C., "Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity," *Parallel Computing,* Vol. 14, 1990, pp. 347-362.
30. Whitley, D., Starkweater, T., "GENITOR II: A Distributed Genetic Algorithm," *Journal of Experimental and Theoretical Artificial Intelligence,* Vol. 2, 1990, pp. 189-214.
31. Whitley, D., Hanson, T., "Optimizing Neural Networks Using Faster, More Accurate Genetic Search," *Proceedings of the Third International Conference on Genetic Algorithms,* 1989, pp. 391-396, San Mateo, CA.
32. Whitley, D., Kauth, J., "GENITOR: A Different Genetic Algorithm," *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence,* 1988, pp. 118-130, Denver CO.

APPENDIX A

MATHEMATICAL MODELS FOR COMPUTER SIMULATION.

Researchers have proposed many models for diesel engines, ranging from steady state performance look-up maps to very complicated multidimensional models. Kao and Moskwa (7) identified three types of diesel engine models: the Quasi-Steady Method, Filling and Emptying Method and Method of Characteristics. However, few of those models were developed for diesel engine control.

Kao and Moskwa (7) "summarized and synthesized" two models from all the previous papers available in diesel engine simulation. They called those models: mean torque production model and cylinder-by-cylinder model. They compared the proposed models with Watson's model (22), identifying their abilities for real time simulation.

A.1. Mean Torque Model.

The mean torque model is a combination of the "Quasi-Steady" and "Filling and Emptying" models. This model assumes average values of pressure, temperature and mass flow. This model is based on the components shown in Figure A.1 (7). The compressor is used to increase the air density consequently increasing the mass of air trapped in the cylinders of the engine. A higher mass of air implies that more fuel could be burn in less time. With that combination we increase the engine output power. From the basic laws of thermodynamics, the compression process raises the air temperature. The intercooler was introduced to increment the mass of air with a minimum temperature rise. Another advantage of the intercooler is reduce the initial temperature of the air at the cylinders and consequently reduce the temperature inside the engine process ("reduced thermal loading"). The air mass is distributed to the cylinder by the intake manifold. After the air is in the cylinders the combustion process occurs as described in Chapter 2. The exhaust gases are collected in the Exhaust Manifold. Those gases move the turbine that is connected to the compressor. This is the turbocharger effect where the exhaust gases allow the compression and the increment in the engine power without an increment in the engine size(8).

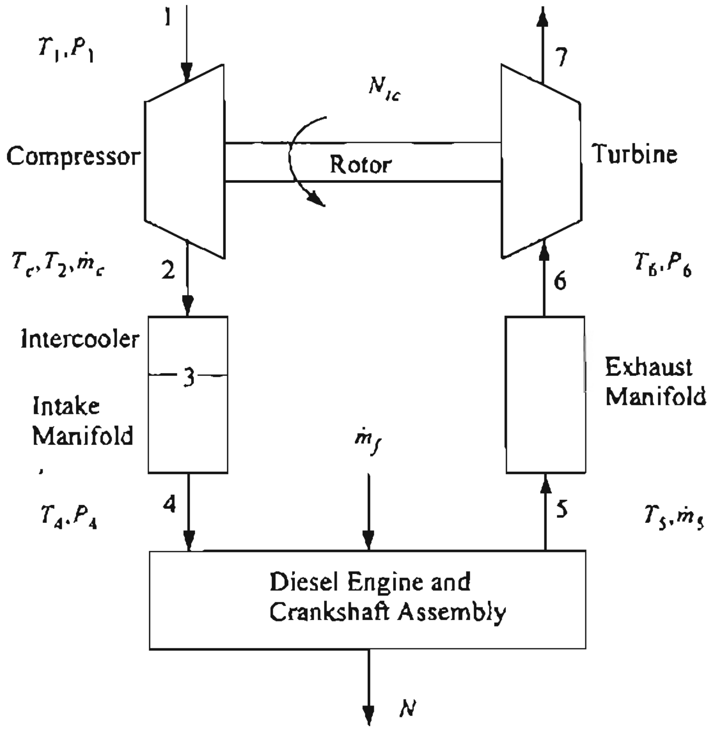


Figure A.1: *Schematic diagram of a turbocharged diesel engine* (7).

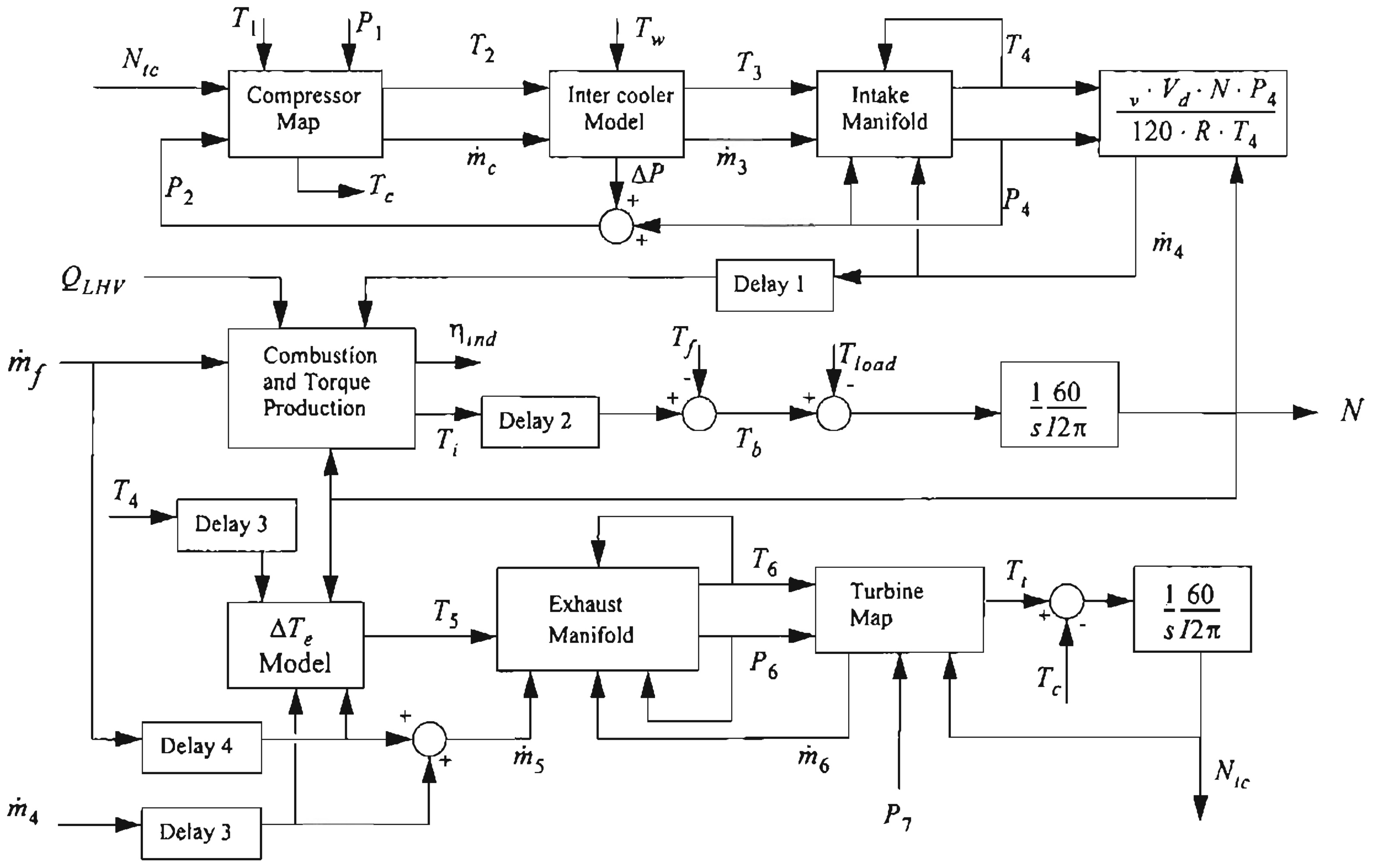


Figure A.2: *Block diagram interaction between submodels for turbocharged diesel engine model* (7).

Figure A.2 shows the different sub-models which make up the Mean Torque Model. Each sub-model can be modeled as:

A.1.1. Compressor model. Here we will create a table or map with compressor data from the manufacturer:

$$\dot{m}_{corr} = f_1\left(N_{corr}, \frac{P_2}{P_1}\right) \tag{A.1}$$

$$\eta_c = f_2\left(N_{corr}, \frac{P_2}{P_1}\right) \tag{A.2}$$

where P_1 is the pressure before the compressor, P_2 is the pressure after the compressor, $\dot{m}_{corr}$ is the corrected mass flow rate, N_{corr} is the corrected turbocharger speed and η_c is the compressor efficiency.

We can use the corrected mass flow rate $\dot{m}_{corr}$ and corrected turbocharger speed N_{corr} in the performance map from the relations:

$$N_{corr} = N_{tc} \cdot \sqrt{\frac{T_{std}}{T_1}} \quad \text{or} \quad \frac{N_{tc}}{T_1} \tag{A.3}$$

$$\dot{m}_{corr} = \frac{\dot{m}_c \cdot \sqrt{\frac{T_1}{T_{std}}}}{\frac{P_1}{P_{std}}} \quad \text{or} \quad \frac{\dot{m}_c \cdot \sqrt{T_1}}{P_1} \tag{A.4}$$

"By looking in the performance map, given the rotor speed of the turbocharger and the pressure ratio across the compressor, the mass flow rate and the efficiency are specified" (7). We can obtain the temperature at the outlet of the compressor and the torque at the compressor from the relations:

$$T_2 = T_1\left\{1 + \frac{1}{\eta_c}\left\{\left(\frac{P_2}{P_1}\right)^{\frac{\gamma-1}{\gamma}} - 1\right\}\right\} \quad \textbf{(A.5)}$$

$$T_c = \frac{m_c C_{pa} T_1}{\eta_c \omega_{tc}}\left\{\left(\frac{P_2}{P_1}\right)^{\frac{\gamma-1}{\gamma}} - 1\right\} \quad \textbf{(A.6)}$$

where T_1 is the temperature before the compressor, T_2 is the temperature after the compressor, m_c is the mass of air after the compressor, C_{pa} is the specific heat for air, γ is the specific heat ratio and ω_{tc} is the turbocharger speed.

A.1.2. Intercooler model. Here Kao and Moskwa (7) used a simple steady-state model. The pressure drop in across the intercooler is computed according to:

$$\Delta P = K\frac{\dot{m}_3^2}{\rho_3} \quad \text{(A.7)}$$

where K is a pipe friction constant, ρ_3 is the air density after the intercooler and $\dot{m}_3$ is the air mass after the intercooler. We have the same mass flow at the inlet and outlet of the intercooler, resulting in a heat exchange stage. That drop in pressure is with respect the intake pressure, therefore we need an estimation of P_4 to obtain:

$$P_2 = P_4 + \Delta P \quad \textbf{(A.8)}$$

The effectiveness (ε) of the intercooler is a nonlinear function of the mass flow:

$$\varepsilon = f(\dot{m}_3) \quad \textbf{(A.9)}$$

We can estimate the temperature at the outlet of the intercooler from the definition of effectiveness by:

$$T_3 = T_2(1-\varepsilon) + \varepsilon T_w \qquad \textbf{(A.10)}$$

where T_w is the coolant temperature.

A.1.3. Intake Manifold Model. We can calculate the average air mass flow into the cylinder with the "speed-density" relation:

$$\dot{m}_a = \frac{\eta_v \cdot \rho_4 \cdot V_d \cdot N}{120} \qquad \textbf{(A.11)}$$

where the air density ρ_4 and the volumetric efficiency η_v are calculated from:

$$\rho_4 = \frac{P_4}{RT_4} \qquad \text{and} \qquad \eta_v = f(N, P_4) \qquad \textbf{(A.12)}$$

where V_d is the displacement volume, N is the engine speed, R is a gas constant, P_4 and T_4 are the pressure and temperature after the Intake manifold.

We can assume that the temperature variations at the intake manifold are small. The pressure at the intake manifold could be estimated from:

$$\dot{P}_{im} = \frac{\gamma R}{V_{im}}\left\{\dot{m}_c T_c - \sum_{cyl} \dot{m}_{im} T_{im}\right\} \qquad \textbf{(A.13)}$$

where:

$$m_{im} = \int(\dot{m}_3 - \dot{m}_4)dt + \text{initial conditions} \qquad \textbf{(A.14)}$$

$$T_4 = \frac{P_4 \cdot V_{im}}{R \cdot m_{im}} \qquad \textbf{(A.15)}$$

We can rewrite Eq. (A.13) by using Eq. (A.11) and Eq. (A.12) as:

$$\dot{P}_{im} = \dot{m}_c \frac{R\gamma T_c}{V_{im}} - \frac{\gamma T_c \eta_v V_d N}{120 V_{im} T_4} P_{im} \quad \textbf{(A.16)}$$

If we assume that the heat transfer and temperature changes are negligible, we can use another model based on the relation:

$$\dot{P}_{im} = \dot{m}_c \frac{R T_{im}}{V_{im}} - \frac{\eta_v V_d N}{120 V_{im}} P_{im} \quad \textbf{(A.17)}$$

A.1.4. Combustion and torque production. This submodel is part of the Diesel Engine and Crankshaft Assembly described in Figure A.1 and detailed in Figure A.2. Here Kao and Moskwa (7) used an "statistical regression to curve-fit empirical indicated efficiency data". The indicated efficiency η_{ind} could be found from the relation:

$$\eta_{ind} = (a_1 + a_2 N + a_3 N^2)(1 - k_1 \Phi^{k_2}) \quad \textbf{(A.18)}$$

where:

$$\Phi = \frac{(F/A)_{actual}}{f_s} \quad \text{and} \quad (F/A)_{actual} = \frac{\dot{m}_f}{\dot{m}_a} \quad \textbf{(A.19)}$$

where $(F/A)_{actual}$ is the actual fuel air ratio and f_s is the stoichiometric[1] fuel air ratio.

The mean indicated torque T_i is:

$$T_i = m_f \cdot Q_{LVH} \cdot \eta_{ind} \quad \textbf{(A.20)}$$

where m_f is the amount of fuel injected and Q_{LVH} is the lower heating value of the fuel.

1. Stoichiometric: "pertaining to or involving substances that are in the exact proportions required for a given reaction" (19).

A.1.5. Engine Friction Model. This submodel is also part of the Diesel Engine and Crankshaft Assembly described in Figure A.1. To determine the friction mean effective pressure $fmep$, Kao and Moskwa (7) use the following relationship:

$$fmep = c_1 + \frac{48N}{1000} + 0.4S_p^2 \tag{A.21}$$

where the parameter c_1 could be determined by experimentation and S_p is the mean piston speed. We can obtain the torque due to friction by:

$$T_f = \frac{fmep \cdot V_d \cdot 1000}{6.28N_r} \tag{A.22}$$

A.1.6. Crankshaft Rotation Model. This submodel explains the relation of the engine load, frictional load and external load on the engine speed. From Newton's second law:

$$T_i(t - \tau_i) - T_f - T_{load} = I \cdot \dot{\omega} \tag{A.23}$$

where a constant engine rotational inertia I is used and τ_i is a delay in the application of the indicated torque T_i.

A.1.7. Valve Flows and Scavenge Flow. This submodel is also part of the Diesel Engine and Crankshaft Assembly described in Figure A.1. For the valve flows, Kao and Moskwa (7) used volumetric efficiency and mean exhaust flow. The authors neglected scavenge flows for the case of medium diesel engine speed due to the fact that the valve overlap is small.

A.1.8. Exhaust Manifold. The exhaust mass flow rate is assumed to be:

$$\dot{m}_{ex}(t) = \dot{m}_f(t - \tau_1) + \dot{m}_{airtrapped}(t - \tau_2) + \dot{m}_{airscavenged} \tag{A.24}$$

where τ_1 and τ_2 are delays. The exhaust temperature T_5 is given by:

$$T_5 = T_4(t - \tau_3) + \Delta T_{Edyna} \tag{A.25}$$

where τ_3 is a delay and:

$$\Delta T_{Edyna} = \Delta T_E + \Delta T_M e^{-t/\tau} \tag{A.26}$$

where ΔT_E is the engine temperature rise, ΔT_M is a transient magnitude offset and is a function of the air fuel ratio f, and τ is the exhaust manifold time constant. The engine temperature rise ΔT_E is given by(2):

$$\Delta T_E = \frac{K}{1 + (\dot{m}_a/\dot{m}_f)} = \frac{K}{1 + f} \tag{A.27}$$

where K is generally plotted versus the air fuel ratio $f = \dot{m}_a/\dot{m}_f$ as shown in Figure A.3 for a typical engine.

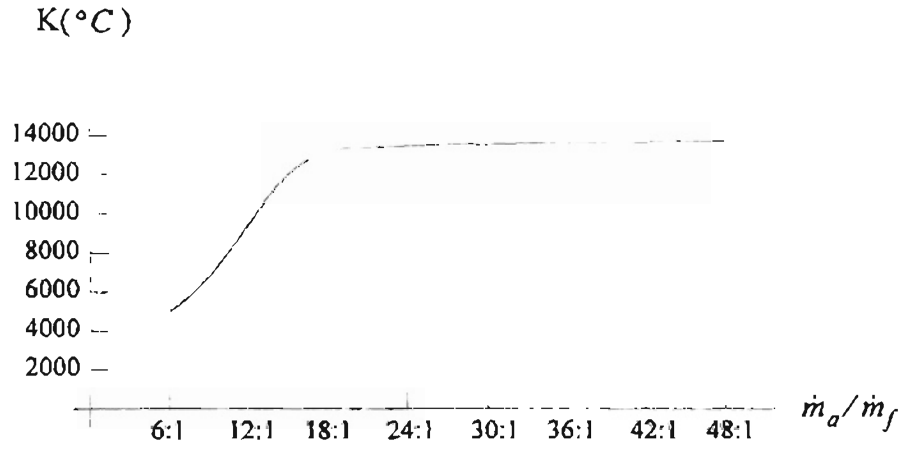


Figure A.3: *Engine temperature rise factor K (2).*

The exhaust manifold pressure is estimated by:

$$\dot{P}_6 = \frac{\gamma_e R_e}{V_{em}}\left[\frac{-\dot{Q}}{C_{pe}} + \dot{m}_5 T_5 + \dot{m}_6 T_6\right] \tag{A.28}$$

where the "gas properties (C_{pe}, R_e, γ_e) ... can be found from curve-fitted equations for hydrocarbon combustion products as a function of A/F ratio and temperature" (7) and:

$$\dot{Q} = ht \cdot A \cdot (T_6 - T_{wall}) \qquad \text{and} \qquad T_{wall} = \frac{\dot{Q}}{m_{wall} C_{pwall}} \tag{A.29}$$

where ht is the convective heat transfer coefficient and can determined by experimentation or can be calculated from:

$$ht = \frac{k \cdot Nu_d}{D} \tag{A.30}$$

where:

$$Nu_d = 0.0483 \cdot Re_d^{0.783} \qquad \text{and} \qquad Re_d = \frac{4 \cdot \dot{m}}{\pi \cdot \mu \cdot D} \tag{A.31}$$

We can estimate the exhaust manifold temperature T_6 from:

$$T_6 = \frac{P_6 \cdot V_{em}}{m_{em} \cdot R_e} \tag{A.32}$$

where:

$$m_{em} = \int(\dot{m}_5 - \dot{m}_6)dt \tag{A.33}$$

A.1.9. Turbine Model. The model is for a constant pressure turbine and is similar to the compressor model, where steady state information could be supplied by the manufacturer. With that information we can construct the tables:

$$\dot{m}_{corr} = f_3\left(N_{corr}, \frac{P_6}{P_7}\right) \tag{A.34}$$

$$\eta_t = f_4\left(N_{corr}, \frac{P_6}{P_7}\right) \tag{A.35}$$

As in the compressor's case, we can use the corrected mass flow rate $\dot{m}_{corr}$ and corrected turbocharged speed N_{corr} in the performance map from the relations:

$$N_{corr} = N_{tc} \cdot \sqrt{\frac{T_{std}}{T_6}} \quad \text{or} \quad \frac{N_{tc}}{T_6} \tag{A.36}$$

$$\dot{m}_{corr} = \frac{\dot{m}_c \cdot \sqrt{\frac{T_6}{T_{std}}}}{\frac{P_6}{P_{std}}} \quad \text{or} \quad \frac{\dot{m}_c \cdot \sqrt{T_6}}{P_6} \tag{A.37}$$

The torque supplied by the turbine is:

$$T_t = \frac{\dot{m}_6 C_{pe} T_6 \eta_t}{\omega_{tc}} \left\{ 1 - \left(\frac{P_7}{P_6}\right)^{\frac{\gamma_e - 1}{\gamma_e}} \right\} \tag{A.38}$$

A.1.10. Turbocharger Rotor Model. This model (without friction) is calculated from the Newton's second law:

$$T_t - T_c = I_{tc}\dot{\omega}_{tc} \tag{A.39}$$

A.2. Cylinder-by-Cylinder Model.

This model is based in the filling and emptying model, where the cylinder pressure with a crankangle-based model. This model is generally used for "cylinder-by-cylinder control, nonlinear state estimation, and dynamic model-based diagnostics"(7). This model is intended for the detailed study of the engine behavior at the cylinder level. This model uses the same turbocharger, intercooler, intake manifold, and exhaust manifold submodels from the previous section. The different submodels are concentrated in the Diesel Engine and Crankshaft Assembly shown in Figure A.1:

A.2.1. Equations from Thermodynamics. The equivalence ratio differential equation from the engine is (7, 22):

$$\frac{dF}{dt} = \left[\frac{1+f_sF}{m}\right]\left[\frac{(1+f_sF)}{f_s}\cdot\frac{dm_{fb}}{dt} - F\frac{dm}{dt}\right] \tag{A.40}$$

The temperature at the cylinder could be found by the relation:

$$\dot{T}_{cyl} = \left[-\frac{RT_{cyl}}{V}\dot{V} + (\dot{Q}_{ht} + h_{for}\cdot m_{fburn} + \sum(h\cdot m)_{in} - \sum(h\cdot m)_{out} - u\dot{m})\frac{1}{m} - \frac{\partial u}{\partial F}\dot{F}\right]\Big/\left(\frac{\partial u}{\partial T}\right) \tag{A.41}$$

where the terms of Eq. (A.41) will be explained in later sections. The mass flow conservation is now:

$$\dot{m} = \sum\dot{m}_{in} - \sum\dot{m}_{out} + \dot{m}_f \tag{A.42}$$

With Eq. (A.42) we can find the mass accumulated in the cylinder (m) with:

$$m = m_a + m_{fb} = \int\dot{m}dt \tag{A.43}$$

The cylinder pressure could be obtained from:

$$P_{cyl} = \frac{mRT_{cyl}}{V} \tag{A.44}$$

where the volume at the cylinder V will be found later. If we want to use heat related analysis we can apply the relation for the closed period:

$$\dot{P}_{cyl} = \frac{\gamma - 1}{V}(\dot{m}_{fburn} \cdot Q_{LVH} + \dot{Q}_{ht}) - \frac{P_{cyl}\gamma}{V}\dot{V} \qquad \textbf{(A.45)}$$

A.2.2. ***Indicated torque.*** The instantaneous indicated torque is found from the relation:

$$T_i = 1000 P_{cyl} \frac{dV}{d\theta} \qquad \textbf{(A.46)}$$

and the average indicated torque is:

$$T_{i,\,average} = \frac{\oint (1000 P_{cyl}) dV}{4\pi} \qquad \textbf{(A.47)}$$

A.2.3. ***Crankshaft Dynamics.*** The nonlinear dynamic crankshaft rotational equation is found from Lagrangian or Newtonian equations:

$$J(\theta) \cdot \ddot{\theta} + \frac{1}{2}\left(\frac{\partial J(\theta)}{\partial \theta} \cdot \dot{\theta}^2\right) = T_i - T_f - T_{load} \qquad \textbf{(A.48)}$$

This equation is important for state estimation, diagnostics, and control in the case of the cylinder-by-cylinder model. The inertia $J(\theta)$ changes according to the crankshaft position.

A.2.4. ***Intake and Exhaust Mass Flows.*** The mass flow depends of the engine operation cycle described in section 2.2. The average intake flow, assuming that the volume is injected during the IVO to IVC period is:

$$\dot{m}_{in} = \frac{\eta_v \cdot V_d \cdot \rho \cdot 6N}{\theta_{IVC} - \theta_{IVO}} \qquad \textbf{(A.49)}$$

We can apply a similar procedure for the average exhaust flow, assuming that the volume is displaced during the EVO to IVO period:

$$\dot{m}_{out} = \frac{m_{EVO} \cdot \frac{CR-1}{CR} \cdot 6N}{\theta_{IVO} - \theta_{EVO}} \tag{A.50}$$

For mass calculation, we need to define the flow in two classes:

- Subsonic flow, when $\frac{P_d}{P_u} > \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma}{\gamma-1}}$, resulting in a flow rate:

$$\dot{m} = C_d A P_u \sqrt{\left\{\frac{2\gamma}{R \cdot T_u(\gamma-1)} \cdot \left[\left(\frac{P_d}{P_u}\right)^{\frac{2}{\gamma}} - \left(\frac{P_d}{P_u}\right)^{\frac{\gamma-1}{\gamma}}\right]\right\}} \tag{A.51}$$

- Sonic flow, when $\frac{P_d}{P_u} \le \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma}{\gamma-1}}$, resulting in a flow rate:

$$\dot{m} = C_d A P_u \sqrt{\left\{\frac{\gamma}{R \cdot T_u} \cdot \left[\frac{2}{\gamma+1}\right]^{\frac{\gamma+1}{\gamma-1}}\right\}} \tag{A.52}$$

We can define the mass flow for the overlap period as:

$$\dot{m} = C_d \cdot A \cdot \sqrt{2\rho_4(P_4 - P_6) \cdot 1000} \tag{A.53}$$

A.2.5. Combustion and Fuel Burning Rate. For this submodel, Kao and Moskwa (7) used the single zone model proposed by Watson (22):

$$ID = 3.45\left(\frac{P_{cyl}}{101.3}\right)^{-1.02} \cdot e^{\frac{2100}{T_{cyl}}} \cdot \int_{tinj}^{ting} \frac{dt}{ID} = 1 \tag{A.54}$$

where the overall equivalence ratio is defined by the relation:

$$\Phi_{OVE} = F_{IVC} + \left(\frac{1}{f_s} + F_{IVC}\right) \cdot \frac{m_f}{m} \tag{A.55}$$

The normalized premixed burning rate is given by:

$$\dot{m}_{pre} = k_{p1} \cdot k_{p2} \cdot t_{norm}^{k_{p1}-1} (1 - t_{norm}^{k_{p1}})^{k_{p2}-1} \tag{A.56}$$

and the normalized diffusion rate is given by:

$$\dot{m}_{fdiff} = k_{d1} \cdot k_{d2} \cdot t_{norm}^{k_{p1}-1} \cdot e^{-k_{d2} \cdot t_{norm}^{k_{d2}}} \tag{A.57}$$

where the constants k_{p1}, k_{p2}, k_{d1}, and k_{d2} were defined by Watson (22). We also have the β value which defines the portion of total fuel that is premixed burned:

$$\beta = 1 - 0.926 \cdot \Phi_{OVE}^{0.37} \cdot ID^{-0.26} \tag{A.58}$$

The combustion time (in seconds) and the normalized time are given by:

$$dtcomb = \frac{125}{6N} \tag{A.59}$$

$$t_{norm} = \frac{\theta - \theta_{ign}}{125} \tag{A.60}$$

where 125 is the crankangle used for combustion. Finally, we obtain the fuel burning rate from:

$$\dot{m}_{fnorm} = \beta \dot{m}_{pre} + (1 - \beta)\dot{m}_{fdiff} \tag{A.61}$$

$$\dot{m}_{fburn} = \frac{m_f \cdot \dot{m}_{fnorm}}{dtcomb} \tag{A.62}$$

where this burning rate is needed in Eq. (A.41) and Eq. (A.45).

A.2.6. Gas and Fuel Properties. The internal energy correlation (u) and the gas constant (R) are given by (7, 21):

$$u(T, F) = \frac{A(T) - B(T) \cdot F}{1 + f_s \cdot F} \tag{A.63}$$

$$R = \frac{0.287 + 0.02F}{1 + f_s \cdot F} \tag{A.64}$$

where $A(T)$ and $B(T)$ are given by:

$$\begin{aligned} A(T) = {} & 0.692T + 39.17x10^{-6}T^2 + 52.9x10^{-9}T^3 \\ & - 228.62x10^{-13}T^4 + 277.58x10^{-17}T^5 \end{aligned} \tag{A.65}$$

$$\begin{aligned} B(T) = {} & 3049.39 - 5.7x10^{-2}T - 9.5x10^{-5}T^2 \\ & + 21.53x10^{-9}T^3 - 200.26x10^{-14}T^4 \end{aligned} \tag{A.66}$$

A.2.7. Cylinder Heat Transfer. For this submodel, Kao and Moskwa (7) used the Eichelberg's heat transfer coefficient:

$$ht = 7.67x10^{-3}\left(\frac{2LN}{60}\right)^{1/3}\left(\frac{P_{cyl} \cdot T_{cyl}}{1000}\right)^{1/2} \tag{A.67}$$

where the heat transfer rate is given by:

$$\dot{Q}_{ht} = ht \cdot A \cdot (T_{wall} - T_{cyl}) \tag{A.68}$$

where T_{wall} is given for the heat transfer relation from the cylinder wall to the coolant.

A.2.8. Cylinder Volume and Area. Kao and Moskwa (7) based this submodel in the cylinder geometry:

$$V = \frac{V_d}{CR - 1} + \frac{\pi B^2}{4}\left[l + r(1 - \cos\theta) - \sqrt{l^2 - r^2(\sin\theta)^2}\right] \tag{A.69}$$

where the cylinder heat transfer area is:

$$A = \alpha\frac{\pi B^2}{4} + \pi B\left[l + r(1 - \cos\theta) - \sqrt{l^2 - r^2(\sin\theta)^2}\right] \qquad \textbf{(A.70)}$$

where $\alpha > 2$ for a general non flat piston and cylinder heat and $\alpha = 2$ for a flat piston and cylinder head. The variation in the cylinder volume is represented by:

$$\frac{dV}{dt} = \left(\frac{\pi B^2}{4}\right) \cdot r \cdot \sin\theta \cdot \frac{d\theta}{dt} \cdot \left(1 + \frac{r \cdot \cos\theta}{\sqrt{l^2 - r^2(\sin\theta)^2}}\right) \qquad \textbf{(A.71)}$$

A.3. Watson's Model.

Watson $_{(22)}$ did an extensive review of the mathematical models for diesel engines available for that time. Watson described the requirements for the simulation as:

- 1 "sufficient detail to reflect design changes, key fuel property changes, and environment changes.
- 2 "ability to accurately predict performance, under steady and transient conditions, and key parameters that limit performance (such as high maximum cylinder pressure).
- 3 "ability to predict parameters that are known to strongly influence exhaust emission, particularly smoke and NO, and noise ...
- 4 "low consumption time and cost so that the model can be used routinely for short-term transients (up to 1 min) and less frequency for complete federal tests cycles, but at reasonable cost.
- 5 "the minimum empirical data requirement." $_{(22)}$

Linear models only meet requirement 1. Watson described two principal models: "filling and emptying" and "method of characteristics". For the "filling and emptying" models the inlet and exhaust manifolds and all cylinders are considered independent thermodynamic models. The equations are solved based on an engine crank-angle, not time base (generally 1 degree steps). We have submodels for model combustion, mass transfer through valves, heat transfer, etc. The "method of characteristics" is a mathematical technique based on hyperbolic partial differential equations. The cylinders are treated in the same way as the previous method, but exhaust (and sometimes inlet) manifolds are treated by solving dynamic gas equations.

The method suggested by Watson is based on "filling and emptying" assuming that all the cylinders behave in an identical manner. Then he reduced the computational time involved in the simulations. The model is based on a turbocharged engine as shown in Figure A.4, where we have the variables:

- W_c	Compressor work.
- W_t	Turbine work.
- Q_{cool}	Heat rejected to charge air cooler.
- Q_{ht}	Heat rejected to cylinder walls.
- Q_{com}	Heat released by combustion.
- W_{pis}	Piston work.
- Q_e	Heat rejected from exhaust manifold.

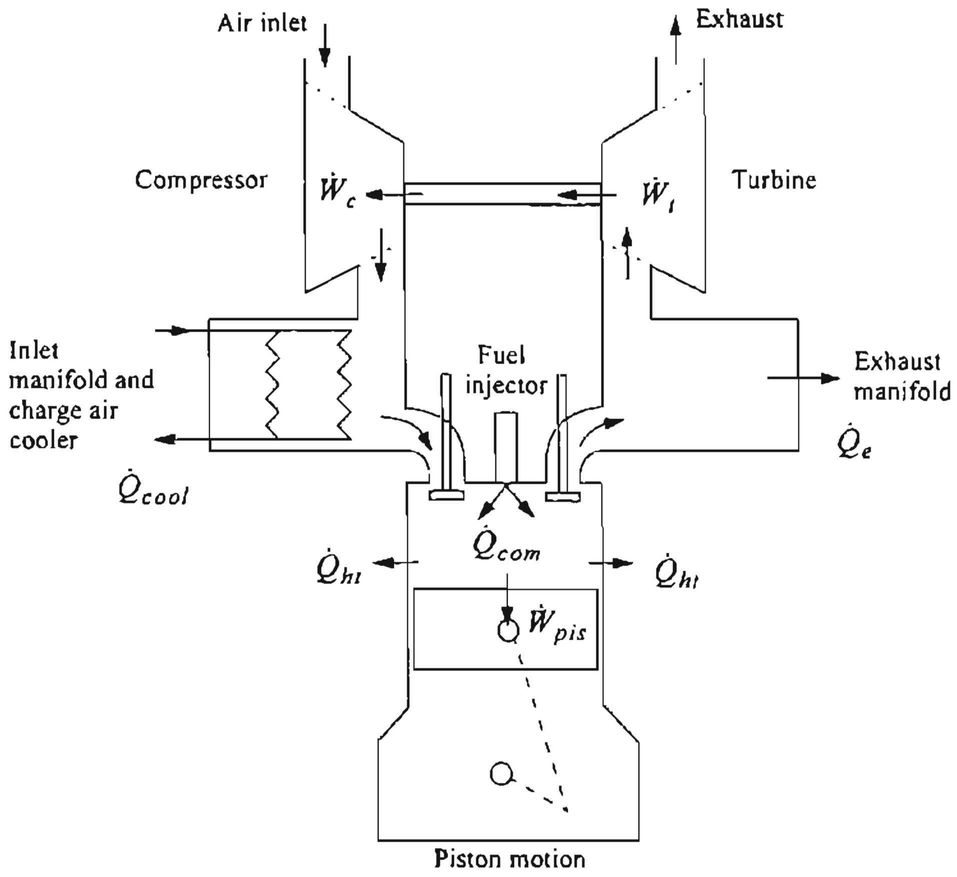


Figure A.4: *Schematic of turbocharged engine* (21, 22).

If we apply the first law of thermodynamics we have:

$$\frac{d}{dt}(mu) = m\frac{du}{dt} + u\frac{dm}{dt} = \sum_{sf}\frac{dQ_{sf}}{dt} - P\frac{dV}{dt} + \sum_{j} h_{oj}\frac{dm_j}{dt} \qquad \text{(A.72)}$$

where m is the mass in combustion, u is the specific internal energy, sf denotes the surfaces with different rates of heat transfer, dQ is the heat released by combustion, P is the pressure of the gas, V is the volume of the gas, h_{oj} is the specific stagnation enthalpy of mass entering or leaving the system.

If we say that the specific internal energy u is only a function of the temperature T and the equivalence ratio F, then:

$$u = u(T, F) \Rightarrow m\frac{du}{dt} = m\left[\frac{\partial u}{\partial T}\frac{dT}{dt} + \frac{\partial u}{\partial F}\frac{dF}{dt}\right] \quad \textbf{(A.73)}$$

By substitution of Eq. (A.73) into Eq. (A.72) and assuming that the gases behave as perfect gases ($PV = mRT$), we obtain:

$$\frac{dT}{dt} = \left[-\frac{RT}{V}\frac{dV}{dt} + \left(\sum_{sf}\frac{dQ_{sf}}{dt} + \sum_{j} h_{oj}\frac{dm_j}{dt} - u\frac{dm}{dt}\right)\frac{1}{m} - \frac{\partial u}{\partial F}\frac{dF}{dt}\right] \Big/ \left(\frac{\partial u}{\partial T}\right) \quad \textbf{(A.74)}$$

We can apply Eq. (A.74) to the manifolds and the cylinders.

By mass conservation we have that:

$$\frac{dm}{dt} = \sum\left[\frac{dm}{dt}\right]_{in} - \sum\left[\frac{dm}{dt}\right]_{out} \quad \textbf{(A.75)}$$

The fuel-air equivalence ratio is defined by:

$$F = f/f_s \quad \textbf{(A.76)}$$

where f is the fuel air ratio and suffix s denotes stoichiometric[1]. The mass of burned fuel (m_{fb}) in a total mass (m) of air and burned is defined by:

$$m_{fb} = \frac{mf_sF}{(1 + f_sF)} \quad \textbf{(A.77)}$$

From Eq. (A.75), Eq. (A.76) and Eq. (A.77), we obtain the term dF/dt from Eq. (A.74):

1. Stoichiometric: "pertaining to or involving substances that are in the exact proportions required for a given reaction" (19).

$$\frac{dF}{dt} = \left[\frac{1+f_sF}{m}\right]\left[\frac{(1+f_sF)}{f_s}\cdot\frac{dm_{fb}}{dt} - F\frac{dm}{dt}\right] \quad \textbf{(A.78)}$$

knowing T, m and V we can find the end of step pressure.

The change in volume, in the case of cylinders, is obtained from the geometry of the piston, crank and connecting rod:

$$\frac{dV}{dt} = \frac{\pi d^2}{4}\left[r\sin\theta\frac{d\theta}{dt} + \frac{r^2\sin\theta\cos\theta}{\sqrt{(l^2 - r^2\sin^2\theta)}}\cdot\frac{d\theta}{dt}\right] \quad \textbf{(A.79)}$$

The engine losses and friction are modeled with the relation:

$$FMEP = 13.79 + 0.005P_{max} + 1.086N\cdot CR \quad \textbf{(A.80)}$$

where: $FMEP$ is the mean effective pressure equivalent of engine losses (kN/m^2), P_{max} is the maximum cylinder pressure (kN/m^2), N is the engine speed and CR is the crank radius.

A diagram of the turbocharged diesel engine is shown in Figure A.5. This diagram includes the interaction of the turbocharger and the engine $_{(21)}$.

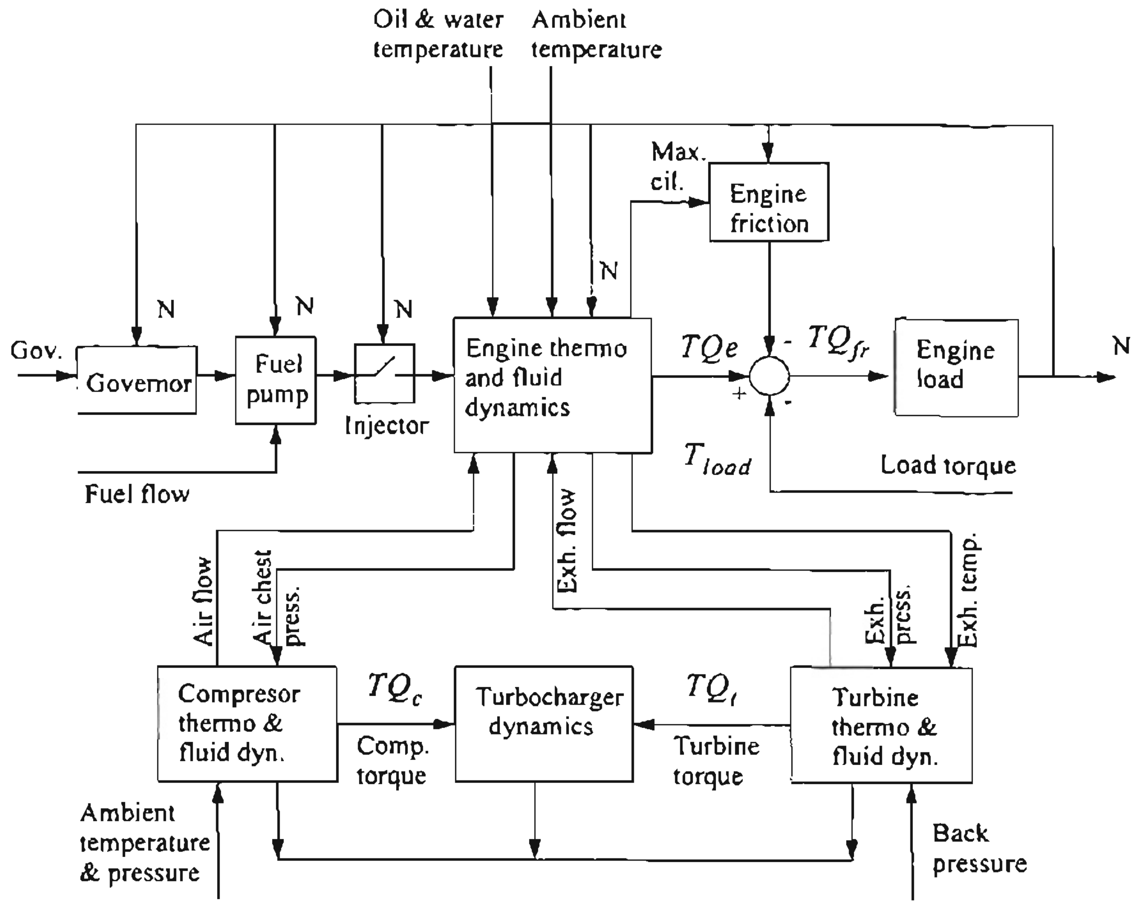


Figure A.5: *Block diagram for a turbocharged diesel engine system* (21).

A.4. Tuken's model.

Tuken et. al. (18) proposed a different model for the experiment shown in Figure A.6 with the block representation in Figure A.7. The electromechanical actuator is described by a third non-linear dynamic model that Tuken et. al. approximated by a linear third order dynamic model plus a time delay. The governor has a mechanical part and a hydraulic part. Then we have mathematical models for each part:

(a) Mechanical part:

$$m\omega^2 r = m_e \frac{d^2x}{dt^2} + \beta\frac{dx}{dt} + Kx + F_L + PA \qquad \textbf{(A.81)}$$

$$F_L = K_L x_a \qquad \textbf{(A.82)}$$

$$P = K_T\Omega \qquad \textbf{(A.83)}$$

(b) Hydraulic part:

$$q = K_m x \qquad \textbf{(A.84)}$$

where:

m = Mass of flyweights

ω = Engine speed

r = Radious of flyweight from the axis of rotation

m_e = Total effective mass referred to axis

β = Viscous friction coefficient of moving parts

K = Spring stiffness

F_L = Load force due to thorttle rack

G_e = Engine torque.

P = Output pressure of the transfer pump

A = Metering-valve piston area

x_a = Throttle position

x = Metered valve position

q = fuel rate

K_m, K_T, K_L = Constants

The engine combustion model is based on the sequential firing of the cylinders, operating in a discontinuous manner. This introduces a delay that is equal to the "actual time between consecutive pistons arriving at the injection point plus a quarter of revolution of the crankshaft (18):

$$T_F = \frac{60h}{2e\omega} + \frac{60}{4\omega} \qquad \text{(A.85)}$$

where:

h = 4 (Number of strokes per cycle)

ω = Speed in rev/min

e = Number of cylinders

T_F = Firing delay (seconds)

The transfer function for the engine combustion is:

$$\frac{G_e(s)}{q(s)} = K_e e^{-T_F s} \qquad \text{(A.86)}$$

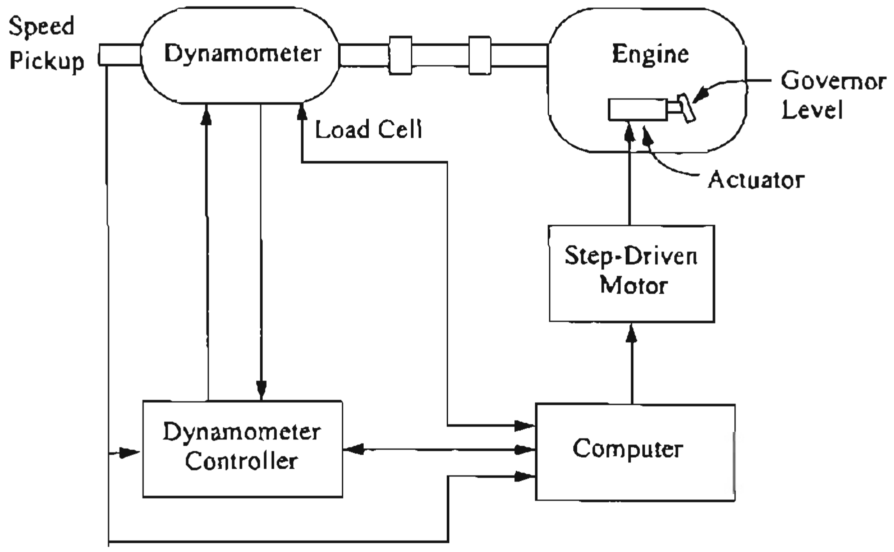


Figure A.6: *Experimental apparatus for Tuken et. al. (18) experiment.*

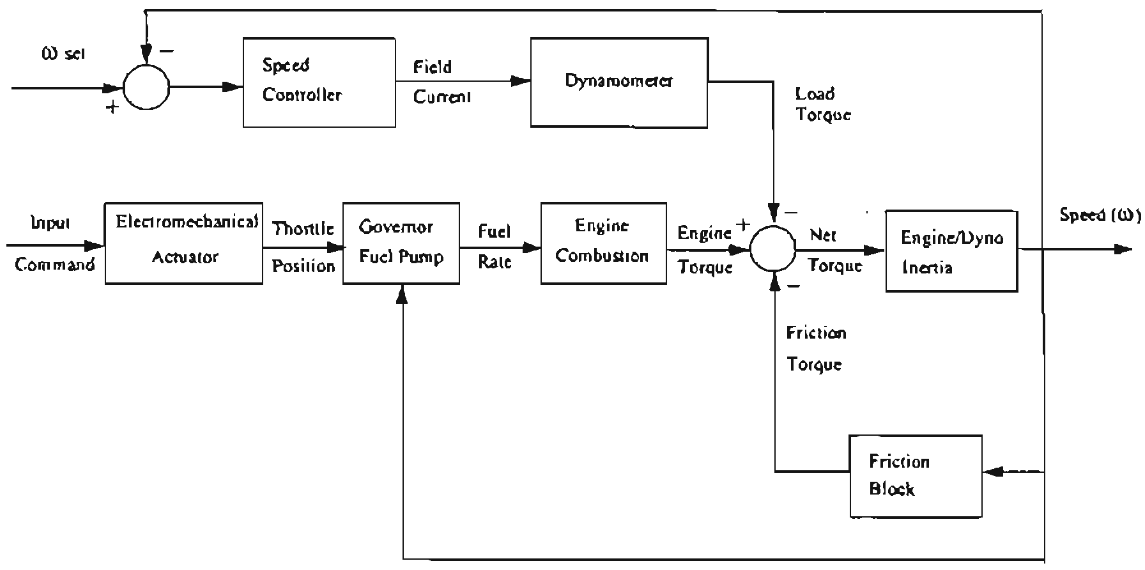


Figure A.7: *Block diagram representation of throttle-torque system (18).*

A.5. Models evaluation.

The models shown in the previous sections have detailed information about the thermodynamics of the process. The experiments that could be done in the main sections will be related with speed control. An important equation for our simulations is Eq. (A.86) where Tuken et. al. $_{(18)}$ defined a relation between the fueling injected to the engine and the torque produced. From the Kao and Moskwa $_{(7)}$ model we will extract the block structure information to construct a neural network model based on the air flow, the fueling and the engine speed. We will consider the friction as a factor in the engine operations with a modification of Eq. (A.21).

VITA

Orlando De Jesús

Candidate for the Degree of

Master of Science

Thesis: REINFORCEMENT LEARNING CONTROL APPLIED TO DIESEL ENGINES

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Caracas, Venezuela, on October 3, 1963, the son of Manuel De Jesús and Lucía N. De Abreu.

Education: Graduated from Creación Guarenas High School, Guarenas, Edo. Miranda, Venezuela, in July 1980; received degrees of Engineer in Electronics (Cum Laude) and Project Management Specialist from Universidad Simón Bolívar, Caracas, Venezuela, in July 1985 and July 1992, respectively. Completed the requirements for the Master of Engineering degree in Electrical and Computer Engineering at Oklahoma State University in December 1998.

Experience: Employed by AETI C.A., Caracas, Venezuela, as a Research Engineer, R&D Manager and Operations Manager from 1985 to 1996; employed by Oklahoma State University as a Research Assistant from 1997 to present.