

PENSE - oPEN Simulation Environment

Engin AYDOGAN Berk DELEVI

2005-06-13

Contents

List of Tables	iv
List of Figures	v
1 Introduction: PENSE 101	1
1.1 What does PENSE mean?	1
1.2 What is it?	1
1.3 How does it work ?	1
1.3.1 Environments	2
1.3.2 Devices	2
1.3.3 Algorithms	3
1.4 Summary	4
2 Device Types	5
2.1 Sources	5
2.2 Plants	5
2.3 Controllers	6
3 Algorithms	7
3.1 Polynomial Based	7
3.2 Fuzzy Logic	8
3.2.1 What is it?	9
3.2.2 How does it work?	9
4 Plants	14
4.1 DC Motor	14
5 Controllers	19
5.1 Fuzzy Logic Controler	19

6	Sources	22
6.1	Voltage Source	22
6.2	PWM	22
7	Example Programs	26
7.1	Fuzzy Logic with PWM controlling a DC Motor	26
7.2	DC Motor	29
7.3	PWM Controller	29
7.4	PWM Motor Control	29
7.5	Fuzzy Logic	29
7.6	Fuzzy Logic Controller	29

List of Tables

List of Figures

1.1	PENSE Object Model	2
1.2	Device Model	3
1.3	A More Advanced Device Model	3
3.1	Membership degree for <i>Normal</i>	10
3.2	Representation of the Fuzzy Logic	13
4.1	Motor velocity-time graph	17
4.2	Motor acceleration-time graph	17
4.3	Motor current-time graph	18
5.1	Speed control with Fuzzy Logic Controller	20
5.2	Fuzzy Logic Controller doesn't overshoot the set point and it does not oscillate.	21
6.1	Pulse Width Modulation	23
6.2	500Hz PWM controlling the motor with 50% duty cycle	24
6.3	Fig. 6.2 Close up	24
6.4	10KHz PWM controlling the motor with 50% duty cycle	24
6.5	Fig. 6.4 Close up	25
7.1	Example System	27
7.2	Fuzzy Logic with PWM unit controlling Maxon 118465	27
7.3	Fig. 7.2 Close Up	28

Abstract

PENSE is a *simulation framework* written in C++ using fully object oriented design patterns and it's designed to be flexible, extendable and portable within itself.

Basically, user connects devices together to realize a system. One can use existing devices in the *libpense* such as Fuzzy Logic Controller, DC Motor, PWM unit and voltage source or create custom devices.

Chapter 1

Introduction: PENSE 101

1.1 What does PENSE mean?

PENSE stands for oPEN Simulation Environment. The word *pense* in Turkish means pliers. Also in French *je pense* means *I think*, so seeing as it's a multi-meaning abbreviation, we agreed that it's appropriate for the job.

1.2 What is it?

PENSE is a *simulation framework* written in C++ that uses fully object oriented design patterns and it's designed to be flexible, extendable and portable within itself. It is supposed to provide an easy to use clean API for programmer to build a system and simulate it.

Basically, it's about connecting built-in or custom devices together and see how do they react by simulating the whole system of devices.

1.3 How does it work ?

We better explain the system from the largest object to the smallest one. *Devices* can be grouped in *environments* to realize a system, which is usually a feedback control system. For instance see Figure 1.1

In Figure 1.1 there is an *Environment* object, which is a container object which contains virtually unlimited devices. Those devices are connected together to form a system. So, let's define what is an *environment* and *device*.

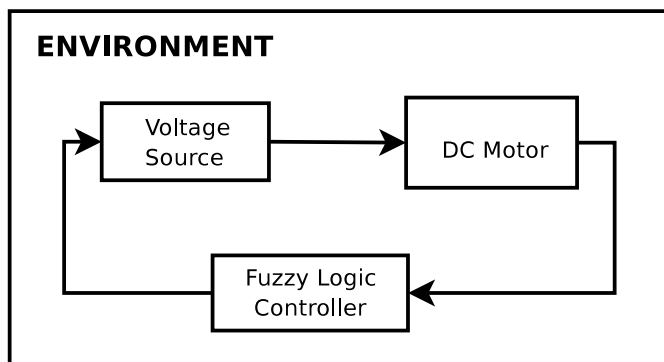


Figure 1.1: PENSE Object Model

1.3.1 Environments

An environment definitely has the parameter frequency. The frequency parameter defaults to 1000000 (million), which means each iteration on the devices in this environment will mean $\frac{1}{1000000}sec$. Of course, API user can change frequency when initializing the environment. Devices in an environment can access environment frequency with parameter f , this parameter is available in all devices and updated when the device is added to the environment. So if one uses parameter f in an algorithm, one should realise it only means frequency of the environment.

Additionally, there could be other parameters which should be shared by all devices in one particular environment, such as; temperature, humidity. Environment object is designed to overcome this common parameters problem. Through the environment object, programmers can assign any value to any parameter on all devices in that particular environment easily.

1.3.2 Devices

A device object is any object derived from the base device model and it is probably the most essential object of the whole framework. Devices contain virtually unlimited *algorithms*, so we can say that devices are containers for the algorithms. When an algorithm is added, the device automatically creates input and output nodes according to the algorithm provided. For instance, as seen in Figure 1.2. In Figure 1.2 algorithm $(\sin(x)\cos(y))z$ is added. Obviously, this algorithm has three parameters; x, y and z. So, three input nodes are created on the device so that other devices' output nodes are connected to these input nodes and values of these parameters are updated.

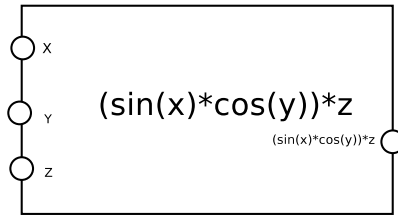


Figure 1.2: Device Model

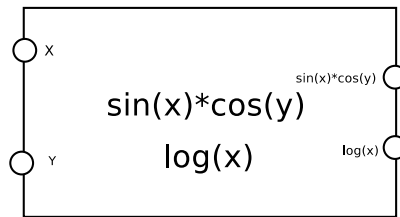


Figure 1.3: A More Advanced Device Model

For each algorithm on the device, there should be one output node so that the result of the algorithm can be sent to other objects. We can say that *output nodes* emit result and *input nodes* omit them. So devices update each others' values.

Devices can contain more than one algorithm, as programmers add algorithms to the device, it will create input and output nodes for each algorithm. Since two different algorithms can contain the same parameter, it's possible for a device to have more than one input node with same name. For instance assume you have added $\sin(x)\cos(y)$ and $\log(x)$ to your device, the device will have two x input nodes and one y input node. On the other hand, device will have exactly the one output node corresponding to each algorithm, in this case our device will have two output nodes. But keep in mind that even though the devices internally create two separate input nodes with same name (x) for these algorithms individually, the input nodes will be represented by one node on the device. This example is illustrated in Figure 1.3

1.3.3 Algorithms

An algorithm is any object derived from base algorithm model. Since all algorithms are supposed to take some inputs, evaluate them and generate an output, this model abstracts these common properties and gives a common interface.

A *Polynomial* object is a very good demonstration of using an algorithm model as base. A polynomial object is basically a wrapper of *GNU libmath-eval* library which can parse equations into tree format and evaluate them. Actually the name Polynomial is misleading, this object can evaluate many more various equations. Polynomial objects can handle logarithmic, arithmetic, polynomial, trigonometric and exponential equations pretty well, that is to say, $(\frac{\sin(x)}{\tan(y)})^{\log(z)}$ is perfectly representable in this object. Of course in a computer friendly manner like; $(\sin(x)/\tan(y))^{\log(x)}$.

Once the *Polynomial* object is created with such an equation, it extracts the parameters of the equation so that programmer can assign values to these parameters and evaluate the equations upon them.

1.4 Summary

In a nutshell, in *PENSE* you can have many *algorithms* wrapped in a *device*, and many devices wrapped in an *environment* sharing some common properties. You can *connect* these devices together and evaluate the system by iterations.

One should realise the fact that *PENSE* itself is not an executable program, but a library for programmers to use. It's designed as a backend for simulation programs. Hence, building a GUI using PENSE is perfectly possible, it's a challenging task to do it properly though. Of course, for convenience some executable demonstration programs for console are provided with PENSE.

Chapter 2

Device Types

PENSE framework provides basic predefined device models which user can use directly. Those models fall into three categories; *Sources*, *Plants* and *Controllers*.

2.1 Sources

All source should be derived from *PENSE::Device::Source* class. This class abstracts common features of source devices and is itself derived from base device class *PENSE::Device::Device*. These features are as follows;

- being able to be turned ON/OFF
- increase/decrease current value

Basically, one would expect to be able to turn a source ON/OFF and tune the output value by increasing/decreasing the current value. So above interface guarantees that any sources derived from base source class will provide those capabilities of sources. As an example to this we have a DC voltage source and PWM unit which you can increase/decrease duty cycles. More information about this in the next chapter.

2.2 Plants

Plants are any device which is supposed to convert some form of energy to another. Like a electric motor which is supposed to convert electric energy to mechanical energy. Many different devices can be derived from base device class *PENSE::Device::Device* to model a device in simulation framework.

Many kind of devices can be implemented by making use of the framework. As an example a DC motor model is provided with *libpense*.

2.3 Controllers

This is the third and the last device type of the framework. All controllers in *libpense* should be derived from *PENSE::Device::Controller*. This base class, as usual, abstract common features of controllers. All controllers must implement methods for setting a *set point* and retrieving it.

Obviously, one would need to set a set point for the controller so that one can control an actual device according to its output. A typical scenario is to connect a *plant's* output to a *controller's* input and connect *controller's* output to the *source* of the plant.

Chapter 3

Algorithms

PENSE framework suggests a common interface for algorithms, so that *very* different algorithms can be developed and treated in a similar same way. It is assumed that all algorithms have three common properties. They all;

- take inputs for each of the parameters.
- evaluate the algorithm for the inputs.
- and give an output.

Two algorithm implementation which satisfies the above conditions are provided with *libpense*.

3.1 Polynomial Based

The name *Polynomial* is actually misleading. *PENSE::Algorithm::Polynomial* class is able to handle logarithmic, arithmetic, polynomial, trigonometric and exponential functions just fine. Keep in mind that *Polynomial* itself is not an actual *algorithm* but a tool to build algorithms.

Basically, you provide a regular equation string to *Polynomial* by using computer notations, then *Polynomial* parses the string, extracts the parameters and get ready for actual work. For instance;

$$a = \frac{k_n(V - RI_o) - \omega}{Rk_n^2(J_r + J_l)} \quad (3.1)$$

should be formatted like;

$$(k_n*(V-R*I_o)-w)/(R*k_n^2*(J_r+J_l))$$

Let's study the given equation Eq. 3.1. Once the *Polynomial* is initialized successfully (i.e. a valid equation is provided) the object will accept values for each of its parameters, which are;

- k_n representing k_n
- V representing V
- R representing R
- I_o representing I_o
- w representing ω
- J_r representing J_r
- J_l representing J_l

An example code snippet is provided below;

```

1 Algorithm::Polynomial p( "(k_n*(V-R*I_o)-w)/(R*k_n^2*(J_r+J_l))" );
2 p["k_n"] = 252.375;
3 p["V"] = 4.8;
4 p["R"] = 2.16;
5 p["I_o"] = 0.029;
6 p["w"] = 0;
7 p["J_r"] = 0.0000000503;
8 p["J_l"] = 0;
9 cout << "Result:" << p.evaluate() << endl;

```

On L1 (line 1) the *Polynomial* object p is created. On the following lines 2-8 ,value assignments occur. And on L9 the polynomial is evaluated and the result is printed on the standart output.

3.2 Fuzzy Logic

Fuzzy logic algorithms are relatively a new concept with respect to PID and such traditional controllers. In *libpense* a so-called type-I fuzzy logic algorithm is implemented; *PENSE::Algorithm::FuzzyLogic*. This algorithm is used in the built-in Fuzzy Logic Controller *PENSE::Device::Controller::FuzzyLogic*.

Even though the Fuzzy Logic concept itself was our actual project, *PENSE* evolved pretty quickly. So quickly that now Fuzzy Logic is only a simple part of the project.

3.2.1 What is it?

In contrast to other algorithms, fuzzy logic's principle is to think like an organic creature; human. In traditional algorithms, engineers always tried to find ways of describing real world problems to computers, to achieve that mathematical equations are developed to model real world problems. In complex problems development of these solutions using these traditional techniques might be time consuming, hence expensive. Fuzzy logic tries to describe the problem to the computer in a more human-like fashion.

3.2.2 How does it work?

Unlike other approaches, in fuzzy logic we define a human readable rules to form the target system. For instance assume we want to control the room temperature, first of all we define simple rules;

- If the room is *hot* then *cool it down*
- If the room is *normal* then *don't change temperature much*
- If the room is *cold* then *heat it up*

Obviously the rules makes sense and easy to understand but there is one simple problem; the computer doesn't have the concepts of *hot*, *normal* or *cold* so we should tell the computer about those words. A typical definition is as follows;

- The room is *hot* if the temperature is between 20°C and 30°C
- The room is *normal* if the temperature is between 18°C and 22°C
- The room is *cold* if the temperature is between 10°C and 20°C

Now that the computer knows of what is *hot*, *normal* and *cold* it can classify an input according to those definitions. As you see definitions' boundaries intersect with each other, that's to say some temperature values are members of two groups. This is essential to the fuzzy logic paradigm and what differs fuzzy logic from other algorithms. Let us evaluate a few samples;

- 20.5°C: This temperature value is member of both *normal* and *hot*.
- 20°C: This temperature value is member of only *normal*.
- 15°C: This temperature value is member of only *cold*.

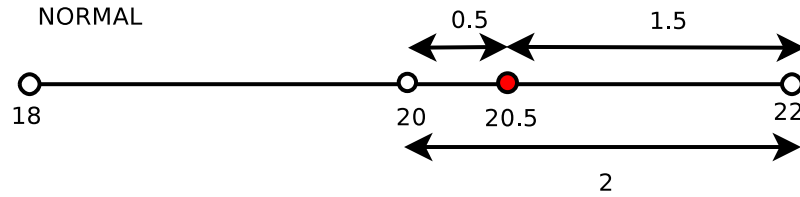


Figure 3.1: Membership degree for *Normal*

Now computer knows how an actual measured input (room temperature) can be classified as but there is a slight problem again. We saw that some inputs belong to more than one category, just like 20.5°C did. If we don't specify a way to understand which group does it belong more or less all the values between 20 and 22 will be the same since they all will belong to both category. Let us evaluate the 20.5°C example again.

As illustrated by the Fig. 3.1 20.5°C is pretty close to *normal*'s center but it's also a member of *hot*. Membership degree is calculated by formula;

$$membership\ degree = \frac{distance\ to\ one\ edge}{\frac{totalwidth}{2}} \quad (3.2)$$

In this particular case we calculate the membership degree of 20.5°C's for *normal* as follows;

$$membership\ degree\ of\ 20.5\ for\ normal = \frac{1.5}{\frac{2}{2}} = \frac{75}{100} = 0.75 \quad (3.3)$$

We know that 20.5°C is also a member of *hot* so we calculate the membership degree also for *hot* in a similar fashion.

$$membership\ degree\ of\ 20.5\ for\ hot = \frac{0.5}{\frac{2}{2}} = \frac{10}{100} = 0.10 \quad (3.4)$$

So, we can, obviously, say that 20.5°C is almost a *normal* temperature but it's just a little bit *hot* for the room.

Now that we have interpreted the measured input value, what's next ? We should give an appropriate reaction to keep the room in target temperature, that's the center of the *normal* which is 20°C. The process we've done so far is called *fuzzifying*. We have *fuzzified* the input value, like; it's almost *normal* but it's a little bit *hot*. Now we should get a *crisp* output value as a reaction, to achieve this we begin a process called *defuzzifying*.

We have defined our input value categories *hot*, *normal* and *cold*. Now we'll define our output value categories which are *calm down*, *don't change much* and *heat it up* because computers, obviously, don't have concepts of calming down or heating up at all. There are a few approaches that we can use here. We can either use hardcoded values for outputs to directly use them on heater/cooler or we output just an adjustment value so that we can tune the heater/cooler according to that.

First choice requires you to enter a valid output range, say, output between 110-220V. Assuming we have a room in an *hot* environment, and we use a cooler (A/C) to control the room temperature, i.e. turning the cooler off will result in temperature increase in the room. In this particular case, if the room is *hot* we should throttle the cooler with full power which is 220V. If the room is *cold* we should throttle the cooler with the lowest possible value which is 110V in this case, and so on.

The other approach is based on tuning the cooler by increasing or decreasing the voltage provided to it. Say, if the room is *hot* then increase the voltage provided to the cooler. If the room is *cold* decrease the voltage provided to the cooler and so on. This approach is straight forward but it has a drawback, this causes oscillation around the set point, so we don't prefer using this approach.

So assuming we're using first approach we should define an output range, in this example we choose the range between 110-220V. Since we have classified the input into three categories *hot*, *normal* and *cold*, we should also define a specific output range for each condition. In this particular case we divide the output range 110-220V into three parts. See Fig. 3.2.

Defuzzifying process is simply finding the center of mass of the outputs according to the fuzzified values and the output ranges. Our basic formula for calculating the weight is as follows:

$$Weigh = Fuzzified\ Value \times Width \tag{3.5}$$

The fuzzified value here refers to the values we've found in Eq. 3.3 and Eq. 3.4. The widths are basically the width of a particular part in the output range. In this case we have three different output parts for each *hot*, *normal* and *cold* we named these parts *low power*, *normal power* and *high power* respectively. Each part has it's width, which can be calculated by simple

extracting the boundaries. For instance for *low power* the width is 110. Now let us calculate the weight of each fuzzified values.

$$\textit{Weight of normal power} = 0.75 \times 50 = 37.5 \quad (3.6)$$

$$\textit{Weight of low power} = 0.10 \times 110 = 11 \quad (3.7)$$

Now that we've found the weights, we should find the center of it. You can think of this center as the equilibrium center. A point which can hold those weight in equilibrium, just like a lever. We can found the center of mass of two weight as follows;

$$\textit{Center of Mass from } M_a = \frac{\textit{distance}}{M_a + M_b} M_b \quad (3.8)$$

So let's find the center of mass for this particular problem.

$$\textit{Center of Mass from normal power} = \frac{220 - 165}{37.5 + 11} 11 = 12.47 \quad (3.9)$$

So the the defuzzified output value must be:

$$\textit{Defuzzified output} = 165 + 12.47 = 177.47 \quad (3.10)$$

As you see we're giving just a little bit more power than normal because the room temperature is just a little bit higher than our set point. For a visual representation of this algorithm please see Fig. 3.2

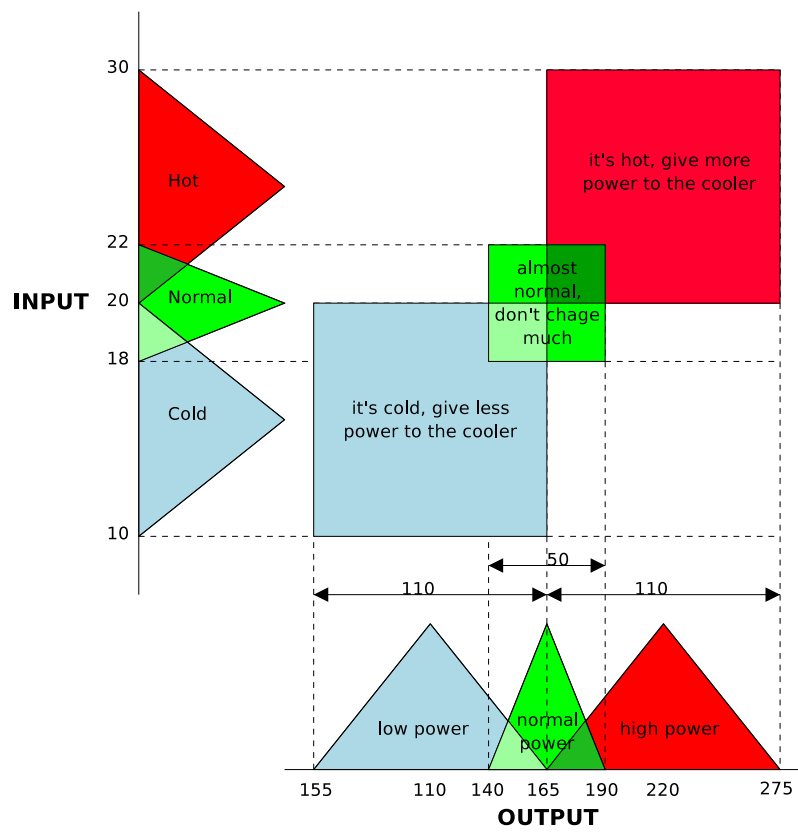


Figure 3.2: Representation of the Fuzzy Logic

Chapter 4

Plants

Plants are any devices which drains energy and transform it into another form of energy. A DC motor modell is provided with *libpense*.

4.1 DC Motor

DC motors transform electrical energy into mechanical energy. The basic power balance formula is as follows:

$$P_{el} = P_{mech} + P_j \quad (4.1)$$

P_{el} is obviouslt the electrical energy provided to the motor. P_{mech} is the mechanical power the motor produces and P_j is the power loss in resistance. Using energy formulas we get the following;

$$VI = \omega M + RI^2 \quad (4.2)$$

V is voltage provided, I is the current begin drained, M is torque, R is the terminal resistance, and ω is the angular velocity. We can expand torque a little bit more, total torque the motor should be dealing with is as follows;

$$M_t = J_r a + J_l a + M_r \quad (4.3)$$

Here J_r is the inertia of the rotor, J_l is the inertia of the load, a is the acceleration $\frac{d\omega}{dt}$ and finally M_r is the torque spent on friction. So substituting M in Eq. 4.2 with Eq. 4.3 gives us;

$$VI = \omega(J_r a + J_l a + M_r) + RI^2 \quad (4.4)$$

Here the current I can be calculated as;

$$I = \frac{V - V_{emf}}{R} \quad (4.5)$$

V is the voltage applied to the motor, V_{emf} is the voltage generated in the motor, sometimes also referred as back EMF and R is the terminal resistance. V_{emf} can be calculated as;

$$V_{emf} = \frac{\omega}{k_n} \quad (4.6)$$

k_n is the speed constant. So Eq. 4.5 can be written as;

$$I = \frac{V k_n - \omega}{R k_n} \quad (4.7)$$

In motor datasheets there's a data called *no load current*, this is the current drained when there is no load on the motor. Which means the current needed to keep the rotor rotating when there's no load. This current is necessary since there's always a friction in the motor. We can calculate the torque spent on friction like this;

$$M_r = k_m I_o \quad (4.8)$$

Here, k_m is the torque constant of the motor, and I_o is the no load current.

Now, let's try to obtain the acceleration formula. First using Eq. 4.4 we can write the following:

$$a = \frac{\frac{VI - RI^2}{\omega} - M_r}{J_r + J_l} \quad (4.9)$$

Then we can write;

$$a = \frac{I(V - RI) - M_r \omega}{\omega(J_r + J_l)} \quad (4.10)$$

We can use Eq. 4.7 in Eq. 4.10 and write the following;

$$a = \frac{\left(\frac{V k_n - \omega}{R k_n}\right)(V - \frac{V k_n - \omega}{k_n}) - M_r \omega}{\omega(J_r + J_l)} \quad (4.11)$$

We can simplify Eq. 4.11 as;

$$a = \frac{\left(\frac{V k_n - \omega}{R k_n}\right)\left(\frac{\omega}{k_n}\right) - M_r \omega}{\omega(J_r + J_l)} \quad (4.12)$$

Let's simplify one step further;

$$a = \frac{\omega(V k_n - \omega - M_r R k_n^2)}{\omega R k_n^2 (J_r + J_l)} \quad (4.13)$$

Obviously ω s will cancel. Now let's substitute M_r with Eq. 4.8.

$$\frac{V k_n - \omega - k_m I_o k_n^2 R}{R k_n^2 (J_r + J_l)} \quad (4.14)$$

k_n and k_m constants are not independent of each other. They should satisfy the following condition;

$$k_m k_n = 1 \quad (4.15)$$

Given above condition, we finally get this;

$$a = \frac{k_n(V - I_o R) - \omega}{R k_n^2 (J_r + J_l)} \quad (4.16)$$

Now that we get $\frac{d\omega}{dt}$, we can also get ω ;

$$\omega = k_n(V - R I_o) - e^{\frac{-t}{R k_n^2 (J_r + J_l)}} \quad (4.17)$$

Even though this model has it's errors this is a fair mathematical DC motor model. With this model, motor goes to it's final angular velocity in a reasonable time which is pretty close to the one given in datasheets. In datasheets a constant called *mechanical time constant* is given, this is the time to be elapsed until the motor reaches 63% of it's final angular velocity.

Fig. 4.1, 4.2 and 4.3 plotted from simulation data of a real motor. Motor specifications are as follows;

- $k_n = 252.374609 \frac{rad}{secV}$
- $J_r = 0.503 \times 10^{-7} kgm^2$
- $J_l = 0.0 kgm^2$
- $I_o = 0.029A$
- *Nominal Voltage* = 4.8V
- $R = 2.16\Omega$

Nominal voltage 4.8 V is applied to the motor for 50ms, for more information motor's order number is 118465.

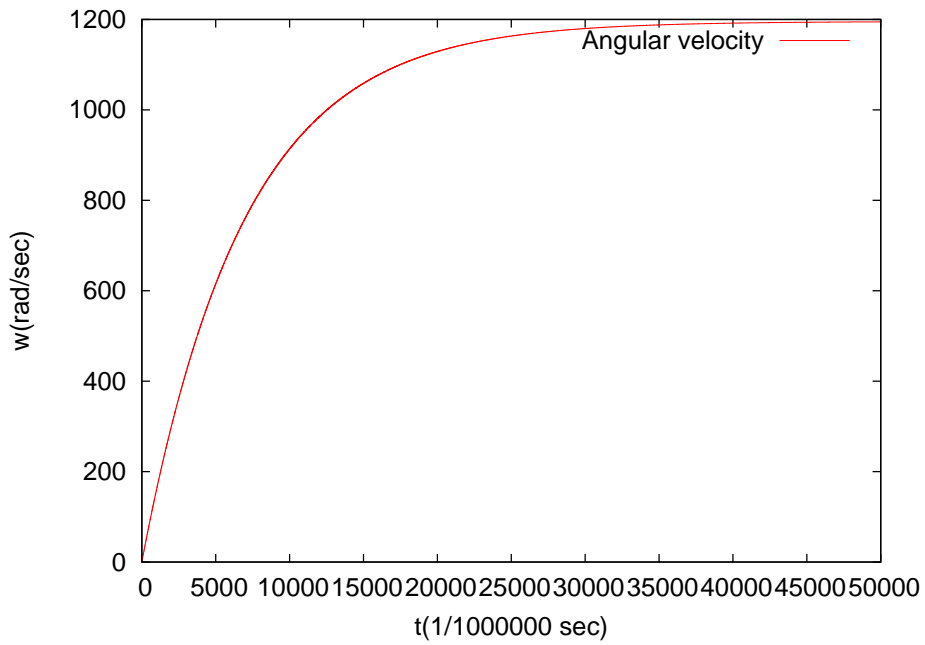


Figure 4.1: Motor velocity-time graph

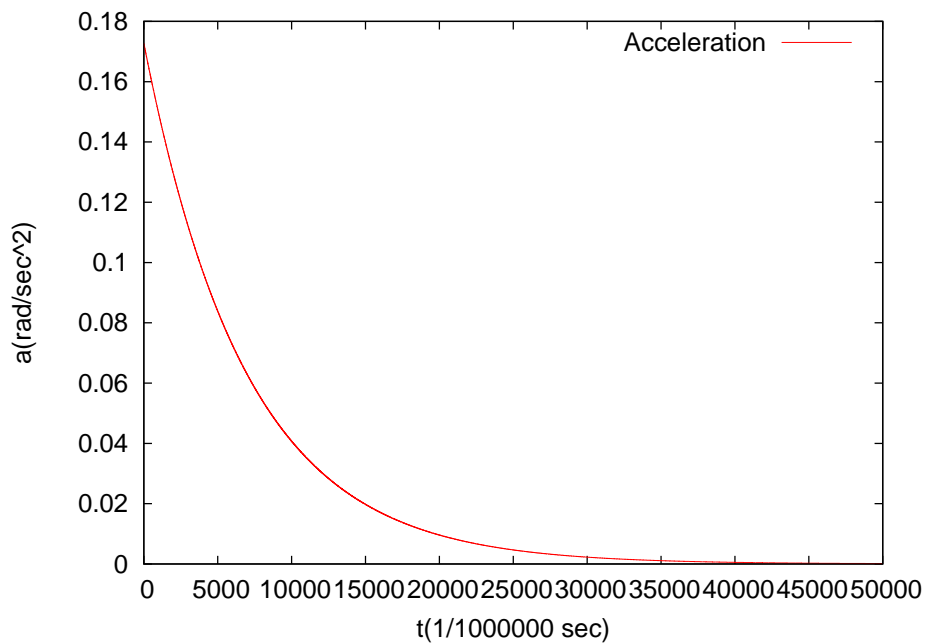


Figure 4.2: Motor acceleration-time graph

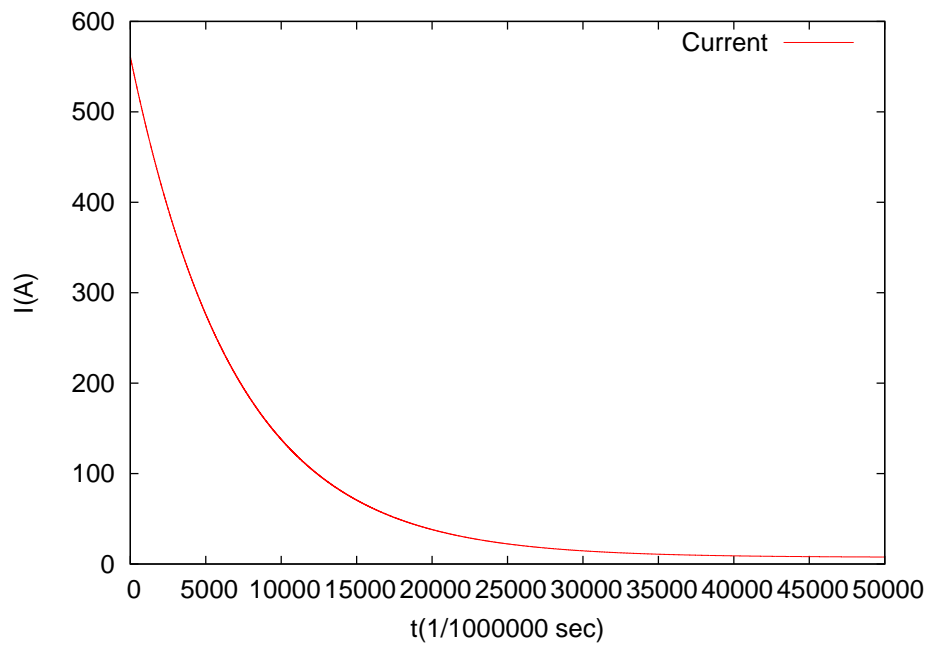


Figure 4.3: Motor current-time graph

Chapter 5

Controllers

PENSE suggest a common interface for controllers too which has one requirement. Each controller should have implement a way to set a *set point* and retrieve it. That is to say, you can set a set point for every controller. A fuzzy logic controller is provided with *libpense*.

5.1 Fuzzy Logic Controler

We know that there a Fuzzy Logic algorithm implemented in *libpense*. By making use of this algorithm this controller is implemented. One should give a set point and an output range for this controller to be inialized. For instance, one can tell that *set point* is 500 and output range is 0-12. This means according to the input value fuzzy logic controller will give an output between 0 and 4.8 to control the device.

We have tried to keep the motor mentioned in chapter 4 at 500 rad/sec. Fig. 5.1 shows the velocity-time graph of the motor when it is feeded with nominal voltage and when it's controlled by the fuzzy logic controller. The system in Fig.1.1 is realized here, so voltage regulation is used to keep the motor in set point. In real world applications DC motors' velocity is controlled by *pulse width modulation* which we'll see in next chapter.

Fig. 5.2 is a close up of Fig. 5.1 to show you how fuzzy logic reaches it's destination *set point* without any overshooting or any oscillations.

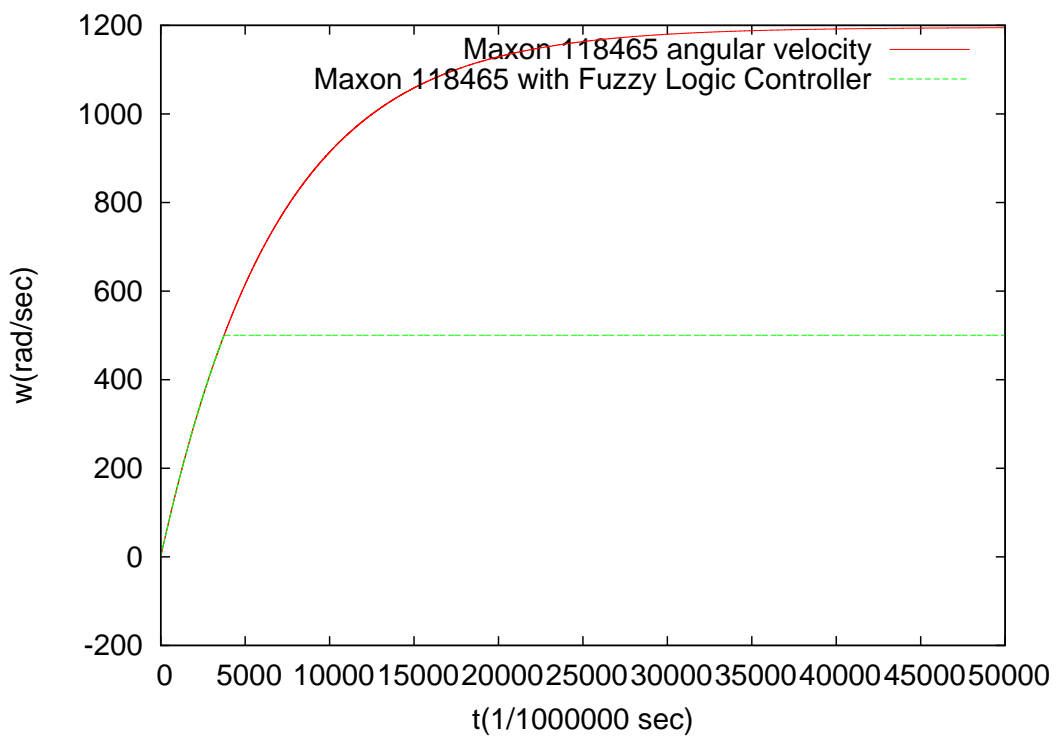


Figure 5.1: Speed control with Fuzzy Logic Controller

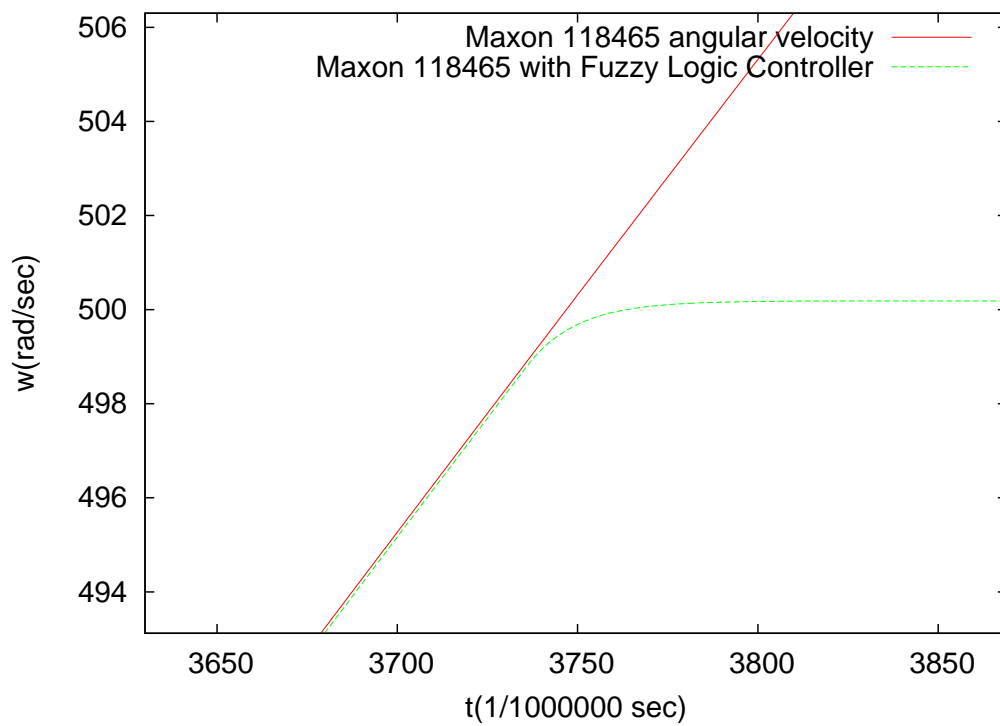


Figure 5.2: Fuzzy Logic Controller doesn't overshoot the set point and it does not oscillate.

Chapter 6

Sources

Sources are any device which provides energy to the plants and system. For further information please see Section 2.1.

6.1 Voltage Source

A simple *voltage source* is provided with *libpense*, which will output the voltage it's been told. Of course this device satisfies the requirements of a *source* which means it can be turned on or off and value of it can be increased or decreased.

When initialized the *voltage source* use can specify the range of this device, say, it can output voltage between 0 and 220V, according to this information the device will always normalize it's output value to be always in this range.

6.2 PWM

Pulse width modulation is used to regulate the voltage by changing the width of the pulse in a specific period of time. PWM unit has two parameters; frequency and duty cycle. The higher the frequency the smoother the output will be. For instance a 500Hz PWM with 60% duty cycles has a period of $\frac{1}{500}$ sec, and during one period 60% of the time it will give ON signal.

Actually *Pulse width modulation* device could be considered as a *controller* but due to it's nature it fits better as a *source* so that we can increase/decrease it's value.

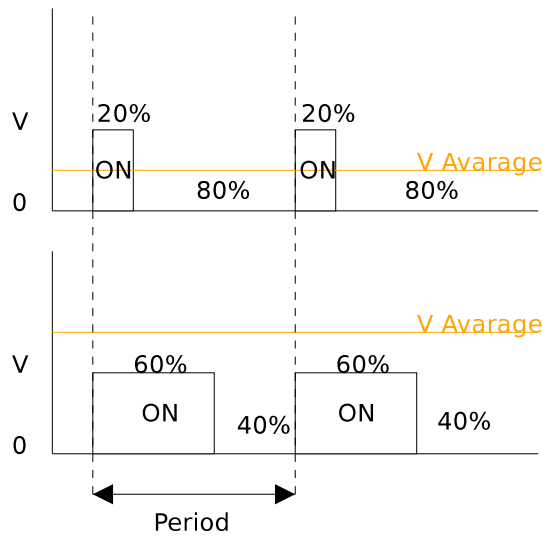


Figure 6.1: Pulse Width Modulation

PWM is usually used as a digital to analog converter. A typical scenario might be a microcontroller decides how much voltage to output and PWM let that amount of voltage to pass, then with a filter we can smooth the signal but this is completely out of scope of this document.

We use PWM because we want to control the motors velocity by not simply regulating the voltage, say, giving less voltage to slow the motor down. Instead we will give full power to the motor but only in a certain amount of time. This will let the motor has more torque, and this is how this is being done in industrial field.

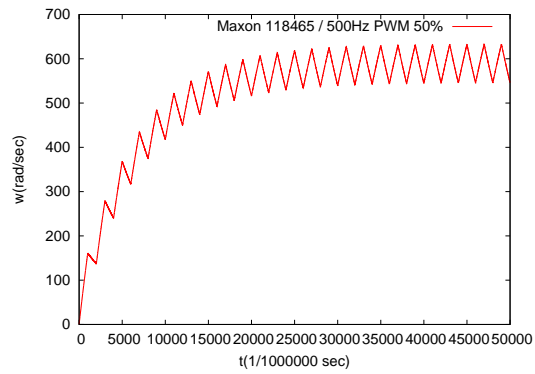


Figure 6.2: 500Hz PWM controlling the motor with 50% duty cycle

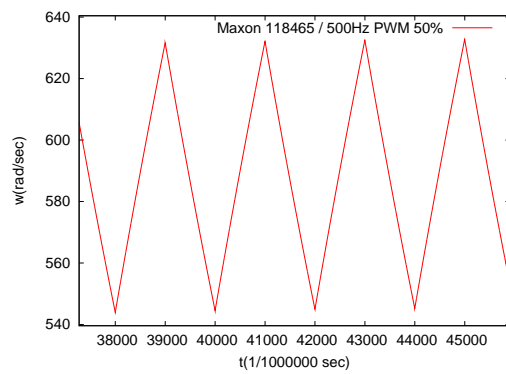


Figure 6.3: Fig. 6.2 Close up

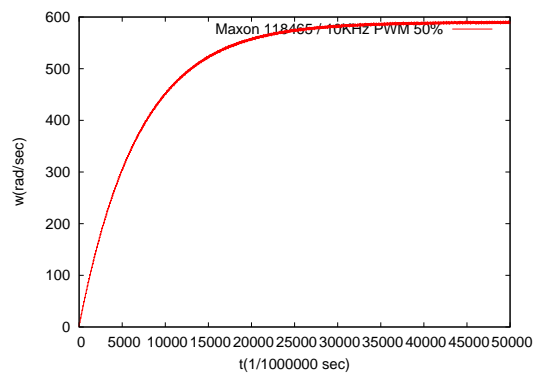


Figure 6.4: 10KHz PWM controlling the motor with 50% duty cycle

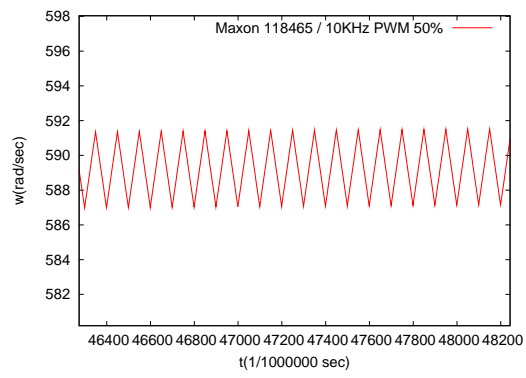


Figure 6.5: Fig. 6.4 Close up

Chapter 7

Example Programs

There is also a *pensedemo* package provided with *libpense* which is a demonstration of how to use *libpense* properly. There are several ready-to-run programs included in *pensedemo*, which are briefly explained in this chapter.

Each executable's name will be given in each section. Also additional statically linked binaries of those executables will be available in *bin* directory of the CD-ROM provided with this dissertation in case use doesn't have the environment to compile *libpense* and *pensedemo*.

7.1 Fuzzy Logic with PWM controlling a DC Motor

As a demonstration of *libpense*, we've constructed a system with built-in devices provided with the library. We've built a feedback control system which is represented in Fig. 7.1. Program executable's name is `fuzzy_pwm_motor_control`, run without any arguments to list available parameters.

In this demonstration we want to control the motors velocity, to achieve that we have a fuzzy logic controller which is continuously measuring the motor's velocity. According to the measured value, the fuzzy logic controller decides a duty cycle for PWM controller. Then PWM controller conducts the voltage provided by the voltage source to the motor as necessary to keep the motor in set point. We have choosed PWM frequency as 10KHz, voltage source as 4.8V. We're using the motor mentioned in previos chapters and our set point is $500rad/sec$. You can see the results in Fig. 7.2 and in Fig. 7.3.

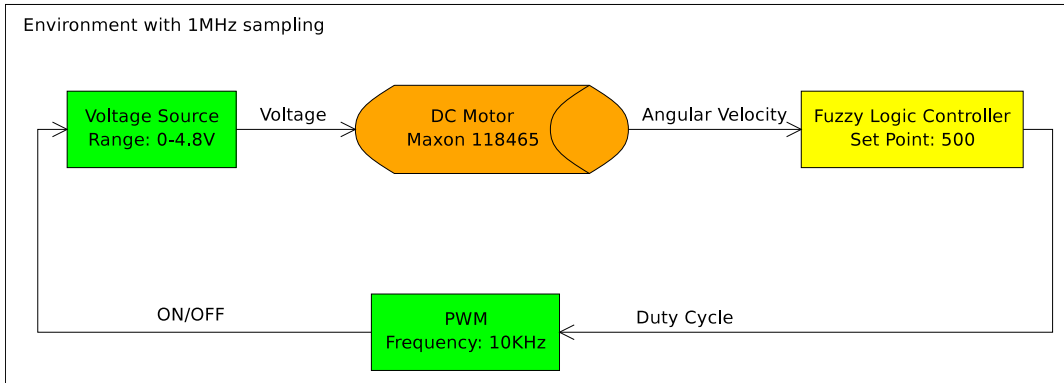


Figure 7.1: Example System

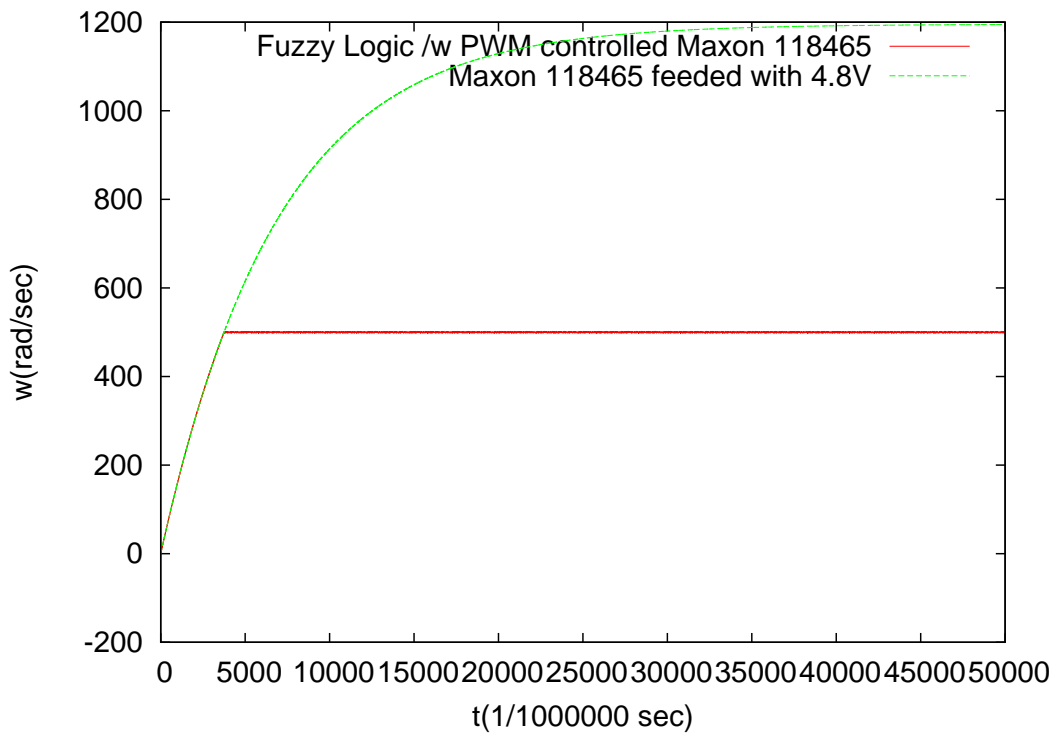


Figure 7.2: Fuzzy Logic with PWM unit controlling Maxon 118465

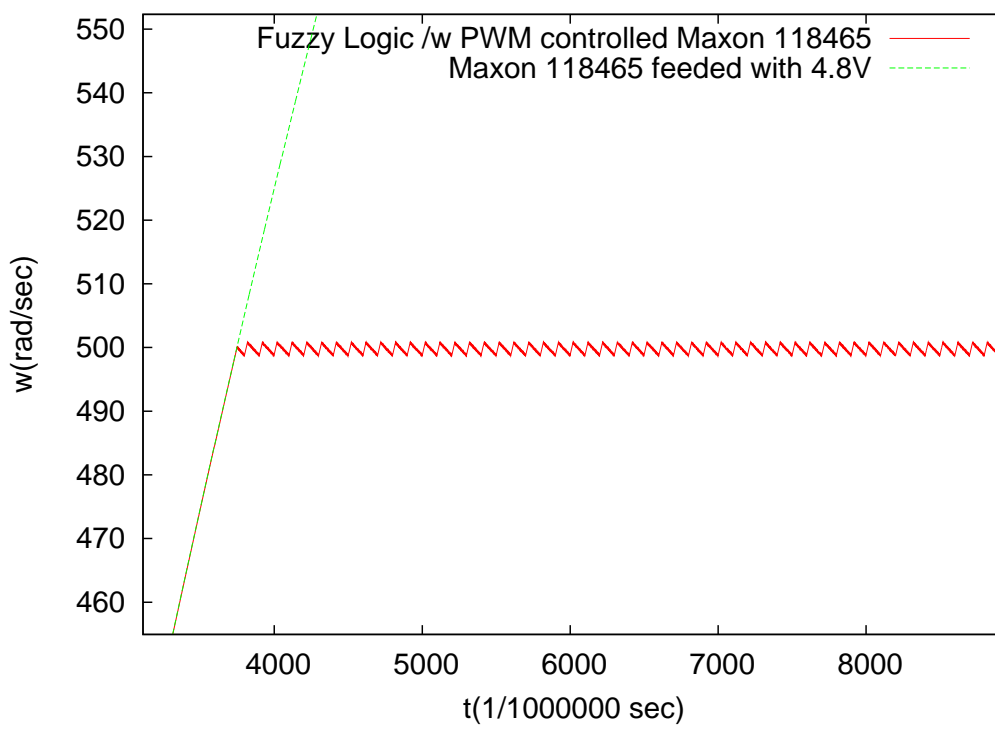


Figure 7.3: Fig. 7.2 Close Up

7.2 DC Motor

This program demonstrates the DC Motor simulation. One can use predefined motor specifications or enter custom values to simulate any DC motor from a given datasheet. Program can output a human readable verbose output or a data file which is ready to be used by *gnuplot*. Program also generates a *gnuplot* script which can plot velocity-time, acceleration-time and current-time graphs of the motor. The executable name is `dc_motor`.

7.3 PWM Controller

This program is for verifying if the PWM controller is working fine, also gives *gnuplot* data to the standard output which can be redirected to a file using the environments IO redirection operators. The program takes environment sampling frequency, PWM frequency and duty cycle percentage as parameters. Executable name is `pwm_controller`.

7.4 PWM Motor Control

This program demonstrates a PWM controller regulated DC motor. PWM frequency, duty cycle percentage and cycles to run simulation are necessary command line parameters of this program. Executable name is `pwm_motor_control`.

7.5 Fuzzy Logic

This command line tool lets user to create arbitrary input and output triangles. Then according to those triangles user can evaluate arbitrary input values with fuzzy logic algorithm. See the README file provided with *pensedemo* for a sample session. Executable name is `fuzzy_logic`.

7.6 Fuzzy Logic Controller

This is essentially same as Sec. 7.1 but in this case there is no PWM unit between Fuzzy Logic controller and voltage source. This program also outputs *gnuplot* data to standard output. Executable name is `fuzzy_logic_controller`.