

Documentation for Single Board Heater System

Rakhi R
Rupak Rokade
Inderpreet Arora
Kannan M. Moudgalya
Kaushik Venkata Belusonti



IIT Bombay
September 27, 2014

Contents

List of Scilab Code	2
1 Block diagram explanation of Single Board Heater System	3
1.1 Microcontroller	3
1.1.1 PWM for heat and speed control	4
1.1.2 Analog to Digital conversion	6
1.2 Instrumentation amplifier	6
1.3 Communication	7
1.3.1 Serial port communication	8
1.3.2 Using USB for Communication	8
1.4 Display and Resetting the setup	8
2 Performing a Local Experiment on Single Board Heater System	12
2.1 Using SBHS on a Windows OS	13
2.1.1 Installing Drivers and Configuring COM Port	13
2.1.2 Steps to Perform a Local Experiment	14
2.2 Using Single Board Heater System on a Linux System	17
2.3 Scilab Code under common_files	20
3 Using Single Board Heater System, Virtually!	26
3.1 Introduction to Virtual Labs at	
IIT Bombay	26
3.2 Evolution of SBHS virtual labs	27
3.3 Current Hardware Architecture	30
3.4 Current Software Architecture	31
3.5 Conducting experiments using the Virtual lab	31
3.5.1 Registration, Login and Slot Booking	33

3.5.2	Configuring proxy settings and executing python based client	34
3.5.3	Executing scilab code	35
4	Identification of Transfer Function of a Single Board Heater System through Step Response Experiment	40
4.1	Procedure to perform Step Test	40
4.2	Determination of First Order Transfer Function	43
4.2.1	Procedure	45
4.3	Determination of Second Order Transfer Function	46
4.3.1	Procedure	47
4.4	Discussion	47
4.5	Conducting Step Test on SBHS, virtually	49
4.6	Scilab Code	49
5	Identification of Transfer Function of a Single Board Heater System through Ramp Response Experiment	59
5.1	About this experiment	59
5.2	Theory	59
5.3	Procedure to perform Ramp Test	61
5.4	Ramp Test Analysis	63
5.5	Discussion	63
5.6	Conducting ramp test on SBHS, virtually	64
5.7	Scilab Code	64
6	Frequency Response Analysis of a Single Board Heater System by the Application of Sine Wave	70
6.1	Theory	70
6.2	Procedure to perform Sine Test	73
6.3	Sine Test Analysis	74
6.4	Conducting Sine Test on SBHS, virtually	77
6.5	Scilab Code	80
7	Controlling Single Board Heater System using PID controller	87
7.1	Theory	87
7.1.1	Proportional Control Action	88
7.1.2	Integral Control Action	89
7.1.3	Derivative Control Action	89

7.2	Ziegler-Nichols Rule for Tuning PID Controllers	90
7.2.1	First Method	90
7.2.2	Second Method	92
7.3	Implementing PI Controller using Trapezoidal Approximation	94
7.3.1	Implementing PI controller using Trapezoidal Approximation on SBHS, virtually	96
7.4	Implementing PI Controller using Backward Difference Approximation	97
7.4.1	Implementing PI Controller using Backward Difference Approximation on SBHS, virtually	98
7.5	Implementing PI Controller using Forward Difference Approximation	99
7.5.1	Implementing PI Controller using Forward Difference Approximation on SBHS, virtually	100
7.6	Implementing PID Controller using Backward Difference Approximation	101
7.6.1	Implementing PID Controller using Backward Difference Approximation on SBHS, virtually	103
7.7	Implementing PID Controller using Trapezoidal Approximation for Integral Mode and Backward Difference Approximation for the Derivative Mode	103
7.7.1	Implementing PID Controller using Trapezoidal Approximation for Integral Mode and Backward Difference Approximation for the Derivative Mode on SBHS, virtually	105
7.8	Implementing PID Controller with Filtering using Backward Difference Approximation	106
7.8.1	Implementing PID Controller with Filtering using Backward Difference Approximation on SBHS, virtually	108
7.9	Scilab Code	109
7.9.1	Scilab code for serial communication	109
7.9.2	Scilab code for PI controller	109
7.9.3	Scilab code for PID controller	112
8	Two Degrees of Freedom (2-DOF) Controller	117
8.1	Introduction to 2-DOF Controller	117
8.2	2-DOF Controller Design using the Pole Placement Method [5]	120
8.3	2-DOF Pole Placement Controller Design and Implementation using SBHS	123

8.3.1	Procedure to use the Scilab/Xcos Code for a Local Experiment	125
8.3.2	Implementing 2-DOF Controller on SBHS, Virtually	127
8.4	Scilab Code for Local Experiment	127
8.5	Scilab Code for Virtual Experiment	147
8.6	Scilab Codes Common for both Local and Virtual Experiments	150
9	PRBS Modeling and Implementation of Pole Placement Controller	156
9.1	Issues with Step Test and an Alternate Approach	158
9.2	Step by step procedure to do PRBS testing	159
9.3	Determination of Discrete Time Transfer Function	160
9.4	Performing PRBS testing on SBHS, locally	162
9.5	Implementing 2DOF pole-placement controller using PRBS model, virtually	162
9.6	Implementing 2DOF pole-placement controller using PRBS model, locally	165
9.7	Scilab Local codes	165
9.7.1	Identification codes	165
9.7.2	Controller codes	170
9.8	Scilab Virtual codes	174
9.8.1	Identification codes	174
9.8.2	Controller codes	179
10	Implementing Internal Model Controller for First Order System on a Single Board Heater System	184
10.1	IMC Design for Single Board Heater System	184
10.2	Step for Designing IMC for Stable Plant	186
10.3	Experimental Results	188
10.3.1	Implementing IMC on SBHS, virtually	188
10.4	Scilab Code	191
11	Design and Implementation of Self Tuning PI and PID Controllers on Single Board Heater System	194
11.1	Introduction	194
11.1.1	Objective	194
11.2	Theory	195
11.2.1	Why a Self Tuning Controller?	195
11.2.2	The Approach Followed	196

11.2.3	Direct synthesis	196
11.3	Ziegler Nichols Tuning	198
11.4	Step Test Experiments and Parmeter Estimation	199
11.4.1	Step Test Experiment	199
11.4.2	Conventional Controller Design	202
11.4.3	Self Tuning Controller Design	203
11.5	Implementation	205
11.5.1	PI Controller	205
11.5.2	PID Controller	206
11.5.3	Self Tuning Controller	209
11.6	Set Point Tracking	209
11.6.1	PI Controller Designed by Direct Synthesis	211
11.6.2	PI Controller using Ziegler Nichols Tuning	213
11.6.3	PID Controller using Ziegler Nichols Tuning	216
11.6.4	Conclusion	217
11.7	Disturbance Rejection	218
11.7.1	PI Controller Designed by Direct Synthesis	218
11.7.2	PI Controller using Ziegler Nichols Tuning	221
11.7.3	PID Controller using Ziegler Nichols Tuning	224
11.7.4	Conclusion	226
11.8	Implementing Self Tuning controller on SBHS, virtually . . .	227
11.9	Scilab Codes	228
11.9.1	Conventional Controller, local	228
11.9.2	Self Tuning Controller, local	233
11.9.3	Conventional Controller, virtual	237
11.9.4	Fan Disturbance in PI Controller	237
11.9.5	Self Tuning Controller, local	241
12	Model Predictive Control in Single Board Heater System using SCILAB	245
12.1	Objective	245
12.2	Single Board Heater System	245
12.3	Model Predictive Control	246
12.4	Implementing MPC	247
12.5	Working of codes	247
12.6	Procedure to implement MPC on SBHS	247
12.7	Description of xcos	248
12.8	Code for MPC (<i>mpc_run.sci</i>)	249

12.9	Other codes used	252
12.10	Experiments conducted to implement MPC	253
12.11	Sample run to implement MPC	254
12.12	Positive Step Change to Set Point and Fan	254
12.13	Negative Step Change to Set Point and Fan	257
12.14	Effect of Tuning parameters: Weighting factors, We and Wu	260
12.15	For same factor of We and Wu	261
	12.15.1 Positive Step Change and (We, Wu)=(1,1) (Expt 1.1)	261
	12.15.2 Positive Step Change and (We, Wu)=(10,10) (Expt 1.2)	264
	12.15.3 Positive Step Change and (We, Wu)=(40,40) (Expt 1.3)	267
	12.15.4 Negative Step Change and (We,Wu)=(1,1) (Expt 2.1)	269
	12.15.5 Negative Step Change and (We, Wu)=(10,10) (Expt 2.2)	271
	12.15.6 Negative Step Change and (We, Wu)=(40,40) (Expt 2.3)	273
12.16	For different We and Wu factors	274
	12.16.1 We =100 and Wu = 2 (Expt 5.1)	275
	12.16.2 We =2 and Wu = 100 (Expt 5.2)	277
	12.16.3 We =10 and Wu = 100 (Expt 5.3)	279
	12.16.4 We =100 and Wu = 10 (Expt 5.4)	281
12.17	Conclusion on Weighting factor experiments	283
12.18	Effect of Control Horizon Paramter, q	283
12.19	For positive step change in Set point and Fan speed	285
	12.19.1 For q =2 (Expt 3.1)	285
	12.19.2 For q =3 (Expt 3.2)	287
	12.19.3 For q =4 (Expt 3.3)	289
12.20	For negative step change in Set point and Fan speed	291
	12.20.1 For q =2 (Expt 4.1)	291
	12.20.2 For q =3 (Expt 4.2)	293
	12.20.3 For q =4 (Expt 4.3)	295
12.21	Conclusion on the effect of Control Horizon parameter	296
12.22	Implementing Model Predictive controller on SBHS, locally	297
12.23	Conclusion for MPC project	297
12.24	Acknowledgement	298
12.25	Appendix	299
12.26	Appendix 1: General Information on Experiments for this Project	299
12.27	Appendix 2: Values of State Space matrices	299
12.28	Appendix 3: Attachments and Contact Information	301
12.29	Scilab Code	301

List of Scilab Code

2.1	comm.sci	20
2.2	init.sci	22
2.3	plotting.sci	23
4.1	label.sci	49
4.2	costf_1.sci	50
4.3	firstorder.sce	50
4.4	costf_2.sci	52
4.5	order_2_heater.sci	52
4.6	secondorder.sce	53
4.7	ser_init.sce	54
4.8	step_test.sci	55
4.9	stepc.sce	55
4.10	steptest.sci	55
4.11	firstorder_virtual.sce	56
4.12	secondorder_virtual.sce	57
5.1	ramp_test.sci	64
5.2	label.sci	64
5.3	cost.sci	65
5.4	cost_approx.sci	66
5.5	ramptest.sci	66
5.6	ramptest.sce	66
5.7	ramp_virtual.sce	67
6.1	sine_test.sci	80
6.2	sinetest.sce	80
6.3	sinetest.sci	81
6.4	sine2.sce	81
6.5	lable.sci	82
6.6	bodeplot.sce	83

6.7	labelbode.sci	83
6.8	TFbode.sce	84
6.9	comparison.sce	84
6.10	sine2_virtual.sce	85
7.1	ser_init.sci	109
7.2	pi_ta.sci	109
7.3	pi_bda.sci	110
7.4	pi_fda.sci	111
7.5	pid_bda.sci	112
7.6	pid_ta_bda.sci	113
7.7	pid_filter.sci	114
7.8	pid_bda_virtual.sce	115
7.9	pid_bda_virtual.sci	115
8.1	twodof_para.sce	127
8.2	twodof.sci	132
8.3	start.sce	133
8.4	cindep.sci	134
8.5	clcoef.sci	135
8.6	colsplit.sci	136
8.7	cosfil_ip.sci	137
8.8	indep.sci	137
8.9	left_prm.sci	138
8.10	makezero.sci	141
8.11	move_sci.sci	142
8.12	polsize.sci	143
8.13	polyno.sci	143
8.14	rowjoin.sci	144
8.15	seshft.sci	145
8.16	t1calc.sci	146
8.17	twodof_para.sce	147
8.18	twodof.sce	148
8.19	twodof.sci	149
8.20	myc2d.sci	150
8.21	desired.sci	150
8.22	polmul.sci	151
8.23	polsplit3.sci	152
8.24	pp_im.sci	153
8.25	xdync.sci	154

8.26	zpowk.sci	155
9.1	ser_init.sce	165
9.2	costfunction.sci	165
9.3	optimize.sce	166
9.4	prbs.sci	168
9.5	prbstest.sci	169
9.6	second_order.sci	169
9.7	start.sce	170
9.8	prbs.sce	170
9.9	prbs_pp.sci	171
9.10	ser_init.sce	171
9.11	start.sce	172
9.12	twodof_para.sce	172
9.13	costfunction.sci	174
9.14	optimize.sce	175
9.15	prbs.sci	177
9.16	prbstest.sci	178
9.17	prbstest.sce	178
9.18	second_order.sci	178
9.19	prbs.sce	179
9.20	prbscontrol-virtual.sci	180
9.21	twodof_para.sce	180
10.1	ser_init.sce	191
10.2	imc.sci	191
10.3	imc_virtual.sce	192
10.4	imc_virtual.sci	192
11.1	ser_init.sce	228
11.2	pi_bda_dist.sci	228
11.3	pi_bda.sci	230
11.4	pid_bda_dist.sci	231
11.5	pid_bda.sci	232
11.6	pi_bda_tuned_dist.sci	233
11.7	pi_bda_tuned.sci	234
11.8	pid_bda_tuned_dist.sci	235
11.9	pid_bda_tuned.sci	236
11.10	pi_bda_dist.sci	237
11.11	pi_bda.sci	238
11.12	pid_bda_dist.sci	239

11.13	pid_bda.sci	240
11.14	pi_bda_tuned_dist.sci	241
11.15	pi_bda_tuned.sci	242
11.16	pid_bda_tuned_dist.sci	242
11.17	pid_bda_tuned.sci	243
12.1	mpc.sci	301
12.2	mpc_init_local.sce	302
12.3	mpc_local.sci	302

Chapter 1

Block diagram explanation of Single Board Heater System

Figure 1.1 shows the block diagram of ‘Single Board Heater System’(SBHS). Microcontroller ATmega16 is used at the heart of the setup. The microcontroller can be programmed with the help of an In-system programmer port(ISP) available on the board. The setup can be connected to a computer via two serial communication ports namely RS232 and USB. A particular port can be selected by setting the jumper to its appropriate place. The communication between PC and setup takes place via a serial to TTL interface. The μ C operates the Heater and Fan with the help of separate drivers. The driver comprises of a power MOSFET. A temperature sensor is used to sense the temperature and feed to the μ C through an Instrumentation Amplifier. Some required parameter values are also displayed along with some LED indications.

1.1 Microcontroller

Some salient features of ATmega16 are listed below:

1. 32 x 8 general purpose registers.
2. 16K Bytes of In-System Self-Programmable flash memory
3. 512 Bytes of EEPROM
4. 1K Bytes of internal Static RAM (SRAM)

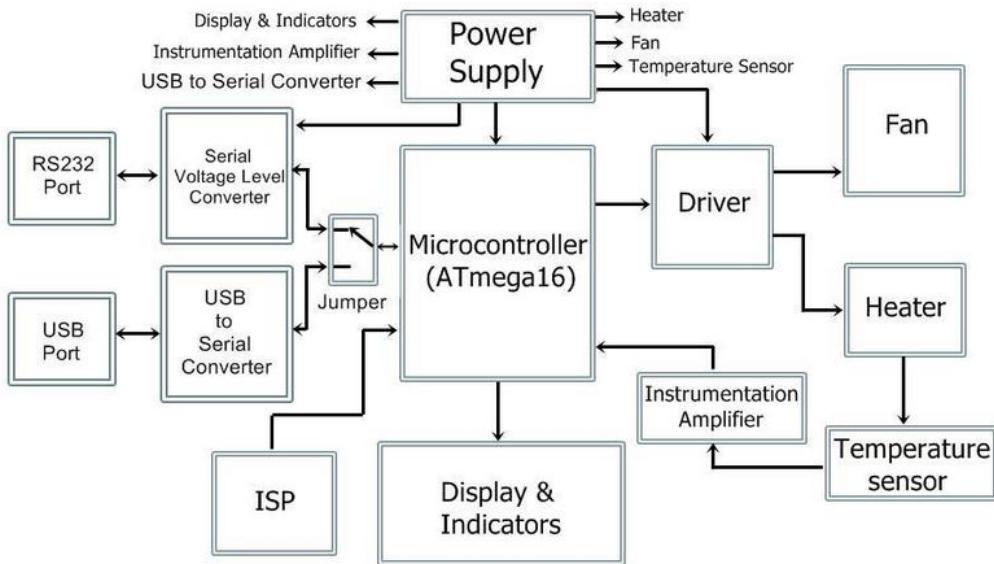


Figure 1.1: Block Diagram

5. Two 8-bit Timer/Counters
6. One 16-bit Timer/Counter
7. Four PWM channels
8. 8-channel,10-bit ADC
9. Programmable Serial USART
10. Up to 16 MIPS throughput at 16 MHz

Microcontroller plays a very important role. It controls every single hardware present on the board, directly or indirectly. It executes various tasks like, setting up communication between PC and the equipment, controlling the amount of current passing through the heater coil, controlling the fan speed, reading the temperature, displaying some relevant parameter values and various other necessary operations.

1.1.1 PWM for heat and speed control

The Single Board Heater System contains a Heater coil and a Fan. The heater assembly consists of an iron plate placed at a distance of about 3.5mm from a

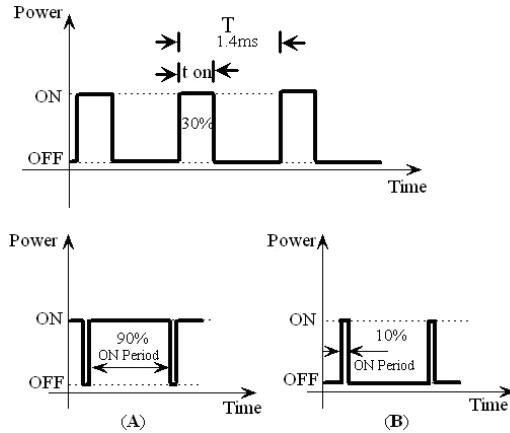


Figure 1.2: Pulse Width Modulation (A): On time is 90% of the total time period, (B): ON time is 10% of total time period

nichrome coil. When current passes through the coil it gets heated and in turn raises the temperature of the iron plate. Altering the heat generated by the coil and also the speed at which the fan is operated, are the objectives of our prime interest. The amount of power delivered to the Fan and Heater can be controlled in various ways. The technique used here is called as PWM (abbreviation of Pulse Width Modulation)technique. PWM is a process in which the duty cycle of the square wave is modulated.

$$\text{Duty cycle} = \frac{T_{ON}}{T} \quad (1.1)$$

Where T_{ON} is the ON time of the wave corresponding to the HIGH state of logic and T is the total time period of the wave. Power delivered to the load is proportional to T_{ON} time of the signal. This is used to control the current flowing through the heating element and also speed of the fan. An internal timer of the microcontroller is used to generate a square wave. The ON time of the square wave depends on a count value set in the internal timer. The pulse width of the waveform can be varied accordingly by varying this count value. Thus, PWM waveform is generated at the appropriate pin of the microcontroller. This generated PWM waveform is used to control the power delivered to the load (Fan and Heater).

A MOSFET is used to switch at the PWM frequency which indirectly controls the power delivered to the load. A separate MOSFET is used to control the power delivered to each of the two loads. The timer is operated at 244Hz.

1.1.2 Analog to Digital conversion

As explained earlier, the heat generated by the heater coil is passed to the iron plate through convection. The temperature of this plate is measured by using a temperature sensor AD590.

Some of the salient features of AD590 include:

1. Linear current output: $1\mu\text{A}/\text{K}$
2. Wide range: -55°C to $+150^\circ\text{C}$
3. Sensor isolation from the case
4. Low cost

The output of AD590 is then fed to the microcontroller through an Instrumentation Amplifier. The signal obtained at the output of the Instrumentation Amplifier is in analog form. It should be converted in to digital form before feeding as an input to the microcontroller. ATmega16 features an internal 8-channel , 10 bit successive approximation ADC (analog to digital converter) with $0\text{-Vcc}(0\text{ to } \text{Vcc})$ input voltage range, which is used for converting the output of Instrumentation Amplifier. An interrupt is generated on completion of analog to digital conversion. Here, ADC is initialize to have $206\ \mu\text{s}$ of conversion time . Digital data thus obtained is sent to the computer via serial port as well as for further processing required for the on-board display.

1.2 Instrumentation amplifier

Instrumentation Amplifiers are often used in temperature measurement circuits in order to boost the output of the temperature sensors. A typical three Op-Amp Instrumentation amplifier is shown in the figure 1.3. The Instrumentation Amplifiers (IAs) are mostly preferred, where the sensor is located at a remote place and therefore is susceptible to signal attenuation, due to their very low DC offsets, high input impedance, very high Common mode rejection ratio (CMRR). The IAs have a very high input impedance and hence do not load the input signal source. IC LM348 is used to construct a 3 Op-Amp IA. IC LM348 contains a set of four Op-Amps. Gain of the amplifier is given by equation 1.2

$$\frac{V_o}{V_2 - V_1} = \left\{ 1 + \frac{2R_f}{R_g} \right\} \frac{R_2}{R_1} \quad (1.2)$$

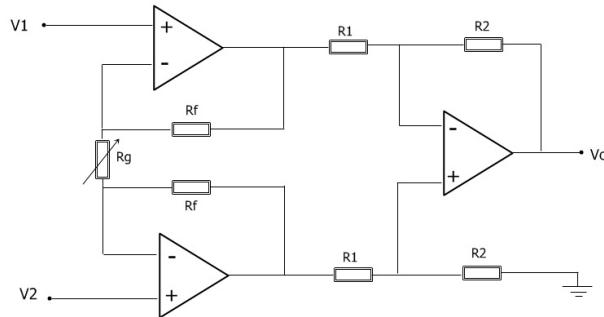


Figure 1.3: 3 Op-Amp Instrumentation Amplifier

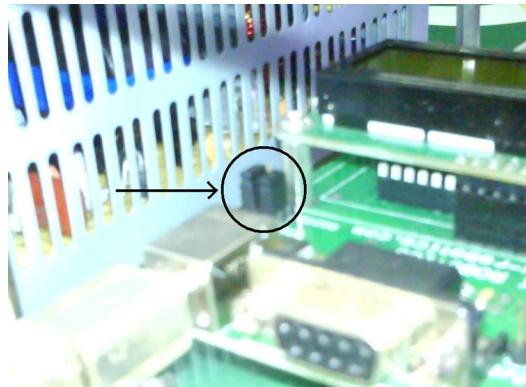


Figure 1.4: Jumper arrangement

The value of R_g is kept variable to change the overall gain of the amplifier. The signal generated by AD590 is in $\mu\text{A}/^\circ\text{K}$. It is converted to $\text{mV}/^\circ\text{K}$ by taking it across a $1 \text{ K}\Omega$ resistor. The $^\circ\text{K}$ to $^\circ\text{C}$ conversion is done by subtracting 273 from the $^\circ\text{K}$ result. One input of the IA is fed with the $\text{mV}/^\circ\text{K}$ reading and the other with 273 mV. The resulting output is now in $\text{mV}/^\circ\text{C}$. The output of the IA is fed to the microcontroller for further processing.

1.3 Communication

The set up has the facility to use either USB or RS232 for communication with the computer. A jumper is been provided to switch between USB and RS232. The voltages available at the TXD terminal of microcontroller are in TTL (transistor-



Figure 1.5: RS232 cable

transistor logic). However, according to RS232 standard voltage level below -5V is treated as logic 1 and voltage level above +5V is treated as logic 0. This convention is used to ensure error free transmission over long distances. For solving this compatibility issue between RS232 and TTL, an external hardware interface IC MAX202 is used. IC MAX202 is a +5V RS232 transceiver.

1.3.1 Serial port communication

Serial port is a full duplex device i.e. it can transmit and receive data at the same time. ATmega16 supports a programmable Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART). Its baud rate is fixed at 9600 bps with character size set to 8 bits and parity bits disabled.

1.3.2 Using USB for Communication

After setting the jumper to USB mode connect the set up to the computer using a USB cable at appropriate ports as shown in the figure 1.8. To make the setup USB compatible, USB to serial conversion is carried out using IC FT232R. Note that proper USB driver should be installed on the computer.

1.4 Display and Resetting the setup

The temperature of the plate, percentage values of Heat and Fan and the machine identification number (MID) are displayed on LCD connected to the microcon-

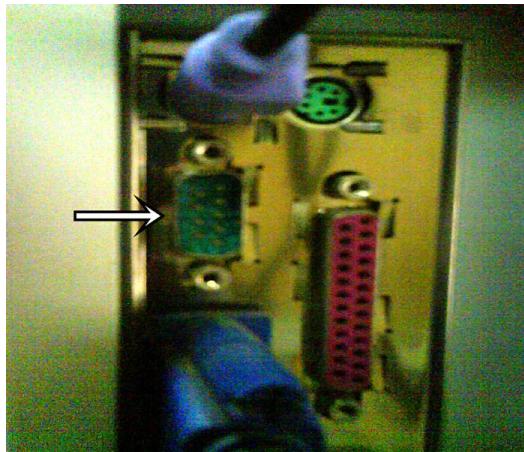


Figure 1.6: Serial port



Figure 1.7: USB communication

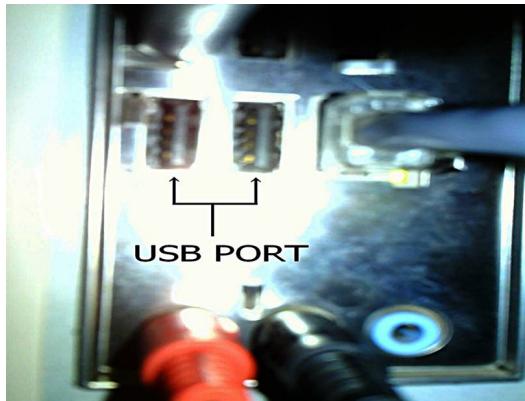


Figure 1.8: USB PORT



Figure 1.9: Display

troller. As shown in figure 1.9, numerals below TEMP indicate the actual temperature of the heater plate in °C. Numerals below HEA and FAN indicate the respective percentage values at which heater and fan are being operated. Numerals below MID corresponds to the device identification number. The set up could be reset at any time using the reset button shown in figure 1.10. Resetting the setup takes it to the standby mode where the heater current is forced to be zero and fan speed is set to the maximum value. Although these reset values are not displayed on the LCD display these are preloaded to the appropriate units.

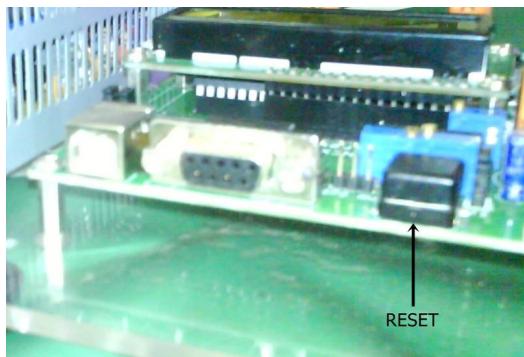


Figure 1.10: Reset

Chapter 2

Performing a Local Experiment on Single Board Heater System

This chapter explains the procedure to use Single Board Heater System locally with Scilab i.e. when you are physically accessing SBHS using your computer. An open loop experiment, step test is used for demonstrating this procedure. The process however remains the same for performing any other experiment explained in this document, unless specified otherwise.

Hardware and Software Requirements

For working with the Single Board Heater system, following components are required:

1. SBHS with USB cable and power cable.
2. PC/Laptop with Scilab software installed. Scilab can be downloaded from:
<http://www.scilab.org>
3. FTDI Virtual Com Port driver corresponding to the OS on your PC. Linux users do not need this. The driver can be downloaded from:
<http://www.ftdichip.com/Drivers/VCP.htm>

2.1 Using SBHS on a Windows OS

This section deals with the procedure to use SBHS on a Windows Operating System. The Operating System used for this document is Windows 7, 32-bit OS. If you are using some other Operating System or the steps explained in section 2.1.1 are not sufficient to understand, refer to the official document available on the main ftdi website at www.ftdichip.com. On the left hand side panel, click on 'Drivers'. In the drop-down menu, choose 'VCP Drivers'. Then on the web page page, click on 'Installation Guides' link. Choose the required OS document. We would now begin with the procedure.

2.1.1 Installing Drivers and Configuring COM Port

After powering ON the SBHS and plugging in the USB cable to the PC (check the jumper settings on the board are set to USB communication) for the very first time, the Welcome to Found New Hardware Wizard dialog box will pop up. Select the option **Install from a list or specific location**. Choose **Search for best driver in these locations**. Check the box **Include this location in the search**. Click on **Browse**. Specify the path where the driver is copied as explained earlier (item no.3) and install the driver by clicking **Next**. Once the wizard has successfully installed the driver, the SBHS is ready for use. Please note that this procedure should be repeated twice.

Now, the communication port number assigned to the computer port to which the Single Board Heater System is connected, via an RS232 or USB cable should be identified. For identifying this port number, right click on **My Computer** and click on **Properties**. Then, select the **Hardware** tab and click on **Device Manager**. The list of hardware devices will be displayed. Locate the **Ports(COM & LPT)** option and click on it. The various communication ports used by the computer will be displayed. If the SBHS is connected via RS232 cable, then look for **Communications Port(COM1)** else look for **USB Serial Port**. For RS232 connection, the port number mostly remains **COM1**. For USB connection it may change to some other number. Note the appropriate COM number. This process is illustrated in figure 2.1

Sometimes the COM port number associated with the USB port after connecting a USB cable may be greater than 9. Since the serial tool box can handle only single digit port number (upto 9), it is necessary to change this COM port number. Following is the procedure to change the COM port number. Double click on the name of the particular port. Click on **Port Settings** tab and then click on

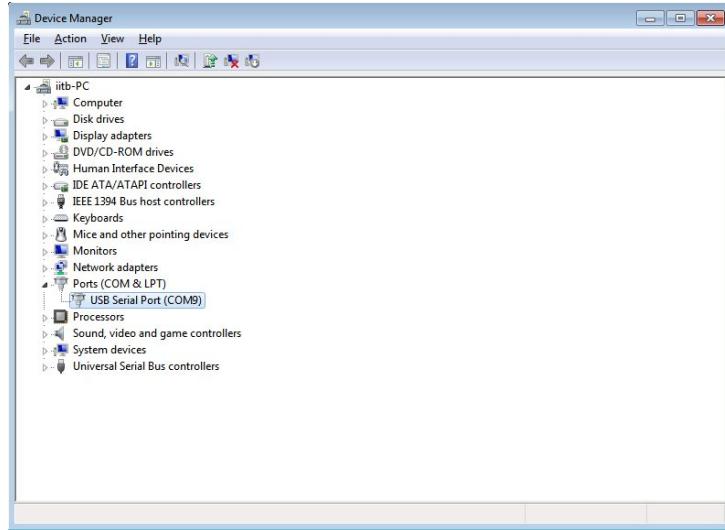


Figure 2.1: Checking Communication Port number

Advanced. In the COM port number drop-down menu, choose the port number to any other number less than 10. This procedure is illustrated in figure 2.2. After following the procedure the COM port number can be verified as described earlier.

Scilab must be installed on your computer. We recommend the use of scilab-5.3.3. This is because all the codes are created and tested using scilab-5.3.3. These codes may very well work in higher versions of scilab but one cannot use the same codes back again in scilab-5.3.3. This is because a software is always backward compatible, never forward compatible. Scilab for windows or linux can be downloaded from scilab.org. However, if scilab-5.3.3 for your OS is not available on scilab.org then one can download it from sbhs.os-hardware.in/downloads. Installation of scilab on windows is very straight-forward. After you download the .exe file one has to double click on it and proceed with the instructions given by the installer. All default options will work. However, note that scilab on windows requires internet connection during installation.

2.1.2 Steps to Perform a Local Experiment

Go to sbhs.os-hardware.in/downloads. Let us take a look at the downloads page. There are two versions of the scilab code. One which can be used with SBHS locally i.e. when you are physically accessing SBHS using your computer and another to be used for accessing SBHS virtually. This section expects you to

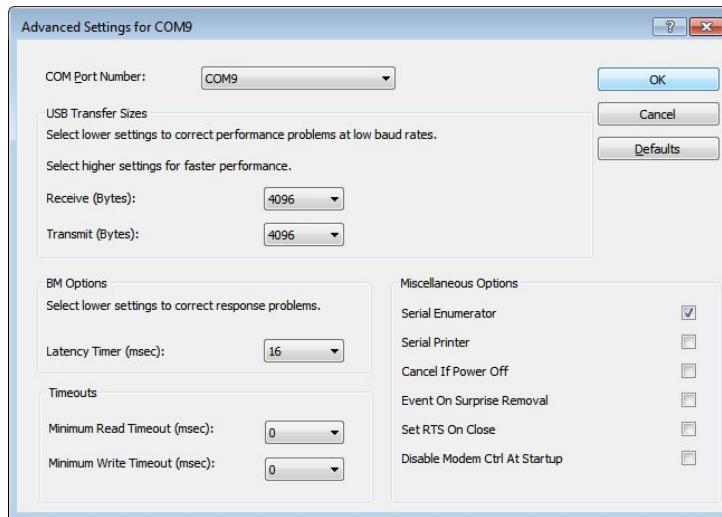


Figure 2.2: Changing Com port number

download the local version. On extracting the file that you will download, you will get a folder **local**. This folder will contain many folders named after the experiment. You will also find a directory named **common-files**. We are going to use the folder named **Step_test**.

1. Launch Scilab from start menu or double click the Scilab icon on the desktop (if any). Before executing any scripts which are dependent on other files or scripts, one needs to change the working directory of Scilab. This will set the directory path in Scilab from where the other necessary files should be loaded. To change the directory, click on file menu and then choose "Change directory". This can also be performed by typing `cd<space>folder path`. Change the directory to the folder **Step_test**. There is another quicker way to make sure you are in the required working directory. Open the experiment folder. Double click on the scilab file you want to execute. Doing so will automatically launch scilab and also automatically change the working directory. To know your working directory at any time, execute the command `pwd` in the scilab console.
2. Next, we have to load the content of **common-files** directory. Notice that this directory is just outside the **Step_test** directory. The **common-files** directory has several functions written in **.sci** files. These functions are required for executing any experiment. To load these functions type

`getd<space>folder path`. The `folder path` argument will be the complete path to `common-files` directory. Since this directory is just outside our `Step_test` directory, the command can be modified to
`getd<space>..\common_files` So now we have all functions loaded.

3. Next we have to load the serial communication toolbox. For doing so we have to execute the `loader.sce` file present in the `common-files` directory. To do so execute the command
`exec<space>..\common_files\loader.sce` or
`exec<space>folder path\loader.sce`.
4. Next, click on `editor` from the menu bar to open the Scilab editor or simply type `editor` on the Scilab console and open the file `ser_init.sce`. Change the value of the variable `port2` to the COM number identified for the connected SBHS. For example, one may enter '`COM5`' as the value for `port2`. Notice that there is no space between `COM` and `5` and `COM5` is in single quotes. Keep all other parameters untouched. Execute this `.sce` file by clicking on the execute button available on the menu bar of scilab editor window. The message `COM Port Opened` is displayed on successful implementation. If there are any errors, reconnecting the USB cable and/or restarting Scilab may help.
5. Next we have to load the function for the step test experiment. This function is written in `step_test.sci` file. Since we do not have to make any changes in this file we can directly execute it from scilab console without opening it. Run the command `exec<space>step_test.sci` in scilab console. The results are illustrated in figure 2.3.
6. Next, type `Xcos` on the Scilab console or click on `Applications` and select `Xcos` to open Xcos environment. Load the `step_test.xcos` file from the `File` menu. The Xcos interface is shown in figure 2.5. The block parameters can be set by double clicking on the block. To run the code click on `Simulation` menu and click on `Start`. After executing the code in Xcos successfully the plots as shown in figure 2.6 will be generated. Note that the values of fan and heater given as input to the Xcos file are reflected on the board display.
7. To stop the experiment click on the `Stop` option on the menu bar of the Xcos environment.

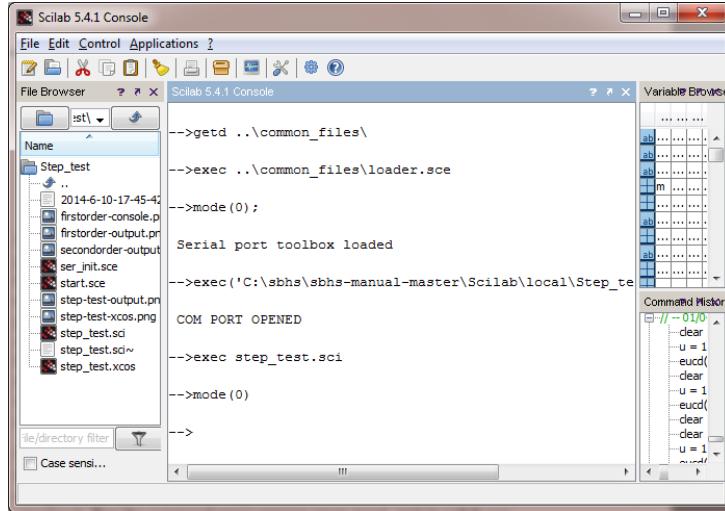


Figure 2.3: Expected responses seen on the console

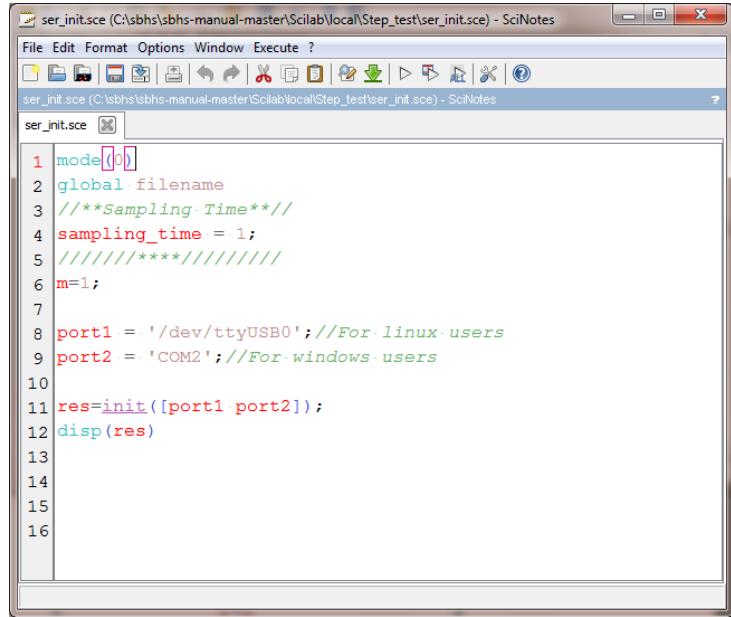
All of the activities mentioned above, from `getd<space>..\common_files` until starting the xcos simulation, are coded in a file named `start.sce`. Executing this file will do all necessary things automatically with just click of a button. This file however assumes three things. These are

1. The location of `common-files` directory is not changed
2. The current working directory is correct
3. The port number mentioned in `ser_init.sce` is correct

2.2 Using Single Board Heater System on a Linux System

This section deals with the procedure to use SBHS on a Linux Operating System. The Operating System used for this document is Ubuntu 12.04. For Linux users, the instructions given in section 2.1 hold true with a few changes as below:

On a linux system, Scilab-5.3.3 can be either installed from available package manager (synaptic in case of Ubuntu) or its portable version can be downloaded from scilab.org or <http://sbhs.os-hardware.in/downloads>. If in-



The screenshot shows a SciNotes window titled "ser_init.sce (C:\sbhs\sbhs-manual-master\Scilab\local\Step_test\ser_init.sce) - SciNotes". The code in the editor is:

```
1 mode(0)
2 global.filename
3 //***Sampling Time***/
4 sampling_time = 1;
5 ////////////****/////////
6 m=1;
7
8 port1 = '/dev/ttyUSB0';//For linux users
9 port2 = 'COM2';//For windows users
10
11 res=init([port1 port2]);
12 disp(res)
13
14
15
16
```

Figure 2.4: Executing script files

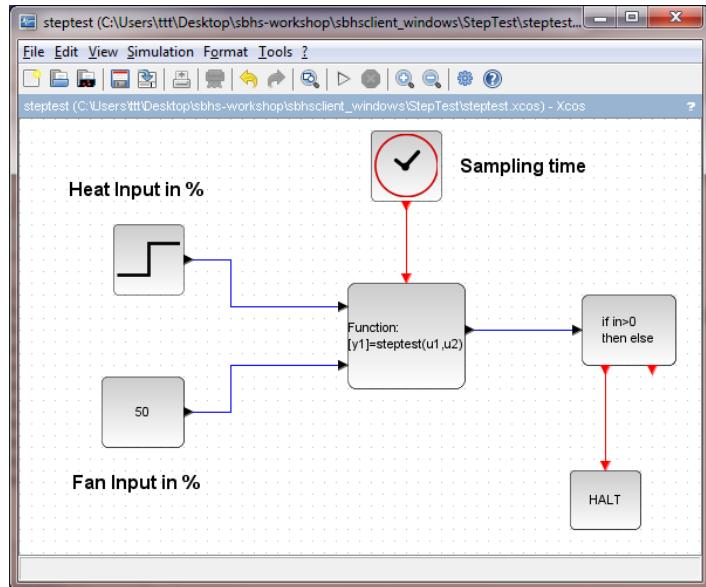


Figure 2.5: Xcos Interface

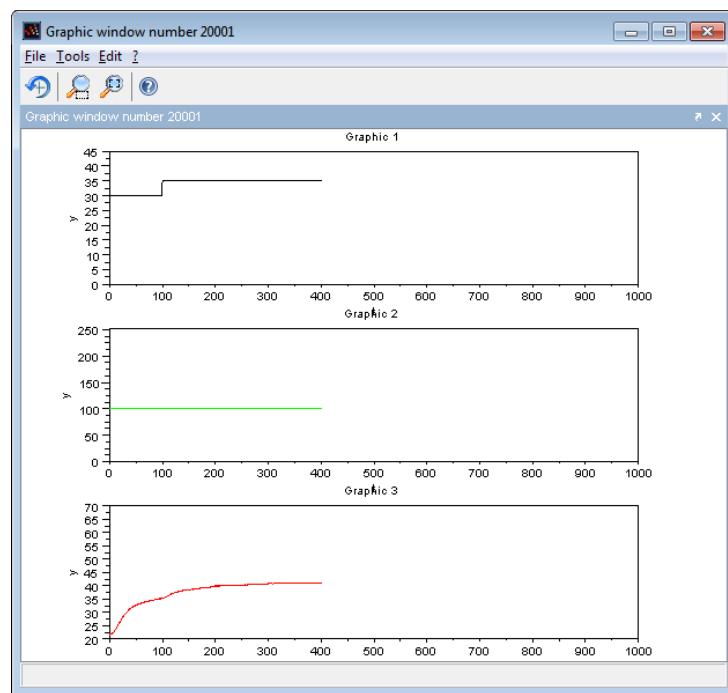


Figure 2.6: Plot obtained after executing step_test.xcos



Figure 2.7: Checking the port number in linux (1)

stalled from a package manager then scilab can be launched by opening a terminal (Alt+Ctrl+T) and executing the command `sudo scilab`. If one downloads the portable version then first the file has to unpacked. This can be done by right clicking on it and choosing **Extract here**. Then one has to open the terminal and change the directory to `scilab/bin`. Then the command `sudo ./scilab` must be executed to launch scilab. Note that scilab must always be launched with sudo permissions to be able to communicate with the SBHS.

FTDI COM port drivers are not required for connecting the SBHS to the PC. After plugging in the USB cable to the PC, check the serial port number by typing `ls /dev/ttyUSB*` on the terminal, refer Fig.2.7.

Note down this number and change the value of the variable `port1` inside the `ser.init.sce` file, refer Fig.2.4.

Except for these changes rest all of the steps mentioned in Section 2.1.2 can be followed.

2.3 Scilab Code under common_files

Scilab Code 2.1 comm.sci

```

1 function [ temp ] = comm( heat , fan )
2     global heatdisp fandisp tempdisp sampling_time m
        name handl filename
3
4 if heat<0
5     heat=0
6 end
```

```

7   if heat>100
8       heat=100
9   end
10  if fan<0
11      fan=0
12 end
13 if fan>100
14     fan=100
15 end
16
17 writeserial(handl,ascii(254)); // Input Heater ,
18 writeserial accepts
19           convert 254 into its
20           string ; so
21           equivalent
22 writeserial(handl,ascii(heat));
23 writeserial(handl,ascii(253)); // Input Fan
24 writeserial(handl,ascii(fan));
25 writeserial(handl,ascii(255)); // To read Temp
26 sleep(100);
27
28 temp = ascii(readserial(handl)); // Read serial
29 returns a string , so
30           convert it to
31           its integer ( ascii )
32           equivalent
33 temp = temp(1) + 0.1*temp(2); // convert to temp with
34           decimal points
35 e.g : 40.7
36
37 A = [m,heat,fan,temp];
38
39 fdfh = file('open',filename,'unknown');
40
41 file('last',fdfh)
42
43 write(fdfh,A,'(7(e11.5,1x))');

```

```
34  
35   file( 'close' , fdfh );  
36  
37 endfunction
```

Scilab Code 2.2 init.sci

```
1 global filename m  
2 function status = init(port)  
3   global handl filename  
4  
5   OS = getos();  
6  
7   if OS == string('Linux')  
8     port_num = port(1);  
9     handl = openserial(port_num , "9600,n,8,0")  
10  else  
11    port_num = port(2);  
12    handl = openserial(port_num , "9600,n,8")  
13  end  
14  
15  if (ascii(handl) ~= [])  
16    status = string('COM PORT OPENED')  
17  else  
18    status = string('ERROR: Check port number or USB  
connection')  
19  end  
20  
21  m = 1;  
22  
23  dt = getdate();  
24  year = dt(1);  
25  month = dt(2);  
26  day = dt(6);  
27  hour = dt(7);  
28  minutes = dt(8);  
29  seconds = dt(9);  
30
```

```

31 file1 = strcat(string([year month day hour minutes
32           seconds]),'-');
33 string txt;
34 filename = strcat([file1, "txt"], '.');
35
endfunction

```

Scilab Code 2.3 plotting.sci

```

1 function [] = plotting(var,low_lim,high_lim)
2
3     global heatdisp fandisp tempdisp setpointdisp
4             sampling_time m
5
6     timeTitle = "No. of samples with sampling time = "
7         +string(sampling_time)
8     if low_lim==[] & high_lim==[]
9         heat_min = low_lim(1)
10        fan_min = low_lim(2)
11        temp_min = low_lim(3)
12        time_min = low_lim(4)
13
14        heat_max = high_lim(1)
15        fan_max = high_lim(2)
16        temp_max = high_lim(3)
17        time_max = high_lim(4)
18
19    else
20        heat_min = 0
21        fan_min = 0
22        temp_min = 20
23        time_min = 0
24
25        heat_max = 100
26        fan_max = 100
27        temp_max = 100
28        time_max = 1000
29
end

```

```

28
29
30
31 if length(var)==3
32     heat = var(1);
33     fan = var(2);
34     temp = var(3);
35
36     heatdisp=[heatdisp;heat];
37     subplot(311);
38     xtitle("",timeTitle,"Heat in percentage")
39     plot2d(heatdisp,rect=[time_min,heat_min,
40             time_max,heat_max],style=1)
41
42     fandisp=[fandisp;fan];
43     subplot(312);
44     xtitle("",timeTitle,"Fan in percentage")
45     plot2d(fandisp,rect=[time_min,fan_min,
46             time_max,fan_max],style=2)
47
48     tempdisp=[tempdisp;temp];
49     subplot(313);
50     xtitle("",timeTitle,"Temperature (deg
51             celcius)")
52     plot2d(tempdisp,rect=[time_min,temp_min,
53             time_max,temp_max],style=5)
54
55
56 elseif length(var) == 4
57
58     heat = var(1);
59     fan = var(2);
60     temp = var(3);
61     setpoint = var(4);
62
63     heatdisp=[heatdisp;heat];
64     subplot(311);
65     xtitle("",timeTitle,"Heat in percentage")

```

```

62 plot2d(heatdisp , rect=[time_min , heat_min ,
63   time_max , heat_max ] , style=1)
64 fandisp=[fandisp ; fan ];
65 subplot(312);
66 xtitle("" ,timeTitle , "Fan in percentage")
67 plot2d(fandisp , rect=[time_min , fan_min ,
68   time_max , fan_max ] , style=2)
69 tempdisp=[tempdisp ; temp ];
70 setpointdisp=[setpointdisp ; setpoint ]
71 subplot(313)
72 xtitle("" ,timeTitle , "Temperature (deg
73   celcius)")
74 plot2d(tempdisp , rect=[time_min , temp_min ,
75   time_max , temp_max ] , style=5)
76 plot2d(setpointdisp , rect=[time_min , temp_min ,
77   time_max , temp_max ] , style=1)
78
79 end
80 endfunction

```

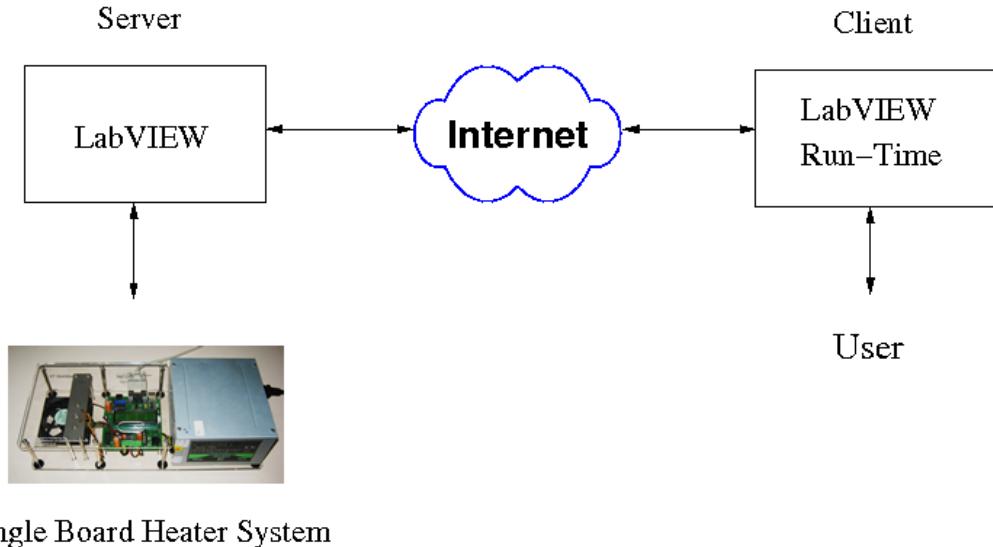
Chapter 3

Using Single Board Heater System, Virtually!

3.1 Introduction to Virtual Labs at IIT Bombay

The concept of virtual laboratory is a brilliant step towards strengthening the education system of an university/college, a metropolitan area or even an entire nation. The idea is to use the ICT i.e. Information and Communications Technology, mainly the Internet for imparting education or exchange of educational information. Virtual Laboratory mainly focuses on providing the laboratory facility, virtually. Various experimental set-ups are hooked up to the internet and made available to use for the external world. Hence, anybody can connect to that equipment over the internet and carry out various experiments pertaining to it. The beauty of this idea is that a college who cannot afford to have some experimental equipments can still provide laboratory support to their students through virtual lab, and all that will cost it is a fair Internet connection! Moreover, the laboratory work does not ends with the college hours, one can always use the virtual lab at any time and at any place assuming the availability of an internet connection.

A virtual laboratory for SBHS is launched at IIT Bombay. Here is the url to access it: vlabs.iitb.ac.in/sbhs/. A set of 36 SBHS are made available to use over the internet 24× 7. These individual kits are made available to the users on hourly basis. We have a slot booking mechanism to achieve this. Since there are 36 SBHS connected with an hours slot for 24 hrs a day, we have 864 one hour slots a day. This means that 864 individual users can access the SBHS in a day for an hour.



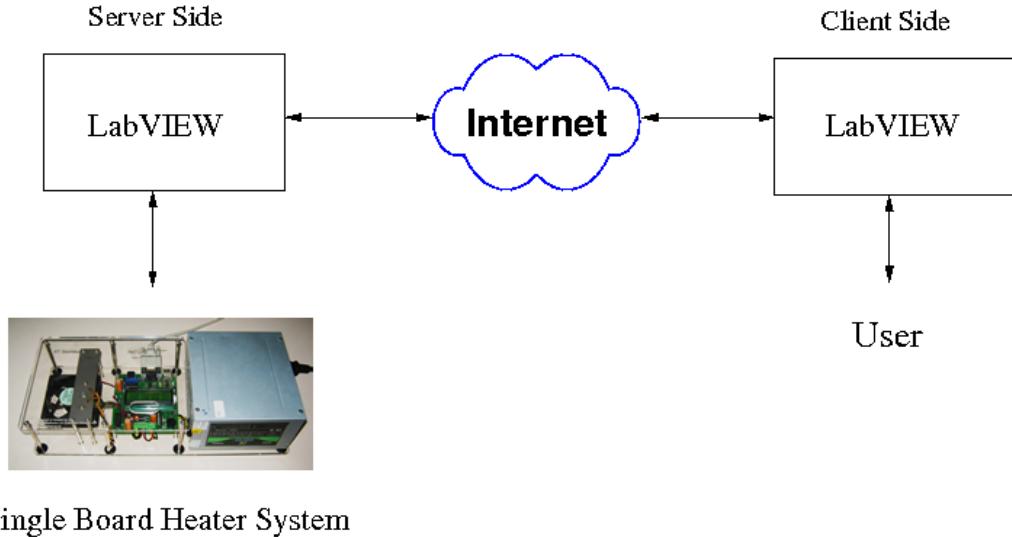
Single Board Heater System

Figure 3.1: SBHS virtual laboratory with remote access using LabVIEW

This also means that up to 6048 users can use the SBHS for an hour in a week and 181440 in a month! A web page is hosted which is the first interface to the user. The user registers/logs in himself/herself here. The user is also supposed to book a slot for accessing the SBHS. A database server maintains a record of the data generated through the web interface. A python script is hosted on the server side and it helps in connecting the user with the corresponding SBHS placed remotely. A free and open source scientific computing Software, Scilab, is used by the user for implementing the experiment on SBHS, in terms of simple Scilab coding.

3.2 Evolution of SBHS virtual labs

In [4], the control algorithm is implemented at the server end and the remote student just keys in the parameters, as shown in Figure 3.1. LabVIEW was used for the implementation of the same. The server end consisted of a computer connected with an SBHS with a full blown copy of LabVIEW installed on it. The client has a LabVIEW run time engine available for free download from the National Instruments website. A few LabVIEW algorithms/experiments were hosted on the server. The client accesses these algorithm/experiment over the Internet using a web browser by entering appropriate parameters.



Single Board Heater System

Figure 3.2: SBHS virtual laboratory with remote access and live data sharing using LabVIEW

It was realized that the learning experience is not complete for this structure. This is because the server hosts some pre-built LabVIEW algorithms and a user can only access these few algorithms. The user can in no way change the program and can only input experimental parameters. Hence, we came up with a new architecture as shown in the Figure 3.2 that used full blown copies of LabVIEW at both server and client ends.

This idea uses the DataSocket technology of LabVIEW. Since now the client is having a complete LabVIEW installation on his/her computer she can now implement her own algorithms. Thus this architecture did provide a complete learning experience to the students. There are some shortcomings as well:

- LabVIEW is expensive and students may not be able to afford to buy it. It is also prohibitively expensive for the Government to distribute it.
- We used the LabVIEW version 8.04, which had restricted scripting language. It was tedious to create new control algorithms in it.

This made us shift to free and open source (FOSS) software. We replaced LabVIEW with Java and Scilab as shown in Figure 3.3. Scilab at the server end is

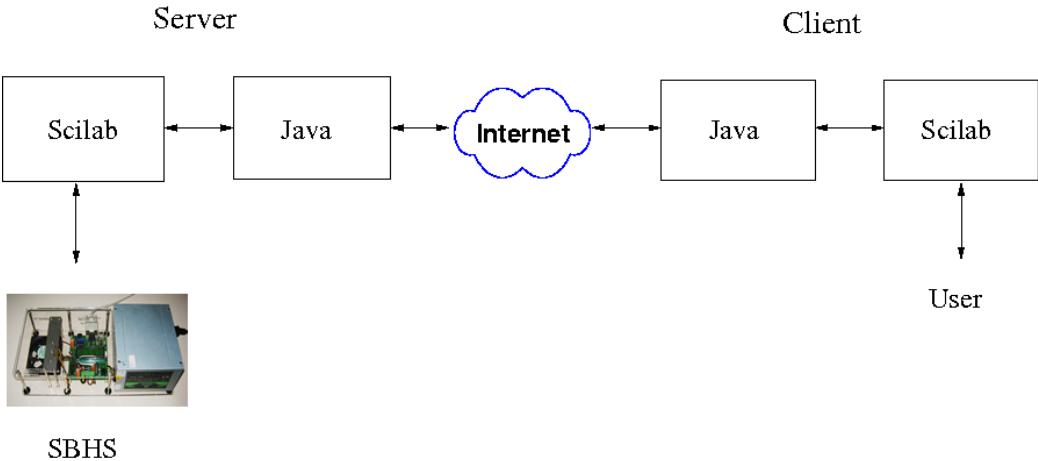


Figure 3.3: SBHS virtual laboratory using open source software

used for communicating with SBHS. Scilab at the client end is used for implementing the algorithms. Java is used at both the server as well as client end for communication over the Internet thereby connecting the client with the server.

For the above solution, we need a dedicated copy of scilab running at the server end for every SBHS. One way to do this is to host it on multiple computers with unique IPs. Hence the number of SBHS we want to host requires as many computer's and public IPs thereby making it expensive. Moreover, it also limits its scalability. The other way to do this is to host multiple java and scilab servers on the same computer. Hosting many copies of Scilab simultaneously requires a powerful computer for the server.

For these reasons we decided to take scilab off the server computer and to use java alone to communicate with the SBHS directly. Java also communicates with the client computer. We connected seven SBHS systems to a USB port through a serial port hub. This architecture was implemented on a Windows Operating System. We faced the following difficulties in this solution.

- When we connected more than one serial hub to a PC, the port ID could not be retrieved correctly. Port ID information is required if we want a student to use the same SBHS for all their experiments during different sessions.
- The experiments required time stamping of the data communicated to and from the server. But this time stamping was not linear and suffered instability.

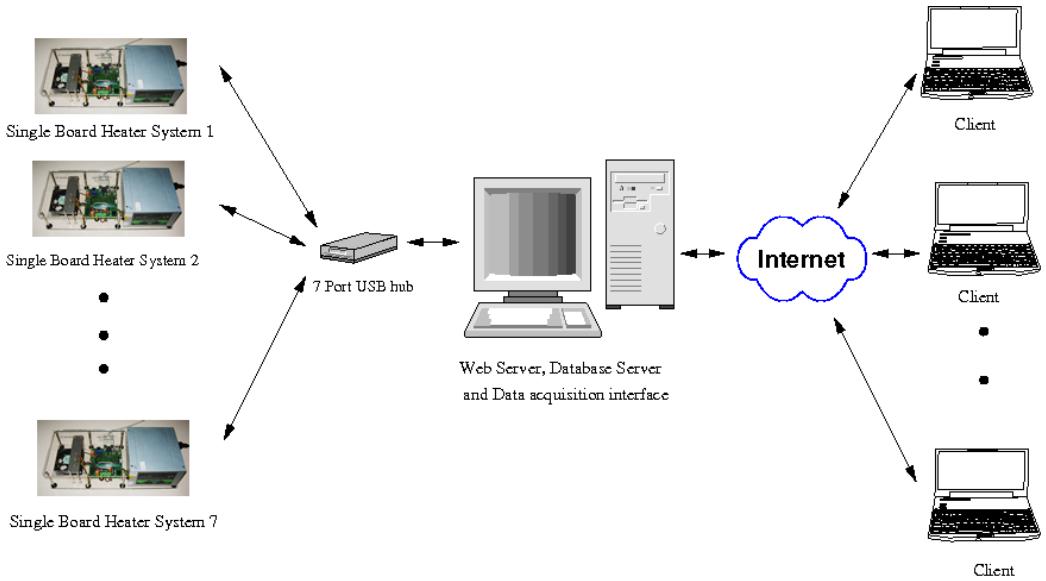


Figure 3.4: Virtual control lab hardware architecture

This made us to completely switch to FOSS with Ubuntu Linux as the OS and is the current structure of the Virtual lab as shown in Figure 3.6

3.3 Current Hardware Architecture

The current hardware architecture of the virtual single-board heater system lab involves 36 single-board heater systems connected to the server via multiple 7 and 10-port USB hubs. The server computer is connected to a high speed inter-network and has enough processing capability to host data acquisition, database, and web servers. It has been successfully tested for the undergraduate Process Control course and the graduate Digital Control and Embedded systems courses conducted at IIT Bombay as well as few workshops over the internet. Currently, this architecture is integrated with a cameras on each SBHS to facilitate live video streaming. This gives the user a feel of remote hands-on.

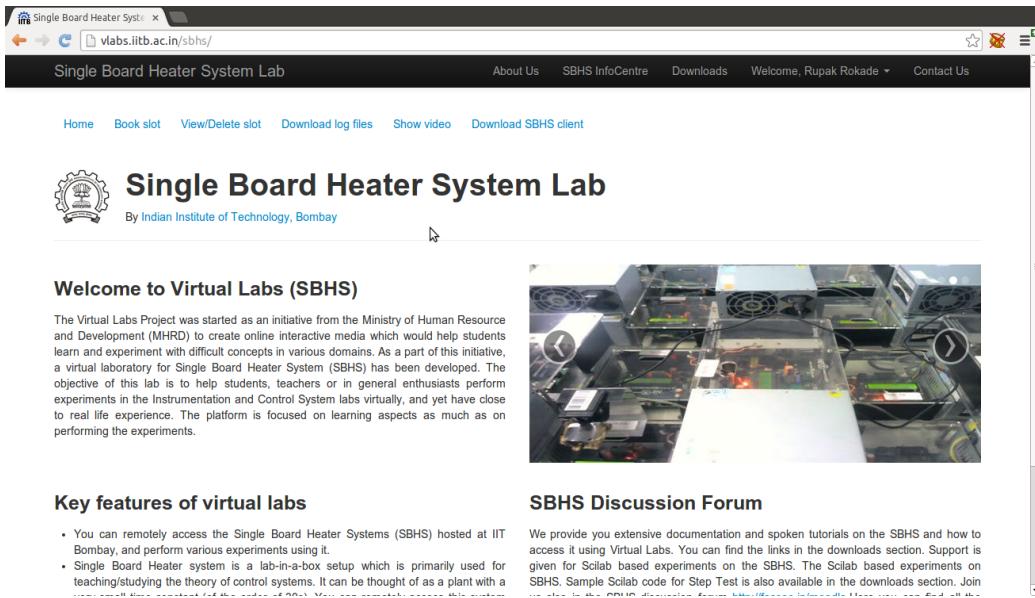


Figure 3.5: Home page of SBHS V Labs

3.4 Current Software Architecture

The current software architecture of this virtual SBHS control lab is shown in Figure 3.6. The server computer runs Ubuntu Linux 12.04.2 OS. It hosts a Apache-MySQL server. The SBHS server is based on Python-Django framework and is linked to Apache server using Apache’s WSGI module. The MySQL database server has the details of all the registered users, their slot details, authentication keys to allow remote access, etc. As shown in Figure ??, the Python-Django server has pages for registration, login, slot booking etc. [9]. On the client end, control algorithms are running in Scilab and a python based client application communicates with virtual labs server over the Internet.

The steps to be performed before and during each experiment are explained next.

3.5 Conducting experiments using the Virtual lab

This section explains the procedure to use Single Board Heater System remotely using Scilab i.e. when you are accessing SBHS remotely using your computer over

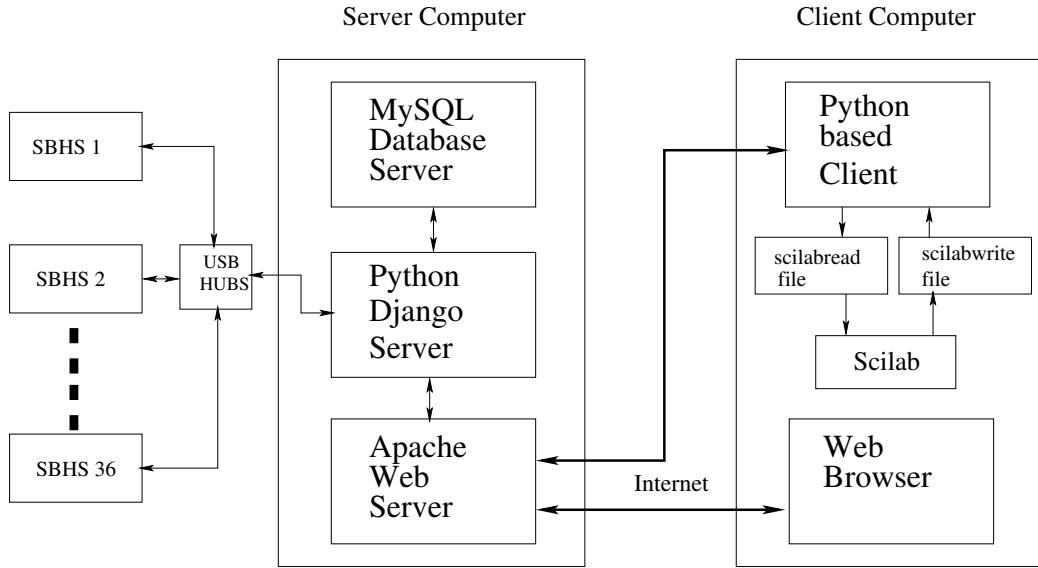


Figure 3.6: Current Architecture of SBHS Virtual Labs

the virtual labs platform. An open loop experiment, step test is used for demonstrating this procedure. The process however remains the same for performing any other experiment explained in this section, unless specified otherwise. Let us first see the required files to be downloaded and installations to be done. Scilab is required to be installed on your computer. Please refer to Section 2.1.1 and Section 2.2 for the procedure to install scilab on Windows and Linux system, respectively.

SBHS scilab code for your OS, under the title **SBHS Virtual Code**, must be downloaded from <http://sbhs.os-hardware.in/downloads>. The code downloaded will be in zip format. After the zip is unpacked, you will see the folder **scilab_codes_windows** or **scilab_codes_linux_32** or **scilab_codes_linux_64** depending on which one you download. All these downloads will have scilab experiment folders such as **Step_test**, **Ramp_Test**, **pid_controller** etc. We will be using the **Step_test** folder. Do not alter the directory structure. If you want to copy or move an experiment outside the directory then make sure you also copy the **common_files** folder. The **common_files** folder must always be one directory outside the experiment folder. Now given that you have scilab installed and working and the required scilab code downloaded, let us see the step-by-step procedure to do a remote experiment.



Figure 3.7: Show Video

3.5.1 Registration, Login and Slot Booking

Go to the website sbhs.os-hardware.in and click on the Virtual labs link available on the left hand side. The home page of Virtual labs is illustrated in Fig 3.5. If you are a first time user, click on the link Login/Register. Fill out the registration form and submit it. If the registration form is submitted successfully, you will receive an activation link on your registered Email id. Use this link to complete the registration process. If you skip this step you will not be able to login. Registration is a one time process and need not be repeated more than once. After completing registration login with your username and password. You should now get the options to Book Slot, Delete Slot etc.

View/Delete slot option allows you to delete your booked slots. This option however will work only for slots booked for the future. You cannot delete a past or the current slot. Download log files option gives you the facility to download your experiment log files. Clicking on it will give you a list of all of the experiments you had performed. Show video option can be used to see the live video feed of your SBHS. Web cameras are mounted on every SBHS. You can see the display of your SBHS as shown in Fig. 3.7.

Clicking on the Book slot option will allow you to book an experiment time slot. Slots are of 55 minutes duration. Click on the Book slot option. If the current slot is free, Book now option will appear. Click on it. Else you have to book an advance slot for the next hour or any other future time using the calender that appears on this page. There is a limit to how many slots one can book in a day. We are allowing only two non-consecutive slots, per user, to be booked in a day. However, there is no limit to how many current slots you book and use. Book an

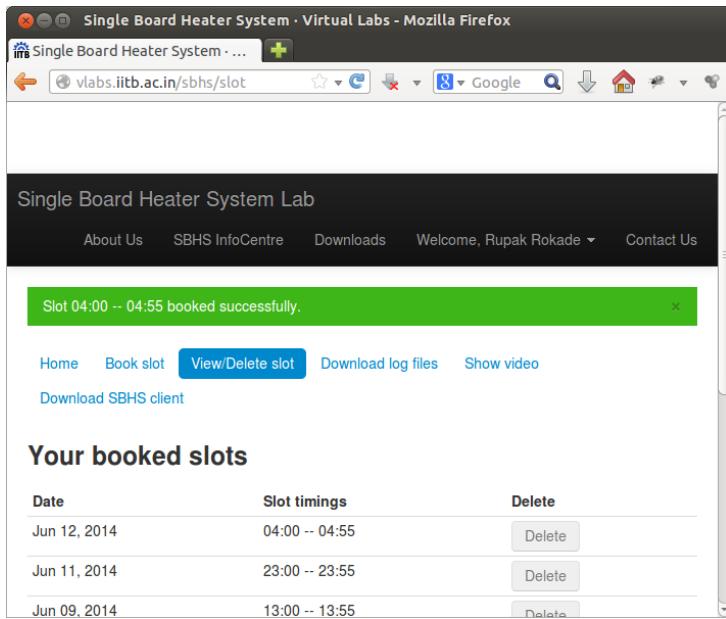


Figure 3.8: Slot booking

experiment slot. Once you successfully book a slot a **Slot booked successfully** message highlighted in green color will appear on the top side. This is shown in Fig. 3.8. It will automatically take you to the View/Delete slot page.

3.5.2 Configuring proxy settings and executing python based client

After booking a slot, the web activity is over. You may close the web browser unless you need it open to see live video feed of your SBHS. The next step is to establish the communication link between the server and your computer. A python based application is created which handles the network communication.

Let us first see how to do the proxy settings if you are behind a proxy network. Open the folder `common_files`. Open the file `config`. This file contains various arguments whose values must be entered to configure proxy.

Do not change the contents of config file if

- You are accessing from inside IIT Bombay OR
- You are accessing from outside IIT Bombay and using an open network

such as at home OR using a mobile internet

Change the contents of config file if

- You are outside IIT Bombay and using a proxy network such as at an institute, office etc.

If you have to put the proxy details, first change the argument `use_proxy = Yes` (Y should be capital in Yes and N should be capital in No). Fill in the other details as per your proxy network. If your proxy network allows un-authenticated login then make the argument `proxy_username` and `proxy_password` blank. This proxy setting has to done only once.

Open the `Step_test` folder. Double click on the file `run`. This will open the client application as shown in Fig. 3.9. Note that for first time execution, it will take a minute to open the client application. It will show various parameters related to the experiment such as SBHS connection, Client version, User login and Experiment status. The green indicators show that the corresponding activity is correct or functional. Here it says that the Client application is been able to connect to the server and the client version being used is the latest. The User login and Experiment status is showing red and will turn green after a registered username and password is entered. If the SBHS is offline or there are some other issues, the corresponding error will be displayed and the respective indicator will turn red. Enter your registered username and password and press login. You should get the message `Ready to execute scilab code`. The application also shows the value of iteration, heat, fan, temperature and time remaining for experimentation. It also shows the name of log file created for the experiment.

3.5.3 Executing scilab code

Inside the `StepTest` folder, if on a windows system, double click on the file `stepc.sce`. This should automatically launch scilab and also open the `stepc.sce` in the scilab editor. It will also automatically change the scilabs working directory. On a linux system, launch scilab manually. Then change the scilab working directory to the folder `StepTest`. This can be done by clicking on `File` menu and then selecting `change current directory`. Next, execute the command `getd`. Scilab command `getd` is used to load all functions defined in all `.sci` files inside a specified folder. Here we have some important function files inside the `..../common_files` directory. Executing this command will load all of

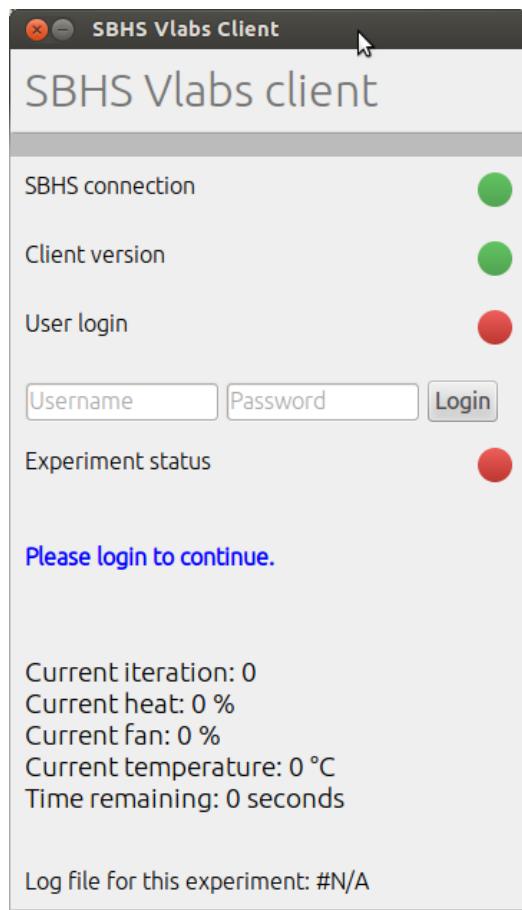


Figure 3.9: Python Client

The screenshot shows a SciNotes editor window with the title bar "stepc.sce (/home/ttt/Desktop/scilab_codes_linux_32/StepTest/stepc.sce)". The menu bar includes File, Edit, Search, Preferences, Window, Execute, and Help. Below the menu is a toolbar with various icons. The main code area contains the following Scilab script:

```

1 mode(0)
2 global fdh.fdt.fncr.fncw.m.err_count.y.limits.sampling_time.m
3
4 //*****
5 sampling_time=1; //In seconds. Fractions are allowed
6 //*****
7 exec_("steptest.sci");
8
9 ok = init();
10
11 if ok~= [] //open xcos only if communication is through (ie reply.h
as come from server)
12 ..... xcos('steptest.xcos');
13 ..... else
14 ..... disp("NO-NETWORK CONNECTION!");
15 ..... return
16 end
17

```

Line 1, Column 0.

Figure 3.10: stepc.sce file

the functions that the experiment needs. Open the file `stepc.sce` using the Open option inside File menu. The file is shown in Fig. 3.10

The experiment sampling time can be set inside the `stepc.sce` file. You may want to change it to a higher value if your network is slow. The default value of 1 second works fine in most cases. On the menu bar, click on Execute option and choose option file with echo. This will execute the scilab code. If the network is working fine, an xcos diagram will open automatically. If it doesn't open then see the scilab console for error messages. If you get a No network connection error message then try executing the scilab code again. The xcos diagram is for the step test experiment as shown in Fig. 3.11. You can set the value of the heat and fan. Keep the default values. On the menu bar of the xcos window, click on start button. This will execute the xcos diagram. If there is no error, you will get a graphic window with three plots. It will show the value of Heat in % Fan in % and ..temperature in degree celcius as shown in Fig. 3.12. After sufficient time of experimentation click on the stop button to stop the experiment. Go to the StepTest folder. Here you will find a logs folder. This folder will have another folder named after your username. It will have the log file for your experiment.

Read the log file name as

`YearMonthDate_hours_minutes_seconds.txt`. This log file contains all the values

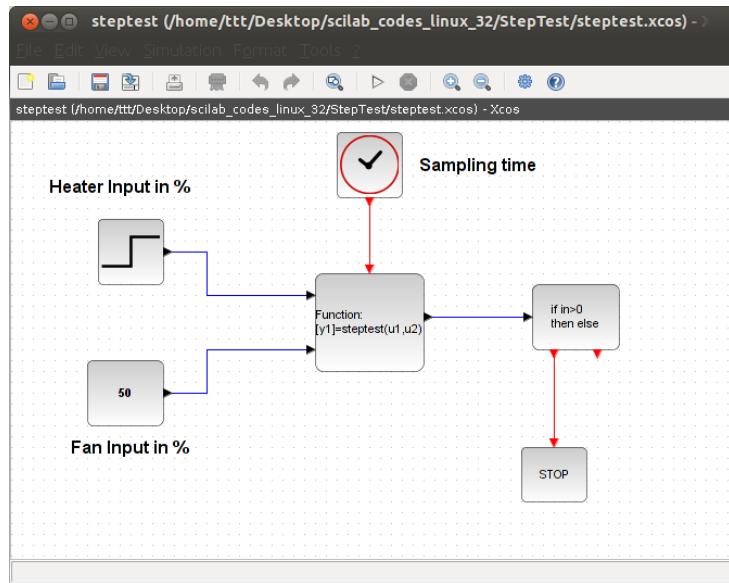


Figure 3.11: Xcos for step test

of heat fan and temperature. It can be used for further analysis.

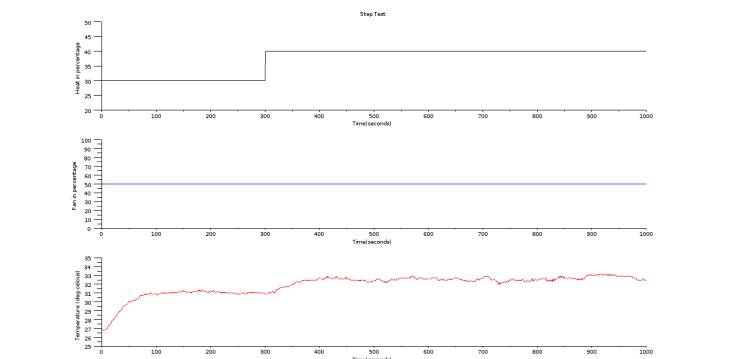


Figure 3.12: Output of Step Test

Chapter 4

Identification of Transfer Function of a Single Board Heater System through Step Response Experiment

The aim of this experiment is to perform step test on a Single Board Heater System and to identify system transfer function using step response data. The target group is anyone who has basic knowledge of control engineering.

We have used Scilab and Xcos as an interface for sending and receiving data. Xcos diagram is shown in figure 4.1. Heater current and fan speed are the two inputs for this system. They are given in percentage of maximum. These inputs can be varied by setting the properties of the input block's properties in Xcos. The plots of their amplitude versus number of collected samples are also available on the scope windows. The output temperature profile, as read by the sensor, is also plotted. The data acquired in the process is stored on the local drive and is available to the user for further calculations.

4.1 Procedure to perform Step Test

The procedure to perform a step test is explained in section 2.1.2.

In the `step_test.xcos` file, open the heater block's parameters to apply a step change of say 5 percent to the heater at operating point of 30 percent of heater after 200 seconds. The block parameters of the step input block will have `Step time = 200`, `Initial value = 30` and `Final value = 35`. Keep the fan input constant at 50 percent. Start the experiment and let it continue until you see

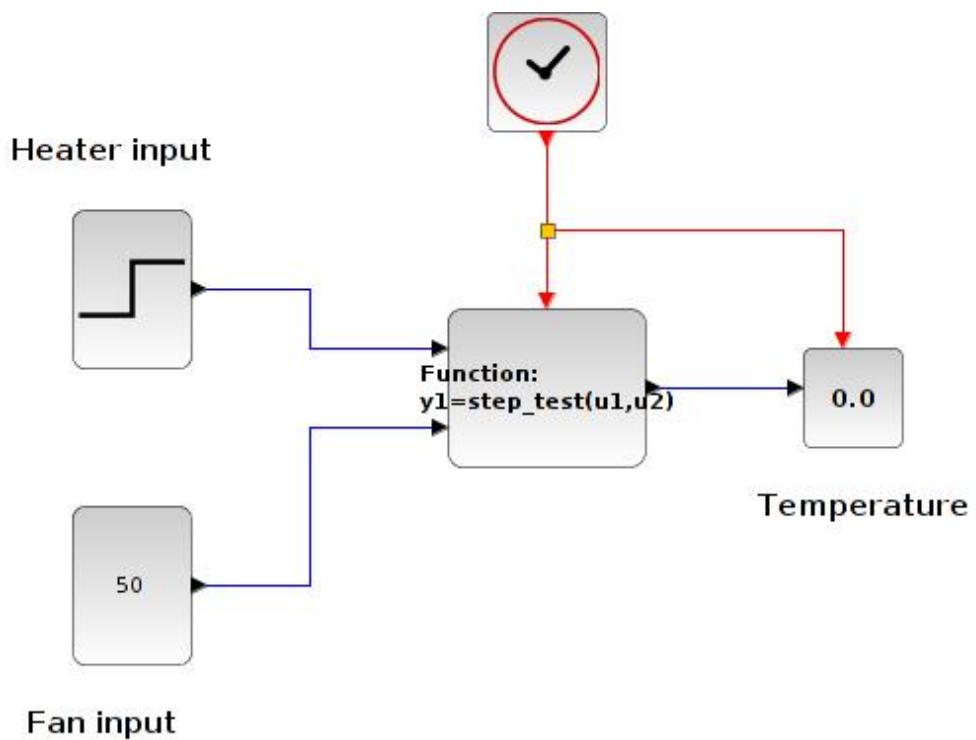


Figure 4.1: Xcos for this experiment

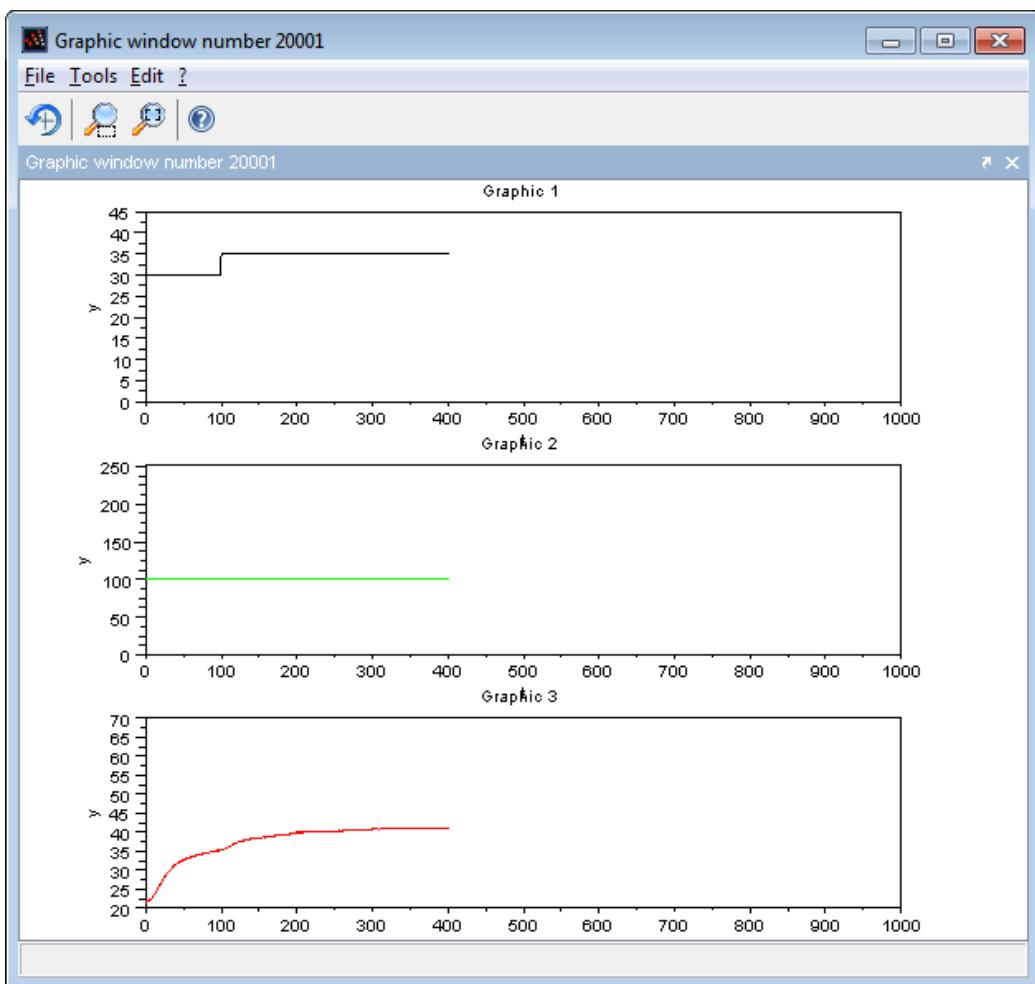


Figure 4.2: Graph shows heater current, fan speed and output temperature

0.100E+00	0.300E+02	0.50E+03	0.195E+02
0.200E+00	0.300E+02	0.50E+03	0.195E+02
.	.	.	.
0.700E+03	0.350E+02	0.50E+03	0.318E+02
0.800E+03	0.350E+02	0.50E+03	0.318E+02

Table 4.1: Step data obtained after performing the Step Test

the temperature reach the steady state.

The step test data file is saved in `Step_test` folder. The name of the file will be the date and time at which the experiment was conducted. Referring to the data file thus obtained as shown in table 4.1, the first column in this table denotes samples. The second column in this table denotes heater in percentage. It starts at 30 and increases with a step size of 5 units. The third column denotes the fan in percentage. It has been held constant at 50 percent. The fourth column refers to the value of temperature.

4.2 Determination of First Order Transfer Function

Identification of the transfer function of a system is important as it helps us to represent the physical system mathematically. Once the transfer function is obtained, one can acquire the response of the system for various inputs without actually applying them to the system.

Consider the standard first order transfer function given below

$$G(s) = \frac{C(s)}{R(s)} \quad (4.1)$$

$$G(s) = \frac{1}{\tau s + 1} \quad (4.2)$$

Rewriting the equation, we get

$$C(s) = \frac{R(s)}{\tau s + 1} \quad (4.3)$$

A step is given as input to the first order system. The Laplace transform of a step function is $\frac{1}{s}$. Hence, substituting $R(s) = \frac{1}{s}$ in equation 4.3, we obtain

$$C(s) = \frac{1}{\tau s + 1} \frac{1}{s} \quad (4.4)$$

Solving $C(s)$ using partial fraction expansion, we get

$$C(s) = \frac{1}{s} - \frac{1}{s + \frac{1}{\tau}} \quad (4.5)$$

Taking the Inverse Laplace transform of equation 4.5, we get

$$c(t) = 1 - e^{-\frac{t}{\tau}} \quad (4.6)$$

From the above equation it is clear that for $t=0$, the value of $c(t)$ is zero. For $t=\infty$, $c(t)$ approaches unity. Also, as the value of 't' becomes equal to τ , the value of $c(t)$ becomes 0.632. τ is called the time constant and represents the speed of response of the system. But it should be noted that, smaller the time constant-faster the system response. By getting the value of τ , one can identify the transfer function of the system.

Consider the system to be first order. We try to fit a first order transfer function of the form

$$G(s) = \frac{K}{\tau s + 1} \quad (4.7)$$

to the Single Board Heater System. Because the transfer function approach uses deviation variables, $G(s)$ denotes the Laplace transform of the gain of the system between the change in heater current and the change in the system temperature. Let the change in the heater current be denoted by Δu . We denote both the time domain and the Laplace transform variable by the same lower case variable. Let the change in temperature be denoted by y . Let the current change by a step of size u . Then, we obtain the following relation between the current and the temperature.

$$y(s) = G(s)u(s) \quad (4.8)$$

$$y(s) = \frac{K}{\tau s + 1} \frac{\Delta u}{s} \quad (4.9)$$

Note that Δu is the height of the step and hence is a constant. On inversion, we obtain

$$y(s) = K[1 - e^{-\frac{t}{\tau}}]\Delta u \quad (4.10)$$

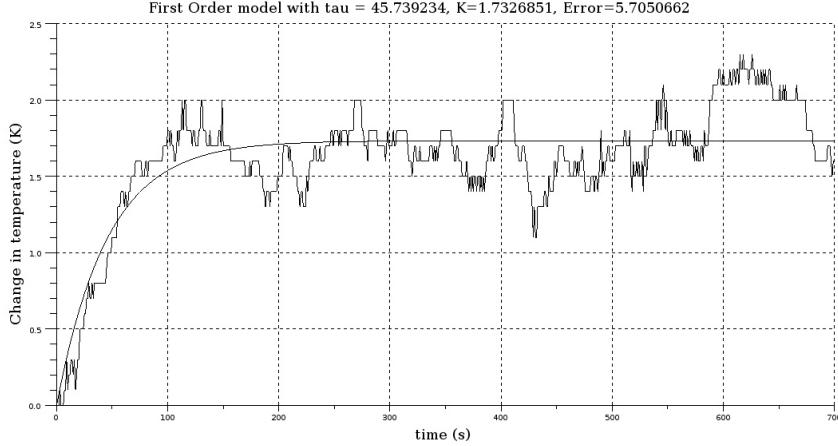


Figure 4.3: Output of the Scilab code `firstorder.sce`

4.2.1 Procedure

1. Copy the step test data file to the folder `Kp-tau-order1`.
2. Change the Scilab working directory to `Kp-tau-order1` folder under `Step Analysis` folder.
3. Open the file `firstorder.sce` and place the name of the data file with extention in the filename field.
4. Save and run this code and obtain the plot as shown in figure 4.3.

This code uses the routines `label.sci` and `costf_1.sci`

The plot thus obtained is reasonably good. See the Scilab plot to get the values of τ and K . The figure 4.3 shows a screen shot of the same. We obtain $\tau = 45.739234$, $K = 1.7326851$. The transfer function obtained here is at the operating point of 30 percentage of heat. If the experiment is repeated at a different operating point, the transfer function obtained will be different. The gain will correspondingly be more at a higher operating point. This means that the plant is faster at higher temperature. Thus the transfer function of the plant varies with the operating point. Let the transfer function we obtain in this experiment be denoted as G_s . We obtain

$$G_s(s) = \frac{1.7326851}{45.739234s + 1} \quad (4.11)$$

4.3 Determination of Second Order Transfer Function

In this section, we explore the efficacy of a second order model of the form

$$G(s) = \frac{K}{(\tau_1 s + 1)(\tau_2 s + 1)} \quad (4.12)$$

The response of the system to a step input of height Δu is given by

$$y(s) = \frac{K}{(\tau_1 s + 1)(\tau_2 s + 1)} \frac{\Delta u}{s} \quad (4.13)$$

Splitting into partial fraction expansion, we obtain

$$y(s) = \frac{K}{\tau_1 \tau_2} \frac{1}{\left(s + \frac{1}{\tau_1}\right)\left(s + \frac{1}{\tau_2}\right)} = \frac{A}{s} + \frac{B}{s + \frac{1}{\tau_1}} + \frac{C}{s + \frac{1}{\tau_2}}$$

Through Heaviside expansion method, we determine the coefficients:

$$\begin{aligned} A &= K \\ B &= -\frac{K\tau_1}{\tau_1 - \tau_2} \\ C &= \frac{K\tau_2}{\tau_1 - \tau_2} \end{aligned}$$

On substitution and inversion, we obtain

$$y(t) = K \left[1 - \frac{1}{\tau_1 - \tau_2} (\tau_1 e^{-t/\tau_1} - \tau_2 e^{-t/\tau_2}) \right] \quad (4.14)$$

We have to determine three parameters K , τ_1 and τ_2 through optimization. Once again, we follow a procedure identical to the first order model. The only difference is that we now have to determine three parameters. Scilab code `secondorder.sce` calculates the gain and two time constants.

4.3.1 Procedure

1. Change the Scilab working directory to the folder `Kp-tau-order2`.
2. Copy the step test data file to this folder.
3. Run the code `secondorder.sce` with the appropriate data file name.

The plot shown in figure 4.4 is obtained. It corresponds to the following transfer function with the parameters written at the top of the plot.

$$G_s(s) = \frac{0.420}{(34.4s + 1)(1s + 1)} \quad (4.15)$$

The fit is much better now. In particular, the initial inflexion is well captured by this second order transfer function.

4.4 Discussion

We summarize our findings now. For the first order analysis, the gain is 1.7326851 and the time constant τ is 45.739234 seconds. For the second order analysis, the initial inflexion is well captured with the two time constants $\tau_1=34.4$, $\tau_2= 1$ and gain = 0.420. Negative steps can also be introduced to make the experiment more informative. One need not keep a particular input constant. By varying both the inputs, one can imagine it to be like a step varying disturbance signal.

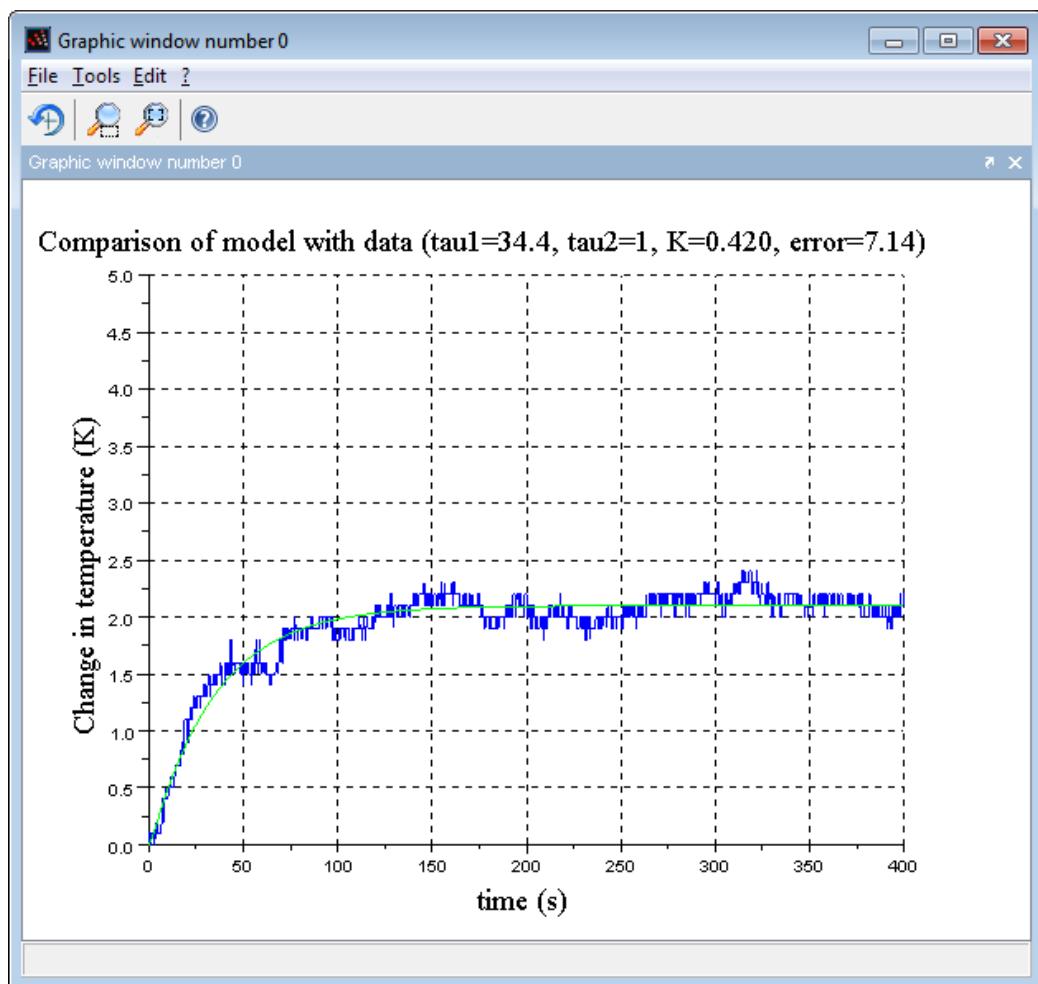


Figure 4.4: Output of the Scilab code `secondorder.sce`

4.5 Conducting Step Test on SBHS, virtually

The step by step procedure for conducting an experiment virtually is explained in section 3.5.

1. Go to virtual folder and then StepTest directory.
2. Perform the experiment using `stepc.sce`
3. Perform first order analysis using `firstorder_virtual.sce`
4. Perform second order analysis using `secondorder_virtual.sce`

The necessary codes are listed in the section 4.6.

4.6 Scilab Code

Scilab Code 4.1 `label.sci`

```
1 // Updated (9 - 12 - 06) , written by Inderpreet Arora
2 // Input arguments : title , xlabel , ylabel and their
   font sizes
3
4 function label(tname , tfont , labelx , labely , xyfont)
5 a = get("current_axes")
6 xtitle(tname , labelx , labely)
7 xgrid
8 t = a.title ;
9 t.font_size = tfont; // Title font size
10 t.font_style = 2; // Title font style
11 t.text = tname;
12
13 u = a.xlabel ;
14 u.font_size = xyfont; // Label font size
15 u.font_style = 2; // Label font style
16
17 v = a.ylabel ;
18 v.font_size = xyfont; // Label font size
19 v.font_style = 2; // Label font style
```

```

20
21 // a . l a b e l _ f o n t _ s i z e = 3 ;
22
23 endfunction ;

```

Scilab Code 4.2 costf_1.sci

```

1 function [f,g,ind] = costf_1(x,ind)
2 kp = x(1); tau = x(2);
3 y_prediction = kp * ( 1 - exp(-t / tau) );
4 f = (norm(y-y_prediction,2))^2;
5 g = numdiff(func_1,x);
6 endfunction
7
8 function f = func_1(x)
9 kp = x(1); tau = x(2);
10 y_prediction = kp * ( 1 - exp(-t / tau) );
11 f = (norm(y-y_prediction,2))^2;
12 endfunction

```

Scilab Code 4.3 firstorder.sce

```

1 mode(0)
2 funcprot(0)
3 filename = "2014-4-17-13-31-57.txt";
4
5 clf
6 exec('costf_1.sci');
7 exec('label.sci');
8 data = fscanfMat(filename);
9 time = data(:, 1);
10 heater = int(data(:, 2));
11 fan = int(data(:, 3));
12 temp = data(:, 4);
13
14 len = length(heater);
15
16

```

```

17 time2 = time - time(1);
18 // time2 = time1 / 1000;
19
20 len = length(heater);
21 heaters1 = [heater(1); heater(1:len-1)];
22 del_heat = heater - heaters1;
23 ind = find(del_heat>1);
24
25 step_instant = ind($)-1;
26
27 t = time(step_instant:len);
28 t = t - t(1);
29 H = heater(step_instant:len);
30 F = fan(step_instant:len);
31 T = temp(step_instant:len);
32 T = T - T(1);
33 delta_u = heater(step_instant + 1) - heater(
    step_instant);
34
35 // finding Kp and Tau between Heater (H) and
   Temperature (T)
36 y = T; // temperature
37 global('y','t');
38 x0 = [.3 40];
39 // [f, xopt, gopt] = optim(costf_1, 'b', [0.1 0.1], [5 100],
   x0, 'ar')
40 [f, xopt] = optim(costf_1, x0);
41 lterr=sqrt(f);
42 kp = xopt(1);
43 tau = xopt(2);
44 y_prediction = kp * (1 - exp(-t/tau));
45 plot2d(t, y_prediction);
46 plot2d(t, y);
47 // label('Showing First Order Model and Experimental
   Results', 4, 'Time (s)', 'Change in temperature (K)
   ', 4);
48
49

```

```

50 title = 'First Order model with tau = ';
51 title = title+string(tau);
52 title = title+', Kp='+string(kp/delta_u);
53 title = title+', Error='+string(1sterr)+';
54 label(title ,4 , 'time (s)', 'Change in temperature (K)'
      ,4);
55
56 kp = kp/delta_u;
57 tau;

```

Scilab Code 4.4 costf_2.sci

```

1 function [f,g,ind] = costf_2(x,ind)
2 kp = x(1); tau1 = x(2); tau2 = x(3);
3 y_prediction = kp * delta_u * (1 - ...
4 (tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
5 /(tau1-tau2));
6 f = (norm(T-y_prediction,2))^2;
7 g = numdiff(func_2,x);
8 endfunction;
9 function f = func_2(x)
10 kp = x(1); tau1 = x(2); tau2 = x(3);
11 y_prediction = kp * delta_u * (1 - ...
12 (tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
13 /(tau1-tau2));
14 f = (norm(T-y_prediction,2))^2;
15 endfunction;

```

Scilab Code 4.5 order_2_heater.sci

```

1 function 1sterr = order_2(t,H,T,limits,no)
2 x0 = [2 200 150];
3 // delta_u = u(2) - u(1); u = u - u(1); y = y - y(1);
4
5 delta_u = H(2)-H(1);
6
7

```

```

8 [f ,xopt ,gopt] = optim(costf_2 , 'b' ,[0 2 1],[18 300
9   350],x0 , 'ar' ,200,200)
10 kp = xopt(1); tau1 = xopt(2); tau2 = xopt(3); lster = 
11   sqrt(f);
12 y_prediction = kp * delta_u * (1 - ...
13   (tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
14   /(tau1-tau2));
15 format('v',6); ord = [T y_prediction]; x = [t t t];
16 // x b a s c ();
17 // p l o t 2 d (x , o r d ) , x g r i d ();
18 plot2d(t,T);
19 plot2d(t,y_prediction);
20
21 title = 'Second Order model with tau1='
22 title = title+string(tau1)+', tau2='+string(tau2)
23 title = title+', K='+string(kp)
24 title = title+', Error='+string(lster)+'
25 label(title ,4 , 'time (s)', 'Change in temperature (K)'
26   ,4);
27 endfunction;

```

Scilab Code 4.6 secondorder.sce

```

1 mode(0)
2 funcprot(0)
3 filename = "2014-4-17-13-31-57.txt"
4 clf
5 exec('costf_2.sci');
6 exec('label.sci');
7 exec ('order_2_heater.sci');

8
9
10 data = fscanfMat(filename);
11 time = data(:, 1);
12 heater = int(data(:, 2));
13 fan = int(data(:, 3));
14 temp = data(:, 4);

15

```

```

16 // times =[ time(1) ; time(1:$-1) ];
17 time2 = time - time(1);
18 // time2 = time1 / 1000;
19
20
21 // find where the step change happens
22
23 len = length(heater);
24 heaters1 = [heater(1); heater(1:len-1)];
25 del_heat = heater - heaters1;
26 ind = find(del_heat>1);
27
28 step_instant = ind($)-1;
29 t = time(step_instant:len);
30 t = t - t(1);
31 H = heater(step_instant:len);
32 F = fan(step_instant:len);
33 T = temp(step_instant:len);
34 T = T - T(1);
35
36 // limits = [0 ,0 ,1000 ,10]; no=10000; // first step
37 // limits = [400 ,0 ,900 ,26]; no=5000; // second step
38 // lterr = order_2(t,H,T,limits,no)
39 lterr = order_2(t,H,T)

```

Scilab Code 4.7 ser_init.sce

```

1 mode(0)
2 global filename
3 // ** Sampling Time */
4 sampling_time = 1;
5 // **** * * * * //
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res=init([port1 port2]);

```

```
12 disp(res)
```

Scilab Code 4.8 step_test.sci

```
1 mode(0)
2 function temp = step_test(heat,fan)
3     temp = comm(heat,fan);
4
5     plotting([heat fan temp],[0 0 20 0],[100 100 40
6         1000])
7
8     m=m+1;
9 endfunction
```

Scilab Code 4.9 stepc.sce

```
1 mode(0)
2 global fdfh fdt fncre fncrew m err_count y limits
3           sampling_time m
4 // *****
5 sampling_time=1; // In seconds. Fractions are allowed
6 // *****
7 exec ("steptest.sci");
8
9 ok = init();
10
11 if ok~= [] // open xcos only if communication is
12   through (ie reply has come from server)
13   xcos('steptest.xcos');
14 else
15   disp("NO NETWORK CONNECTION!");
16 return
17 end
```

Scilab Code 4.10 steptest.sci

```

1 function [ stop ] = steptest(heat,fan)
2
3     [ stop ,temp ] = comm(heat,fan); // Never edit this
4         line
5     plotting([heat fan temp],[0 0 25 0],[100 100 50
6         1000])
7
8 endfunction

```

Scilab Code 4.11 firstorder_virtual.sce

```

1 mode(0)
2 filename = "2014Jun17_19_18_40.txt"
3
4 clf
5 exec('costf_1.sci');
6 exec('label.sci');
7 data = fscanfMat(filename);
8 time = data(:, 5);
9 heater = int(data(:, 2));
10 fan = int(data(:, 3));
11 temp = data(:, 4);
12
13
14 len = length(heater);
15
16 time1 = time - time(1);
17 time2 = time1/1000;
18
19 len = length(heater);
20 heaters1 = [heater(1); heater(1:len-1)];
21 del_heat = heater - heaters1;
22 ind = find(del_heat>1);
23
24 step_instant = ind($)-1;
25
26 t = time2(step_instant:len);
27 t = t - t(1);

```

```

28 H = heater(step_instant:len);
29 F = fan(step_instant:len);
30 T = temp(step_instant:len);
31 T = T - T(1);
32 delta_u = heater(step_instant + 1) - heater(
    step_instant);

33
34 // finding Kp and Tau between Heater (H) and
   Temperature (T)
35 y = T; // temperature
36 global('y','t');
37 x0 = [.3 40];
38 // [f, xopt, gopt] = optim(costf_1, 'b', [0.1 0.1], [5 100],
   x0, 'ar')
39 [f, xopt] = optim(costf_1, x0);
40 lterr=sqrt(f);
41 kp = xopt(1);
42 tau = xopt(2);
43 y_prediction = kp * (1 - exp(-t/tau));
44 plot2d(t, y_prediction);
45 plot2d(t, y);
46 title = 'First Order model with tau = ';
47 title = title+string(tau);
48 title = title+', Kp='+string(kp/delta_u);
49 title = title+', Error='+string(lterr)+';
50 label(title, 4, 'time (s)', 'Change in temperature (K)'
      ,4);
51 kp = kp/delta_u
52 tau

```

Scilab Code 4.12 secondorder_virtual.sce

```

1 mode(0)
2 filename = "2014Jun17_19_18_40.txt";
3 clf
4 exec('costf_2.sci');
5 exec('label.sci');
6 exec ('order_2_heater.sci');

```

```

7
8
9  data = fscanfMat(filename);
10 time = data(:,5);
11 heater = int(data(:, 2));
12 fan = int(data(:, 3));
13 temp = data(:, 4);
14
15 // times =[ time(1) ; time(1:$-1) ];
16 time1 = time - time(1);
17 time2 = time1/1000;
18
19
20 // find where the step change happens
21
22 len = length(heater);
23 heaters1 = [heater(1); heater(1:len-1)];
24 del_heat = heater - heaters1;
25 ind = find(del_heat>1);
26
27 step_instant = ind($)-1;
28 t = time2(step_instant:len);
29 t = t - t(1);
30 H = heater(step_instant:len);
31 F = fan(step_instant:len);
32 T = temp(step_instant:len);
33 T = T - T(1);
34
35 // limits = [ 0 ,0 ,500 ,10 ]; no = 10000; // first step
36 // limits = [ 400 ,0 ,900 ,26 ]; no = 5000; // second step
37 lterr = order_2(t,H,T,)


```

Chapter 5

Identification of Transfer Function of a Single Board Heater System through Ramp Response Experiment

The aim of this experiment is to perform ramp test on the Single Board Heater System and to identify the system transfer function using ramp response data. The target group is anyone who has basic knowledge of control engineering.

5.1 About this experiment

We have used Scilab 5.3.3 and Xcos for sending and receiving data. This interface is shown in figure 5.1. Heater current and fan speed are the two inputs to the SBHS system. They are given in percentage of maximum output. These inputs can be varied through the Xcos interface by setting the properties of the input blocks in Xcos. The data acquired in the process is stored in the local drive and is available to the user for further analysis.

5.2 Theory

Identification of the transfer function of a system is important as it helps us model the physical system mathematically. Once the transfer function is obtained, one

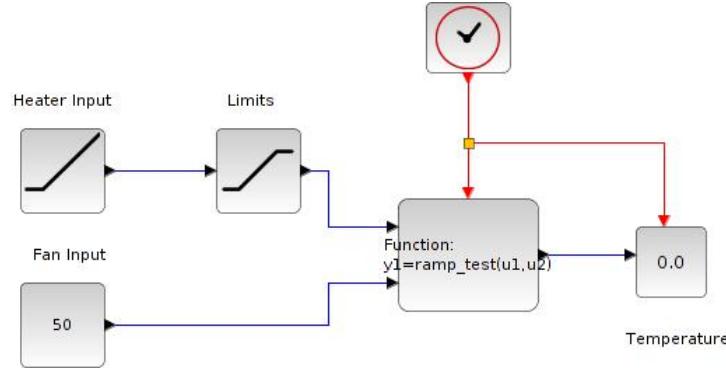


Figure 5.1: Xcos for ramp test experiment

can find out the response of the system to various inputs without actually applying them to the system.

Consider the standard first order transfer function given below

$$G(s) = \frac{C(s)}{R(s)} \quad (5.1)$$

$$G(s) = \frac{K}{\tau s + 1} \quad (5.2)$$

Combining the previous two equations, we get

$$C(s) = K \left\{ \frac{R(s)}{\tau s + 1} \right\} \quad (5.3)$$

Let us consider the case of giving a ramp input to this first order system. The Laplace transform of a ramp function with slope = v is $\frac{v}{s^2}$. Substituting $R(s) = \frac{v}{s^2}$ in equation 5.3, we obtain

$$C(s) = \frac{K}{\tau s + 1} \frac{v}{s^2} \quad (5.4)$$

$$= \frac{A}{s} + \frac{B}{s^2} + \frac{C}{\tau s + 1} \quad (5.5)$$

Solving $C(s)$ using Heaviside expansion approach, we get

$$C(s) = Kv \left\{ \frac{1}{s^2} - \frac{\tau}{s} + \frac{\tau^2}{\tau s + 1} \right\} \quad (5.6)$$

Taking the Inverse Laplace transform of the above equation, we get

$$c(t) = Kv \left\{ t - \tau + \tau e^{\frac{-t}{\tau}} \right\} \quad (5.7)$$

The difference between the reference and output signal is the error signal $e(t)$. Therefore,

$$e(t) = r(t) - c(t) \quad (5.8)$$

$$e(t) = Kv t - Kv t + Kv \tau - Kv \tau e^{\frac{-t}{\tau}} \quad (5.9)$$

$$e(t) = Kv \tau (1 - e^{\frac{-t}{\tau}}) \quad (5.10)$$

Normalizing equation 5.10 for $t \gg \tau$, we get

$$e(t) = \tau \quad (5.11)$$

This means that the error in following the ramp input is equal to τ for large value of t [7]. Hence, smaller the time constant τ , smaller the steady state error.

5.3 Procedure to perform Ramp Test

Follow the procedure explained in section 2.1.2.

1. Change the Scilab working directory to Ramp_Test folder.
2. Execute the code `ramp_test.sci` instead of `step_test.sci`.
3. Open the Xcos file `ramp_test.xcos`. Give a ramp input to the system with some value for slope. For this experiment, we have chosen slope = 0.1.
4. Double click on the ramp input block labeled as “Heater input”. Change the following values in the respective fields- slope = 0.1, start time = 200, initial output = 20.
5. Keep the fan constant at 100.

The first column of table 5.1 denotes samples. The second column denotes heater in percentage. The third column denotes the fan. It has been held constant at 100 units. The last column denotes the plate temperature.

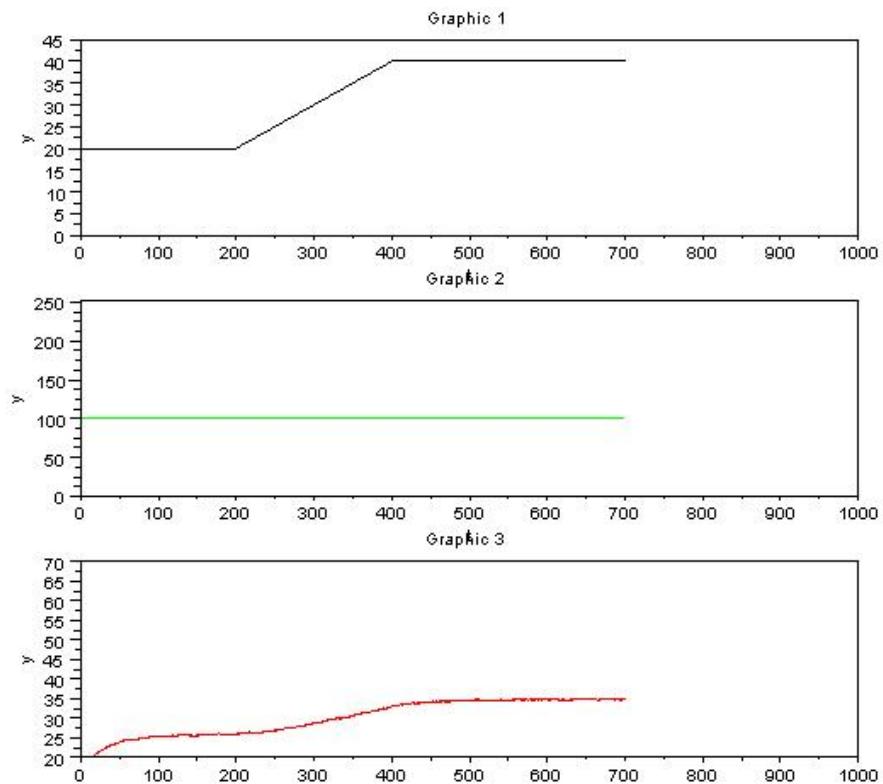


Figure 5.2: Screen shot of ramp test experiment

```

0.000E+00  0.100E+02  0.100E+03  0.216E+02
0.100E+00  0.100E+02  0.100E+03  0.216E+02
.
.
.
0.251E+03  0.300E+02  0.100E+03  0.291E+02
0.251E+03  0.300E+02  0.100E+03  0.291E+02

```

Table 5.1: Ramp data obtained after performing the ramp test

5.4 Ramp Test Analysis

After completing the ramp test experiment, let us do the analysis.

1. Change the directory to `Ramp Analysis`.
2. Execute the file `ramp.sce`.

On executing this file, you get the values of K_p , τ and K_p approx and τ approx on the Scilab plot. You will also get a plot of the ramp response calculated using the equation 5.7 for K_p and τ values.

i401, $K_p = 0.1755018$, $\tau_{\text{Approx}} = 35.184915$, $K_p_{\text{Approx}} = 0.1755$

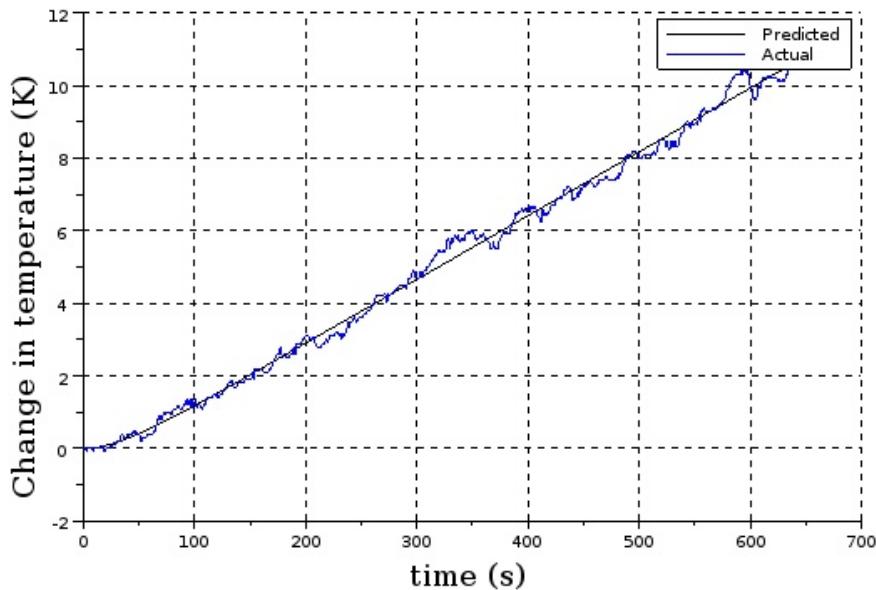


Figure 5.3: Ramp response for K_p and τ

5.5 Discussion

We summarize our findings now. The experiment has been performed by varying the heater current and keeping the fan speed constant. However, the user is encouraged to experiment using different combinations of fan speed and heater

current. Negative ramp can also be used to make the experiment more informative. It is not necessary to keep a particular input constant. For example, you can try giving a step input to the disturbance signal, i.e., the fan input. The system can also be treated as a second order system. This consideration is necessary as it increases the accuracy of the acquired transfer function [6].

5.6 Conducting ramp test on SBHS, virtually

The step by step procedure for conducting an experiment virtually is explained in section 3.5.

1. Go to virtual folder and then RampTest directory
2. Execute `ramptest.sce`
3. Perform ramp test analysis by executing `ramp_virtual.sce` under Ramp_Analysis directory

The necessary codes are listed in the section 5.7.

5.7 Scilab Code

Scilab Code 5.1 ramp_test.sci

```
1 mode(0)
2 function temp = ramp_test(heat ,fan )
3     temp = comm(heat ,fan );
4
5     plotting ([heat  fan  temp ],[0  0  20  0],[100  100  40
6                               1000])
7
8     m=m+1;
9 endfunction
```

Scilab Code 5.2 label.sci

```
1 mode(-1);
2 // Updated (9 - 12 - 06) , written by Inderpreet Arora
```

```

3 // Input arguments : title , xlabel , ylabel and their
4   font sizes
5
6 function label(tname, tfont, labelx, labely, xyfont)
7 a = get("current_axes")
8 xtitle(tname, labelx, labely)
9 xgrid
10 t = a.title;
11 t.font_size = tfont; // Title font size
12 t.font_style = 2; // Title font style
13 t.text = tname;
14
15 u = a.x_label;
16 u.font_size = xyfont; // Label font size
17 u.font_style = 2; // Label font style
18
19 v = a.y_label;
20 v.font_size = xyfont; // Label font size
21 v.font_style = 2; // Label font style
22 // a.label_font_size = 3;
23
24 endfunction;

```

Scilab Code 5.3 cost.sci

```

1 function f = func_1(x)
2   k = x(1);
3   tau = x(2);
4   y_prediction = k*(t + tau*(exp(-t/tau) - 1));
5   f = (norm(y - y_prediction, 2))^2;
6 endfunction
7
8 function [f, g, ind1] = cost(x, ind1)
9   k = x(1);
10  tau = x(2);
11  y_prediction = k*(t + tau*(exp(-t/tau) - 1));
12  f = (norm(y - y_prediction, 2))^2;

```

```
13     g = numdiff(func_1,x);  
14 endfunction
```

Scilab Code 5.4 cost_approx.sci

```
1 function f = func_approx(x)  
2     k = x(1);  
3     tau = x(2);  
4     y_p_approx = k*(t_approx - tau);  
5     f = (norm(y_approx - y_p_approx,2))^2;  
6 endfunction  
7  
8 function [f,g,ind] = cost_approx(x,ind)  
9     k = x(1);  
10    tau = x(2);  
11    y_p_approx = k*(t_approx - tau);  
12    f = (norm(y_approx - y_p_approx,2))^2;  
13    g = numdiff(func_approx,x);  
14 endfunction
```

Scilab Code 5.5 ramptest.sci

```
1 function [stop] = ramptest(heat,fan)  
2  
3     [stop,temp] = comm(heat,fan); // Never edit this  
4         line  
4     plotting([heat fan temp]);  
5  
6 endfunction
```

Scilab Code 5.6 ramptest.sce

```
1 mode(0)  
2 global fdfh fdt fncre fncw m err_count y limits  
3         sampling_time m  
4 // *****
```

```

5 sampling_time=1; // In seconds. Fractions are allowed
6 // *****
7 exec ("ramptest.sci");
8
9 ok = init();
10
11 if ok~= [] // open xcos only if communication is
   through (ie reply has come from server)
12   xcos('ramptest.xcos');
13 else
14   disp("NO NETWORK CONNECTION!");
15 return
16 end

```

Scilab Code 5.7 ramp_virtual.sce

```

1 mode(-1);
2
3 // filename = "20 Apr 2012_15_10_35.txt"; // complete
   path of the saved data file
4 filename ="29Apr2014_14_32_06.txt";
5 slope = 0.1; // change this to the slope that you have
   used in the experiment
6 ind1=3;
7 // Ramp Analysis
8 exec('cost_approx.sci');
9 exec('cost.sci');
10 exec('label.sci');

11
12 data = fscanfMat(filename);
13 time = data(:, 5);
14 heater = int(data(:, 2));
15 fan = int(data(:, 3));
16 temp = data(:, 4);

17
18 len = length(heater);
19 heaters1 = [heater(1); heater(1:$-1)];

```

```

21 del_heat = abs(heater - heaters1);
22 ind = find(del_heat >.5);
23
24 t = time(ind(2):ind($-1));
25 t=t/1000
26 H = heater(ind(2):ind($-1));
27 T = temp(ind(2):ind($-1));
28
29 t = t - t(1);
30 T = T - T(1);
31
32 y = T;
33 x0 = [.5 100]
34 global('y','t');
35
36 [f, xopt] = optim(cost,x0);
37 kp = xopt(1)/slope
38 tau = xopt(2)
39
40 len = length(t);
41 halfway = ceil(len/2);
42
43 t_approx = t(halfway:len);
44 y_approx = y(halfway:len);
45 global('y_approx','t_approx');
46
47 [f_approx,xopt_approx] = optim(cost_approx,x0);
48 kp_approx = xopt_approx(1)/slope;
49 tau_approx = xopt_approx(2);
50
51 // Display and Plot
52 disp('kp = ');
53 disp(kp);
54 disp('tau = ');
55 disp(tau);
56 disp('kp_approx = ');
57 disp(kp_approx);
58 disp('tau_approx = ');

```

```
59 disp(tau_approx);
60
61 y_p = kp*slope*(t + tau*(exp(-t/tau) - 1));
62 y_p_approx = kp_approx*slope*(t_approx - tau_approx);
63 y_p_approx = y_p_approx';
64 plot2d(t,[y_p,T]);
65 label('Showing First Order Model and Experimental
    Results for kp and tau',4,'Time (s)', 'Change in
    Temperature (Predicted ,Actual)',4);
66 legend(['Predicted';'Actual']);
```

Chapter 6

Frequency Response Analysis of a Single Board Heater System by the Application of Sine Wave

The aim of this experiment is to do a frequency response analysis of a Single Board Heater System by the application of sine wave. The target group is anyone who has basic knowledge of control engineering.

We have used Scilab with Xcos as an interface for sending and receiving data. This interface is shown in figure 6.1. Heater current and fan speed are the two inputs to the system. The heater current is varied sinusoidally. A provision is made to set the parameters related to it like frequency, amplitude and offset. The temperature profile thus obtained is the output.

In this experiment we are applying a sine change in the heater current by keeping the fan speed constant. After application of sine change, wait for sufficient amount of time to allow the temperature to reach a steady-state.

6.1 Theory

Frequency response of a system means its steady-state response to a sinusoidal input. For obtaining a frequency response of a system, we vary the frequency of the input signal over a spectrum of interest. The analysis is useful and simple because it can be carried out with the available signal generators and measuring devices.

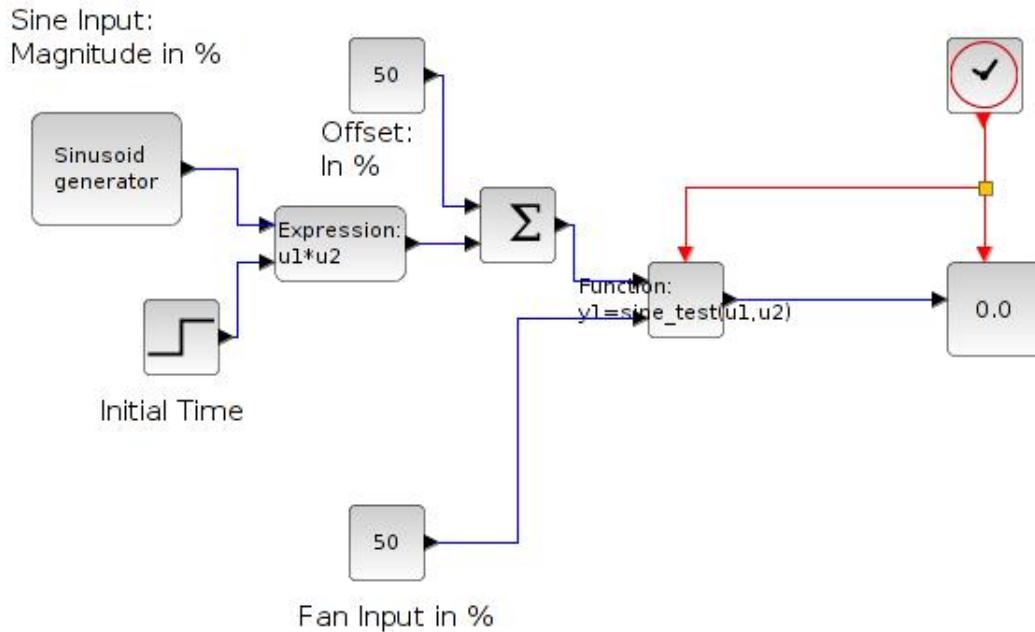


Figure 6.1: Xcos for this experiment

Consider a sinusoidal input

$$U(t) = A \sin \omega t \quad (6.1)$$

The Laplace transform of the above equation yields

$$U(s) = \frac{A\omega}{s^2 + \omega^2} \quad (6.2)$$

Consider the standard first order transfer function given below

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K}{s + 1} \quad (6.3)$$

Replacing the value of $U(s)$ from equation 6.2, we get

$$Y(s) = \frac{KA\omega}{(\tau s + 1)(s^2 + \omega^2)} \quad (6.4)$$

$$= \frac{KA}{\omega^2 \tau^2 + 1} \left[\frac{\omega \tau^2}{\tau s + 1} - \frac{\tau s \omega}{s^2 + \omega^2} + \frac{\omega}{s^2 + \omega^2} \right] \quad (6.5)$$

Taking Laplace Inverse, we get

$$y(t) = \left[\frac{KA}{\omega^2\tau^2 + 1} \right] \left[\omega\tau e^{\frac{-t}{\tau}} - \omega\tau \cos(\omega t) + \sin(\omega t) \right] \quad (6.6)$$

The above equation has an exponential term $e^{\frac{-t}{\tau}}$. Hence, for large value of time, its value will approach to zero and the equation will yield a pure sine wave. One can also use trigonometric identities to make the equation look more simple.

$$y(t) = \left[\frac{KA}{\sqrt{\omega^2\tau^2 + 1}} \right] [\sin(\omega t) + \phi] \quad (6.7)$$

where,

$$\phi = -\tan^{-1}(\omega\tau) \quad (6.8)$$

By observing the above equation, one can easily make out that for a sinusoidal input the output is also sinusoidal but has some phase difference. Also, the amplitude of the output signal, \hat{A} , has become a function of the input signal frequency, ω .

$$\hat{A} = \frac{KA}{\sqrt{\omega^2\tau^2 + 1}} \quad (6.9)$$

The amplitude ratio (AR) can be calculated by dividing both sides by the input signal amplitude A.

$$AR = \frac{\hat{A}}{A} = \frac{K}{\sqrt{\omega^2\tau^2 + 1}} \quad (6.10)$$

Dividing the above equation by the process gain K yields the normalized amplitude ratio (AR_n)

$$AR_n = \frac{AR}{K} = \frac{1}{\sqrt{\omega^2\tau^2 + 1}} \quad (6.11)$$

Because the process steady state gain is constant, the normalized amplitude ratio is often used for frequency response analysis [8].

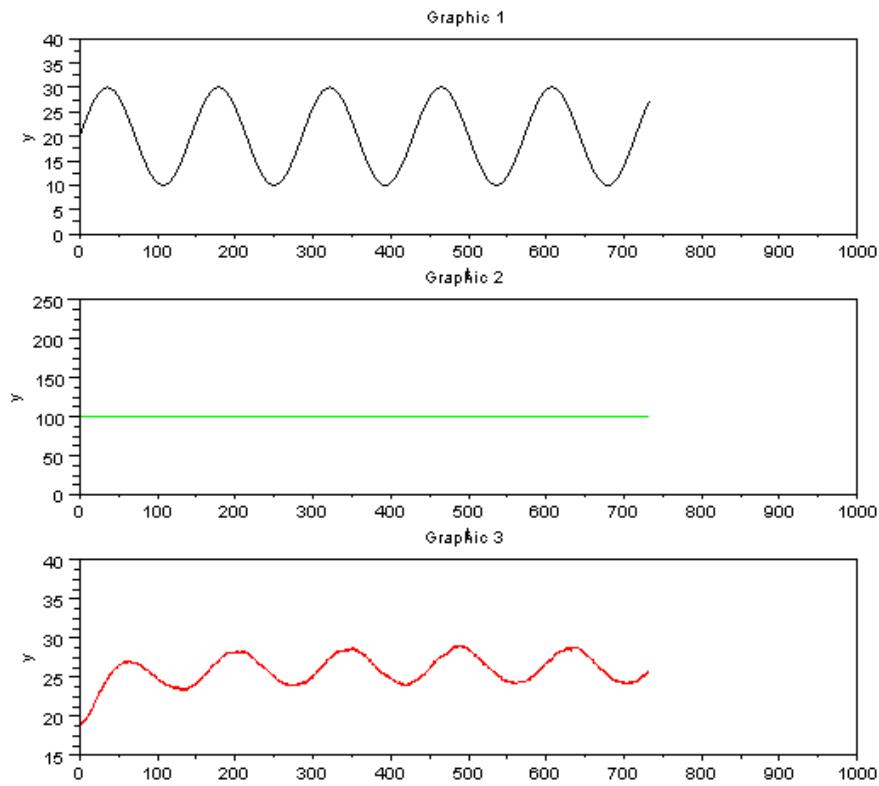


Figure 6.2: Plot for sine input 0.007Hz

6.2 Procedure to perform Sine Test

Follow the procedure explained in section 2.1.2.

1. Change the current working directory of Scilab to the folder **Sine_Test**.
2. Execute the code **sinetest.sce** and **sinetest.sci**.
3. Open the Xcos file **sine_test.xcos**.
4. Initiate a sine input to the system by setting sinusoid generator block properties with some value of the frequency (0.007Hz) and amplitude (10).

Note that at high frequencies the plant output is not sinusoidal, which is not of any use. Hence, avoid choosing frequencies above 0.04Hz .

0.100E+00	0.200E+02	0.100E+03	0.239E+02
0.200E+00	0.201E+02	0.100E+03	0.238E+02
0.300E+00	0.201E+02	0.100E+03	0.238E+02
.	.	.	.
.	.	.	.
0.749E+03	0.300E+02	0.100E+03	0.301E+02
0.749E+03	0.300E+02	0.100E+03	0.302E+02
0.749E+03	0.300E+02	0.100E+03	0.302E+02

Table 6.1: Data obtained after application of sine input of 0.04Hz

The sine test data file is shown in table 6.1. Referring to table 6.1 the first column represents samples. The second column represents heater in percentage. Here, it is sinusoidally varied. The third column represents fan in percentage. Note that its value is 100 throughout the experiment. The fourth column represents the output temperature. It should be taken into consideration that all the values mentioned in the data file are in percentage of maximum output, except for the temperature which is in $^{\circ}\text{C}$.

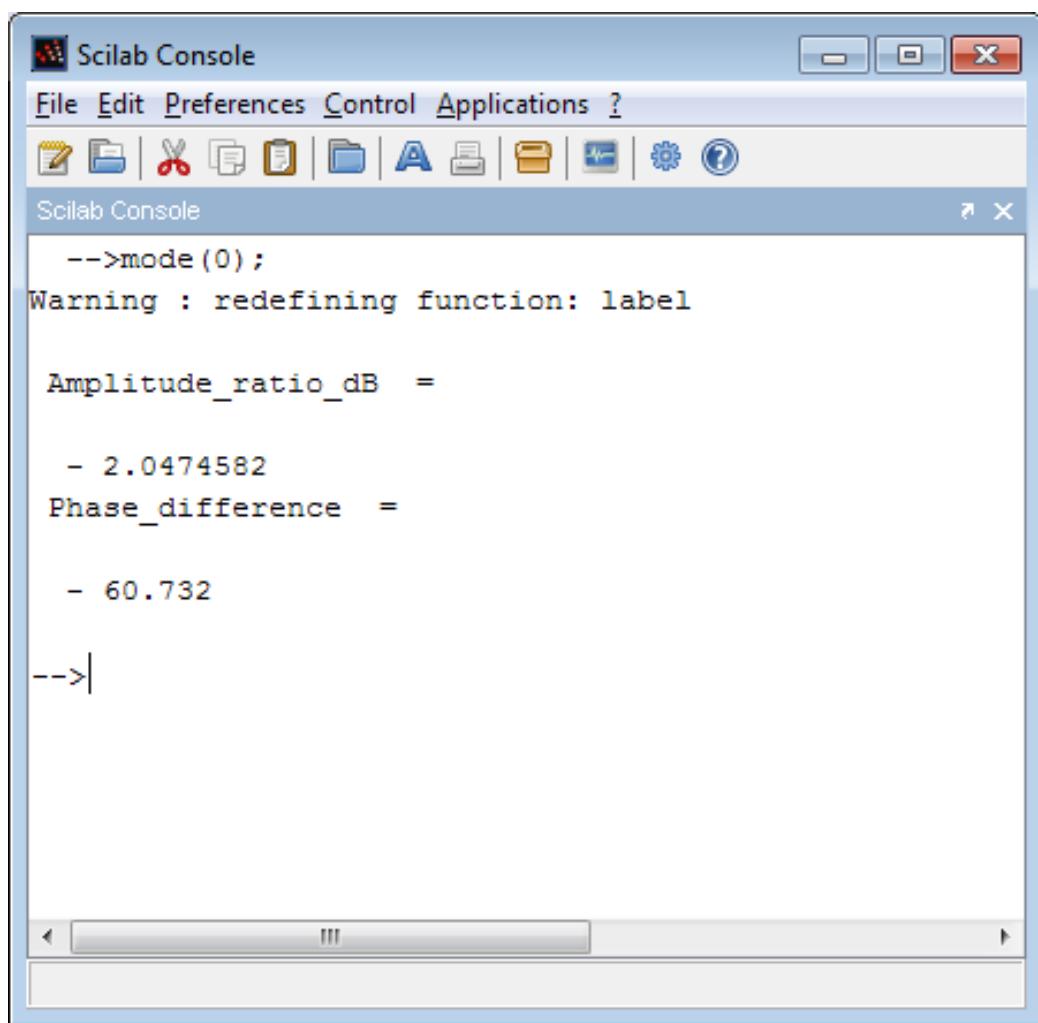
6.3 Sine Test Analysis

Now let us calculate amplitude ratio and phase difference.

1. Change the current working directory of Scilab to the folder **Sine_Analysis**.
2. Copy the data files generated after the completion of the experiment into the **Sine_Analysis** folder.
3. Place the arguments **f** and **filename** in the Scilab code **sine2.sce** for the calculation of the above parameters and execute it. Here **f** means input frequency.

It could be seen from figure 6.3 that the amplitude ratio turns out to be -2.047dB and phase difference to be -60.732° . The plot thus obtained is shown in figure 6.4

Repeat this calculation over a range of frequencies and note down the values of amplitude ratio in dB and phase difference. Input these values for the appropriate



The image shows a screenshot of the Scilab Console window. The title bar reads "Scilab Console". The menu bar includes "File", "Edit", "Preferences", "Control", "Applications", and "?". The toolbar below the menu contains icons for file operations like Open, Save, Print, and others. The main console area displays the following text:

```
-->mode(0);  
Warning : redefining function: label  
  
Amplitude_ratio_dB =  
  
- 2.0474582  
Phase_difference =  
  
- 60.732  
-->|
```

Figure 6.3: Scilab Output

Plot of sine input in heater and the corresponding temperature profile

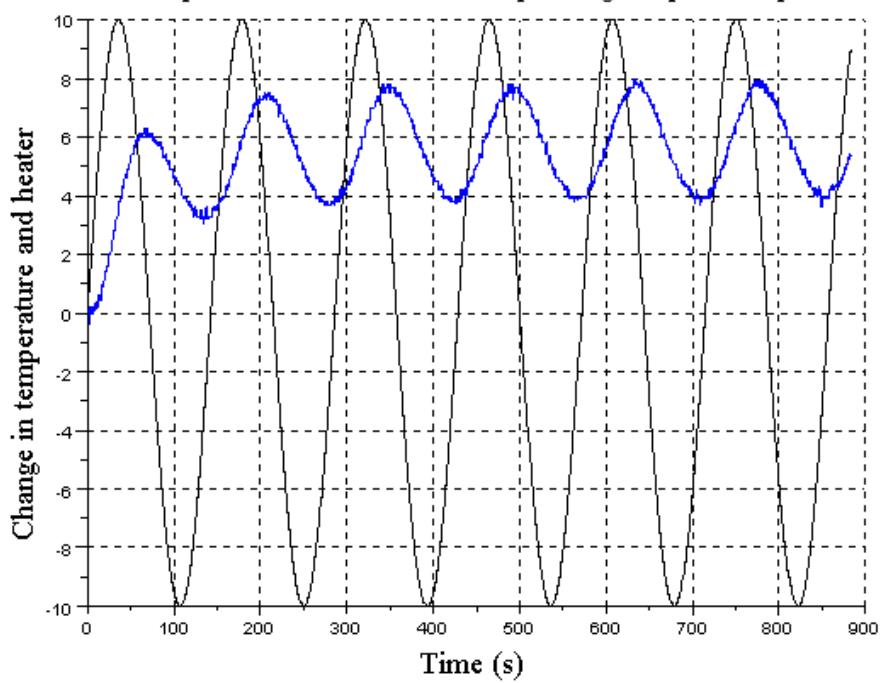


Figure 6.4: Plot of Input and Output vs time

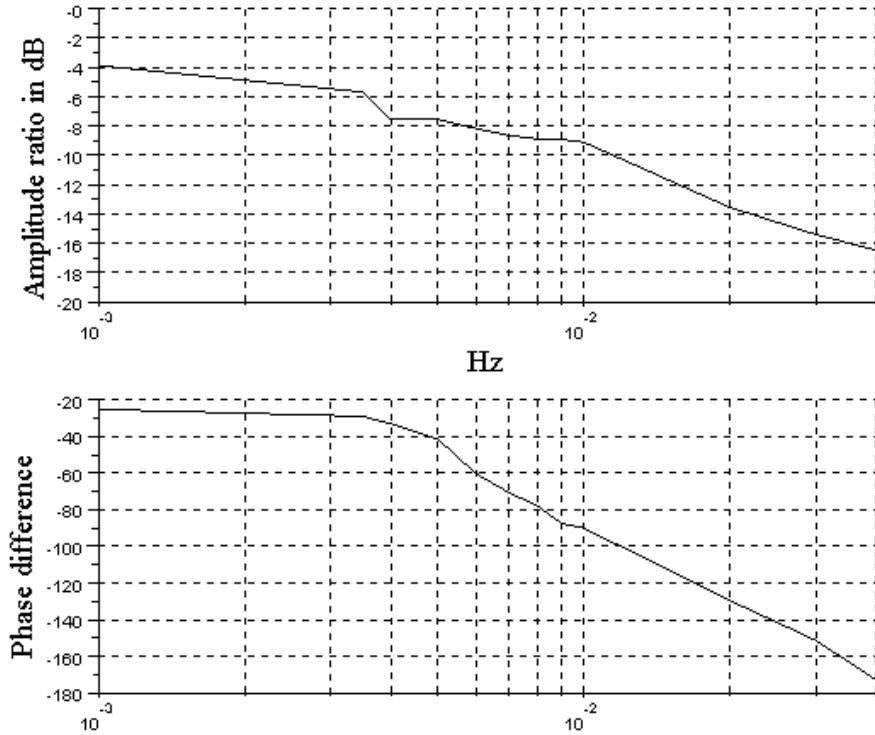


Figure 6.5: Bode plot obtained from the plant

frequencies into the Scilab code `TFbode.sce` and execute it to get a Bode plot of the plant which is illustrated in figure 6.5.

Bode plot can be obtained directly from the plant's second order transfer function [6] with the help of Scilab code `TFbode.sce`, as shown in figure 6.6. A visual comparison of the two Bode plots can be done to validate the Bode diagram obtained from the plant.

To compare the two plots, we plot it on the same graph as shown in figure 6.7

6.4 Conducting Sine Test on SBHS, virtually

The step by step procedure for conducting an experiment virtually is explained in section 3.5.

1. Go to virtual folder and then `SineTest` directory .

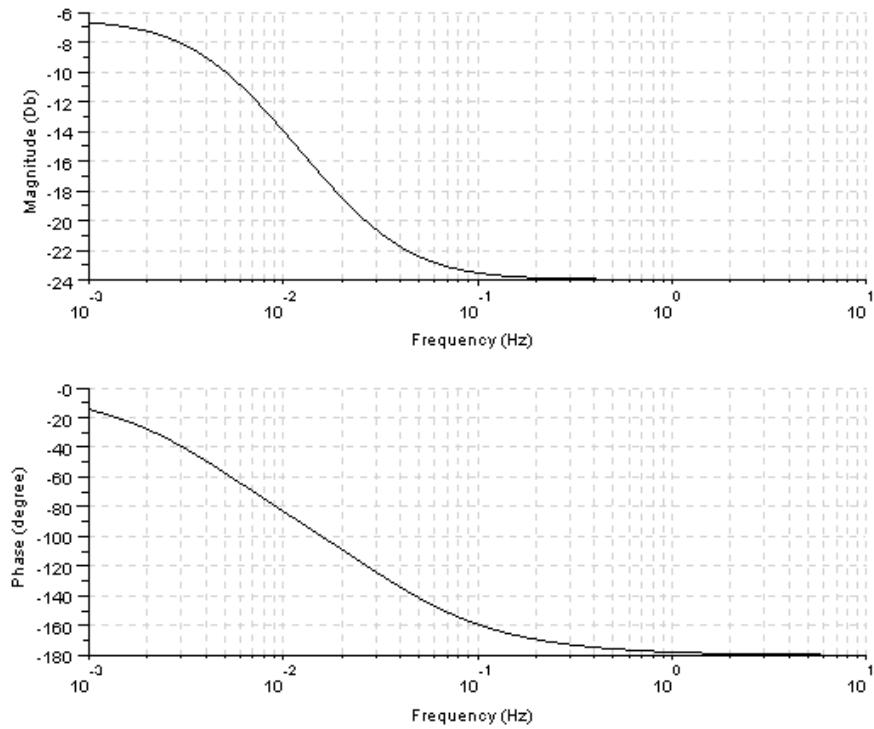


Figure 6.6: Bode plot obtained through plant's transfer function

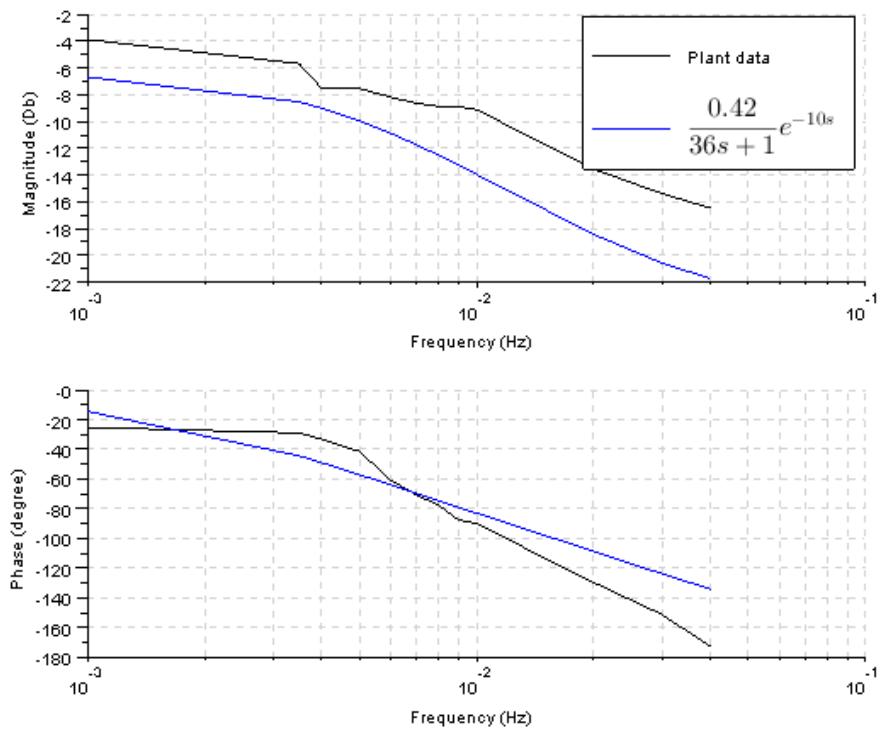


Figure 6.7: Comparison of Bode plots

2. Execute `sinetest.sce`.
3. Perform sine test analysis by executing the file `sine2_virtual.sce` in `Sine_Analysis` directory under `virtual` folder.

The necessary codes are listed in the section 6.5.

6.5 Scilab Code

Scilab Code 6.1 sine_test.sci

```

1 mode(0)
2 function temp = sine_test(heat,fan)
3     temp = comm(heat,fan);
4
5     plotting([heat fan temp],[0 0 20 0],[100 100 40
1000])
6
7     m=m+1;
8 endfunction

```

Scilab Code 6.2 sinetest.sce

```

1 mode(0)
2 global fdfh fdt fngr fncl m err_count y limits
sampling_time m
3
4 // *****
5 sampling_time=1; // In seconds. Fractions are allowed
6 // *****
7 exec ("sinetest.sci");
8
9 ok = init();
10
11 if ok~= [] // open xcos only if communication is
through (ie reply has come from server)
12 xcos('sinetest.xcos');
13 else

```

```

14     disp ("NO NETWORK CONNECTION!");  

15     return  

16 end

```

Scilab Code 6.3 sinetest.sci

```

1 function [ stop ] = sinetest(heat , fan )  

2  

3     [ stop , temp ] = comm(heat , fan ); // Never edit this  

4         line  

5     plotting ([ heat  fan  temp ]);  

6  

7 endfunction

```

Scilab Code 6.4 sine2.sce

```

1 mode(0);  

2 // filename= ' sine001 ' ; // Enter the data file name in  

3 // single quotes  

4 filename= ' sine01 ' ; // Enter the data file name in  

5 // single quotes  

6 f=0.006; // Enter the frequency  

7 data7=fscanfMat(filename);  

8 exec('labelbode.sci');  

9 T = data7(:,1); fan = data7(:,3); // T is time , fan is  

10 fan speed  

11 u = data7(:,2)-data7(1,2); y = data7(:,4)-data7(1,4);  

12 // u is current , y is temperature  

13 period=ceil(1/f);  

14 p=length(u);  

15 sampling = T(3)-T(2); // sampling time  

16 index = round((period)/sampling); // calculating the  

17 duration of last cycle of waveform  

18 times=T($-index:$);  

19 temp = y($-index:$); // output for last cycle  

20 heater = u($-index:$); // input for last cycle

```

```

18 [ max_heater , pointer1 ] = max( heater ); // determining max
    amplitude and index for last cycle of input ( index
    is relative to last cycle )
19 [ max_temp , pointer2 ] = max( temp ); // determining max
    amplitude and index for last cycle of input ( index
    is relative to last cycle )
20 pointer1 = pointer1 + (p-index); // conversion of index
    for input in terms of complete data period
21 pointer2 = pointer2 + (p-index); // conversion of index
    for output in terms of complete data period
22 Amplitude_ratio_dB = 20*log10(y(pointer2)/u(pointer1))
    // To find gain in dB
23 Phase_difference = 360*f*(pointer1-pointer2)*sampling
    // phase difference in degrees
24 // Phase_difference = -((pointer1 - pointer2) / (1 / f)) * 360
25
26 plot2d(T,[u y]);
27 label('Plot of sine input in heater and the
    corresponding temperature profile',4,'Time (s)','
    Change in temperature and heater',4);
28 // legend(['Heater';'Temperature']);

```

Scilab Code 6.5 label.sci

```

1 // Updated (9-12-06), written by Inderpreet Arora
2 // Input arguments : title , xlabel , ylabel and their
    font sizes
3 function label(tname , tfont , labelx , labely , xyfont)
4 a = get("current_axes")
5 xtitle(tname , labelx , labely)
6 xgrid
7 t = a.title;
8 t.font_size = tfont; // Title font size
9 t.font_style = 2; // Title font style
10 t.text = tname;
11 u = a.xlabel;
12 u.font_size = xyfont; // Label font size
13 u.font_style = 2; // Label font style

```

```

14 v = a.y_label;
15 v.font_size = xyfont; // Label font size
16 v.font_style = 2; // Label font style
17 // a.label_font_size = 3;
18 endfunction;

```

Scilab Code 6.6 bodeplot.sce

```

1 // bodeplot
2 exec('labelbode.sci');
3 x=[0.001,0.0035,0.004,0.005,0.006,0.007,...
4 0.008,0.009,0.01,0.02,0.03,0.04]; // Input frequency (Hz)
5 y=[-3.87,-5.67,-7.53,-7.53,-8.17,-8.64,...
6 -8.87,-8.90,-9.11,-13.55,-15.39,-16.47]; // Amplitude
      ratio (dB)
7 subplot(2,1,1);
8 plot2d(x,y,rect=[0.001,-20,0.04,0],logflag="ln");
9 xgrid();
10 y=[-25.2,-28.98,-33.11,-41.4,-60.48,-70.56,...
11 -77.76,-87.48,-90,-129.6,-151.2,-172.8]; // Phase
      difference (degree)
12 title = ''
13 label(title,4,'Hz','Amplitude ratio in dB',4);
14 subplot(2,1,2);
15 plot2d(x,y,rect=[0.001,-180,0.04,-20],logflag="ln");
16 label(title,4,'','Phase difference',4);
17 subplot(2,1,2);
18 xgrid();
19
20 // s = poly(0,'s')
21 // h = syslin('c',(0.475/(124.827*s^2+57.26*s+1)))
22 // bode(h,0.001,0.04);

```

Scilab Code 6.7 labelbode.sci

```

1 // Updated (9-12-06), written by Inderpreet Arora
2 // Input arguments : title , xlabel , ylabel and their
      font sizes

```

```

3
4 function label(tname , tfont , labelx , labely , xyfont)
5 a = get("current_axes")
6 xtitle(tname , labelx , labely)
7 xgrid
8 t = a.title ;
9 t.font_size = tfont; // Title font size
10 t.font_style = 2; // Title font style
11 t.text = tname;
12 u = a.x_label;
13 u.font_size = xyfont; // Label font size
14 u.font_style = 2; // Label font style
15 v = a.y_label;
16 v.font_size = xyfont; // Label font size
17 v.font_style = 2; // Label font style
18 // a.label_font_size = 3;
19 endfunction;

```

Scilab Code 6.8 TFbode.sce

```

1 s=poly(0 , 's ')
2 dt=10; // delay time
3 // h = syslin ('c' , ((0.510/(65.49*s+1))) ) // transfer
   function using first order pade ' approximation
4 tf=((0.475/(36*s+1))*((-dt/2)*s+1/(dt/2)*s+1));
5 bode(h,0.001,10);

```

Scilab Code 6.9 comparison.sce

```

1 s=poly(0 , 's ');
2 frq = [0.001,0.0035,0.004,0.005,0.006,0.007, ...
3 0.008,0.009,0.01,0.02,0.03,0.04]; // Input frequency (Hz)
4 dt=10; // delay time
5
6 tf=((0.475/(36*s+1))*((-dt/2)*s+1/(dt/2)*s+1)); //
   transfer function using pade ' approximation
7 h=syslin('c',tf);
8

```

```

9 [frq1,rep]=repfreq(h,frq);
10 [dB1,phi1]=dbphi(rep);
11 title = 'From actual plant data';
12 dB = [-3.87,-5.67,-7.53,-7.53,-8.17,-8.64, ...
13 -8.87,-8.90,-9.11,-13.55,-15.39,-16.47]; // Amplitude
14 ratio (dB)
15 phi = [-25.2,-28.98,-33.11,-41.4,-60.48, ...
16 -70.56,-77.76,-87.48,-90,-129.6,-151.2,-172.8]; // Phase
17 difference (degree)
18 bode([frq],[dB;dB1],[phi;phi1])
19 legend(['Plant data';'$\frac{0.42}{36s+1}e^{-10s}$'])
20
21 // transfer function using pade approximation

```

Scilab Code 6.10 sine2_virtual.sce

```

1 mode(0);
2 filename= '29Apr2014_14_03_40.txt'; // Enter the data
3 file name in single quotes
4 f=0.01; // Enter the frequency
5 data6=fscanfMat(filename);
6 data7=data6(2:$,:);
7 exec('labelbode.sci');
8 T = data7(:,5); fan = data7(:,3); // T is time, fan is
fan speed
9 u = data7(:,2)-data7(1,2); y = data7(:,4)-data7(1,4);
// u is current, y is temperature
10
11 period=ceil(1/f);
12 p=length(u);
13 sampling = T(3)-T(2); // sampling time
14 sampling = sampling/1000;
15 index = round((period)/sampling); // calculating the
duration of last cycle of waveform
16 times=T($-index:$);
17 temp = y($-index:$); // output for last cycle
18 heater = u($-index:$); // input for last cycle

```

```

19 [ max_heater , pointer1 ] = max( heater ); // determining max
    amplitude and index for last cycle of input ( index
    is relative to last cycle )
20 [ max_temp , pointer2 ] = max( temp ); // determining max
    amplitude and index for last cycle of input ( index
    is relative to last cycle )
21 pointer1 = pointer1 + (p-index); // conversion of index
    for input in terms of complete data period
22 pointer2 = pointer2 + (p-index); // conversion of index
    for output in terms of complete data period
23 Amplitude_ratio_dB = 20*log10(y(pointer2)/u(pointer1))
    // To find gain in dB
24 Phase_difference = 360*f*(pointer1-pointer2)*sampling
    // phase difference in degrees
25 // Phase_difference = -((pointer1 - pointer2) / (1 / f)) * 360
26
27
28 del_T = T-T(1);
29 del_T = del_T/1000;
30 plot2d(del_T,[u y]);
31 label('Plot of sine input in heater and the
    corresponding temperature profile',4,'Time (s)','
    Change in temperature and heater',4);
32 // legend(['Heater','Temperature']);

```

Chapter 7

Controlling Single Board Heater System using PID controller

The aim of this experiment is to apply a PID controller to the Single Board Heater System. The target group is anyone who has basic knowledge of control engineering.

Scilab is used with Xcos as an interface for sending and receiving data. This interface is shown in figure 7.1. Heater current and fan speed are the two inputs to the system. The inputs are provided in percentage of maximum output. The parameters related to PID controller (K, τ_i, τ_d) can be set in Xcos. In this experiment, the fan speed is kept constant. The output temperature profile, read by the sensor, is also plotted. The data acquired in the process is stored on the local drive and is available to the user for further calculations.

7.1 Theory

A PID controller tries to minimize the error between measured variable and the setpoint by calculating the error and then taking a suitable corrective action. Note that the output of interest is called the measured variable or process variable, the difference between the setpoint and the measured variable is called the error and the control action taken to minimize the error is given as input to the process in the form of the manipulated variable. A PID controller does not simply add or subtract the error in order to calculate control action but instead uses three distinct control features, namely, Proportional, Integral and Derivative. Thus, a PID controller has three separate parameters.

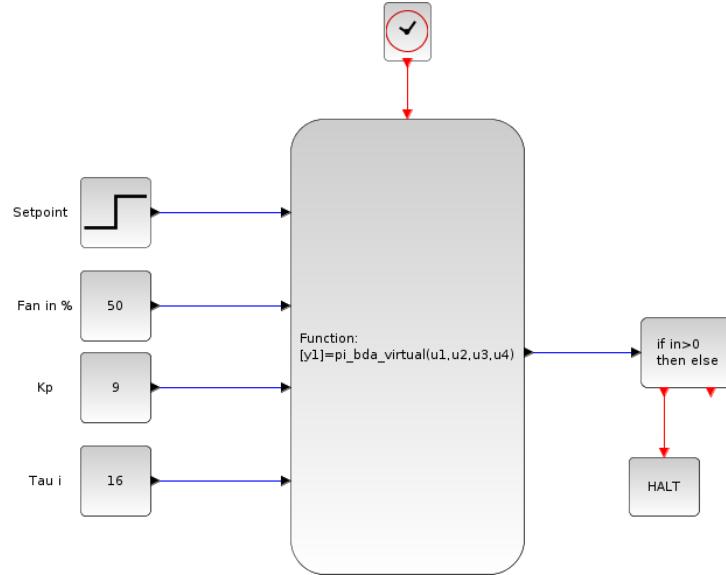


Figure 7.1: Xcos interface for this experiment

7.1.1 Proportional Control Action

This parameter generates a control action based on the current value of the error. In a more simplified sense, if the error is +2, the control action is -2. The proportional action can be generated by multiplying the error with a proportional constant- K_p . Mathematical representation of the same is given below,

$$P = K_p e(t) \quad (7.1)$$

where,

P is the proportional output

K_p is the proportional gain

$e(t)$ is the error signal

The value of K_p is very important. A large value of K_p may lead to instability of the system. In contrast, a smaller value of K_p may decrease the controller's sensitivity towards error. The problem involved in using only proportional action is that the control action will never settle down to its target value and will always retain a steady-state error.

7.1.2 Integral Control Action

This parameter generates a control action depending on the history of errors. It means that the action is based on the sum of the recent errors. It is proportional to both the magnitude as well as duration of the error. The summation of the error over a period of time gives a value of the offset that should have been corrected previously. The integral action can thus be generated by multiplying this accumulated error with an integral gain K_i . Mathematical representation of the same is given below.

$$I = K_i \int_0^t e(t)dt \quad (7.2)$$

where,

I is the integral output

K_i is the integral gain ($K_i = K_p/\tau_i$, where, τ_i is the integral time)

The integral action tends to accelerate the control action. However, since it looks only at the past values of the error, there is always a possibility of it causing the present values to overshoot the setpoint values.

7.1.3 Derivative Control Action

As the name suggests, a derivative parameter generates a control action by calculating the rate of change of error. A derivative action is thus generated by multiplying the value of rate of change of error with a derivative gain K_d . Mathematical representation of the same is given below.

$$D = K_d \frac{d}{dt} e(t) \quad (7.3)$$

where,

D is the derivative output

K_d is the derivative gain ($K_d = K_p/\tau_d$, where, τ_d is the derivative time)

The derivative action slows down the rate of change of the controller output. A derivative controller is quite useful when the error is continuously changing with time. One should, however, avoid using it alone. This is because there is no output when the error is zero and when the rate of change of error is constant.

When all the above control actions are summed up and used together, the final equation becomes

$$PID = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d}{dt} e(t) \quad (7.4)$$

The above equation represents an ideal form of PID controller. This means that the integral controller can be used independently. However, it is not a good decision since, the integral action begins only after the error exists for some amount of time. The proportional controller however begins as soon as the error starts existing. Hence, the integral controller is often used in conjunction with a proportional controller. This is popularly known as PI controller and the equation for Proportional Integral action becomes,

$$PI = K_p e(t) + \left(K_p / \tau_i \right) \int_0^t e(t) dt \quad (7.5)$$

$$= K_p \left\{ e(t) + (1/\tau_i) \int_0^t e(t) dt \right\} \quad (7.6)$$

Similarly, as discussed before, independent use of derivative controller is also not desirable. Moreover, if the process contains high frequency noise then the derivative action will tend to amplify the noise. Hence, derivative controller is also used in conjunction with Proportional or Proportional Integral controller popularly known as PD or PID, respectively. Therefore the equation for Proportional Derivative action becomes,

$$PD = K_p e(t) + K_p \tau_d \frac{d}{dt} e(t) \quad (7.7)$$

$$= K_p \left\{ e(t) + \tau_d \frac{d}{dt} e(t) \right\} \quad (7.8)$$

Finally, writing the equation for PID controller,

$$PID = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt + \tau_d \frac{d}{dt} e(t) \right\} \quad (7.9)$$

7.2 Ziegler-Nichols Rule for Tuning PID Controllers

There are many rules to tune a PID controller. We shall see the two popular methods suggested by Ziegler-Nichols.

7.2.1 First Method

Ziegler-Nichols rule determines the values of gain K , integral time τ_i and derivative time τ_d based on the step response characteristics of a given plant. In this

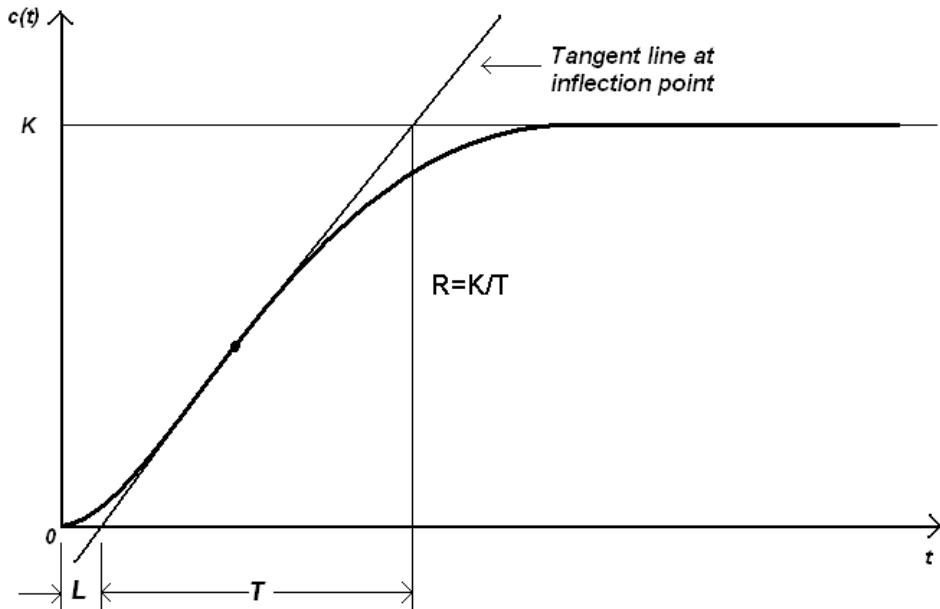


Figure 7.2: Reaction curve [5]

method, one can experimentally obtain the response of a plant to a step input, as shown in figure 7.2. This method is applicable only when the response to the step input exhibits S-shaped curve [7].

As shown in figure 7.2, by drawing the tangent line at the inflection point and determining the intersection of the tangent line with the time axis and the line $c(t) = K$, we get two constants, namely, delay time L and time constant T .

Ziegler and Nichols suggested to set the values of K, τ_i, τ_d according to the formula shown in table 7.1. Notice that the PID controller tuned by the Ziegler-Nichols rule gives,

$$G_c(s) = K_p \left(1 + \frac{1}{T_i s} + T_d s \right) \quad (7.10)$$

$$= 1.2 \frac{T}{L} \left(1 + \frac{1}{2Ls} + 0.5Ls \right) \quad (7.11)$$

$$= 0.6T \frac{\left(s + \frac{1}{L} \right)^2}{s} \quad (7.12)$$

Thus, the PID controller has a pole at the origin and double zeros at $s = -1/L$.

Type of controller	K	τ_i	τ_d
P	$\frac{1}{RL}$	∞	0
PI	$\frac{0.9}{RL}$	$3L$	0
PID	$\frac{1.2}{RL}$	$2L$	$0.5L$

Table 7.1: Ziegler-Nichols tuning rule based on step response of plant

Type of controller	K	τ_i	τ_d
P	$0.5K_u$	∞	0
PI	$0.45K_u$	$\frac{1}{0.2}P_u$	0
PID	$0.6K_u$	$0.5P_u$	$0.125P_u$

Table 7.2: Ziegler-Nichols tuning rule for instability tuning method

7.2.2 Second Method

The second method is also known as ‘instability method’[6]. This is a closed loop method in which the integral and derivative gains of the PID controller are made zero with a unity value for proportional gain. A setpoint change is made and the temperature profile is observed for some time. The temperature would most likely maintain a steady-state with some offset. The gain is increased to a next distinct value (say 2) with a change in the setpoint. The procedure is repeated until the temperature first varies with sustained oscillations. It is necessary that the output (temperature) should have neither under damped nor over damped oscillations. At this particular frequency of sustained oscillations, the corresponding value of K_p is noted and is called as the critical gain K_{cr} . The corresponding period of oscillation is known as P_{cr} . Refer to figure 7.3.

The various P, PI and PID parameters are then calculated with the help of table 7.2. Using the Ziegler-Nichols method explained earlier, the following values were obtained. Refer to figure 7.4.

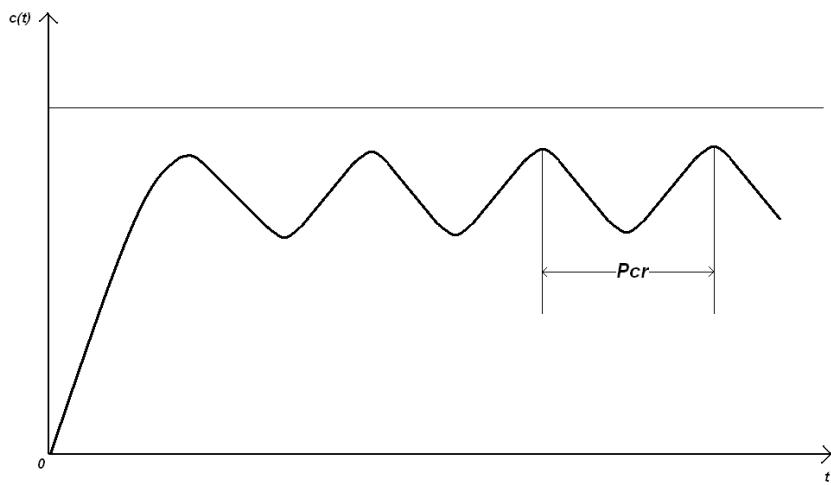


Figure 7.3: Ziegler-Nichols instability tuning method

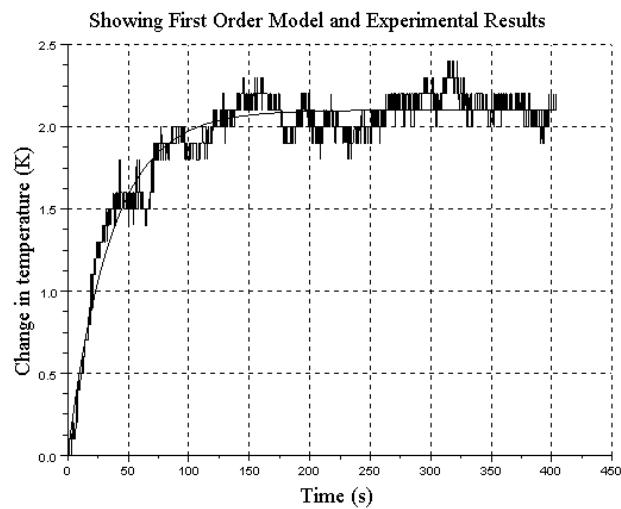


Figure 7.4: Refer to ‘Step Test’ experiment [6]

$$\begin{aligned}L &= 6 \text{ s} \\T &= 193 \text{ s}\end{aligned}$$

For PI

$$\begin{aligned}K &= 6.031 \\ \tau_i &= 18\end{aligned}$$

For PID

$$\begin{aligned}K &= 8 \\ \tau_i &= 12 \\ \tau_d &= 3\end{aligned}$$

While performing the experiment, fine tunning of K, τ_i, τ_d may be required.

7.3 Implementing PI Controller using Trapezoidal Approximation

Figure 7.5 shows Xcos diagram for implementing PI controller. The PI controller in continuous time is given by,

$$u(t) = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt \right\} \quad (7.13)$$

On taking the Laplace transform, we obtain

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} \right\} e(t) \quad (7.14)$$

By mapping controller given in equation 7.14 to the discrete time domain using trapezoidal approximation

$$u(n) = K \left\{ 1 + \frac{T_s}{2\tau_i} \frac{z+1}{z-1} \right\} e(n) \quad (7.15)$$

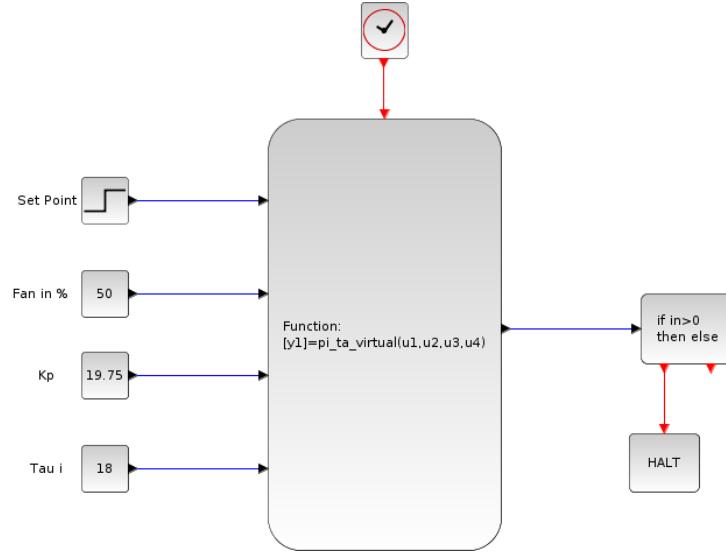


Figure 7.5: Xcos for PI controller available as `pi_ta_virtual.xcos`

On cross multiplying, we obtain

$$(z - 1)u(n) = K \left\{ (z - 1) + \frac{T_s}{2\tau_i}(z + 1) \right\} e(n) \quad (7.16)$$

We divide by z and then by using shifting theorem, we obtain

$$u(n) - u(n - 1) = K \left\{ e(n) - e(n - 1) + \frac{T_s}{2\tau_i}e(n) + \frac{T_s}{2\tau_i}e(n - 1) \right\} \quad (7.17)$$

The PI controller is usually written as

$$u(n) = u(n - 1) + s_0e(n) + s_1e(n - 1) \quad (7.18)$$

where

$$s_0 = K \left(1 + \frac{T_s}{2\tau_i} \right) \quad (7.19)$$

$$s_1 = K \left(-1 + \frac{T_s}{2\tau_i} \right) \quad (7.20)$$

For implementing the PI controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

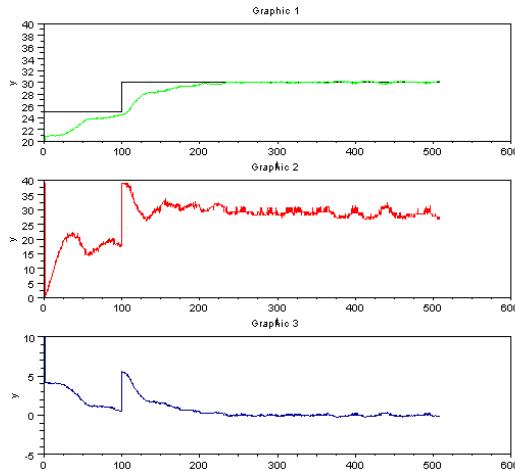


Figure 7.6: PI controller (Trapezoidal Approximation) output

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_ta.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pi_ta.xcos` instead of `step_test.xcos`.

The output of Xcos is shown in figure 7.6. Figure shows three plots. First subplot shows setpoint and output temperature profile. Second subplot shows control effort and third subplot shows error between setpoint and plant output.

7.3.1 Implementing PI controller using Trapezoidal Approximation on SBHS, virtually

For implementing above PI controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_ta_virtual.sce` instead of `step_test.sce`.
3. In step 6, execute the file `pi_ta_virtual.xcos` instead of `step_test.xcos`.

7.4 Implementing PI Controller using Backward Difference Approximation

The PI controller in continuous time is given by

$$u(t) = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt \right\} \quad (7.21)$$

On taking the Laplace transform, we obtain

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} \right\} e(t) \quad (7.22)$$

By mapping controller given in equation 7.22 to the discrete time domain using Backward difference approximation:

$$u(n) = K \left\{ 1 + \frac{T_s}{\tau_i} \frac{z}{z-1} \right\} e(n) \quad (7.23)$$

On cross multiplying, we get

$$(z-1)u(n) = K \left\{ (z-1) + \frac{T_s}{\tau_i} (z) \right\} e(n) \quad (7.24)$$

We divide by z and then by using shifting theorem, we obtain

$$u(n) - u(n-1) = K \left\{ e(n) - e(n-1) + \frac{T_s}{\tau_i} e(n) \right\} \quad (7.25)$$

The PI controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) \quad (7.26)$$

where

$$s_0 = K \left(1 + \frac{T_s}{\tau_i} \right) \quad (7.27)$$

$$s_1 = -K \quad (7.28)$$

For implementing the PI controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

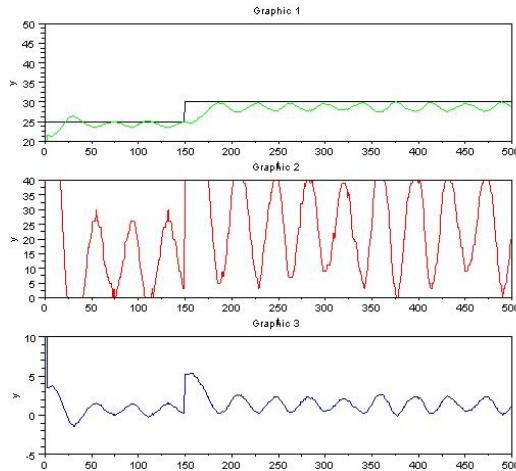


Figure 7.7: PI controller (Backward Difference Approximation) output

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_bda.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pi_bda.xcos` instead of `step_test.xcos`.

The Xcos output is shown in figure 7.7. Figure shows three plots. First subplot shows setpoint and output temperature profile. Second subplot shows control effort and third subplot shows error between setpoint and plant output.

7.4.1 Implementing PI Controller using Backward Difference Approximation on SBHS, virtually

For implementing above PI controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_bda_virtual.sce` instead of `step_test.sce`.
3. In step 6, execute the file `pi_bda_virtual.xcos` instead of `step_test.xcos`.

7.5 Implementing PI Controller using Forward Difference Approximation

The PI controller in continuous time is given by

$$u(t) = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt \right\} \quad (7.29)$$

On taking the Laplace transform, we obtain

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} \right\} e(t) \quad (7.30)$$

By mapping controller given in equation 7.30 to the discrete time domain using forward difference formula, we get

$$u(n) = K \left\{ 1 + \frac{T_s}{\tau_i} \frac{1}{z - 1} \right\} e(n) \quad (7.31)$$

On cross multiplying, we get

$$(z - 1)u(n) = K \left\{ (z - 1) + \frac{T_s}{\tau_i} \right\} e(n) \quad (7.32)$$

We divide by z and then by using shifting theorem, we get

$$u(n) - u(n - 1) = K \left\{ e(n) - e(n - 1) + \frac{T_s}{\tau_i} e(n - 1) \right\} \quad (7.33)$$

The PI controller is usually written as

$$u(n) = u(n - 1) + s_0 e(n) + s_1 e(n - 1) \quad (7.34)$$

where

$$s_0 = K \quad (7.35)$$

$$s_1 = K \left(-1 + \frac{T_s}{\tau_i} \right) \quad (7.36)$$

For implementing the PI controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

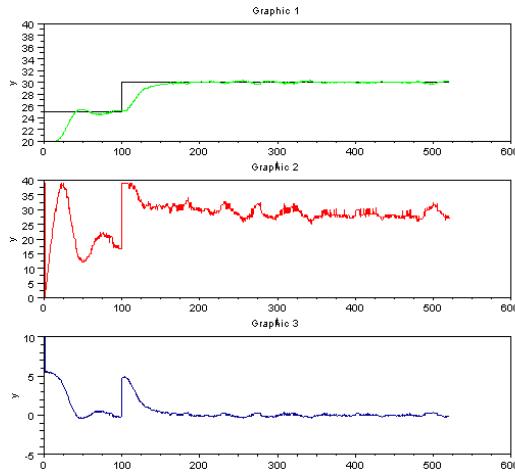


Figure 7.8: PI controller implementation (Forward Difference Approximation)

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_fda.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pi_fda.xcos` instead of `step_test.xcos`.

The Xcos output is shown in figure 7.8. Figure shows three plots. First sub plot shows setpoint and output temperature profile. Second sub plot shows control effort and third sub plot shows error between setpoint and plant output.

7.5.1 Implementing PI Controller using Forward Difference Approximation on SBHS, virtually

For implementing above PI controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pi_fda_virtual.sce` instead of `step_test.sce`.
3. In step 6, execute the file `pi_fda_virtual.xcos` instead of `step_test.xcos`.

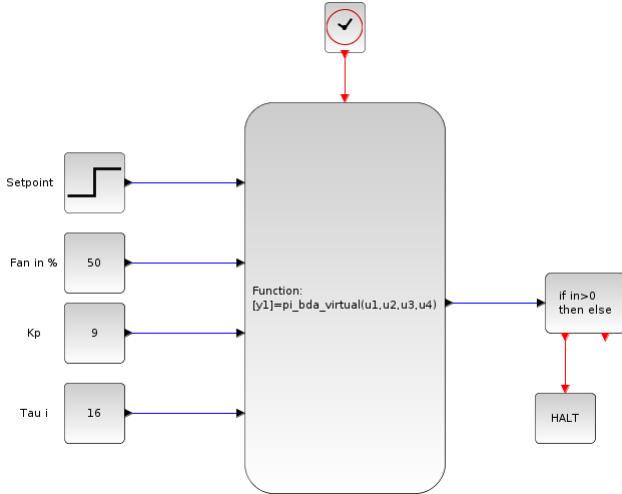


Figure 7.9: Xcos for PID controller available as `pid_bda_virtual.xcos`

7.6 Implementing PID Controller using Backward Difference Approximation

Figure 7.9 shows Xcos diagram for implementing PID controller.

The PID controller in continuous time is given by

$$u(t) = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt + \tau_d \frac{de(t)}{dt} \right\} \quad (7.37)$$

On taking the Laplace transform, we obtain

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} + \tau_d s \right\} e(t) \quad (7.38)$$

By mapping controller given in equation 7.38 to the discrete time domain using backward difference formula, we get

$$u(n) = K \left\{ 1 + \frac{T_s}{\tau_i} \frac{z}{z-1} + \frac{\tau_d}{T_s} \frac{z-1}{z} \right\} e(n) \quad (7.39)$$

On cross multiplying, we obtain

$$(z^2 - z)u(n) = K \left\{ (z^2 - z) + \frac{T_s}{\tau_i} z^2 + \frac{\tau_d}{T_s} (z - 1)^2 \right\} e(n) \quad (7.40)$$

We divide by z^2 and by using shifting theorem, we get

$$\begin{aligned} u(n) - u(n-1) &= K \left\{ e(n) - e(n-1) + \frac{T_s}{\tau_i} e(n) \right. \\ &\quad \left. + \frac{\tau_d}{T_s} [e(n) - 2e(n-1) + e(n-2)] \right\} \end{aligned} \quad (7.41)$$

The PID controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) + s_2 e(n-2) \quad (7.42)$$

where

$$s_0 = K \left[1 + \frac{T_s}{\tau_i} + \frac{\tau_d}{T_s} \right] \quad (7.43)$$

$$s_1 = K \left[-1 - 2 \frac{\tau_d}{T_s} \right] \quad (7.44)$$

$$s_2 = K \left[\frac{\tau_d}{T_s} \right] \quad (7.45)$$

For implementing the PID controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pid_bda.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pid_bda.xcos` instead of `step_test.xcos`.

The output of Xcos is shown in figure 7.10. Figure shows three plots. First sub plot shows setpoint and output temperature profile. Second sub plot shows control effort and third sub plot shows error between setpoint and plant output.

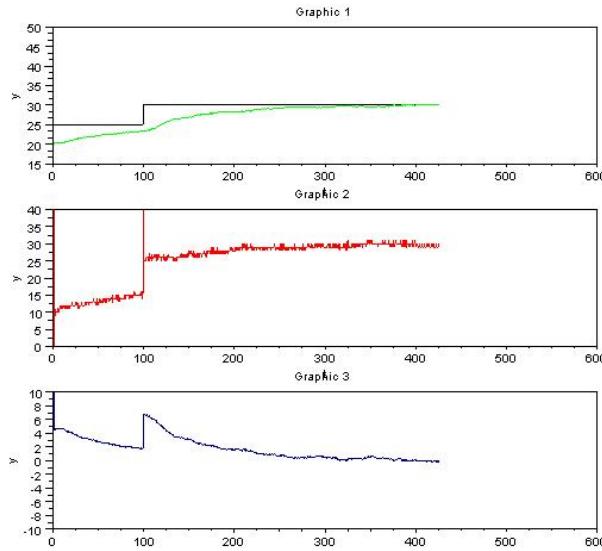


Figure 7.10: PID controller (Backward Difference Approximation) output

7.6.1 Implementing PID Controller using Backward Difference Approximation on SBHS, virtually

For implementing above PI controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pid_bda_virtual.sce` instead of `step_test.sce`.
3. In step 6, execute the file `pid_bda_virtual.xcos` instead of `step_test.xcos`.

7.7 Implementing PID Controller using Trapezoidal Approximation for Integral Mode and Backward Difference Approximation for the Derivative Mode

The PID controller in continuous time is given by

$$u(t) = K \left\{ e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt + \tau_d \frac{de(t)}{dt} \right\} \quad (7.46)$$

On taking the Laplace transform, we obtain

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} + \tau_d s \right\} e(t) \quad (7.47)$$

By mapping controller given in equation 7.47 to the discrete time domain using trapezoidal approximation for integral mode and backward difference approximation for the derivative mode, we get

$$u(n) = K \left\{ 1 + \frac{T_s}{2\tau_i} \frac{z+1}{z-1} + \frac{\tau_d}{T_s} \frac{z-1}{z} \right\} e(n) \quad (7.48)$$

On cross multiplying, we obtain

$$(z^2 - z)u(n) = K \left\{ (z^2 - z) + \frac{T_s}{2\tau_i} (z^2 + z) \frac{\tau_d}{T_s} (z-1)^2 \right\} e(n) \quad (7.49)$$

We divide by z^2 and then by using shifting theorem, we get

$$\begin{aligned} u(n) - u(n-1) &= K \left\{ e(n) - e(n-1) + \frac{T_s}{2\tau_i} e(n) + e(n-1) \right. \\ &\quad \left. + \frac{\tau_d}{T_s} [e(n) - 2e(n-1) + e(n-2)] \right\} \end{aligned} \quad (7.50)$$

The PID controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) + s_2 e(n-2) \quad (7.51)$$

where

$$s_0 = K \left[1 + \frac{T_s}{2\tau_i} + \frac{\tau_d}{T_s} \right] \quad (7.52)$$

$$s_1 = K \left[-1 + \frac{T_s}{2\tau_i} - 2 \frac{\tau_d}{T_s} \right] \quad (7.53)$$

$$s_2 = K \frac{\tau_d}{T_s} \quad (7.54)$$

For implementing the PID controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

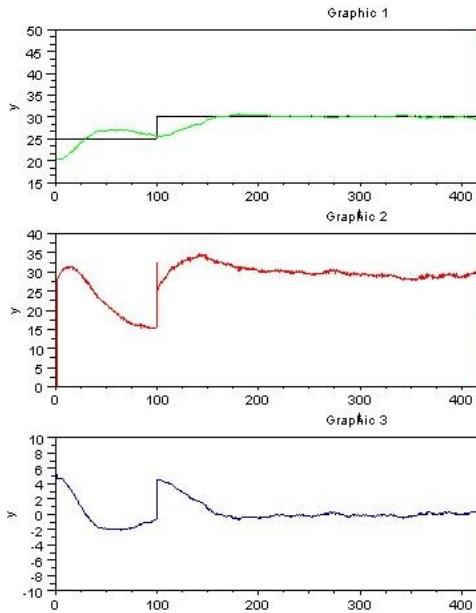


Figure 7.11: PID controller (TA - BDA) implementation

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pid_ta_bda.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pid_ta_bda.xcos` instead of `step_test.xcos`.

The Xcos output is shown in figure 7.11. Figure shows three plots. First subplot shows setpoint and output temperature profile. Second subplot shows control effort and third subplot shows error between setpoint and plant output.

7.7.1 Implementing PID Controller using Trapezoidal Approximation for Integral Mode and Backward Difference Approximation for the Derivative Mode on SBHS, virtually

For implementing above PID controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

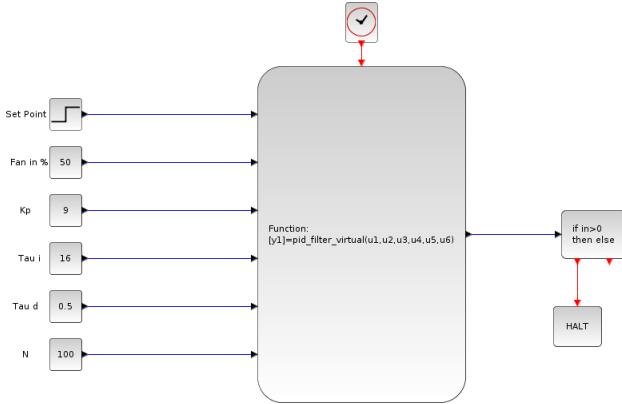


Figure 7.12: Xcos for PID controller with filtering available as pidN_virtual.xcos

1. In step 1, change the directory to the folder pid_controller instead of StepTest.
2. In step 5, execute the file pid_ta_bda_virtual.sce instead of step_test.sce.
3. In step 6, execute the file pid_ta_bda_virtual.xcos instead of step_test.xcos.

Due to the introduction of derivative action, control effort shows lots of fluctuations. By using filtered form of PID, we can make derivative mode implementable.

7.8 Implementing PID Controller with Filtering using Backward Difference Approximation

Figure 7.12 shows Xcos diagram for implementing PID controller with filtering.

PID filtered form is given by

$$u(t) = K \left\{ 1 + \frac{1}{\tau_i s} + \frac{\tau_d s}{1 + \frac{\tau_d s}{N}} \right\} e(t) \quad (7.55)$$

where N is large number of the order of 100.

By mapping controller given in equation 7.55 to the discrete time domain using backward difference formula, we get

$$u(n) = K \left(1 + \frac{T_s}{\tau_i} \frac{1}{1-z^{-1}} + \frac{\tau_d(1-z^{-1})}{1+\frac{\tau_d(1-z^{-1})}{N}} \right) e(n) \quad (7.56)$$

$$u(n) = K \left(1 + \frac{T_s}{\tau_i} \frac{1}{1-z^{-1}} + \frac{Nr_1(1-z^{-1})}{1+r_1z^{-1}} \right) e(n) \quad (7.57)$$

where

$$r_1 = -\frac{\frac{\tau_d}{N}}{\frac{\tau_d}{N} + T_s} \quad (7.58)$$

On cross multiplying, we obtain

$$\begin{aligned} (1-z^{-1})(1+r_1z^{-1})u(n) &= K[(1-z^{-1})(1+r_1z^{-1}) \\ &\quad + \frac{T_s}{\tau_i}(1+r_1z^{-1}) + \frac{\tau_d}{T_s}(1-z^{-1})^2]e(n) \end{aligned} \quad (7.59)$$

Simplifying and then by using shifting theorem, we obtain

$$\begin{aligned} u(n) + (r_1 - 1)u(n-1) \\ -r_1u(n-2) &= K \left[1 + \frac{T_s}{\tau_i} - Nr_1 \right] e(n) \\ &\quad + K \left[r_1 \left(1 + \frac{T_s}{\tau_i} + 2N \right) - 1 \right] e(n-1) \\ &\quad - K [r_1(1+N)] e(n-2) \end{aligned} \quad (7.60)$$

Hence

$$\begin{aligned} u(n) &= r_1u(n-2) - (r_1 - 1)u(n-1) \\ &\quad + s_0e(n) + s_1e(n-1) + s_2e(n-2) \end{aligned} \quad (7.61)$$

where

$$s_0 = K \left[1 + \frac{T_s}{\tau_i} - Nr_1 \right] \quad (7.62)$$

$$s_1 = K \left[r_1 \left(1 + \frac{T_s}{\tau_i} + 2N \right) - 1 \right] \quad (7.63)$$

$$s_2 = -K [r_1(1+N)] \quad (7.64)$$

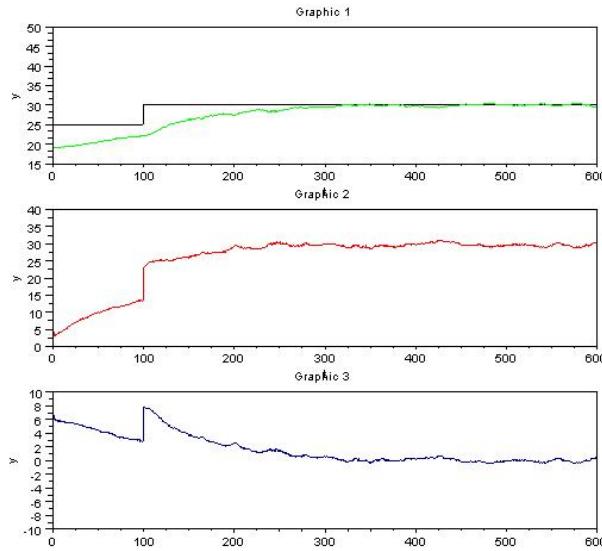


Figure 7.13: PID controller (with filtering) implementation

For implementing the PID controller, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

1. In step 1, change the directory to the folder `pid_controller` instead of `StepTest`.
2. In step 5, execute the file `pid_filter.sci` instead of `step_test.sci`.
3. In step 6, execute the file `pidN.xcos` instead of `step_test.xcos`.

The Xcos output is shown in figure 7.13. Figure shows three plots. First subplot shows setpoint and output temperature profile. Second subplot shows control effort and third subplot shows error between setpoint and plant output.

By comparing figure 7.10 and figure 7.13, it is clear that introduction of filtered form of PID reduces fluctuations in control effort.

7.8.1 Implementing PID Controller with Filtering using Backward Difference Approximation on SBHS, virtually

For implementing the PID controller virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder pid_controller instead of StepTest.
2. In step 5, execute the file pid_filter_virtual.sce instead of step_test.sce.
3. In step 6, execute the file pidN_virtual.xcos instead of step_test.xcos.

7.9 Scilab Code

7.9.1 Scilab code for serial communication

Scilab Code 7.1 ser_init.sci used for serial communication

```

1 mode(0)
2 global filename m
3 // ** Sampling Time ** //
4 sampling_time = 1;
5 // // // // * * * // // // // //
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res = init([port1 port2]);
12 disp(res)
```

7.9.2 Scilab code for PI controller

Scilab Code 7.2 pi_ta.sci

```

1 mode(0)
2 function temp = pi_ta(setpoint,fan,K,Ti)
3     global heatdisp fandisp tempdisp setpointdisp
        sampling_time m name temp heat_in fan_in C0
        u_old u_new e_old e_new
4
5 Ts=sampling_time;
6 e_new = setpoint - temp;
```

```

7
8 S0=K(1+Ts/(2*Ti));
9 S1=K*(-1+(Ts/(2*Ti)));
10 u_new = u_old+(S0*e_new)+(S1*e_old);
11
12
13 u_old = u_new;
14 e_old = e_new;
15
16 heat = u_new;
17 temp = comm(heat,fan);
18
19 plotting([heat fan temp setpoint],[0 0 20 0],[100
100 40 1000])
20
21 m=m+1;
22 endfunction

```

Scilab Code 7.3 pi_bda.sci

```

1 // global temp heat fan sampling_time m heatdisp
   fandisp tempdisp x name Ts
2
3 mode(0)
4 function temp = pi_bda(setpoint,fan,K,Ti)
5     global heatdisp fandisp tempdisp setpointdisp
       sampling_time m name temp heat_in fan_in C0
       u_old u_new e_old e_new
6
7 Ts = sampling_time;
8 e_new = setpoint - temp;
9
10
11 S0=K(1+(Ts/Ti));
12 S1=-K;
13
14
15

```

```

16 u_new = u_old+ S0*e_new+ S1*e_old ;
17
18
19 u_old = u_new ;
20 e_old = e_new ;
21
22 heat = u_new ;
23 temp = comm(heat ,fan ) ;
24
25 plotting([heat fan temp setpoint],[0 0 20 0],[100
100 40 1000])
26
27 m=m+1 ;
28 endfunction

```

Scilab Code 7.4 pi_fda.sci

```

1 mode(0)
2 function temp = pi_fda(setpoint ,fan ,K, Ti)
3     global heatdisp fandisp tempdisp setpointdisp
        sampling_time m name temp heat_in fan_in C0
        u_old u_new e_old e_new
4
5 Ts=sampling_time ;
6 e_new = setpoint - temp ;
7
8
9 S0=K*(1+((Ts / Ti ))) ;
10 S1=-K;
11 u_new = u_old+(S0*e_new)+(S1*e_old) ;
12
13 u_old = u_new ;
14 e_old = e_new ;
15
16 heat = u_new ;
17
18 temp = comm(heat ,fan ) ;
19

```

```

20     plotting([heat fan temp setpoint],[0 0 20 0],[100
21             100 40 1000])
22
23     m=m+1;
endfunction

```

7.9.3 Scilab code for PID controller

Scilab Code 7.5 pid_bda.sci

```

1 mode(0)
2
3 function [temp] = pid_bda(setpoint,fan,K,Ti,Td)
4 global temp heat_in fan_in C0 u_old u_new e_old e_new
e_old_old
5
6 global heatdisp fandisp tempdisp setpointdisp
sampling_time m name
7
8 e_new = setpoint - temp;
9
10 Ts=sampling_time;
11
12 S0=K*(1+(Ts/Ti)+(Td/Ts));
13 S1=K*(-1-((2*Td)/Ts));
14 S2=K*(Td/Ts);
15
16 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
17
18 u_old = u_new;
19 e_old_old = e_old;
20 e_old = e_new;
21
22
23 heat = u_new;
24 temp = comm(heat,fan);
25

```

```

26     plotting([heat fan temp setpoint],[0 0 20 0],[100
27             100 40 1000])
28
29     m=m+1;
endfunction

```

Scilab Code 7.6 pid_ta_bda.sci

```

1
2 mode(0)
3
4 function [temp] = pid_ta_bda(setpoint,fan,K,Ti,Td)
5 global temp heat_in fan_in C0 u_old u_new e_old e_new
6     e_old_old
7 global heatdisp fandisp tempdisp setpointdisp
8     sampling_time m name
9
10    e_new = setpoint - temp;
11
12    Ts=sampling_time;
13
14    S0=K*(1+(Ts/(2*Ti))+(Td/Ts));
15    S1=K*(-1+(Ts/(2*Ti))-(2*Td/Ts));
16    S2=(K*Td/Ts);
17
18    u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
19
20    u_old = u_new;
21    e_old_old = e_old;
22    e_old = e_new;
23
24
25    heat = u_new;
26
27    temp = comm(heat,fan);
28
29    plotting([heat fan temp setpoint],[0 0 20 0],[100
30             100 40 1000])

```

```

28
29      m=m+1;
30  endfunction


---


Scilab Code 7.7 pid_filter.sci
1  mode(0)
2
3  function temp = pid(setpoint,fan,K,Ti,Td,N)
4      global heatdisp fandisp tempdisp setpointdisp
         sampling_time m name temp heat_in fan_in C0
         u_old u_new e_old e_new e_old_old r1 u_old_old
5
6  Ts = sampling_time;
7  e_new = setpoint - temp;
8
9  r1=-(Td/N)/((Td/N)+Ts));
10
11 S0=K*(1+(Ts/Ti)-(N*r1));
12 S1=K*((r1*(1+(Ts/Ti)+(2*N))-1);
13 S2=-K*r1*(1+N);
14
15 u_new = r1*u_old_old -(r1-1)*u_old + S0*e_new + S1*
   e_old + S2*e_old_old;
16
17 u_old_old = u_old;
18 u_old = u_new;
19 e_old_old = e_old;
20 e_old = e_new;
21
22
23 heat = u_new;
24
25 temp = comm(heat,fan);
26
27 plotting([heat fan temp setpoint],[0 0 20 0],[100
   100 40 1000])
28

```

```

29     m=m+1;
30 endfunction

```

Scilab Code 7.8 pid_bda_virtual.sce

```

1 mode(0);
2 // For scilab 5.1.1 or lower version users, use scicos
   command to open scicos diagrams instead of xcos
3
4 global fdfh fdt fngr fncl m err_count y limits
   sampling_time m
5
6 // *****
7 sampling_time=1; // In seconds. Fractions are allowed
8 // *****
9
10 exec ("pid_bda_virtual.sci");
11
12 ok = init();
13
14 if ok~= [] // open xcos only if communication is
   through (ie reply has come from server)
15 xcos('pid_bda_virtual.xcos');
16 else
17 disp("NO NETWORK CONNECTION!");
18 return
19 end

```

Scilab Code 7.9 pid_bda_virtual.sci

```

1 mode(0);
2 // PI Controller using trapezoidal approximation.
3 // Heater input is passed as input argument to
   introduce control effort u(n)
4 // Fan input is passed as input argument which is kept
   at constant level
5 // Range of Fan input : 20 to 252
6 // Temperature is read

```

```

7
8 function [ stop ] = pid_bda_virtual( setpoint ,fan ,K,Ti ,Td
    )
9
10 global temp heat C0 u_old u_new e_old e_new fdfh fdt
    fnrcr fnchw m err_count stop q heatdisp fandisp
    tempdisp setpointdisp limits m x sampling_time
    e_old_old
11
12 e_new = setpoint - temp ;
13
14
15 Ts=1 ;
16 S0=K*(1+(Ts / Ti )+(Td/Ts )) ;
17 S1=K*(-1-((2*Td) / Ts )) ;
18 S2=K*(Td/Ts ) ;
19
20 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old ;
21
22 heat=u_new ;
23
24 u_old = u_new ;
25 e_old_old = e_old ;
26 e_old = e_new ;
27
28 [ stop ,temp ] = comm(heat ,fan ); // N e v e r   e d i t   t h i s
    l i n e
29 plotting([heat fan temp setpoint],[0 0 30 0],[100
    100 50 1000])
30
31 endfunction

```

Chapter 8

Two Degrees of Freedom (2-DOF) Controller

In this chapter, we discuss the implementation of a 2-DOF controller using the SBHS. We also cover the basics of 2-DOF controller theory and design.

8.1 Introduction to 2-DOF Controller

Controllers are broadly divided into two categories: feedback and feed forward controllers. Feed forward controllers are those that take control action before a disturbance affects the plant. But this requires an ability to sense the disturbance accurately. Moreover, exact knowledge of the plant is also needed. As a result, a feed forward control strategy is rarely used alone.

A feedback control strategy is shown in figure 8.1. The reference r and the output y are continuously compared to generate error e , which is fed to the controller $G_c(z)$, to take appropriate control action. u is the controller output that is fed to the plant. Unlike feed forward controllers, exact knowledge of the plant $G(z)$ and the disturbance v is not necessary in this case. Feedback controllers are further classified as One Degree of Freedom (1-DOF) controllers and Two Degrees of Freedom (2-DOF) controllers. Degree of freedom refers to the number of parameters that are free to vary in a system. A higher degree of freedom controller makes the plant less susceptible to disturbances.

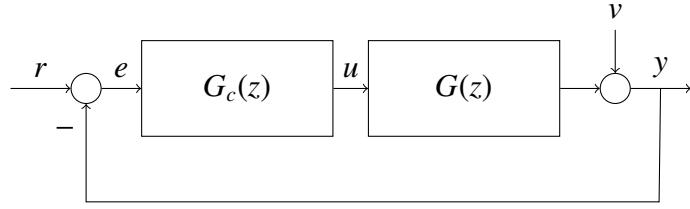


Figure 8.1: Feed back control strategy

The expression for output, $Y(z)$ of the system shown in figure 8.1 is given by

$$Y(z) = \frac{G(z)G_c(z)}{1 + G(z)G_c(z)}R(z) + \frac{1}{1 + G(z)G_c(z)}V(z) \quad (8.1)$$

This expression can be written in mixed notation [5] as

$$y(n) = \frac{G(z)G_c(z)}{1 + G(z)G_c(z)}r(n) + \frac{1}{1 + G(z)G_c(z)}v(n) \quad (8.2)$$

Let,

$$T(z) = \frac{G(z)G_c(z)}{1 + G(z)G_c(z)}, S(z) = \frac{1}{1 + G(z)G_c(z)} \quad (8.3)$$

Therefore,

$$y(n) = T(z)r(n) + S(z)v(n) \quad (8.4)$$

The controller has to track the reference input as well as eliminate the effect of external disturbance. So ideally, we want $T = 1$ and $S = 0$. But, it is not possible to achieve both the requirements simultaneously using this control strategy. This control strategy is called **One Degree of Freedom**, abbreviated as 1-DOF.

A **Two Degrees of Freedom** controller is as shown in figure 8.2. Here, G_b and G_f together constitute the controller. G_b is in the feedback path and is used to eliminate the effect of disturbances, whereas G_f is in the feed forward path and is used to help the output track the reference input.

The expression for control effort u in figure 8.2 is given by

$$u(n) = r(n)G_f - y(n)G_b \quad (8.5)$$

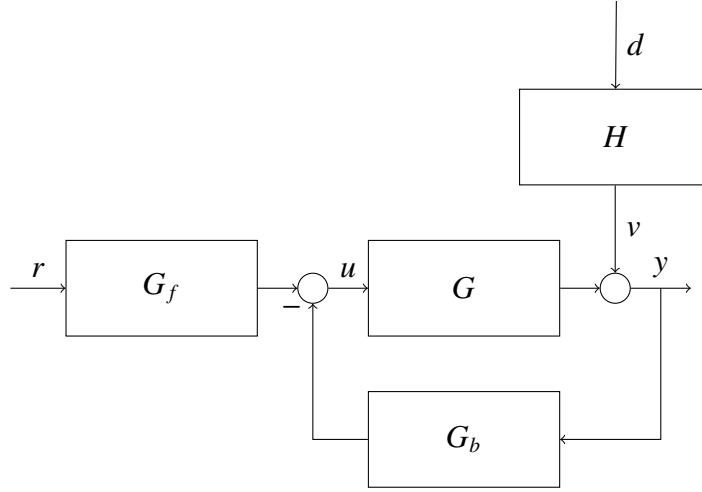


Figure 8.2: 2DOF feed back control strategy

Let

$$G_b = \frac{S_c}{R_c}, G_f = \frac{T_c}{R_c} \quad (8.6)$$

where R_c , S_c and T_c are polynomials in z^{-1} .

We get

$$R_c(z)u(n) = T_c(z)r(n) - S_c(z)y(n) \quad (8.7)$$

Consider a plant whose model is given by

$$A(z)y(n) = z^{-k}B(z)u(n) + v(n) \quad (8.8)$$

Substituting equation 8.7 in equation 9.4, we get

$$Ay(n) = z^{-k}\frac{B}{R_c}\left[T_c r(n) - S_c y(n)\right] + v(n) \quad (8.9)$$

Solving for $y(n)$,

$$\left(\frac{R_c A + z^{-k} B S_c}{R_c}\right)y(n) = z^{-k}\frac{B T_c}{R_c}r(n) + v(n) \quad (8.10)$$

This can also be written as

$$y(n) = z^{-k} \frac{BT_c}{\phi_{cl}} r(n) + \frac{R_c}{\phi_{cl}} v(n) \quad (8.11)$$

where ϕ_{cl} is the closed loop characteristic polynomial given by

$$\phi_{cl} = R_c(z)A(z) + z^{-k}B(z)S_c(z) \quad (8.12)$$

We want the following conditions to be satisfied while designing a controller.

1. The zeros of ϕ_{cl} should be inside the unit circle, so that the closed-loop system becomes stable
2. The value of $z^{-k} \frac{BT_c}{\phi_{cl}}$ must be close to unity, so that reference tracking is achieved
3. The value of $\frac{R_c}{\phi_{cl}}$ must be as small as possible to achieve disturbance rejection

We shall now see the pole placement controller approach to design a 2-DOF controller.

8.2 2-DOF Controller Design using the Pole Placement Method [5]

A 2-DOF pole placement controller is shown in figure 8.3. We will not consider the effect of external disturbance in the design. The controller will be designed for setpoint tracking. We want the desired output, Y_m , of the system to be related to the setpoint R in the following manner:

$$Y_m(z) = \gamma z^{-k} \frac{B_r}{\phi_{cl}} R(z) \quad (8.13)$$

ϕ_{cl} is the desired closed loop characteristic polynomial obtained from the desired region analysis. Please refer to [5] for more information on desired region analysis. γ is chosen such that Y_m equals the setpoint at steady-state. Therefore γ is given by,

$$\gamma = \frac{\phi_{cl}(1)}{B_r(1)} \quad (8.14)$$

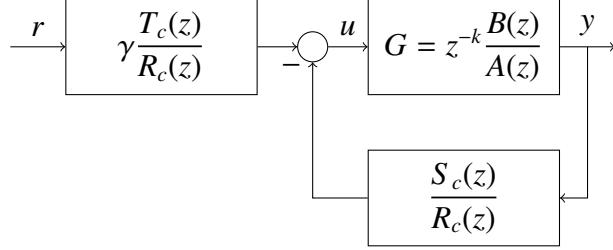


Figure 8.3: 2-DOF pole placement controller

Simplifying the block diagram shown in figure 8.3 yields

$$Y = \gamma z^{-k} \frac{BT_c}{AR_c + z^{-k}BS_c} R \quad (8.15)$$

We have dropped the argument of z for convenience

We want the output Y of the system to be equal to the desired output Y_m . Equating equations 8.13 and 8.15 we get

$$\frac{BT_c}{AR_c + z^{-k}BS_c} = \frac{B_r}{\phi_{cl}} \quad (8.16)$$

We can expect some cancelations between the numerator and the denominator polynomials in the LHS, thereby making $\deg B_r < \deg B$. But the cancelations, if any, must be between *stable* poles and zeros. One should avoid the cancelation of an unstable pole with an unstable zero.

Let us split the factors of the numerator and denominator polynomials, B and A , of the plant into *good* and *bad* factors. Therefore, we write A and B as

$$A = A^g A^b, B = B^g B^b \quad (8.17)$$

We also define R_c , S_c and T_c as

$$R_c = B^g R_1 \quad (8.18)$$

$$S_c = A^g S_1 \quad (8.19)$$

$$T_c = A^g T_1 \quad (8.20)$$

Hence, equation 8.16 becomes

$$\frac{B^g B^b A^g T_1}{A^g A^b B^g R_1 + z^{-k} B^g B^b A^g S_1} = \frac{B_r}{\phi_{cl}} \quad (8.21)$$

After cancelling out the common factors, we obtain

$$\frac{B^b T_1}{A^b R_1 + z^{-k} B^b S_1} = \frac{B_r}{\phi_{cl}} \quad (8.22)$$

We obtain,

$$B^b T_1 = B_r \quad (8.23)$$

$$A^b R_1 + z^{-k} B^b S_1 = \phi_{cl} \quad (8.24)$$

Equation 8.24 is known as the Aryabhatta's identity and can be used to solve for R_1 and S_1 . One can choose T_1 in many ways. If we choose $T_1 = S_1$ the 2-DOF controller is reduced to a 1-DOF controller. Let us choose $T_1 = 1$. Therefore equation 8.23 becomes

$$B^b = B_r \quad (8.25)$$

The expression for γ now becomes

$$\gamma = \frac{\phi_{cl(1)}}{B^b(1)} \quad (8.26)$$

and the desired closed loop transfer function will be

$$\frac{Y_m(z)}{R(z)} = \gamma z^{-k} \frac{B^b}{\phi_{cl}} \quad (8.27)$$

One can see that the open loop plant model imposes two limitations on the closed loop transfer function.

1. The bad portion of the open loop model cannot be canceled out and it appears in the closed loop model.
2. The open loop plant delay cannot be removed or minimized, i.e., the closed loop model cannot be made faster than the open loop model.

8.3 2-DOF Pole Placement Controller Design and Implementation using SBHS

We obtain a first order transfer function of the plant using the step test approach. Refer to the chapter on Step Test using SBHS for more details. The model obtained is

$$G(s) = \frac{0.42}{35.6s + 1} \quad (8.28)$$

with a time constant of $\tau = 35.6s$ and gain $K = 0.42$

After discretization with sampling time = 1 s, we obtain

$$G(z) = \frac{0.0116304z^{-1}}{1 - 0.9723086z^{-1}} \quad (8.29)$$

Refer to the Scilab code `myc2d.sci`¹. We shall now define good and bad factors as

$$\begin{aligned} A^g &= 1 - 0.9723086z^{-1} \\ A^b &= 1 \\ B^g &= 0.0116304 \\ B^b &= 1 \end{aligned}$$

Let us now define the transient specifications. We choose Rise Time = 100 s and Overshoot (ϵ) = 5%. Number of samples in one rise time (N_r), [5], is calculated as

$$\begin{aligned} N_r &\leq \frac{\text{Rise time}}{\text{Sampling time}} \\ &= 100 \\ \therefore \omega &= \frac{\pi}{2N_r} \\ &= 0.015708 \end{aligned}$$

¹Go the folder dc inside the 2dof_controller folder. Now go to the folder scilab and locate `myc2d.sci`

and

$$\begin{aligned}\rho &\leq \epsilon^{\omega/\pi} \\ &= 0.98513\end{aligned}$$

The closed loop characteristic polynomial is given by,

$$\begin{aligned}\phi_{cl} &= 1 - 2\rho \cos \omega z^{-1} + \rho^2 z^{-2} \\ &= 1 - 1.9700229z^{-1} + 0.9704870z^{-2}\end{aligned}$$

Refer to the Scilab code `desired.sci` to calculate N_r , ω , ρ and ϕ_{cl} . The code is available in the same location as `myc2d.sci`.

But according to equation 8.24,

$$A^b R_1 + z^{-k} B^b S_1 = \phi_{cl}$$

Recall that we have not considered external disturbance in the block diagram shown in figure 8.3. However, we can still up to some extent take care of the disturbances. This is achieved by using the Internal Model Principle. If a model of step is present inside the loop, step disturbances can be rejected [5]. We can apply this by forcing R_c to have this term. A step model is given by

$$1(z) = \frac{1}{1 - z^{-1}} = \frac{1}{\Delta}$$

Therefore,

$$R_c = B^g \Delta R_1$$

Δ has a root which lies on the unit circle. Hence it has to be treated as a bad part and should not be canceled out. Hence, we should make sure that all of the occurrences of R_1 have this term.

Therefore,

$$\phi_{cl} = A^b \Delta R_1 + z^{-k} B^b S_1 \quad (8.30)$$

Hence,

$$A^b \Delta R_1 + z^{-k} B^b S_1 = 1 - 1.9700229z^{-1} + 0.9704870z^{-2} \quad (8.31)$$

This expression is known as the Aryabhatta Identity and is solved using rigorous matrix calculations. The explanation of this operation is not considered here. Refer to [5] for more details on Aryabhatta's Identity. Refer to the Scilab code `pp_im.sci`, which is used to split the denominator and numerator polynomials of the plant transfer function into good and bad factors, and solving the Aryabhatta's Identity given in equation 8.31. On solving equation 8.31, we get

$$\begin{aligned} R_c &= 0.0116304 - 0.0229175z^{-1} + 0.0112871z^{-2} \\ S_c &= 0.0004641 - 0.0004512z^{-1} \\ T_c &= 1 - 0.9723z^{-1} \\ \gamma &= 0.0004641 \end{aligned}$$

All the above calculations are incorporated into a single Scilab code named `twodof_para.sce`².

8.3.1 Procedure to use the Scilab/Xcos Code for a Local Experiment

1. Change the directory to `local/2dof_controller`.
2. Execute the command `getd dc/scilab`
3. Open the Scilab code `twodof_para.sce`. Define the variable `TFcont` with first order transfer function (or second order transfer function) of your SBHS. Execute the Scilab code. With this, the 2-DOF controller parameters have been calculated. Figure 8.6 shows the calculated 2-DOF controller parameters on the Scilab console.
4. Open the Scilab code `twodof.sci`. Make sure that the first order control law (or second order control law in case of second order plant transfer function) is uncommented and the second order control law (or first order control law in case of second order transfer function) is commented.

²All the Scilab codes are given at the end of this chapter in the section 8.4

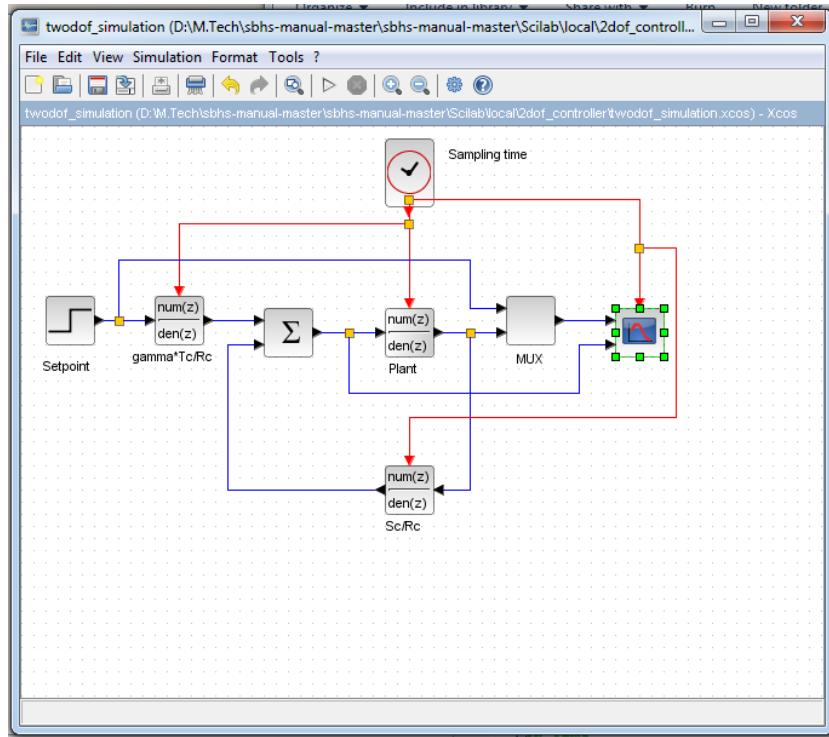


Figure 8.4: Xcos diagram for simulating 2-DOF controller

- Now follow the steps for performing a local experiment, as explained in the Chapter 2. Note that all occurrences of `step_test` will now be `twodof`. `step_test.sci` will be `twodof.sci`. `step_test.sce` will be `twodof.sce` and `step_test.xcos` will be `twodof.xcos`.

In the `twodof.xcos` Xcos file, change the setpoint to the desired value. Make sure the period of the clock block is the same as the sampling time used for discretisation. The Xcos diagram is shown in figure 8.7.

You can also use the Xcos file `twodof_simulation.xcos`, shown in figure 8.4, to simulate your controller before implementing on SBHS. This will help you validate your controller. You need to execute `getd dc/scilab` and then execute the file `twodof_para.sce`. Now run `twodof_simulation.xcos`. Figure 8.5 shows the simulation results. Note that, execution of this Xcos file is not mandatory for performing a virtual experiment.

The performance of the controller is shown in figure 8.8. It is seen that the

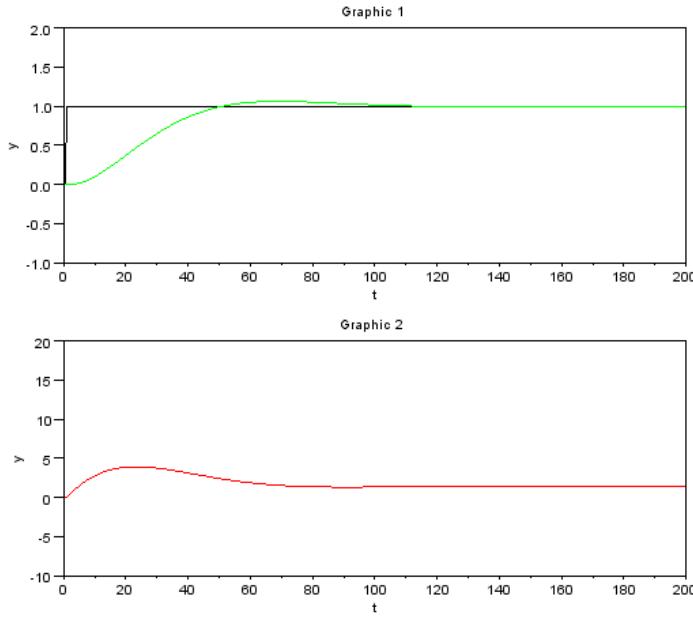


Figure 8.5: Simulation results after executing `twodof_simulation.xcos`

output (temperature) tracks the setpoint irrespective of the step changes in the fan speed. We see that the overshoot turns out to be 6% and rise time turns out to be 60 seconds, which is acceptable.

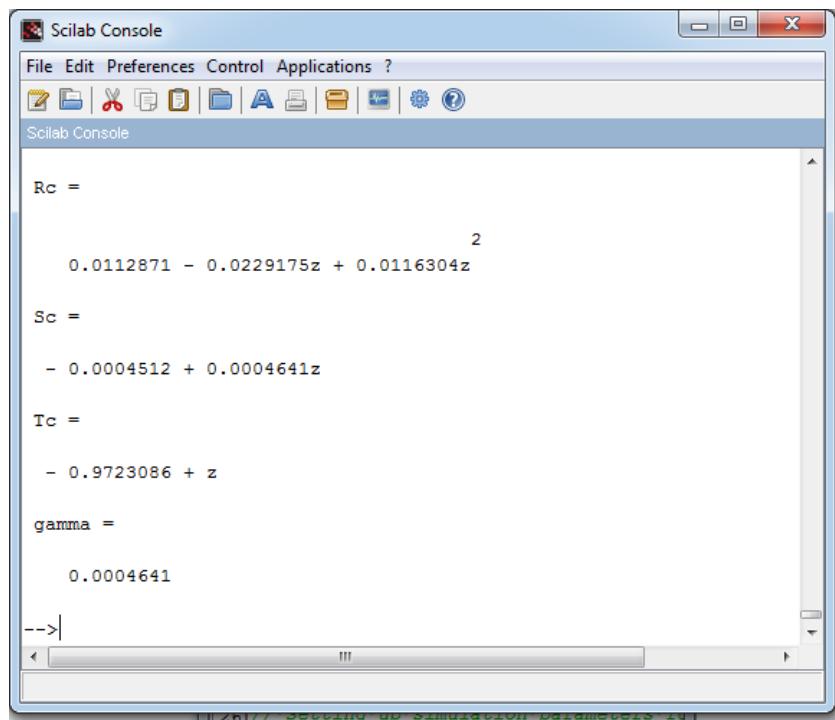
8.3.2 Implementing 2-DOF Controller on SBHS, Virtually

The step by step procedure for conducting a virtual experiment is explained in Chapter 3. After logging in using the SBHS Client, open Scilab. Change the directory to `virtual/2dof_controller`. Make sure you input the correct transfer function and control law as explained in 8.3.1. Now follow the steps for performing a virtual experiment, as explained in Chapter 3. Note that all occurrences of `step_test` will now be `twodof`.

All the required Scilab codes are listed in the section 8.5.

8.4 Scilab Code for Local Experiment

Scilab Code 8.1 `twodof_para.sce`



The image shows a screenshot of the Scilab software interface, specifically the 'Scilab Console' window. The window has a title bar 'Scilab Console' and a menu bar with options: File, Edit, Preferences, Control, Applications, and ?. Below the menu bar is a toolbar with various icons. The main area of the window displays the following Scilab code output:

```
Rc =  
      2  
    0.0112871 - 0.0229175z + 0.0116304z  
  
Sc =  
    - 0.0004512 + 0.0004641z  
  
Tc =  
    - 0.9723086 + z  
  
gamma =  
    0.0004641  
-->|
```

Figure 8.6: Scilab output for `twodof_para.sce`

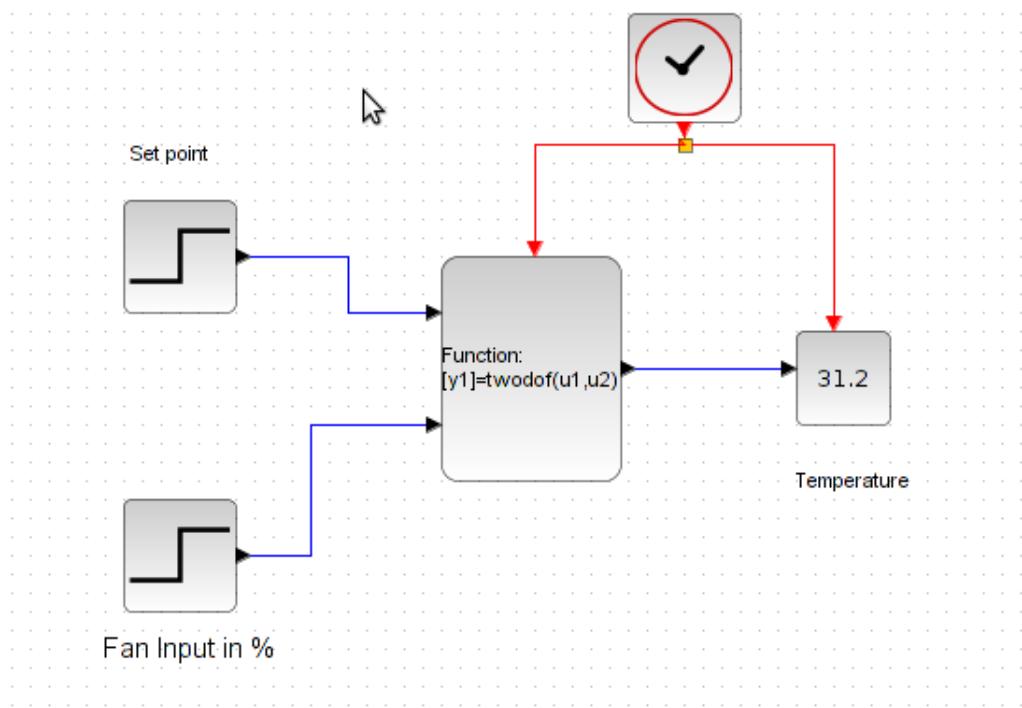


Figure 8.7: Xcos diagram for 2-DOF controller

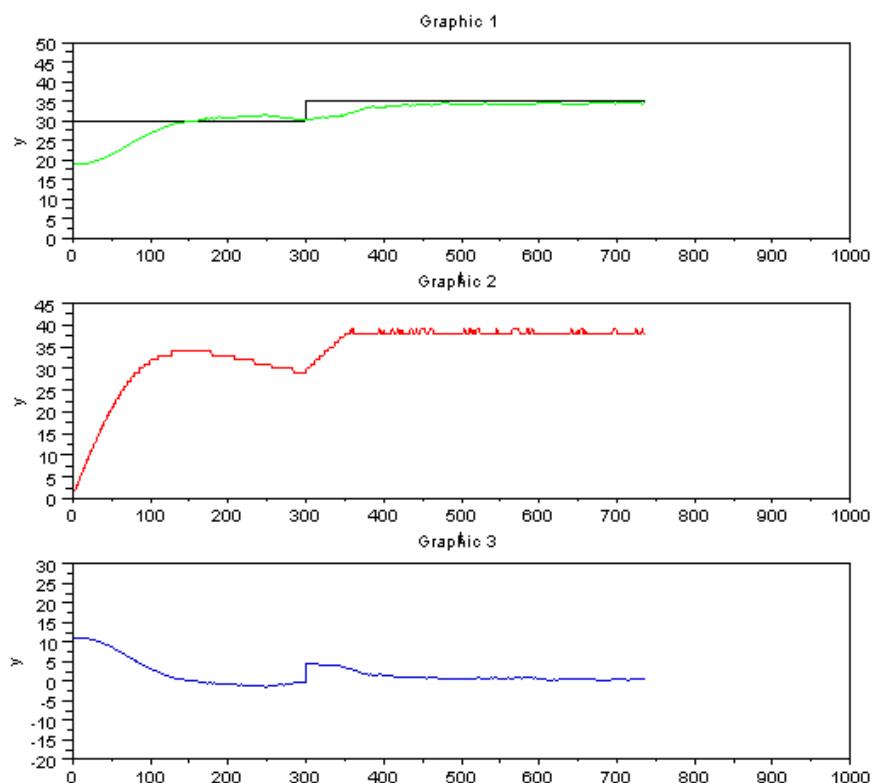


Figure 8.8: Implementation of 2-DOF controller

```

1 mode(0)
2 s=%s;
3 z=%z;
4 global Rc Sc Tc gamm
5 // TFcont = syslin('c', -280.14/((s-31.32)*(s+100)*(s
+31.32)));
6 // TFcont = syslin('c', 0.667/((73.5*s+1)*(1*s+1))) //
second order
7 TFcont = syslin('c', 0.668/(75.013*s+1)) // first order
8 SScont = tf2ss(TFcont);
9 // TFdisc = ss2tf(SScont);
10 Ts = 0.5;
11 [B,A,k] = myc2d(SScont,Ts);
12
13 // polynomials are returned
14 [Ds,num,den] = ss2tf(SScont);
15 num = clean(num); den = clean(den);
16
17 // Transient specifications
18 rise = 35; epsilon = 0.05;
19 phi = desired(Ts,rise,epsilon);
20
21 // Controller design
22 Delta = [1 -1];
23 [Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi); // with integral
24
25 // Setting up simulation parameters for basic.cos
26 st = 0.0001; // desired change in h, in m.
27 t_init = 0; // simulation start time
28 t_final = 0.5; // simulation end time
29
30 // Setting up simulation parameters for c_ss_cl.cos
31 N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;
32
33 [Tc1,Rc1] = cosfil_ip(Tc,Rc); // Tc / Rc
34 [Sc2,Rc2] = cosfil_ip(Sc,Rc); // Sc / Rc
35
36 [Bp] = cosfil_ip(B,1);

```

```

37 [Ap] = cosfil_ip(A,1);
38
39 [Tcp1,Tcp2] = cosfil_ip(Tc,1); // Tc / 1
40 [Np,Rcp] = cosfil_ip(N,Rc); // 1 / Rc
41 [Scp1,Scp2] = cosfil_ip(Sc,1); // Sc / 1
42 [Cp,Dp] = cosfil_ip(C,D); // C / D
43
44 // R c 1 = R c ( 1 ) ; R c 2 = R c ( 2 ) ; R c 3 = R c ( 3 ) ; R c 4 = R c ( 4 ) ;
45 // S c 1 = S c ( 1 ) ; S c 2 = S c ( 2 ) ;
46 // S c 3 = S c ( 3 ) ;
47 // T c 1 = T c ( 1 ) ; T c 2 = T c ( 2 ) ;
48 // T c 3 = T c ( 3 ) ;
49 disp(Rcp,'Rc =')
50 disp(Scp1,'Sc =')
51 disp(Tcp1,'Tc =')
52 disp(gamm,'gamma =')
53
54
55
56 // u _ o l d _ o l d = 1 ;
57 // u _ o l d = 1 ;
58 // r _ o l d _ o l d = 1 ;
59 // r _ o l d = 1 ;
60 // y _ o l d _ o l d = 1 ;
61 // y _ o l d = 1 ;

```

Scilab Code 8.2 twodof.sci

```

1 mode(0)
2 function [temp] = twodof(setpoint,fan)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
   e_old_old
4
5 global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name
6
7 global temp u_old_old u_old r_old_old r_old y_old_old
   y_old u_new heat r_new y_new

```

```

8
9 Ts=sampling_time;
10
11 r_new = setpoint;
12 y_new = temp;
13 et = setpoint-temp;
14
15 // u_new = (1 / Rc(1)) * (gamm*Tc(1) * r_new + gamm*Tc(2) *
16 r_old + r_old * Tc(3) * gamm - Sc(1) * y_new - Sc(2) *
17 y_old - Sc(3) * y_old * old - Rc(2) * u_old - Rc(3) *
18 u_old * old); // second order control law
19
20 u_new = (1/Rc(1))*(gamm*Tc(1)*r_new + gamm*Tc(2)*r_old
21 -Sc(1)*y_new -Sc(2)*y_old-Rc(2)*u_old - Rc(3)*
22 u_old * old); // first order control law
23
24 u_old * old = u_old;
25
26 u_old = u_new;
27 r_old * old=r_old;
28 r_old = r_new;
29 y_old * old=y_old;
30 y_old = y_new;
31
32 heat = u_new;
33 temp = comm(heat,fan);
34
35 plotting([heat fan temp setpoint],[0 0 20 0],[100
36 100 40 1000])
37
38 m=m+1;
39
40 endfunction

```

Scilab Code 8.3 start.sce

```

1 getd ../common_files/
2 getd dc/scilab
3 exec ../common_files/loader.sce

```

```

4
5 exec ser_init.sce
6
7 exec twodof_para.sce
8 exec twodof.sci
9
10 xcos twodof.xcos

```

Scilab Code 8.4 cindep.sci

```

1 // Updated ----No change
2 // function b = cindep( S , gap )
3 // used in XD + YN = C . all rows except the last of
4 // are assumed to
5 // be independent . The aim is to check if the last
6 // row is dependent on the
7 // rest and if so how . The coefficients of dependence
8 // is sent in b
9 function b = cindep( S , gap )
10
11 if argn(2) == 1
12     gap = 1.0e8 ;
13 end
14 eps = 2.2204e-016;
15 [rows , cols] = size(S);
16 if rows > cols
17     ind = 0;
18 else
19     sigma = svd(S);
20     len = length(sigma);
21     if (sigma(len)/sigma(1) <= (eps*max(i , cols)))
22         ind = 0;                                // not independent
23     else
24         if or(sigma(1:len-1) ./ sigma(2:len)>=gap)
25             ind = 0;                                // not dependent
26         else
27             ind = 1;                                // independent
28         end

```

```

26     end
27 end
28 if ind
29   b = [];
30 else
31   b = S(rows,:)/S(1:rows-1,:);
32   b = makezero(b,gap);
33 end
34 endfunction

```

Scilab Code 8.5 clcoef.sci

```

1 // Updated ----- No change
2 // H. Kwakernaak, July , 1990
3 // Modified by Kannan Moudgalaya in Nov. 1992
4
5 function [P,degP] = clcoef(Q,degQ)
6
7 [rQ,cQ] = polsize(Q,degQ);
8
9 if and(and(Q==0))
10   P = zeros(rQ,cQ);
11   degP = 0;
12 else
13   P = Q; degP = degQ; rP = rQ; cP = cQ;
14   j = degP+1;
15   while j >= 0
16     X = P(:,(j-1)*cP+1:j*cP)
17     if max(sum(abs(X'))) < (1e-8)*max(sum(abs(P)))
18       P = P(:,1:(j-1)*cP);
19       degP = degP-1;
20     else
21       j = 0;
22     end
23     j = j-1;
24   end
25 end
26 endfunction

```

Scilab Code 8.6 colsplit.sci

```
1 // colsplit
2 // The command
3 // [P1 , degP1 , P2 , degP2 ] = colsplit (P , degP , p1 , p2 )
4 // produces two polynomial matrix P1 and P2 . P1
5 // consists of the first
6 // p1 columns of P and P2 consists of the remaining p2
7 // columns of P .
8
9
10 function [P1 ,degP1 ,P2 ,degP2 ] = colsplit(P ,degP ,p1 ,p2 )
11
12 if isempty(P)
13     P1 = [] ; P2 = [] ;
14     degP1 = 0 ; degP2 = 0 ;
15     return ;
16 end
17
18 [rP ,cP ] = polsize (P ,degP ) ;
19 if p1 < 0 | p1 > cP | p2 < 0 | p2 > cP | p1+p2 ~= cP
20     error( 'colsplit: Inconsistent numbers of columns' );
21 end
22 rP1 = rP ; rP2 = rP ; cP1 = p1 ; cP2 = p2 ;
23 degP1= degP ; degP2 = degP ;
24
25 if p1 == 0
26     P1 == [] ; P2 = P ;
27 elseif p2 == 0
28     P1 = P ; P2 = [] ;
29 else
30     P1 = zeros (rP1 ,( degP1+1)*cP1 ) ; P2 = zeros (rP2 ,(
31         degP2+1)*cP2 ) ;
32     for i = 1:degP+1
```

```

32     P1(:,(i-1)*cP1+1:i*cP1) = P(:,(i-1)*cP+1:(i-1)*
33                               cP+cP1);
34     P2(:,(i-1)*cP2+1:i*cP2) = P(:,(i-1)*cP+cP1+1:i*
35                               cP);
36   end
37 end
38 endfunction;

```

Scilab Code 8.7 cosfil_ip.sci

```

1 // Updated (31-7-07)
2 // Input arguments are numerator and denominator
3 // polynomials' coefficients in ascending
4 // powers of z^-1
5
6 // Scicos blocks need input polynomials
7 // with positive powers of z
8
9 function [nume,deno] = cosfil_ip(num,den)
10
11 [Nn,Nd] = polyno(num,'z');
12 [Dn,Dd] = polyno(den,'z');
13 nume = Nn*Dd;
14 deno = Nd*Dn;
15
16 endfunction;

```

Scilab Code 8.8 indep.sci

```

1 // Updated ----No change
2 // function b = indep(S,gap)
3 // determines the first row that is dependent on the
4 // previous rows of S.
5 // The coefficients of dependence is returned in b
6 function b = indep(S,gap)
7 if argn(2) == 1
8     gap = 1.0e8;

```

```

9         end
10    [rows ,cols ] = size(S);
11    ind = 1;
12    i = 2;
13    eps = 2.2204e-016;
14    while ind & i <= rows
15        sigma = svd(S(1:i,:));
16        len = length(sigma);
17        if (sigma(len)/sigma(1) < (eps*max(i ,cols )))
18            ind =0;
19        else
20            shsig = [sigma(2:len);sigma(len)];
21            if or( (sigma ./ shsig) > gap)
22                ind = 0;
23            else
24                ind = 1;
25                i = i+1;
26            end
27        end
28
29    end
30    if ind
31        b = [];
32
33    else
34        c = S(i,:)/S(1:i-1,:);
35        c = makezero(c,gap);
36        b = [-c 1];
37    end
38 endfunction

```

Scilab Code 8.9 left_prm.sci

```

1 //  f u n c t i o n [ B , degB ,A , degA ,Y , degY ,X , degX ] = ...
2 //  l e f t _ p r m ( N , degN ,D , degD ,job , gap )
3 //
4 //  d o e s  t h r e e  d i f f e r e n t  t h i n g s  a c c o r d i n g  t o  i n t e g e r s
5 //  t h a t  ' j o b '  t a k e s

```

```

5 // job = 1.
6 // this is the default. It is always done for all
jobs.
7 // -1
-1
8 // Given ND , returns coprime B and A where ND = A
B
9 // It is enough if one sends the first four input
arguments
10 // If gap is required to be sent, then one can send
either 1 or a null
11 // entry for job
12 // job = 2.
13 // first solve for job = 1 and then solve XA + YB = I
14 // job = 3.
15 // used in solving XD + YN = C
16 // after finding coprime factorization, data are
returned
17 //
18 // convention: the variable with prefix deg stand for
degrees
19 // of the corresponding polynomial matrices
20 //
21 // input:
22 // N: right fraction numerator polynomial matrix
23 // D: right fraction denominator polynomial matrix
24 // N and D are not necessarily coprime
25 // gap: variable used to zero entries; default value
is 1.0e+8
26 //
27 // output
28 // b and A are left coprime num. and den. polynomial
matrices
29 // X and Y are solutions to Aryabhatta identity, only
for job = 2
30
31 function [B,degB,A,degA,Y,degY,X,degX] = left_prm(N,
degN,D,degD,job,gap)

```

```

32 if argn(2) == 4 | argn(2) == 5
33 gap = 1.0e8 ;
34 end
35 // pause
36 if argn(2) == 4,
37 job = 1; end
38 [F,degF] = rowjoin(D,degD,N,degN);
39 [Frows,Fbcols] = polysize(F,degF);           // Fbcols =
40 Fcols = Fbcols * (degF+1);                   // actual
41 columns of F
42 T1 = []; pr = []; degT1 = 0; T1rows = 0; shft = 0;
43 S=F; sel = ones(Frows,1); T1bcols = 1;
44 abar = (Fbcols + 1):Frows;                  // a superbar
45 of B-C. Chang
46 while isempty(T1) | T1rows < Frows - Fbcols
47 Srows = Frows*T1bcols; // max actual columns of
48 result
49 [T1,T1rows,sel,pr] = ...
50 t1calc(S,Srows,T1,T1rows,sel,pr,Frows,
51 Fbcols,abar,gap);
52 [T1rows,T1cols] = size(T1);
53 if T1rows < Frows - Fbcols
54 T1 = [T1 zeros(T1rows,Frows)];
55 T1bcols = T1bcols + 1;                      // max. block
56 columns of result
57 degT1 = degT1 + 1;                          // degree of
58 result
59 shft = shft +Fbcols;
60 S = seshft(S,F,shft);
61 sel = [ sel; sel(Srows-Frows+1:Srows) ];
62 rowvec = (T1bcols-1)*Frows+(Fbcols+1):T1bcols
63 * Frows;
64 abar = [ abar rowvec];                     // A superbar
65 of B-C. chang
66 end
67 end

```

```

61 [B,degB,A,degA] = colsplit(T1,degT1,Fbcols,Frows-
   Fbcols);
62 [B,degB] = clcoef(B,degB);
63 B = -B;
64 [A,degA] = clcoef(A,degA);
65 // pause
66 if job == 2
   S = S(mtbl_logical(sel),:);
   // columns
68 [redSrows,Scols] = size(S);
69 C = [eye(Fbcols,Fbcols) zeros(Fbcols,Scols-
   Fbcols)]; // append with zeros
70 T2 = C/S;
71 T2 = makezero(T2,gap);
72 T2 = move_sci(T2,find(sel),Srows);
73 [X,degX,Y,degY] = colsplit(T2,degT1,Fbcols,Frows-
   -Fbcols);
74 [X,degX] = clcoef(X,degX);
75 [Y,degY] = clcoef(Y,degY);
76 elseif job == 3
77 Y = S;
78 degY = sel;
79 X = degT1;
80 degX = Fbcols;
81 else
82   if job ~= 1
83     error('Message from left prm: no legal job
           number specified')
84   end
85 end
86 endfunction

```

Scilab Code 8.10 makezero.sci

```

1 // Updated
2 // function B = makezero(B, gap)
3 // where B is a vector and gap acts as a tolerance
4

```

```

5 function B = makezero(B, gap)
6
7 if argn(2) == 1
8     gap = 1.0e8;
9 end
10 temp = B(find(B));           // non zero entries of B
11 temp = -gsort(-abs(temp));  // absolute values sorted
12 len = length(temp);
13 ratio = temp(1:len-1) ./ temp(2:len); // each ratio >1
14 min_ind = min(find(ratio>gap));
15 if ~isempty(min_ind)
16     our_eps = temp(min_ind+1);
17     zeroind = find(abs(B)<=our_eps);
18     B(zeroind) = zeros(1,length(zeroind));
19 end
20 endfunction

```

Scilab Code 8.11 move_sci.sci

```

1 // function result = move_sci(b, nonred, max_sci)
2 // Moves matrix b to matrix result with the
   information on where to move,
3 // decided by the indices of nonred.
4 // The matrix result will have as many rows as b has
   and max number of columns.
5 // b is augmented with zeros to have nonred number of
   columns;
6 // The columns of b put into those of result as
   decided by nonred.
7
8 function result = move_sci(b,nonred,max_sci)
9 [brows,bcols] = size(b);
10 b = [b zeros(brows,length(nonred)-bcols)];
11 result = zeros(brows,max_sci);
12 result(:,nonred') = b;
13 endfunction

```

Scilab Code 8.12 polisize.sci

```
1 // Updated ----- No change
2 // function [rQ ,cQ ] = polsize ( Q , degQ )
3 // FUNCTION polsize TO DETERMINE THE DIMENSIONS
4 // OF A POLYNOMIAL MATRIX
5 //
6 // H. Kwakernaak , August , 1990
7
8 function [rQ ,cQ ] = polsize ( Q , degQ )
9
10 [rQ ,cQ ] = size ( Q ) ; cQ = cQ / ( degQ + 1 ) ;
11 if abs ( round ( cQ ) - cQ ) > 1e-6
12     error ( ' polsize : Degree of input inconsistent with
           number of columns ' );
13 else
14     cQ = round ( cQ );
15 end
16 endfunction
```

Scilab Code 8.13 polyno.sci

```
1 // Updated ( 1 - 8 - 07 )
2 // Operations :
3 // Polynomial definition
4 // Flipping of coefficients
5 // Variable ----- passed as input argument ( either ,
       ' s ' or ' z ' )
6 // Both num and den are used mostly used in scicos
       files ,
7 // to get rid of negative powers of z
8
9 // Polynomials with powers of s need to
10 // be flipped only
11
12 function [ polynu , polyde ] = polyno ( zc , a )
13 zc = clean ( zc );
14 polynu = poly ( zc ( length ( zc ) : - 1 : 1 ) , a , ' coeff ' );
```

```

15   if a == 'z'
16     polyde = %z^(length(zc) - 1);
17   else
18     polyde = 1;
19   end
20
21 // Scicos (4.1) Filter block shouldn't have constant /
22 // constant
22   if type(polynu)==1 & type(polyde)==1
23     if a == 'z'
24       polynu = %z; polyde = %z;
25     else
26       polynu = %s; polyde = %s;
27     end;
28   end;
29
30 endfunction

```

Scilab Code 8.14 rowjoin.sci

```

1 // Updated -----No change
2 // function [P, degP] = rowjoin(P1, degP1, P2, degP2)
3 // MATLAB FUNCTION rowjoin TO SUPERPOSE TWO POLYNOMIAL
4 // MATRICES
5
6 // H. Kwakernaak, July, 1990
7
8 function [P, degP] = rowjoin(P1, degP1, P2, degP2)
9
10 [rP1, cP1] = polsize(P1, degP1);
11 [rP2, cP2] = polsize(P2, degP2);
12 if cP1 ~= cP2
13   error('rowjoin: Inconsistent numbers of columns');
14 end
15
16 rP = rP1+rP2; cP = cP1;
17 if degP1 >= degP2
18   degP = degP1;

```

```

19 else
20     degP = degP2;
21 end
22
23 if isempty(P1)
24     P = P2;
25 elseif isempty(P2)
26     P = P1;
27 else
28     P = zeros(rP,(degP+1)*cP);
29     P(1:rP1,1:(degP1+1)*cP1) = P1;
30     P(rP1+1:rP,1:(degP2+1)*cP2) = P2;
31 end
32 endfunction

```

Scilab Code 8.15 seshft.sci

```

1 // Updated ----- No change
2 // function C = seshft(A,B,N)
3 // given A and B matrices, returns C = [ <-A-> 0
4 //                                     0 <-B->] with B
5 // shifted east by N cols
6
6 function C = seshft(A,B,N)
7 [Arows , Acols ] = size(A);
8 [Brows , Bcols ] = size(B);
9 if N >= 0
10    B = [zeros(Brows ,N) B];
11    Bcols = Bcols + N;
12 elseif N < 0
13    A = [zeros(Arows ,abs(N)) A];
14    Acols = Acols +abs(N);
15 end
16 if Acols < Bcols
17    A = [A zeros(Arows ,Bcols-Acols ) ];
18 elseif Acols > Bcols
19    B = [B zeros(Brows ,Acols-Bcols ) ];
20 end

```

```

21   C = [A
22     B];
23 endfunction

```

Scilab Code 8.16 t1calc.sci

```

1 // Updated
2 // function [T1 , T1rows , sel , pr ] = ...
3 // t1calc (S , Srows , T1 , T1rows , sel , pr , Frows , Fbcols , abar ,
4 // gap )
5 // calculates the coefficient matrix T1
6 // redundant row information is kept in sel : redundant
// rows are marked
7 // with zeros . The undeleted rows are marked with
8 // ones .
9
10
11 function [T1 , T1rows , sel , pr ] = t1calc (S , Srows , T1 , T1rows
12   , sel , pr , Frows , Fbcols , abar , gap )
13 b = 1; // vector of
14 primary red. rows
15
16 while (T1rows < Frows - Fbcols) & or(sel==1) & ~
17 isempty(b)
18   S = clean(S);
19   b = indep(S(mtlb_logical(sel),:),gap); // send
20   selected rows of S
21   if ~isempty(b)
22     b = clean(b);
23     b = move_sci(b,find(sel),Srows);
24     j = length(b);
25     while ~(b(j) & or(abar==j)) // pick largest
26       non zero entry
27       j = j-1; // of coeff .
28       belonging to abar
29     if ~j
30       fprintf( '\nMessage from t1calc ,
31           called from left_prm\n\n' )

```

```

22             error( 'Denominator is noninvertible
23             ')
24         end
25     end
26     if ~or(j<pr & pmodulo(pr,Frows) == pmodulo(j ,
27         Frows)) // pr(2), pr(1)
28         T1 = [T1; b]; // condition
29         is not violated
30         T1rows = T1rows +1; // accept this
31         vector
32     end // else don't
33     accept
34     pr = [pr; j]; // update
35     prime red row info
36     while j <= Srows
37         sel(j) = 0;
38         j = j + Frows;
39     end
40 end
41 endfunction

```

8.5 Scilab Code for Virtual Experiment

Scilab Code 8.17 twodof_para.sce

```

1 mode(0)
2 global Rc Sc Tc gamm u_old_old u_old r_old_old r_old
3 y_old_old y_old u_new r_new y_new
4 s=%s;
5 z=%z;
6 // TFcont = syslin('c',0.593/((47.21*s+1)*(1.373*s+1)))
7 ;// second order
8 // TFcont = syslin('c',0.594/(49.19*s+1)) // first order
9 TFcont = syslin('c',0.42/(35.61*s+1)); // first order
10 SScont = tf2ss(TFcont);
11 Ts = 1;

```

```

10 [B,A,k] = myc2d(SScont,Ts);
11
12 // polynomials are returned
13 [Ds,num,den] = ss2tf(SScont);
14 num = clean(num); den = clean(den);
15
16 // Transient specifications
17 rise = 100; epsilon = 0.05;
18 phi = desired(Ts,rise,epsilon);
19
20 // Controller design
21 Delta = [1 -1];
22 [Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi,Delta); // with
    integral
23
24 // initial values
25 u_old_old = 0;
26 u_old = 0;
27 r_old_old = 0;
28 r_old = 0;
29 y_old_old = 0;
30 y_old = 0;

```

Scilab Code 8.18 twodof.sce

```

1 mode(0)
2 global fdfh fdt fnr fncw m err_count y limits
    sampling_time m heat temp
3
4 // *****
5 sampling_time=1; // In seconds. Fractions are allowed
6 // *****
7 getd('dc/scilab');
8 exec("twodof_para.sce")
9 exec ("twodof.sci");
10
11 ok = init();
12

```

```

13  if ok~= [] // open xcos only if communication is
14    through ( ie reply has come from server )
15    xcos( 'twodof.xcos' );
16  else
17    disp ("NO NETWORK CONNECTION!");;
18  return
19
end

```

Scilab Code 8.19 twodof.sci

```

1 function [ stop ] = twodof( setpoint , fan )
2   global temp u_old_old u_old r_old_old r_old
3           y_old_old y_old u_new heat r_new y_new
4
5   r_new = setpoint ;
6   y_new = temp ;
7   // u_new = ( 1 / Rc ( 1 ) ) * ( gamm * Tc ( 1 ) * r_new + gamm * Tc ( 2 )
8   // * r_old + r_old_old * Tc ( 3 ) * gamm - Sc ( 1 ) * y_new - Sc
9   // ( 2 ) * y_old - Sc ( 3 ) * y_old_old - Rc ( 2 ) * u_old - Rc
10  // ( 3 ) * u_old_old ) ; // second order control law
11
12  u_new = ( 1 / Rc ( 1 ) ) * ( gamm * Tc ( 1 ) * r_new + gamm * Tc ( 2 ) *
13    r_old - Sc ( 1 ) * y_new - Sc ( 2 ) * y_old - Rc ( 2 ) * u_old - Rc
14    ( 3 ) * u_old_old ) ; // first order control law
15  heat = u_new ;
16  [ stop , temp ] = comm( heat , fan ) ; // Never edit this
17  // line
18  plotting ([ heat fan temp setpoint ] , [ ] , [ ]) ;
19
20 endfunction

```

8.6 Scilab Codes Common for both Local and Virtual Experiments

Scilab Code 8.20 myc2d.sci

```
1 // Updated (26-7-07)
2 // 9.2
3 // function [B,A,k] = myc2d(G,Ts)
4 // Produces numerator and denominator of discrete
   transfer
5 // function in powers of z^{-1}
6 // G is continuous transfer function; time delays are
   not allowed
7 // Ts is the sampling time, all in consistent time
   units
8 // User defined function
9 // -----
10
11 function [B,A,k] = myc2d(G,Ts)
12 H = ss2tf(dscr(G,Ts));
13 num1 = coeff(H('num'));
14 den1 = coeff(H('den'));// -----
15 A = den1(length(den1):-1:1);
16 num2 = num1(length(num1):-1:1); // flip
17 nonzero = mtlb_find(num1);
18 first_nz = nonzero(1);
19 B = num2(first_nz:length(num2)); // -----
20 k = length(den1) - length(num1);
21 endfunction
```

Scilab Code 8.21 desired.sci

```
1 // Updated (26-7-07)
2 // 9.4
3 function [phi,dphi] = desired(Ts,rise,epsilon)
4
5 Nr = rise/Ts; omega = %pi/2/Nr; rho = epsilon^(omega/
```

```

    %pi);
6 phi = [1 -2*rho*cos(omega) rho^2]; dphi = length(phi)
    -1;
7 endfunction;

```

Scilab Code 8.22 polmul.sci

```

1 // Updated ----- No change
2 // polmul
3 // The command
4 // [C, degC] = polmul(A, degA, B, degB)
5 // produces the polynomial matrix C that equals the
// product A*B of the
6 // polynomial matrices A and B.
7 //
8 // H. Kwakernaak, July , 1990
9
10
11 function [C,degC] = polmul(A,degA,B,degB)
12 [rA,cA] = polysize(A,degA);
13 [rB,cB] = polysize(B,degB);
14 if cA ~= rB
15     error('polmul: Inconsistent dimensions of input
// matrices');
16 end
17
18 degC = degA+degB;
19 C = [];
20 for k = 0:degA+degB
21     mi = 0;
22     if k-degB > mi
23         mi = k-degB;
24     end
25     ma = degA;
26     if k < ma
27         ma = k;
28     end
29 Ck = zeros(rA,cB);

```

```

30   for i = mi:ma
31     Ck = Ck + A(:, i*cA+1:(i+1)*cA)*B(:, (k-i)*cB
32       +1:(k-i+1)*cB);
33   end
34   C = [C Ck];
35 endfunction

```

Scilab Code 8.23 polssplit3.sci

```

1 // Updated (18 -7 -07)
2 // 9.11
3 // function [goodpoly , badpoly ] = polssplit3 ( fac , a )
4 // Splits a scalar polynomial of  $z^{-1}$  into good and
5 // factors . Input is a polynomial in increasing degree
6 // of
7 //  $z^{-1}$ . Optional input is a , where a <= 1 .
8 // Factors that have roots outside a circle of radius
9 // a or
10 // with negative roots will be called bad and the rest
11 // good . If a is not specified , it will be assumed as
12 // 1 .
13
14
15
16
17
18 // extract good and bad roots
19 badindex = mtlb_find((abs(rts)>=a-1.0e-5)|(real(rts)
20 <-0.05));
21 badpoly = coeff(poly(rts(badindex) , 'z'));
22 goodindex = mtlb_find((abs(rts)<a-1.0e-5)&(real(rts)
23 >=-0.05));
24 goodpoly = coeff(poly(rts(goodindex) , 'z'));

```

```

23
24 // scale by equating the largest terms
25 [m, index] = max(abs(fac));
26 goodbad = convol(goodpoly, badpoly);
27 goodbad = goodbad(length(goodbad):-1:1);
28 factor1 = fac(index)/goodbad(index);
29 goodpoly = goodpoly * factor1;
30 goodpoly = goodpoly(length(goodpoly):-1:1);
31 badpoly = badpoly(length(badpoly):-1:1);
32 endfunction;

```

Scilab Code 8.24 pp_im.sci

```

1 // Updated (27-7-07)
2 // 9.8
3 // function [Rc , Sc , Tc , gamma , phit ] = pp_im (B , A , k , phi ,
4 // Delta )
5 // -----
6 function [Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi,Delta)
7
8 // Setting up and solving Aryabhatta identity
9 [Ag,Ab] = polsplit3(A); dAb = length(Ab) - 1;
10 [Bg,Bb] = polsplit3(B); dBb = length(Bb) - 1;
11
12 [zk,dzk] = zpowk(k);
13
14 [N,dN] = polmul(Bb,dBb,zk,dzk);
15 dDelta = length(Delta)-1;
16 [D,dD] = polmul(AB,dAb,Delta,dDelta);
17 dphi = length(phi)-1;
18
19 [S1,dS1,R1,dR1] = xdync(N,dN,D,dD,phi,dphi);
20
21 // Determination of control law
22 Rc = convol(Bg,convol(R1,Delta)); Sc = convol(Ag,S1);
23 Tc = Ag; gamm = sum(phi)/sum(Bb);
24 endfunction;

```

Scilab Code 8.25 xdync.sci

```
1 // Updated ----No change
2 // function [Y, degY ,X ,degX ,B ,degB ,A ,degA ] = xdync(N ,
3 // given coefficient matrix in T1 , primary redundant
4 // row information sel ,
5 // solves XD + YN = C
6 // calling order changed on 16 April 2005. Old order :
7 // function [B ,degB ,A ,degA ,Y ,degY ,X ,degX ] = xdync(N ,
8 //degN ,D ,degD ,C ,degC ,gap )
9
10 function [Y,degY ,X ,degX ,B ,degB ,A ,degA ] = xdync(N,degN ,
11 D,degD ,C,degC ,gap )
12 if argn(2) == 6
13     gap = 1.0e+8;
14 end
15
16 [F,degF] = rowjoin(D,degD ,N,degN );
17
18 [Frows ,Fbcols ] = polsize(F,degF); // Fbcols = block
19 // columns
20
21 [B,degB ,A,degA ,S ,sel ,degT1 ,Fbcols ] = left_prm(N,degN ,D
22 ,degD ,3 ,gap );
23 // if issoln (D ,degD ,C ,degC ,B ,degB ,A ,degA )
24     [Crows ,Ccols ] = size(C);
25     [Srows ,Scols ] = size(S);
26     S = clean(S);
27     S = S(mtlb_logical(sel ),:);
28     T2 =[];
```

```

29             b = move_sci(b, find(sel), Srows);
30             T2 =[T2; b];
31         end
32
33     [X,degX,Y,degY] = colsplit(T2,degT1,Fbcols,Frows-
34     Fbcols);
35
36     [X,degX] = clcoef(X,degX);
37     [Y,degY] = clcoef(Y,degY);
38     Y = clean(Y); X = clean(X);
endfunction

```

Scilab Code 8.26 zpowk.sci

```

1 // Updated (26 - 7 - 07)
2 // 9.6
3 // -----
4
5 function [zk,dzk] = zpowk(k)
6 zk = zeros(1,k+1); zk(1,k+1) = 1;
7 dzk = k;
8 endfunction

```

Chapter 9

PRBS Modeling and Implementation of Pole Placement Controller

The aim of this chapter is to do PRBS testing on Single Board Heater System by the application of PRBS signal and to design a pole-placement controller. The target group is anyone who has basic knowledge of control engineering.

We have used Scilab with Xcos as an interface for sending and receiving data. This interface is shown in figure 9.1. Heater current and fan speed are the two inputs to the system. The heater current is varied with a PRBS signal. A provision is made to set the parameters like PRBS amplitude and offset value. A provision is also made to time the occurrence of the PRBS input using a step block. The value of step time in the step block has to be chosen carefully. Sufficient amount of time should be given to allow the temperature to reach a steady-state before the PRBS signal is applied. In this experiment we are keeping the fan speed constant at 50%. The temperature profile thus obtained is the output.

The first half of this chapter is dedicated to do system identification of the SBHS system using the response obtained for a PRBS (Pseudo Random Binary Sequence) input. In the second half, a pole-placement controller is designed using this model and implemented on SBHS.

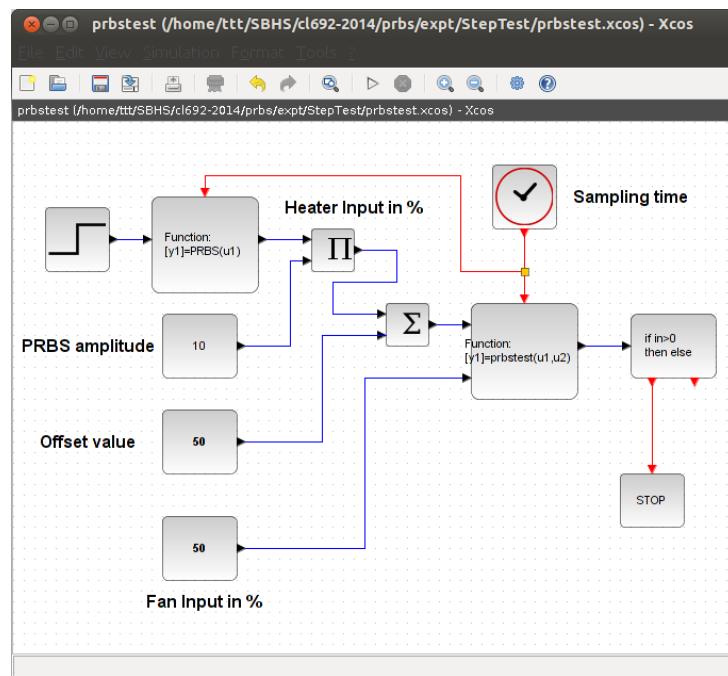


Figure 9.1: Xcos for PRBS testing experiment

9.1 Issues with Step Test and an Alternate Approach

SBHS is an example of a heater. Suppose you are working in a full scale plant. Current control system designed to control one of the heaters of the plant is lousy and your supervisor asks you to design a new controller from scratch. The first step you need to do is identification of the heater transfer function. The catch is, the plant is currently operational. You can't shut the plant down to identify the heater transfer function. You have to do it while the heater is operating in the plant. You might think of giving the heater a positive step and measuring the response in the controlled temperature. This will increase the temperature of the component being heated for the period of time step is applied. However, if the process is sensitive to temperature of the component (distillation, for example), it will go off the desired course and the output of the whole plant will be affected and will be undesirable.

There is an alternate approach which is widely used in industry. The input given to the heater for identification is not step, but a **pseudo-random binary sequence (PRBS)**. The concept behind PRBS is that the input is perturbed in such a way that the time average of the input is the value at which it is being operated currently. Thus, some positive and some negative steps can be given. This results in some positive and some negative changes in the temperature which leads to the time average of the performance of the plant remaining the same. Thus, PRBS testing can be done in a working plant without affecting the plant performance unlike step testing. A typical PRBS and corresponding plant output is shown in figure 9.2

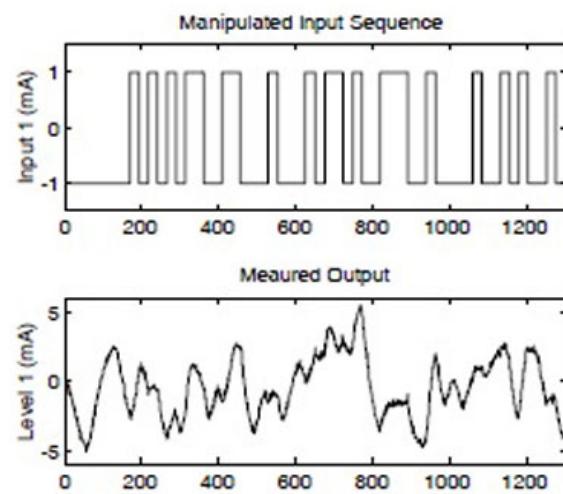


Figure 9.2: PRBS testing input and output [Image source: CL 686 Advanced Process Control, Spring 2013-14 lecture slides. Prof. S. C. Patwardhan, IIT Bombay]

9.2 Step by step procedure to do PRBS testing

Similar to Chapter 4 and 5, in this section we will find the transfer function model of SBHS. But there are two major differences. First difference is that we will give a Pseudo Random Binary Sequence to the heater input of SBHS and the second difference is that we will find the discrete time transfer function. A Pseudo Random Binary Sequence is nothing but a signal whose amplitude varies between two limits randomly at any given time. An illustration of the same is given in figure 9.3. A PRBS signal can be easily generated using the `rand()` function in Scilab. Scilab code to generate the PRBS signal is given at the end of this chapter. Figure 9.1 shows the Xcos diagram for PRBS testing. The PRBS amplitude and offset value to the input can be adjusted using the relevant blocks.

The steps to be followed to conduct PRBS test experiment virtually remains same as explained in section 3.5. only for the following differences

- The working folder is prbs/identification/
- Use the `getd` command `getd ../../common_files`
- In section 3.5 read all instances of `step_test` as `prbstest`

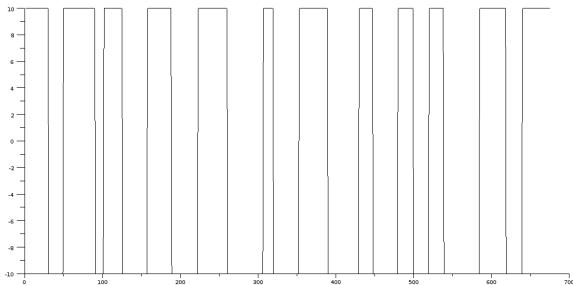


Figure 9.3: A Pseudo Random Binary Sequence

The PRBS response obtained after executing the experiment successfully is shown in figure 9.4

9.3 Determination of Discrete Time Transfer Function

System identification is carried out to identify the transfer function between the input signal to the system and output from the system. Firstly, a transfer function with unknown parameters is assumed. The system is given a known input and its response is obtained and then the values of the unknown parameters is chosen such that the sum of squares of the errors is minimized. Here, the error is the difference between the actual output and the output predicted by the transfer function model assumed. For the given SBHS system, we assume a second order transfer function:

$$G(z) = \frac{b_1 + b_2 z^{-1}}{1 + a_1 z^{-1} + a_2 z^{-2}} z^{-d} \quad (9.1)$$

The unknown parameters a_1, a_2, b_1, b_2 and d are to be obtained through the response of the system to the known inputs. a_1, a_2, b_1, b_2 are real numbers and d is the plant delay which is an integer. For these model parameters estimation, we use a pseudo random binary sequence (PRBS) input. Since the optimization over discrete variables (d in this case) is a very difficult routine for computers, we assume a value for d and then optimize over a_1, a_2, b_1, b_2 . The optimization problem, then, becomes:

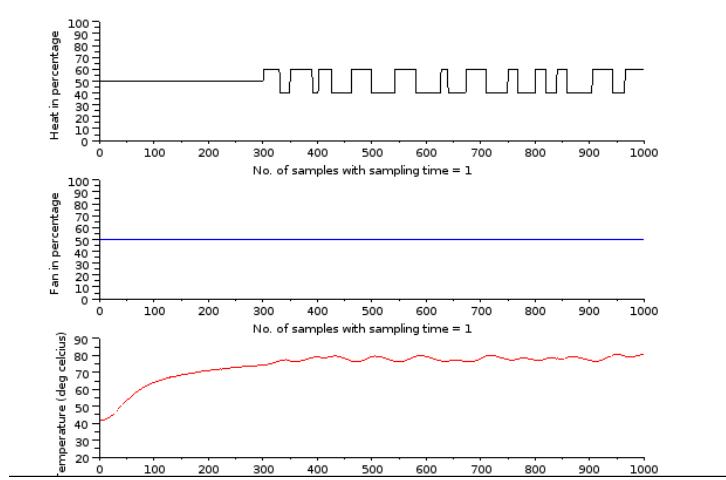


Figure 9.4: PRBS testing response

$$(\hat{b}_1, \hat{b}_2, \hat{a}_1, \hat{a}_2) = \underset{b_1, b_2, a_1, a_2}{\operatorname{argmin}} \sum_{i=0}^N (y(k) - \hat{y}(k))^2 \quad (9.2)$$

Here, $y(k)$ is the output obtained from the system- so it is known. $\hat{y}(k)$ is the estimated output using y the model assumed, which can be written as a difference equation:

$$\hat{y}(k) = -a_1\hat{y}(k-1) - a_2\hat{y}(k-2) + b_1u(k-d) + b_2u(k-1-d) \quad (9.3)$$

The optimization is performed using the optimization routine “optim” of Scilab. Copy the data file to the folder `prbs_analysis`. Change the Scilab working directory to `prbs_analysis` folder. Open the file `optimize.sce` and put the name of the data file (with extention) in the filename field. Save and run this code and obtain the plot as shown in figure 4.3. This code uses the routines `label.sci`, `costfunction.sci` and `second_order.sci`. This code will give optimized values for a_1, a_2, b_1, b_2 which can be used to define a second order discrete time transfer function as given in equation 9.1. The results generated after executing optimization routine over the data file obtained earlier is shown in figure 9.5 and figure 9.6. The initial values were chosen to be $[0.5 \ 0.5 \ 0.5 \ 0.5]$. The value of err along with the fit obtained has to be used to change the initial values and the value of $delay$. The transfer function thus obtained is

Showing Second Order Model and Experimental Results

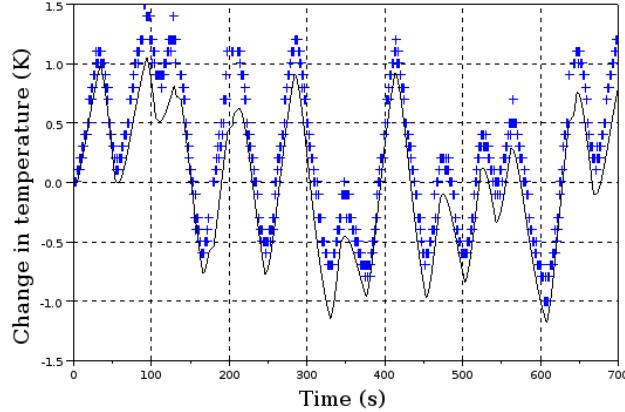


Figure 9.5: PRBS testing response

$$G(z) = \frac{0.0057384 - 0.0057355z^{-1}}{1.9529968z^{-1} + 0.9162750z^{-2}} z^{-4} \quad (9.4)$$

9.4 Performing PRBS testing on SBHS, locally

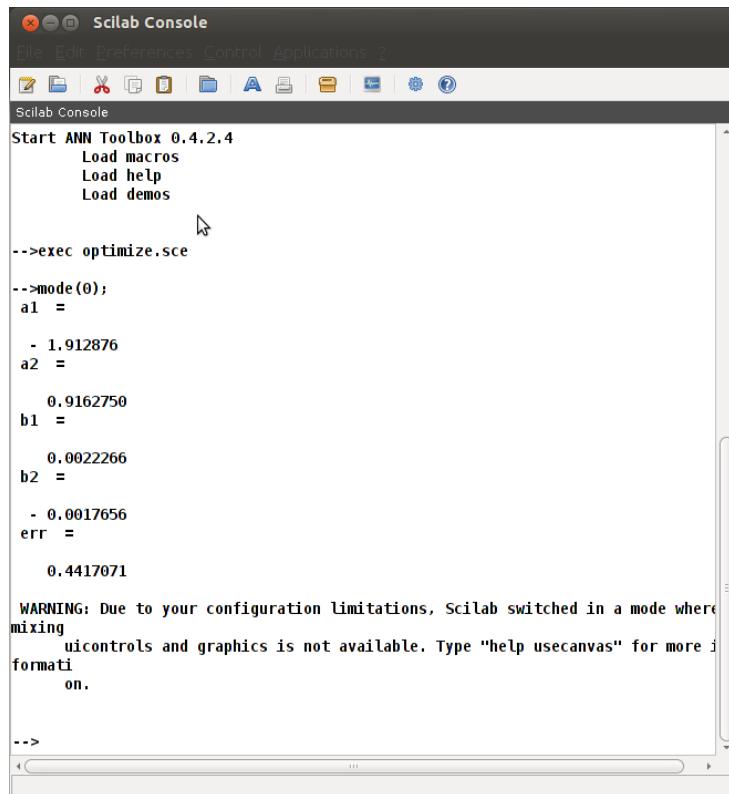
The step by step procedure for conducting an experiment locally remains same as explained in section 2.1.2 with the following changes

- The working folder is prbs/identification/
- Use the getd command `getd/common_files`
- In section 2.1.2 read all instances of `step_test` as `prbstest`

The steps to identify the transfer remains same as explained in the section 9.6

9.5 Implementing 2DOF pole-placement controller using PRBS model, virtually

For deriving the Two degrees of freedom control law, please refer to the chapter 8.2 The controller was designed for the given transient conditions, rise time = 10



The image shows a screenshot of the Scilab Console window. The title bar reads "Scilab Console". The menu bar includes "File", "Edit", "Preferences", "Control", "Applications", and "Help". The toolbar contains icons for file operations like Open, Save, and Print, along with other tools. The main console area displays the following text:

```
Start ANN Toolbox 0.4.2.4
  Load macros
  Load help
  Load demos

-->exec optimize.sce
-->mode(0);
a1 =
 - 1.912876
a2 =
  0.9162750
b1 =
  0.0022266
b2 =
 - 0.0017656
err =
  0.4417071

WARNING: Due to your configuration limitations, Scilab switched in a mode where
mixing
    uicontrols and graphics is not available. Type "help usecanvas" for more i
formati
    on.

-->
```

Figure 9.6: PRBS testing response

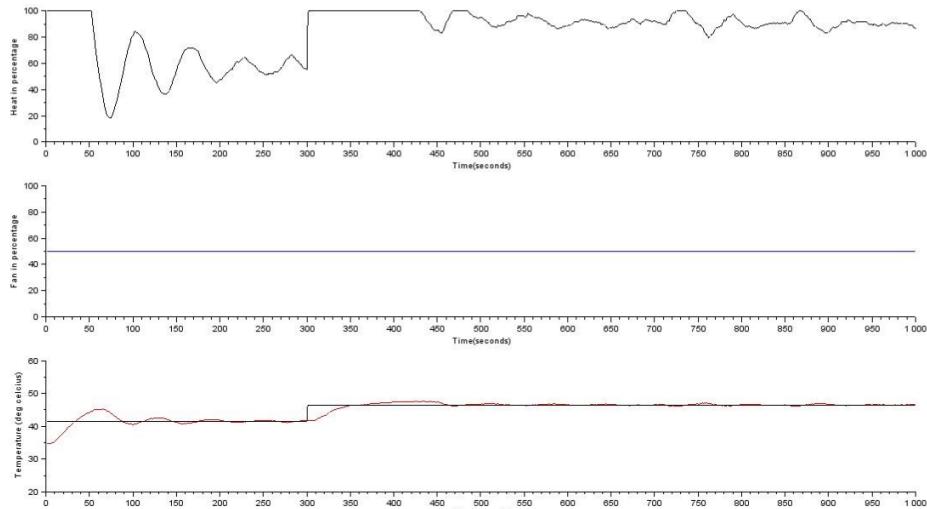


Figure 9.7: 2dof controller response

sec, overshoot = 0.1. The experimental result and performance of the controller for setpoint temperature change from 38.00 to 43.00 degree C, i.e. 5 degree C positive step change, has been shown below in Fig 9.7

The parameters for the 2-DOF pole-placement controller obtained are shown here

$$Tc = 1 - 1.9444137z^{-1} + 0.9447818z^{-2}$$

$$Sc = 0.0337719 - 0.0656666z^{-1} + 0.0319071z^{-2}$$

$$Rc = 10^{-9}(4377900 - 12034140z^{-1} + 11094713z^{-2} - 3436740.5z^{-3} + 3.469D^{-09}z^{-4} - 147850.06z^{-5} + 146117.57z^{-6})$$

$$\gamma = 0.0337719$$

As can be observed from the graph of temperature vs. time (third subplot) in Fig 9.7, the overshoot criteria was satisfied very easily. The rise time criteria is observed to be more than 30 sec. This can be satisfied with experimentation. The parameters are computed by the file `twodof_para.sce`.

The steps to be followed to conduct PRBS test experiment virtually remains same as explained in section 3.5. only for the following differences

- The working folder is prbs/controller/
- Use the getd command `getd ../../common_files`

- In section 3.5 read all instances of step_test as prbstest

9.6 Implementing 2DOF pole-placement controller using PRBS model, locally

The step by step procedure for conducting an experiment locally remains same as explained in section 2.1.2 with the following changes

- The working folder is prbs/controller/
- Use the getd command getd ../../common_files
- In section 3.5 read all instances of step_test as prbstest

9.7 Scilab Local codes

9.7.1 Identification codes

Scilab Code 9.1 ser_init.sce

```

1 mode(0)
2 global filename m
3 // ** Sampling Time ** //
4 sampling_time = 1;
5 // // // // * * * // // // // //
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res=init([port1 port2]);
12 disp(res)
```

Scilab Code 9.2 costfunction.sci

```

1 function [f,g,ind] = costfunction(x,ind)
2     global delay;
```

```

3     y_prediction = second_order(u, x);
4     if size(y) ~= size(y_prediction) then
5         y_prediction = y_prediction';
6     end
7     f = (norm(y-y_prediction,2))^2;
8     g = numdiff(func_1,x);
9  endfunction
10
11 function f = func_1(x)
12     global delay;
13     y_prediction = second_order(u, x);
14     if size(y) ~= size(y_prediction) then
15         y_prediction = y_prediction';
16     end
17     f = (norm(y-y_prediction,2))^2;
18 endfunction

```

Scilab Code 9.3 optimize.sce

```

1 mode(0);
2 // Change filename here
3 filename = "30Apr2014_12_30_50.txt";
4 clf
5
6
7 exec('costfunction.sci');
8 exec('label.sci');
9 exec('second_order.sci');
10
11
12 data = fscanfMat(filename);
13
14 time = data(:, 1);
15
16 heater = int(data(:, 2));
17
18 fan = int(data(:, 3));
19

```

```

20 temp = data(:, 4);
21
22 ss_op_pt = heater(2);
23 for i=2:length(heater)
24     if heater(i) ~= ss_op_pt then
25         startTime = i;
26         break
27     end
28 end
29
30
31 time1 = time - time(1);
32 time2 = time1/1000;
33
34 baseheat = heater(5);
35 heater = heater(startTime:length(heater));
36 heater = heater - baseheat;
37
38 len = length(heater);
39
40 temp = temp(startTime:length(temp));
41 temp = temp - temp(1);
42
43 time = time2(startTime:length(time));
44 time = time - time(1);
45
46 t = time;
47 y = temp;
48 u = heater;
49
50 x0 = [0.2 0.2 0.5 0.5]; // Change initial guess here
51 delay = 5; // Change delay here
52 global delay;
53 [f, xopt] = optim(costfunction, x0);
54
55 a1 = xopt(1)
56 a2 = xopt(2)
57 b1 = xopt(3)

```

```

58 b2 = xopt(4)
59
60 y_pred = second_order(u, xopt);
61
62 if size(y) ~= size(y_pred) then
63     y_pred = y_pred';
64 end
65 err = norm(y - y_pred)/norm(y)
66
67 plot(t, y, "+");
68 plot(t, y_pred, "k");
69 // plot(t, u/10, "r");
70
71 label('Showing Second Order Model and Experimental
    Results',4,'Time (s)', 'Change in temperature (K)'
    ,4);

```

Scilab Code 9.4 prbs.sci

```

1 function u=PRBS(active)
2 if active == 0 then
3 u = 0;
4 else
5 global PRBSu PRBScount;
6 if PRBSu == [] then
7 PRBSu = 1;
8 PRBScount = 30;
9 end
10 if PRBScount == 0 then
11 PRBSu = -1 * PRBSu;
12 PRBScount = int(rand()*40)+10;
13 else
14 PRBScount = PRBScount - 1;
15 end
16 u = PRBSu;
17 global PRBSu PRBScount;
18 end
19 endfunction

```

Scilab Code 9.5 prbstest.sci

```
1 mode(0)
2 function [temp] = prbstest(heat,fan)
3 global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name
4
5
6 temp = comm(heat,fan);
7
8 plotting([heat fan temp],[0 0 20 0],[100 100 40
   1000])
9
10 m=m+1;
11 endfunction
```

Scilab Code 9.6 second_order.sci

```
1 function y = second_order(u, params)
2     // Do not change anything here
3     a1 = params(1);
4     a2 = params(2);
5     b1 = params(3);
6     b2 = params(4);
7     global delay;
8     // End
9
10    // You should write your code below this line
11 N=length(u);
12
13    // Defining y vector
14    // from t=0 up to t=delay -1, y=0, so in scilab y at
      indices i=1 up to i=delay is 0
15 y(1:1:delay) = 0;
16    // First nonzero output is only due to input u at t
      =0, i.e. in scilab input u at i=1
```

```

17     y(delay+1) = b1*u(1);
18     // After that y(i) can be defined as follows
19     for i=delay+2:1:N
20         y(i)=-a1*y(i-1)-a2*y(i-2)+b1*u(i-delay)+b2*u(i-
21             -1-delay);
22     end
23
24     // y =
25 endfunction

```

Scilab Code 9.7 start.sce

```

1 getd ../../common_files/
2
3 exec ../../common_files/loader.sce
4
5 exec ser_init.sce
6 exec prbs.sci
7 exec prbstest.sci
8
9 xcos prbstest.xcos

```

9.7.2 Controller codes

Scilab Code 9.8 start.sce

```

1 mode(0)
2 global fdfh fdt fncre fncrew m err_count y limits
      sampling_time m
3
4 global scn scd tcn tcd rcn rcd gamm
5
6 getd "dc/scilab"
7
8 // *****
9 sampling_time=1; // In seconds. Fractions are allowed

```

```

10 // **** */
11 exec ("prbstest-virtual.sci");
12 exec ("twodof_para.sce");
13 // exec ("sbhs_control.sci");
14
15
16
17 // [scn, scd, tcn, tcd, rcn, rcd, gamm] = sbhs_control()
18
19 ok = init();
20
21 if ok~=[] // open xcos only if communication is
22     through (ie reply has come from server)
23     xcos('prbstest-virtual.xcos');
24 else
25     disp("NO NETWORK CONNECTION!");
26 return
27 end

```

Scilab Code 9.9 prbs_pp.sce

```

1 mode(0)
2 function [temp] = prbs_pp(heat,fan, setpoint)
3 global heatdisp fandisp tempdisp setpointdisp
4           sampling_time m name
5
6 temp = comm(heat,fan);
7
8 plotting([heat fan temp setpoint],[0 0 20 0],[100
9           100 40 1000])
10
11 m=m+1;
12 endfunction

```

Scilab Code 9.10 ser_init.sce

```

1 mode(0)

```

```

2 global filename m
3 // ** Sampling Time ** //
4 sampling_time = 1;
5 // **** * * * * //////
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res=init([port1 port2]);
12 disp(res)

```

Scilab Code 9.11 start.sce

```

1 getd ../../common_files/
2 getd dc/scilab
3 exec ../../common_files/loader.sce
4
5 exec ser_init.sce
6 exec prbs_pp.sci
7
8 exec twodof_para.sce
9
10 xcos prbs_pp.xcos

```

Scilab Code 9.12 twodof_para.sce

```

1 mode(0)
2 global Rc Sc Tc gamm
3 global scn scd tcn tcd rcn rcd gamm
4 s=%s;
5 z=%z;
6
7
8 Ts = sampling_time;
9
10 // Transfer function
11 A = [1 -1.87 0.87];

```

```

12 B= [0.0020 -0.0015];
13 k = 3;
14
15
16 rise = 10;
17 epsilon = 0.1;
18 Nr = rise/Ts;
19
20 // Transient specifications
21 // rise = 10; epsilon = 0.05;
22 phi = desired(Ts,rise,epsilon);
23
24 // Controller design
25 Delta = [1 -1];
26 [Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi); // with integral
27
28 // Setting up simulation parameters for basic.cos
29 st = 0.0001; // desired change in h, in m.
30 t_init = 0; // simulation start time
31 t_final = 0.5; // simulation end time
32
33 // Setting up simulation parameters for c_sscl.cos
34 N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;
35
36 [Tc1,Rc1] = cosfil_ip(Tc,Rc); // Tc / Rc
37 [Sc2,Rc2] = cosfil_ip(Sc,Rc); // Sc / Rc
38
39 [Bp] = cosfil_ip(B,1);
40 [Ap] = cosfil_ip(A,1);
41
42 [Tcp1,Tcp2] = cosfil_ip(Tc,1); // Tc / 1
43 [Np,Rcp] = cosfil_ip(N,Rc); // 1 / Rc
44 [Scp1,Scp2] = cosfil_ip(Sc,1); // Sc / 1
45 [Cp,Dp] = cosfil_ip(C,D); // C / D
46
47 // Rc1 = Rc (1) ; Rc2 = Rc (2) ; Rc3 = Rc (3) ; Rc4 = Rc (4) ;
48 // Sc1 = Sc (1) ; Sc2 = Sc (2) ;
49 // Sc3 = Sc (3) ;

```

```

50 // Tc1 = Tc ( 1 ) ; Tc2 = Tc ( 2 ) ;
51 // Tc3 = Tc ( 3 ) ;
52 Rcp
53 Scp1
54 Tcp1
55 gamm
56
57
58
59 scn = poly(Sc(length(Sc):-1:1), 'z', 'coeff');
60 tcn = poly(Tc(length(Tc):-1:1), 'z', 'coeff');
61 rcn = poly(Rc(length(Rc):-1:1), 'z', 'coeff');
62
63 scd = z^(length(Sc)-1);
64 rcd = z^(length(Rc)-1);
65 tcd = z^(length(Tc)-1);

```

9.8 Scilab Virtual codes

9.8.1 Identification codes

Scilab Code 9.13 costfunction.sci

```

1 function [f,g,ind] = costfunction(x,ind)
2   global delay;
3   y_prediction = second_order(u, x);
4   if size(y) ~= size(y_prediction) then
5     y_prediction = y_prediction';
6   end
7   f = (norm(y-y_prediction,2))^2;
8   g = numdiff(func_1,x);
9 endfunction
10
11 function f = func_1(x)
12   global delay;
13   y_prediction = second_order(u, x);
14   if size(y) ~= size(y_prediction) then

```

```

15         y_prediction = y_prediction';
16     end
17     f = (norm(y-y_prediction,2))^2;
18 endfunction

```

Scilab Code 9.14 optimize.sce

```

1 mode(0);
2 // filename = "prbs.txt"; // Change filename here
3 // filename = "29 Apr 2014_17_03_57.txt";
4 // filename = "30 Apr 2014_12_30_50.txt";
5 filename = "02May2014_16_23_16.txt";
6 clf
7
8 exec('costfunction.sci');
9 exec('label.sci');
10 exec('second_order.sci');
11 // data = fscanfMat(filename);
12 // heater1 = int(data(:, 2));
13 // len = length(heater1);
14 // heater_new = [heater1(1); heater1(1:len-1)];
15 // del_heater = heater1 - heater_new;
16 // ind = find(del_heater >1);
17 // heater = heater1(ind(2):len);
18 //
19 // time = data(ind(2):len, 5);
20 //
21 // fan = int(data(ind(2):len, 3));
22 // temp = data(ind(2):len, 4);
23
24
25 data = fscanfMat(filename);
26
27 time = data(:, 5);
28
29 heater = int(data(:, 2));
30
31 fan = int(data(:, 3));

```

```

32
33 temp = data(:, 4);
34
35 ss_op_pt = heater(2);
36 for i=2:length(heater)
37     if heater(i) ~= ss_op_pt then
38         startTime = i;
39         break
40     end
41 end
42
43
44 time1 = time - time(1);
45 time2 = time1/1000;
46
47 baseheat = heater(5);
48 heater = heater(startTime:length(heater));
49 heater = heater - baseheat;
50
51 len = length(heater);
52
53 temp = temp(startTime:length(temp));
54 temp = temp - temp(1);
55
56 time = time2(startTime:length(time));
57 time = time - time(1);
58
59 t = time;
60 y = temp;
61 u = heater;
62
63 x0 = [0.2 0.2 0.5 0.5]; // Change initial guess here
64 delay = 5; // Change delay here
65 global delay;
66 [f, xopt] = optim(costfunction, x0);
67
68 a1 = xopt(1)
69 a2 = xopt(2)

```

```

70 b1 = xopt(3)
71 b2 = xopt(4)
72
73 y_pred = second_order(u, xopt);
74
75 if size(y) ~= size(y_pred) then
76     y_pred = y_pred';
77 end
78 err = norm(y - y_pred)/norm(y)
79
80 plot(t, y, "+");
81 plot(t, y_pred, "k");
82 // plot(t, u/10, "r");
83
84 label('Showing Second Order Model and Experimental
    Results', 4, 'Time (s)', 'Change in temperature (K)'
    , 4);

```

Scilab Code 9.15 prbs.sci

```

1 function u=PRBS(active)
2 if active == 0 then
3     u = 0;
4 else
5     global PRBSu PRBScount;
6     if PRBSu == [] then
7         PRBSu = 1;
8         PRBScount = 30;
9     end
10    if PRBScount == 0 then
11        PRBSu = -1 * PRBSu;
12        PRBScount = int(rand()*40)+10;
13    else
14        PRBScount = PRBScount - 1;
15    end
16    u = PRBSu;
17    global PRBSu PRBScount;
18 end

```

```
19 endfunction
```

Scilab Code 9.16 prbstest.sci

```
1 function [ stop ] = prbstest(heat , fan )
2
3     [ stop , temp ] = comm(heat , fan ); // N e v e r   e d i t   t h i s
4         l i n e
5     plotting([heat fan temp]);
6
6 endfunction
```

Scilab Code 9.17 prbstest.sce

```
1 mode(0)
2 global fdfh fdt fnrcr fnccw m err_count y limits
3         sampling_time m
4
5 // *****
6 sampling_time=1;      // In seconds . Fractions are allowed
7 // *****
8 exec ("prbstest.sci");
9 exec ("prbs.sci");
10
11 ok = init();
12
13 if ok~= [] // open xcos only if communication is
14     through ( ie reply has come from server )
15     xcos('prbstest.xcos');
16 else
17     disp("NO NETWORK CONNECTION!");
18     return
19
19 end
```

Scilab Code 9.18 second_order.sci

```
1 function y = second_order(u , params)
```

```

2      // Do not change anything here
3      a1 = params(1);
4      a2 = params(2);
5      b1 = params(3);
6      b2 = params(4);
7      global delay;
8      // End
9
10     // You should write your code below this line
11     N=length(u);
12
13     // Defining y vector
14     // from t=0 up to t=delay-1, y=0, so in scilab y at
15     // indices i=1 up to i=delay is 0
16     y(1:1:delay) = 0;
17     // First nonzero output is only due to input u at t
18     // =0, i.e. in scilab input u at i=1
19     y(delay+1) = b1*u(1);
20     // After that y(i) can be defined as follows
21     for i=delay+2:1:N
22         y(i)=-a1*y(i-1)-a2*y(i-2)+b1*u(i-delay)+b2*u(i
23         -1-delay);
24     end
25
26     // y = ?
27
28
29 endfunction

```

9.8.2 Controller codes

Scilab Code 9.19 prbs.sce

```

1 mode(0)
2 global fdfh fdt fncre fncrew m err_count y limits
   sampling_time m
3
4 global scn scd tcn tcd rcn rcd gamm
5

```

```

6 getd "dc/scilab"
7
8 // *****
9 sampling_time=1; // In seconds. Fractions are allowed
10 // *****
11 exec ("prbscontrol-virtual.sci");
12 exec ("twodof_para.sce");
13 // exec ("sbhs_control.sci");
14
15
16
17 // [scn,scd,tcn,tcd,rcn,rcd,gamm] = sbhs_control()
18
19 ok = init();
20
21 if ok~= [] // open xcos only if communication is
   through (ie reply has come from server)
   xcos('prbscontrol-virtual.xcos');
22 else
23 disp("NO NETWORK CONNECTION!");
24 return
25
26 end

```

Scilab Code 9.20 prbscontrol-virtual.sci

```

1 function [stop,temp] = probstest(heat,fan,setp)
2   global scn scd tcn tcd rcn rcd gamm
3
4   [stop,temp] = comm(heat,fan); // Never edit this
   line
5   plotting([heat fan temp setp]);
6
7 endfunction

```

Scilab Code 9.21 twodof_para.sce

```

1 mode(0)
2 global Rc Sc Tc gamm

```

```

3  global scn scd tcn tcd rcn rcd gamm
4  s=%s;
5  z=%z;
6  // // TFcont = syslin('c', -280.14/((s-31.32)*(s+100)*(s
+31.32))) ;
7  // // TFcont = syslin('c', 0.593/((47.21*s+1)*(1.373*s+1)))
// second order
8  // // TFcont = syslin('c', 0.594/(49.19*s+1)) // first
order
9  // SScont = tf2ss(TFcont);
10 // // TFDisc = ss2tf(SScont);
11 // Ts = 1;
12 // [B,A,k] = myc2d(SScont, Ts);
13 //
14 // // polynomials are returned
15 // [Ds, num, den] = ss2tf(SScont);
16 // num = clean(num); den = clean(den);
17
18 Ts = sampling_time;
19
20 // Transfer function for Part - B
21 A = [1 -1.9529968 0.9531269];
22 B= [0.0057384 -0.0057355];
23 k = 3;
24
25 // Transfer function for Part - A
26 // B = [0.0043779 -0.0043266];
27 // A = [1 -1.9444137 0.9447818];
28 // k = 5;
29
30 rise = 10;
31 epsilon = 0.1;
32 Nr = rise/Ts;
33
34 // Transient specifications
35 // rise = 10; epsilon = 0.05;
36 phi = desired(Ts, rise, epsilon);
37

```

```

38 // Controller design
39 Delta = [1 -1];
40 [Rc, Sc, Tc, gamm] = pp_im(B, A, k, phi); // with integral
41
42 // Setting up simulation parameters for basic.cos
43 st = 0.0001; // desired change in h, in m.
44 t_init = 0; // simulation start time
45 t_final = 0.5; // simulation end time
46
47 // Setting up simulation parameters for c_ss_cl.cos
48 N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;
49
50 [Tc1, Rc1] = cosfil_ip(Tc, Rc); // Tc / Rc
51 [Sc2, Rc2] = cosfil_ip(Sc, Rc); // Sc / Rc
52
53 [Bp] = cosfil_ip(B, 1);
54 [Ap] = cosfil_ip(A, 1);
55
56 [Tcp1, Tcp2] = cosfil_ip(Tc, 1); // Tc / 1
57 [Np, Rcp] = cosfil_ip(N, Rc); // 1 / Rc
58 [Scp1, Scp2] = cosfil_ip(Sc, 1); // Sc / 1
59 [Cp, Dp] = cosfil_ip(C, D); // C / D
60
61 // Rc1 = Rc (1) ; Rc2 = Rc (2) ; Rc3 = Rc (3) ; Rc4 = Rc (4) ;
62 // Sc1 = Sc (1) ; Sc2 = Sc (2) ;
63 // Sc3 = Sc (3) ;
64 // Tc1 = Tc (1) ; Tc2 = Tc (2) ;
65 // Tc3 = Tc (3) ;
66 Rcp
67 Scp1
68 Tcp1
69 gamm
70
71
72
73 scn = poly(Sc(length(Sc):-1:1), 'z', 'coeff');
74 tcn = poly(Tc(length(Tc):-1:1), 'z', 'coeff');
75 rcn = poly(Rc(length(Rc):-1:1), 'z', 'coeff');

```

76

77 scd = z^(length(Sc)-1);
78 rcd = z^(length(Rc)-1);
79 tcd = z^(length(Tc)-1);

Chapter 10

Implementing Internal Model Controller for First Order System on a Single Board Heater System

This experiment aims to implement an Internal Model Controller for first order systems on a Single Board Heater System. The target group is anyone possessing basic knowledge of control engineering.

Scilab is used with Xcos as an interface for sending and receiving data. This interface is shown in figure 10.1. Fan speed and heater current are the two inputs to the system. For this experiment, the heater current is the control effort or manipulated variable. The fan input is considered to be the external disturbance.

10.1 IMC Design for Single Board Heater System

Internal Model Controller contains explicit model of plant [5]. The closed loop system can be stabilized with the use of a stable open loop transfer function and a stable controller. The IMC is mainly used for stable plants.

Let the transfer function of the stable plant be denoted by $G_p(z)$ and its model is denoted by $G(z)$. Hence

$$y(n) = G(z)u(n) + \xi(n) \quad (10.1)$$

where:

$y(n)$ = plant output

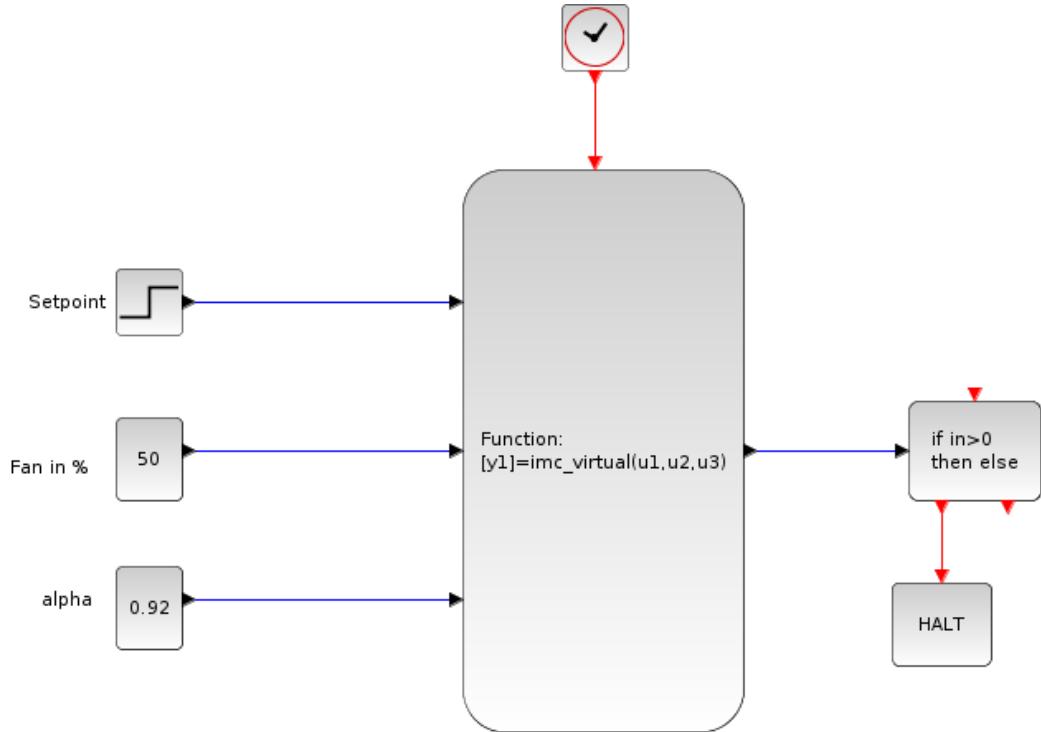


Figure 10.1: Xcos interface for this experiment

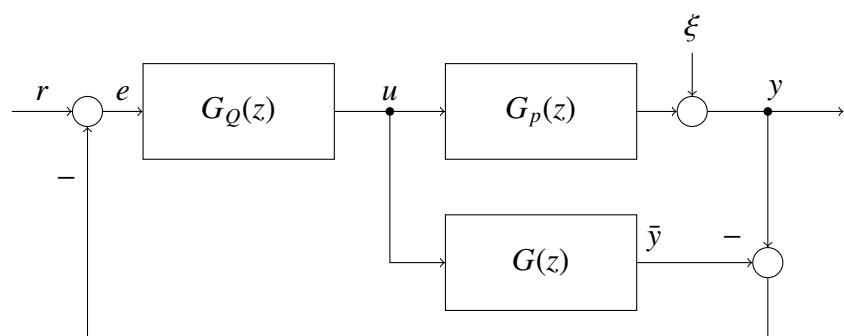


Figure 10.2: IMC feedback configuration

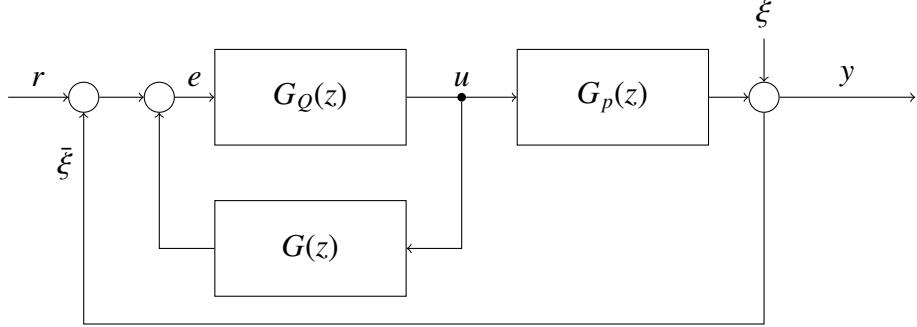


Figure 10.3: Feedback configuration

$u(n)$ = plant input

$\xi(n)$ = noise

For noise rejection with $y=0$, we require $G_Q = G_p^{-1}$ and $G = G_p$, i.e., for stable G_Q we require an approximate inverse of G . Also, for internal stability, transfer function between any two points in the feedback loop must be stable [5].

10.2 Step for Designing IMC for Stable Plant

IMC design refers to obtaining a realizable G_Q that is stable and approximately inverse of G . This can be achieved by inverting the delay free plant model so that G_Q is realizable. For non-minimum phase part of the plant, reciprocal polynomial is used for stable controller. Negative real part of the plant should be replaced with the steady state equivalent of that part to avoid oscillatory nature of control effort. Low pass filter must be used to avoid the high frequency components because of the model mismatch. The SBHS is modeled as-

$$G = Z^{-1} \frac{0.01163}{1 - 0.9723Z^{-1}} \quad (10.2)$$

Inverting delay free plant, we get

$$\frac{A}{B} = \frac{1 - 0.9723Z^{-1}}{0.01163} \quad (10.3)$$

Comparing plant model with equation

$$G = Z^{-1} \frac{B^g B^- B^{nm+}}{A} \quad (10.4)$$

We get,

$$B^g = 0.01163 \quad (10.5)$$

$$B^- = 1 \quad (10.6)$$

$$B^{nm+} = 1 \quad (10.7)$$

$$A = 1 - 0.9723Z^{-1} \quad (10.8)$$

For the stable system, internal model controller is given by

$$G_Q = \frac{A}{B^g B_s^- B_r^{nm+}} G_f \quad (10.9)$$

$$G_Q = \frac{1 - 0.9723Z^{-1}}{0.01163} \frac{1 - \alpha}{1 - \alpha Z^{-1}} \quad (10.10)$$

Now,

$$G_c = \frac{G_Q}{1 - GG_Q} \quad (10.11)$$

$$\frac{u}{e} = \frac{\frac{1 - 0.9723Z^{-1}}{0.01163} \frac{1 - \alpha}{1 - \alpha Z^{-1}}}{1 - Z^{-1} \frac{0.01163}{1 - 0.9723Z^{-1}} \frac{1 - 0.9723Z^{-1}}{0.01163} \frac{1 - \alpha}{1 - \alpha Z^{-1}}} \quad (10.12)$$

After simplifying, we get

$$\frac{u}{e} = \frac{1 - \alpha}{0.01163} \frac{1 - 0.9723Z^{-1}}{1 - Z^{-1}} \quad (10.13)$$

$$\frac{u}{e} = b \frac{1 - 0.9723Z^{-1}}{1 - Z^{-1}} \quad (10.14)$$

where,

$$b = \frac{1 - \alpha}{0.01163} \quad (10.15)$$

Hence,

$$u(n) = u(n - 1) + b[e(n) - 0.9723e(n - 1)] \quad (10.16)$$

For implementing the IMC given above, please follow the steps illustrated in section 2.1.2 of this document with the following changes:

1. In step 1, change the directory to the folder `imc_controller` instead of `StepTest`.
2. In step 5, execute the file `imc.sci` instead of `step_test.sci`.
3. In step 6, execute the file `imc.xcos` instead of `step_test.xcos`.

The output of Xcos is shown in figure 10.4. Figure shows three plots. First sub plot shows setpoint and output temperature profile. Second sub plot shows control effort and third sub plot shows error between setpoint and plant output.

10.3 Experimental Results

By comparing the two graphs, we can say that for $\alpha = 0.92$ the response of the controller is sluggish. For $\alpha = 0.85$, the controller starts responding quickly and no overshoots are seen in the temperature profile.

10.3.1 Implementing IMC on SBHS, virtually

For implementing the IMC virtually, please follow the steps illustrated in section 3.5 of this document with the following changes:

1. In step 1, change the directory to the folder `imc_controller` instead of `StepTest`.
2. In step 5, execute the file `imc_virtual.sce` instead of `step_test.sce`.
3. In step 6, execute the file `imc.xcos` instead of `step_test.xcos`.

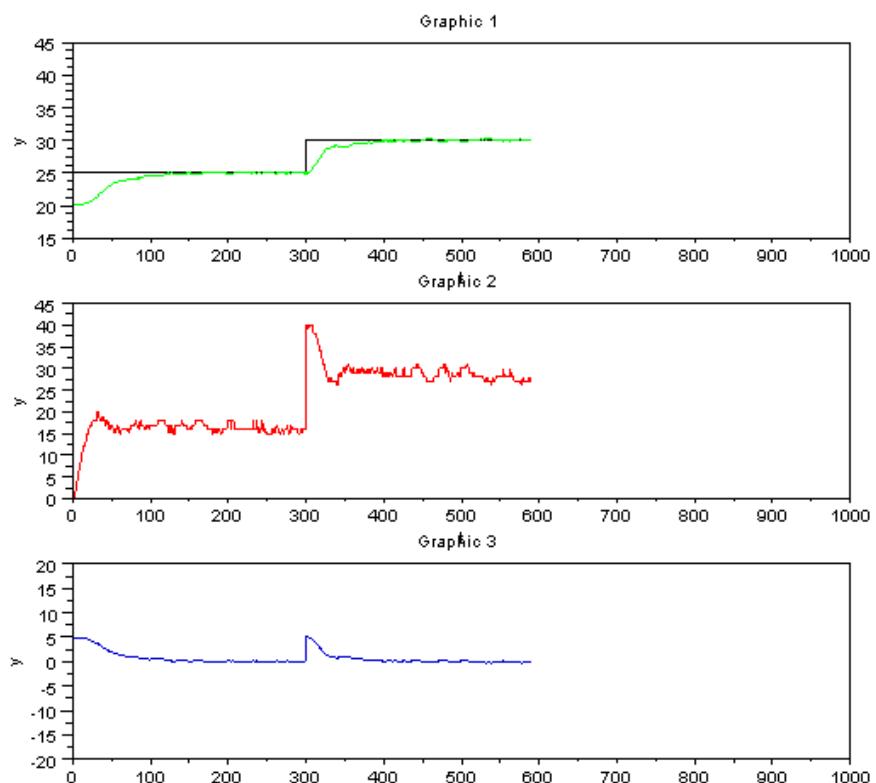


Figure 10.4: Experimental results with IMC for $\alpha = 0.92$

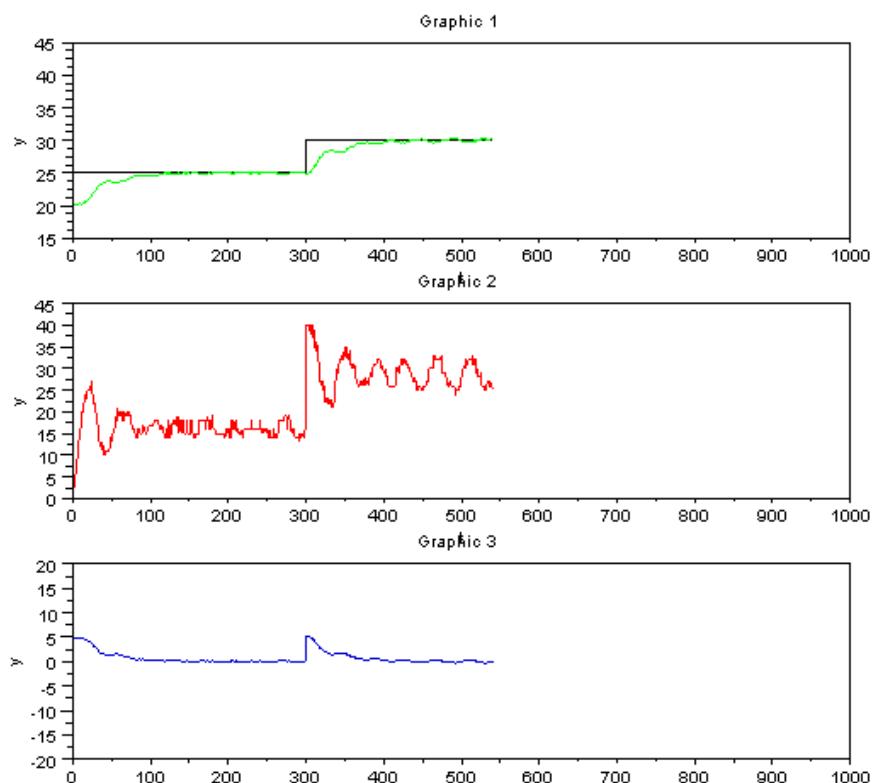


Figure 10.5: Experimental results with IMC for $\alpha = 0.85$

10.4 Scilab Code

Scilab Code 10.1 ser_init.sce

```
1 mode(0)
2 global filename m
3 // ** Sampling Time ** //
4 sampling_time = 1;
5 // // // // * * * // // // // //
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res=init([port1 port2]);
12 disp(res)
```

Scilab Code 10.2 imc.sci

```
1 mode(0)
2 function [temp] = imc(setpoint,fan,alpha)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
   e_old_old
4
5 global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name
6
7 e_new = setpoint - temp;
8 b=((1-alpha)/0.01163);
9
10 u_new = u_old + b*(e_new - (0.9723*e_old));
11
12
13 u_old = u_new;
14 e_old = e_new;
15
16
17 heat = u_new;
```

```

18
19     temp = comm(heat,fan);
20
21     plotting([heat fan temp setpoint],[0 0 20 0],[100
22           100 40 1000])
23
24     m=m+1;
endfunction

```

Scilab Code 10.3 imc_virtual.sce

```

1 mode(0);
2 // For scilab 5.1.1 or lower version users, use scicos
   command to open scicos diagrams instead of xcos
3
4 global fdfh fdt fngr fncl m err_count y limits
   sampling_time m
5
6 // *****
7 sampling_time=1; // In seconds. Fractions are allowed
8 // *****
9
10 exec ("imc_virtual.sci");
11
12 ok = init();
13
14 if ok~= [] // open xcos only if communication is
   through (ie reply has come from server)
15   xcos('imc.xcos');
16 else
17   disp("NO NETWORK CONNECTION!");
18 return
19 end

```

Scilab Code 10.4 imc_virtual.sci

```

1 mode(0)
2

```

```

3
4 function [ stop ] = imc_virtual( setpoint , fan , alpha )
5
6 global temp heat C0 u_old u_new e_old e_new fdfh fdt
    fnrcr fnrcw m err_count stop q heatdisp fandisp
    tempdisp setpointdisp limits m x sampling_time
    e_old_old
7
8 e_new = setpoint - temp;
9
10 b=((1-alpha)/0.01163);
11 u_new = u_old + b*(e_new - (0.9723*e_old));
12
13
14 heat=u_new;
15 u_old = u_new;
16 e_old = e_new;
17
18
19 [ stop ,temp ] = comm(heat,fan); // Never edit this
    line
20 plotting([heat fan temp setpoint],[0 0 30 0],[100
    100 50 1000])
21
22 endfunction

```

Chapter 11

Design and Implementation of Self Tuning PI and PID Controllers on Single Board Heater System

11.1 Introduction

This chapter presents design and implementation of Self Tuning PI and PID controllers on Single Board Heater System done by Mr. Vikas Gayasen.¹

When a plant is wired in a close loop with a PID controller, the parameters- K_c , τ_i and τ_d determine the variation of the manipulated input that is given to the controller. This, in turn, determines the variation of the controlled variable, when a set point is given. Suitable values of these parameters can be found out when plant transfer function is known. However, with large changes in the controlled variable, there may be appreciable changes in the plant transfer function itself. Therefore, it is needed to dynamically update the controller parameters according to the transfer function.

11.1.1 Objective

The objective is to design and implement an algorithm that would dynamically update the values of the controller parameters that are used to control the temperature in the Single Board Heater System (SBHS).

¹Copyright: Mr. Vikas Gayasen, student of Prof. Kannan Moudgalya, IIT Bombay for Process Control course, 2010

11.2 Theory

11.2.1 Why a Self Tuning Controller?

The transfer function of SBHS is assumed as

$$\Delta T = \frac{K_p}{\tau_p s + 1} \Delta H + \frac{K_f}{\tau_f s + 1} \Delta F \quad (11.1)$$

ΔT : Temperature change

ΔF : Fan input change

ΔH : Heater input change

The values of K_p , K_f , τ_s and τ_f can be found by conducting step test experiments. Using these values, the parameters (K_c , τ_i and τ_d) of the PID controller can be defined using methods like Direct Synthesis of Ziegler Nichols Tuning. However, when the apparatus is used in over a large range of temperature, the values of the plant parameters (K_p , K_f , τ_s and τ_f) may change. The new values would give new values of PID controller parameters. However, in a conventional PID controlled system, the parameters K_c , τ_i and τ_d are defined beforehand and are not changed when the system is working. Therefore, we might have a situation in which the PID controller is working with unsuitable values that may not give the desired performance. Therefore, it becomes necessary to change or update the values of the PID parameters so that the plant gives the optimum performance.

11.2.2 The Approach Followed

Variable Description:

- Manipulated Variable: Heater Input
- Disturbance Variable: Fan Input
- Controlled Variable: Temperature

Several open loop step test experiments were performed (giving step changes in the heater input) and the values of K_p and τ_p were found from the results of each experiment by fitting the Inverse Laplace transform of the assumed transfer function with the experimental data. These values were plotted with respect to the corresponding average temperatures. From these plots, correlations were found for both K_p and τ_p as functions of temperature. From correlations of K_p and τ_p , the PID parameters were calculated as functions of temperature. Thus, in the new PID controller, the values of K_c , τ_i and τ_d were calculated using the temperature of the system. For the calculation of PID settings, two approaches: Direct Synthesis and Ziegler-Nichols Tuning were followed.

11.2.3 Direct synthesis

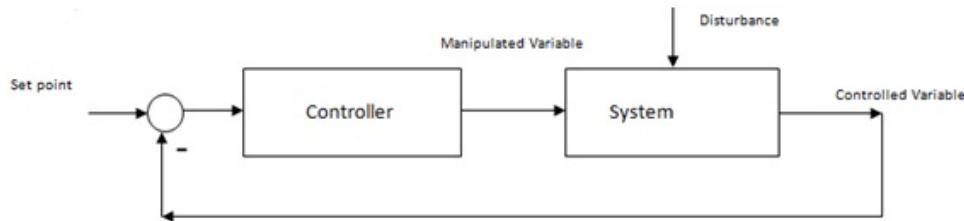


Figure 11.1: Closed loop circuit

We have

$$V(s) = \frac{G_c(s)G(s)}{1 + G_c(s)G(s)} \quad (11.2)$$

where

$V(s)$: Overall closed-loop transfer function

$G_c(s)$: Controller transfer function

$G(s)$: System transfer function.

Therefore,

$$G_c(s) = \frac{1}{G(s)} \frac{V(s)}{1 - V(s)} \quad (11.3)$$

Let the desired closed loop transfer function be of the form

$$V(s) = \frac{1}{(\tau_{cl}s + 1)} \quad (11.4)$$

$$G(s) = \frac{K_p}{(\tau_ps + 1)} \quad (11.5)$$

By using the equations for $G(s)$ and $V(s)$, we get

$$G(c) = K_c \left(1 + \frac{1}{\tau_s}\right) \quad (11.6)$$

where,

$$K_c = \frac{1}{K_p} (\tau_p / \tau_{cl}) \quad (11.7)$$

$$\tau_i = \tau_p \quad (11.8)$$

When K_p and τ_p are known as a function of time, the values of K_c and τ_i can be found as functions of temperature as well.

11.3 Ziegler Nichols Tuning

For Ziegler Nichols Tuning, we use the step response of the open loop experiment.

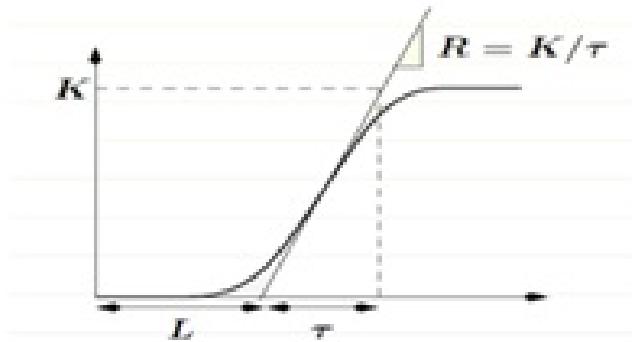


Figure 11.2: Tangent approach to Ziegler Nichols Tuning

	K_c	τ_i	τ_d
P	1/RL		
PI	0.9/RL	3L	
PID	1.2/RL	2L	.5L

Table 11.1: Ziegler Nichols PID Settings

Table 11.1 gives the PID settings. In this approach too, for every open step test, K and τ are found and correlated as function of average temperature and PID settings are then found as functions of temperature.

Note: For a first order transfer function we assume

- $K_p \approx K$
- $\tau_p \approx \tau$

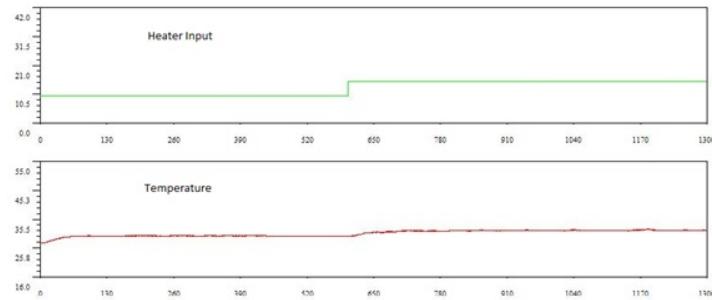


Figure 11.3: Step response for heater reading from 10 to 15

11.4 Step Test Experiments and Parameter Estimation

Several open loop step test experiments were conducted and the values of the open loop parameters were found by curve fitting.

11.4.1 Step Test Experiment

11.4.1.1 Step change in heater reading from 10 to 15

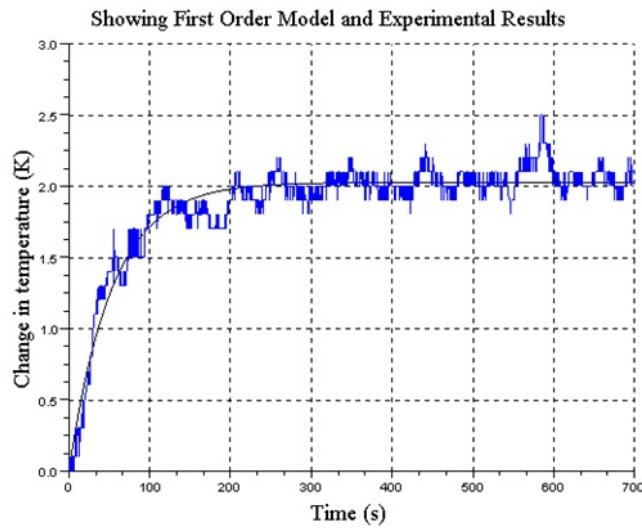


Figure 11.4: Step response for heater reading from 10 to 15 in terms of deviation

11.4.1.2 Step Change in Heater Reading from 20 to 25

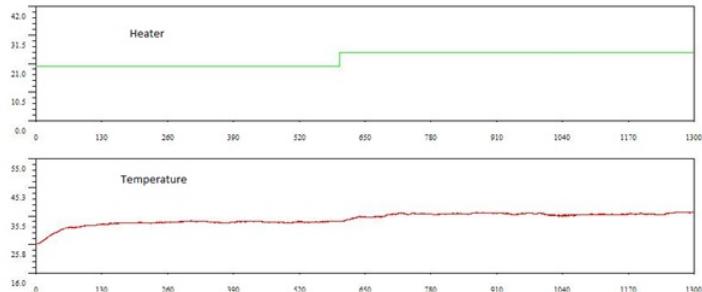


Figure 11.5: Step response for Heater Reading from 20 to 25

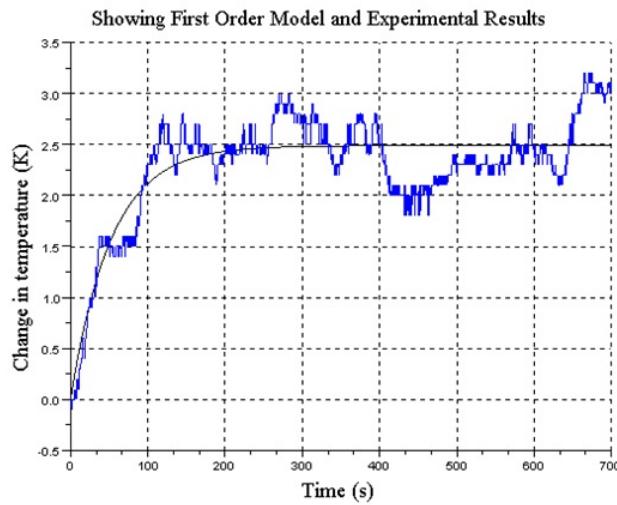


Figure 11.6: Step response for heater reading from 20 to 25 in terms of deviation

11.4.1.3 Step Change in Heater Reading from 30 to 35

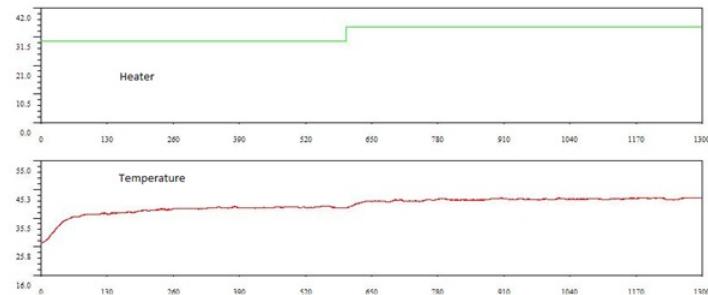


Figure 11.7: Step response for heater reading from 30 to 35

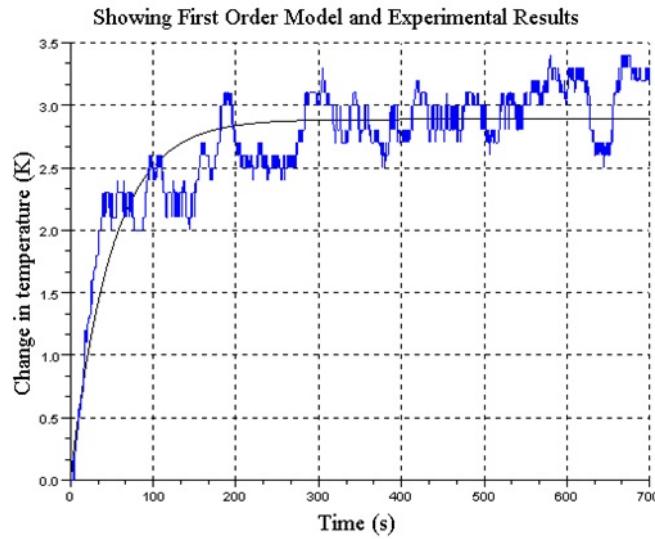


Figure 11.8: Step response for heater reading from 30 to 35 in terms of deviation

11.4.1.4 The Open Loop Parameters

Initial Heater Reading	Final Heater Reading	Average Temperature($^{\circ}\text{C}$)	K_p	τ_p
10	15	31.57	0.41	53.37
20	25	36.00	0.50	52.64
30	35	41.79	0.58	49.21

Table 11.2: Open loop parameters

It can be seen from the graphs that there is a lag of approximately 6 seconds in each experiment.

11.4.2 Conventional Controller Design

1. PI Controller using Ziegler Nichols Tuning with the results of the first step test experiment:
 - $K_c = 19.75$
 - $\tau_i = 18$

2. PID Controller using Ziegler Nichols Tuning with the results of the first step test experiment:
 - $K_c = 26.327$
 - $\tau_i = 12$
 - $\tau_d = 3$

3. PI Controller Using Direct Synthesis with the results of the second step test experiment (τ_{cl} is taken as $\tau_p/2$):
 - $K_c = 4.02$
 - $\tau_i = 52.645$

11.4.3 Self Tuning Controller Design

The graphs showing the variation of K_p and τ_p are shown below:

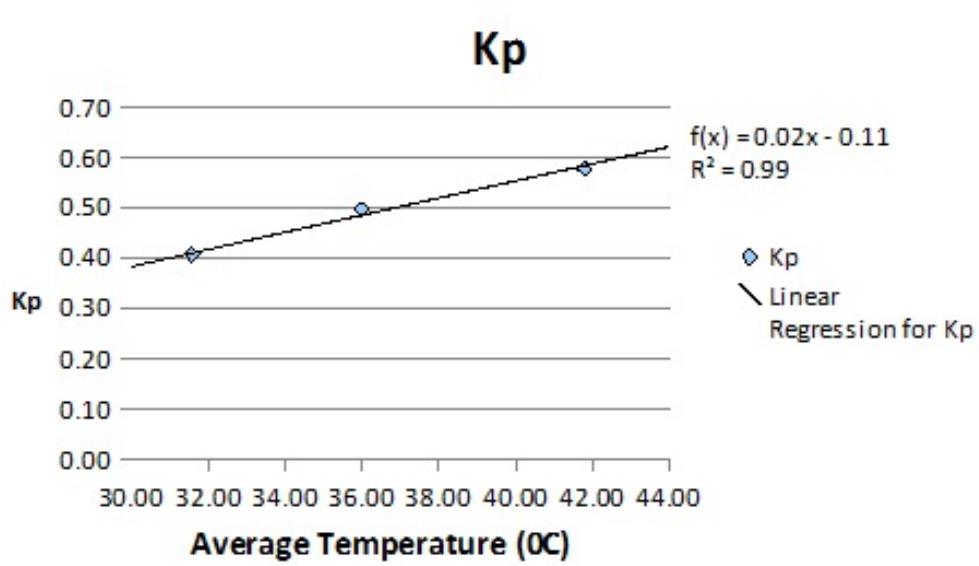


Figure 11.9: Variation of K_p with temperature

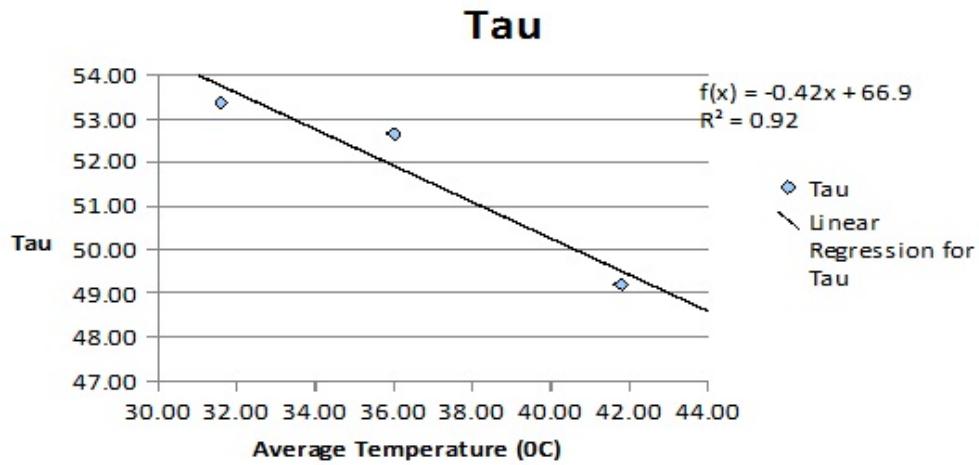


Figure 11.10: Variation of τ_p with temperature

1. PI Controller using Ziegler Nichols Tuning:

$$L = 6$$

$$R = (0.016 \times T - 0.114) / (66.90 - 0.415 \times T) \text{ where } T \text{ is the temperature}$$

$$K_c = 0.9(66.90 - 0.415T) / 6(0.016T - 0.114)$$

$$= (60.21 - 0.3735T) / (0.096T - 0.684)$$

$$\tau_i = 3 \times 6 = 18$$

2. PID Controller using Ziegler Nichols Tuning:

$$L = 6$$

$$R = (0.016 \times T - 0.114) / (66.90 - 0.415 \times T) \text{ where } T \text{ is the temperature}$$

$$K = 1.2(66.90 - 0.415T) / 6(0.016T - 0.114)$$

$$= (80.28 - 0.498T) / (0.096T - 0.684)$$

$$\tau_i = 2 \times 6 = 12$$

$$\tau_d = 0.5 \times 6 = 3$$

3. PI Controller using Direct Synthesis (τ_{cl} is taken as $\tau_p/2$):

$$K = 2/(0.016 \times T - 0.114)$$

$$\tau_i = (66.90 - 0.415 \times T) \text{ where } T \text{ is the temperature}$$

11.5 Implementation

11.5.1 PI Controller

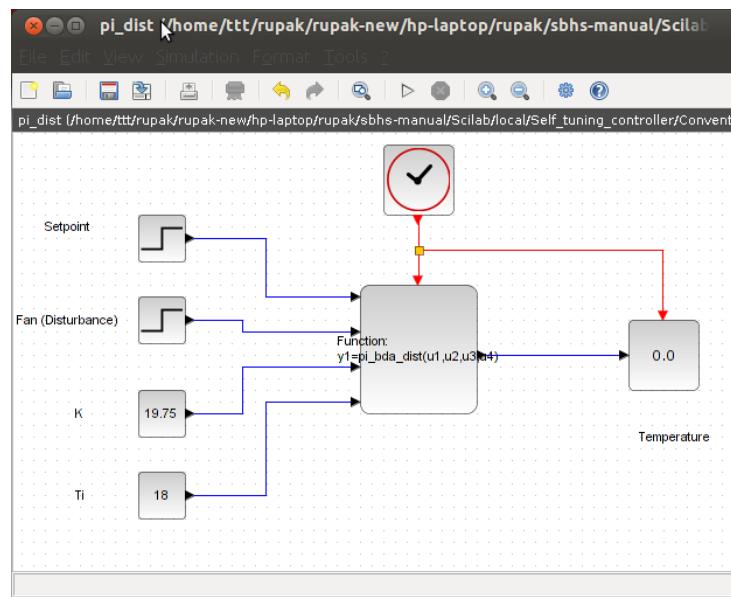


Figure 11.11: Xcos diagram for PI controller

The PI Controller in continuous time is given by

$$u(t) = K \left[e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt \right] \quad (11.9)$$

On taking Laplace transform, we obtain

$$u(s) = K \left[1 + \frac{1}{\tau_i s} \right] e(s) \quad (11.10)$$

By mapping equation 11.10 to discrete time interval using Backward Difference Approximation, we get

$$u(n) = K \left[1 + \frac{T_s}{\tau_i} \frac{z}{z-1} \right] e(n) \quad (11.11)$$

On cross multiplication, we obtain

$$(z-1) \times u(n) = K \left[(z-1) + \frac{T_s}{\tau_i} (z) \right] e(n) \quad (11.12)$$

We divide by z, and using the shifting theorem, we obtain

$$u(n) - u(n-1) = K \left[e(n) - e(n-1) + \frac{T_s}{\tau_i} e(n) \right] \quad (11.13)$$

The PI Controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) \quad (11.14)$$

where,

$$\begin{aligned} s_0 &= K \left(1 + \frac{T_s}{\tau_i} \right) \\ s_1 &= -K \end{aligned}$$

11.5.2 PID Controller

The PID Controller in continuous time is given by

$$u(t) = K \left[e(t) + \frac{1}{\tau_i} \int_0^t e(t) dt + \tau_d \frac{de(t)}{dt} \right] \quad (11.15)$$

On taking Laplace Transform, we obtain

$$u(s) = K \left[1 + \frac{1}{\tau_i s} + \tau_d s \right] e(s) \quad (11.16)$$

By mapping equation 11.16 to discrete time interval by using the Trapezoidal Approximation for integral mode and Backward Difference Approximation for derivative mode, we get

$$u(n) = K \left[1 + \frac{T_s}{\tau_i} \frac{z}{z-1} + \frac{\tau_d}{T_s} \frac{z-1}{z} \right] e(n) \quad (11.17)$$

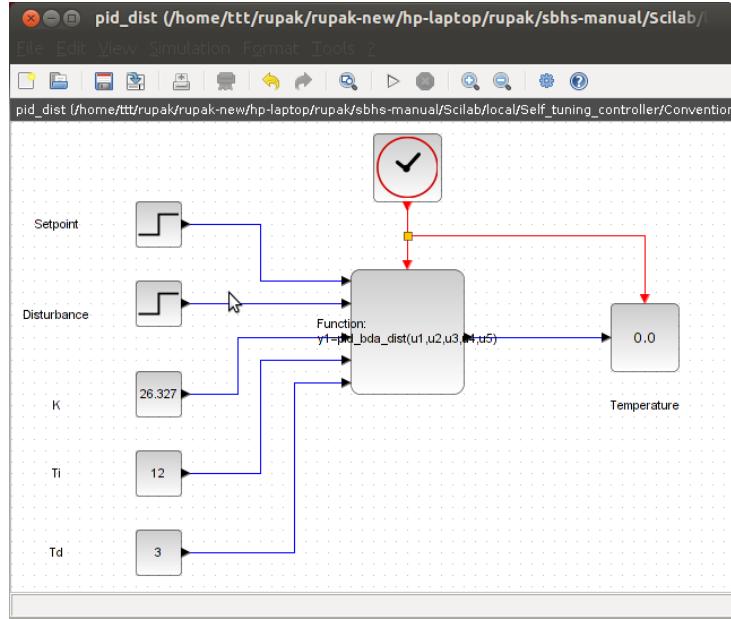


Figure 11.12: Xcos diagram for PID controller

On cross multiplication, we obtain

$$(z^2 - z) \times u(n) = K \left[(z^2 - z) + \frac{T_s}{\tau_i} (z^2) + \frac{\tau_d}{T_s} (z - 1)^2 \right] e(n) \quad (11.18)$$

We divide by z, and using the shifting theorem, we obtain

$$u(n) - u(n - 1) = K \left[e(n) - e(n - 1) + \frac{T_s}{\tau_i} e(n) + \frac{\tau_d}{T_s} \{e(n) - 2e(n - 1) + e(n - 2)\} \right] \quad (11.19)$$

The PID Controller is usually written as

$$u(n) = u(n - 1) + s_0 e(n) + s_1 e(n - 1) + s_2 e(n - 2) \quad (11.20)$$

where,

$$s_0 = K \left(1 + \frac{T_s}{\tau_i} + \frac{\tau_d}{T_s} \right) \quad (11.21)$$

$$s_1 = K \left[-1 - 2 \frac{\tau_d}{T_s} \right] \quad (11.22)$$

$$s_2 = K \left[\frac{\tau_d}{T_s} \right] \quad (11.23)$$

11.5.3 Self Tuning Controller

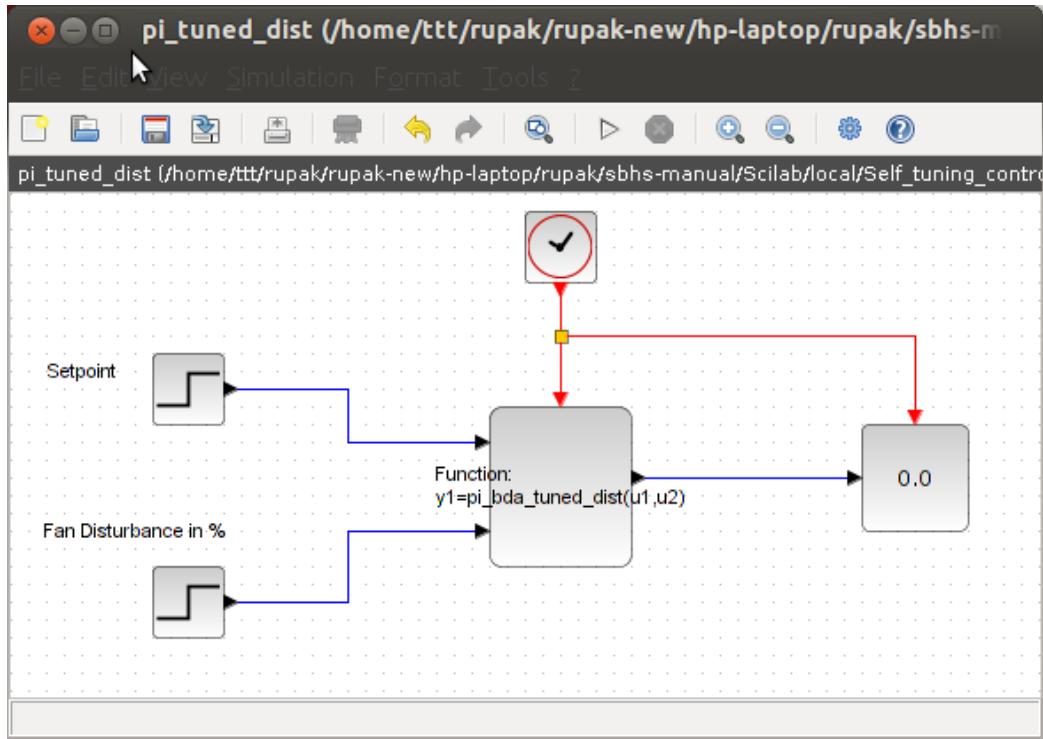


Figure 11.13: Xcos diagram for self tuning controller

The parameters of the controller are determined dynamically using the temperature values for every sampling time. For this, the formulae derived in section 11.4.3 are used. The formulae for the control effort are same as the conventional PI and PID controllers. So the PI/PID settings are calculated for every sampling time and the control effort is calculated thereafter using the formulae derived for conventional controllers.

11.6 Set Point Tracking

The main aim of the controller is to track the set point and to reject disturbances. When the set point of the controlled variable (temperature in this case) is changed, the controller should work in such a manner that the actual temperature follows

the set point as close as possible.

In this project, several experiments were conducted with the self tuning and conventional PI/PID Controllers. Table 11.3 shows the set point changes given during the various experiments that were conducted with conventional and self tuning controllers designed using several methods.

	Conventional Controller	Self Tuning Controller
Direct Synthesis PI	32^0C to 37^0C 35^0C to 45^0C	32^0C to 37^0C 35^0C to 45^0C
Ziegler Nichols PI	32^0C to 37^0C 35^0C to 45^0C 40^0C to 45^0C	32^0C to 37^0C 35^0C to 45^0C 35^0C to 45^0C
Ziegler Nichols PID	31^0C to 45^0C 32^0C to 37^0C	32^0C to 46^0C 32^0C to 37^0C

Table 11.3: Set point changes in experiments conducted for set point tracking

11.6.1 PI Controller Designed by Direct Synthesis

The results of the experiments carried out for the self tuning PI controller using direct synthesis method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the green line) in the SBHS. The lower plot shows the control effort.

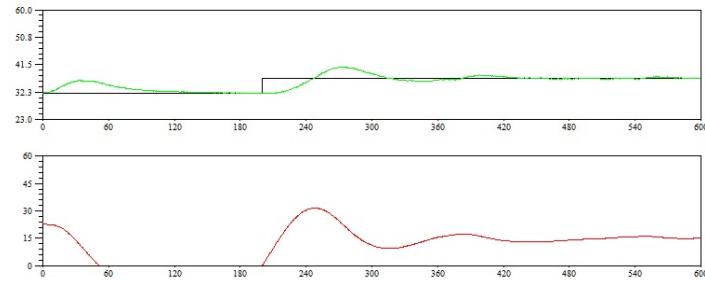


Figure 11.14: Result for self tuning controller designed using Direct Synthesis for set point going from 32°C to 37°C

Although there is a small overshoot, the controller is able to make the actual temperature follow the set point temperature quite closely. Looking at higher values of set point changes, the result for set point change going from 35°C to 45°C is shown.

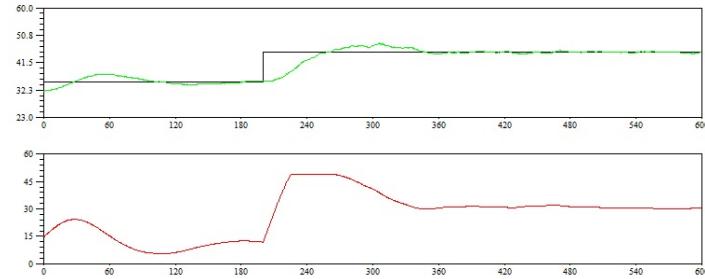


Figure 11.15: Result for self tuning controller designed using Direct Synthesis for set point going from 35°C to 45°C

For a higher set point change also, the controller is able to make the temperature follow the set point closely. Notice the abrupt change in the control effort as soon as the step change in the set point is encountered.

For comparison, results of experiments done with conventional PI controller designed using the Direct Synthesis method are also shown.

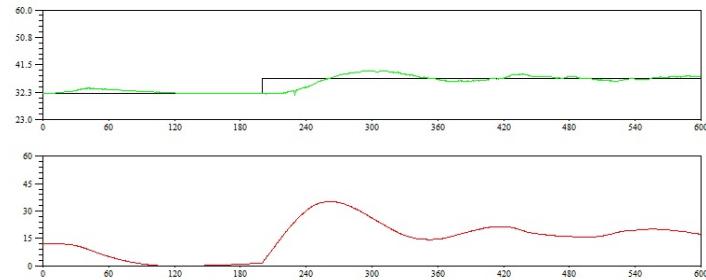


Figure 11.16: Result for conventional controller designed using Direct Synthesis for set point going from 32°C to 37°C

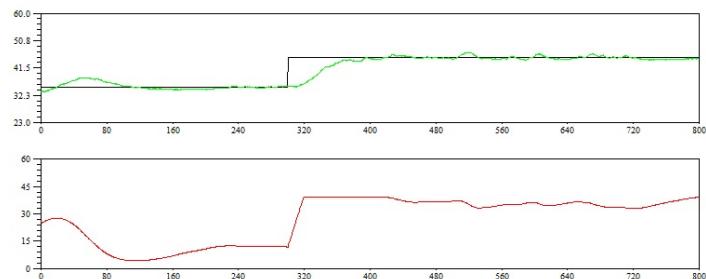


Figure 11.17: Result for conventional controller designed using Direct Synthesis for Set Point going from 35°C to 45°C

As it can been seen from the graph, the self tuning controller stabilised the temperature faster.

11.6.2 PI Controller using Ziegler Nichols Tuning

The results of the experiments carried out for the self tuning PI controller using Ziegler Nichols tuning method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the green line) in the SBHS. The lower plot shows the control effort.

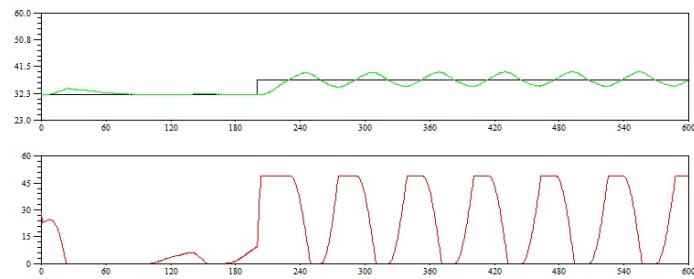


Figure 11.18: Result for self tuning controller designed using Ziegler Nichols Tuning for set point going from 32°C to 37°C

Although there are oscillations, the temperature remains near the set point. The result for a higher value of set point change is also shown.

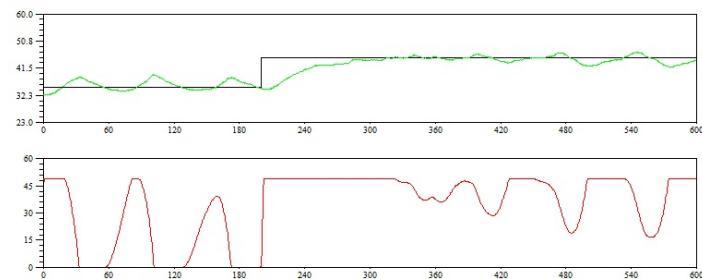


Figure 11.19: Result for self tuning controller designed using Ziegler Nichols Tuning for set point going from 35°C to 45°C

For this experiment, the controller is able to make the temperature follow the set point closely. The fluctuations may be due to noises and the surrounding conditions. The plot for result of an experiment with another value of set point change is also shown.

In this experiment too, the controller is able to keep the temperature close to the set point and it stabilises fast.

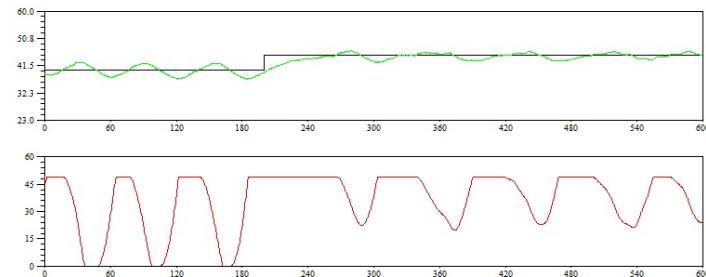


Figure 11.20: Result for self tuning controller designed using Ziegler Nichols Tuning for set point going from 40°C to 45°C

For comparison, results of experiments done with conventional PI controller designed using the Ziegler Nichols method are also shown.

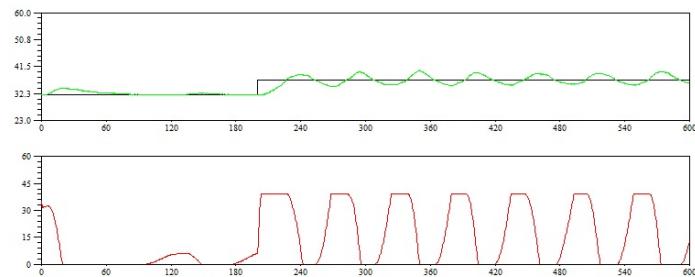


Figure 11.21: Result for conventional controller designed using Ziegler Nichols Tuning for set point going from 32°C to 37°C

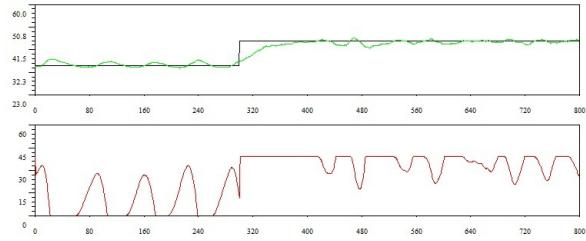


Figure 11.22: Result for conventional controller designed using Ziegler Nichols Tuning for set point going from 35°C to 45°C

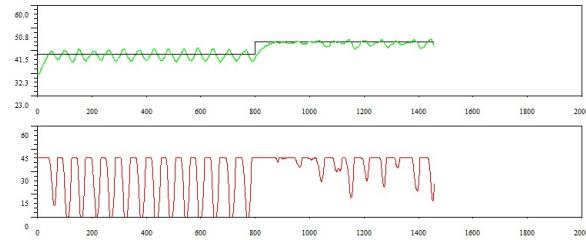


Figure 11.23: Result for conventional controller designed using Ziegler Nichols Tuning for set point going from 40°C to 45°C

For set point change from 40°C to 45°C , the self tuning controller showed small oscillations, but the conventional controller shows bigger oscillations.

11.6.3 PID Controller using Ziegler Nichols Tuning

The results of the experiments carried out for the self tuning PID controller using Ziegler Nichols tuning method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the purple line) in the SBHS. The lower plot shows the control effort.

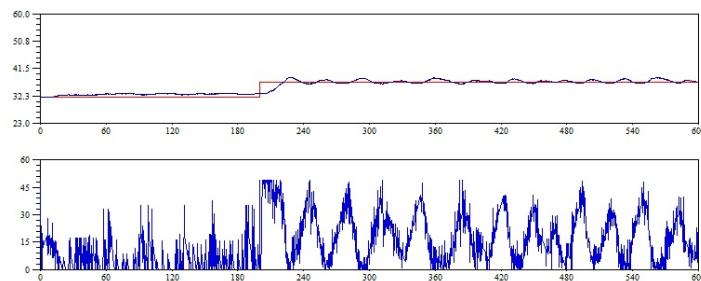


Figure 11.24: Result for self tuning PID controller designed using Ziegler Nichols Tuning for set point going from 32°C to 37°C

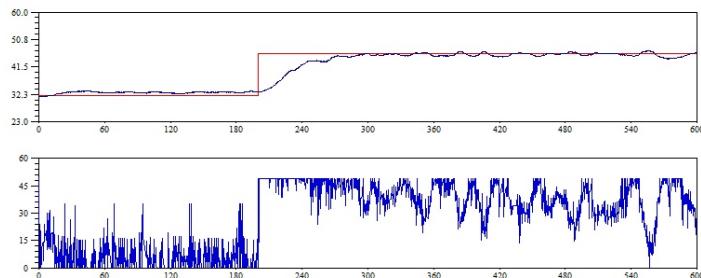


Figure 11.25: Result for self tuning PID controller designed using Ziegler Nichols Tuning for set point going from 32°C to 46°C

From the graph it can be seen that for both the experiments, the self tuning PID controller is able to keep the temperature close to the set point and the stabilisation is also fast. For comparison, plots for experiments conducted with conventional PID controller designed using Ziegler Nichols method are also shown.

From the graph, we can see that the conventional PID controller is not able to make the temperature close to the set point when the set point value is 45°C . The self tuning PID controller had successfully brought the temperature to 45°C .

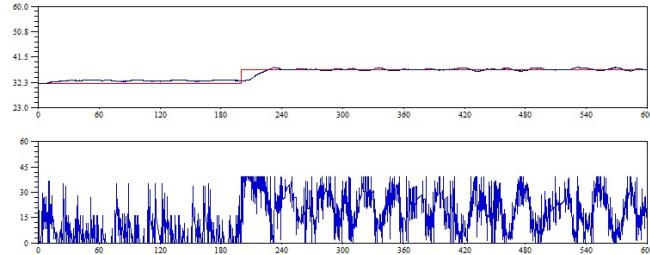


Figure 11.26: Result for conventional PID controller designed using Ziegler Nichols Tuning for set point going from 32°C to 37°C

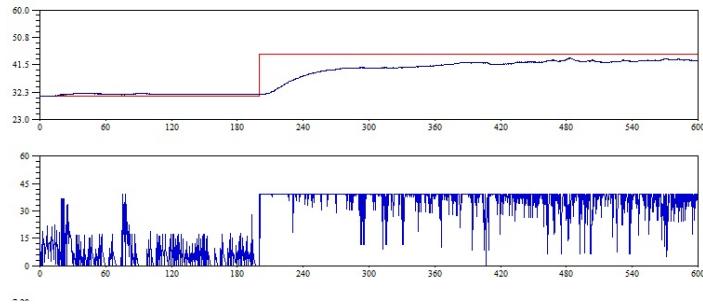


Figure 11.27: Result for conventional PID controller designed using Ziegler Nichols Tuning for set point going from 31°C to 45°C

11.6.4 Conclusion

The self tuning PI controller is able to accomplish the aim of keeping the temperature as close as possible to the set point. Although it may show some initial overshoot or oscillation for some values of set point change, the time needed for stabilisation is low. There may also some cases where the conventional controller shows bigger oscillations than the self tuning controller.

The PI controllers, both conventional and self tuning, show oscillations for some values of set point change. To eliminate the oscillations, when we use the PID Controller, the self tuning design definately seems to be a better option because for higher values of set point change, the self tuning PID controller shows a better performance than the conventional controller as seen in section 11.6.3.

11.7 Disturbance Rejection

Apart from tracking the set point, the system should also be able to reject disturbances. There may be several factors influencing the controlled variable and not all of them can be manipulated. Therefore, it becomes necessary for the controller to not let the changes in the non-manipulated variables to affect the controlled variable. This is called Disturbance Rejection.

In this system, the disturbance variable is the fan input. Therefore, the controller has to work in such a way that changes in the fan input does not affect the temperature in the SBHS.

In this project, several experiments were conducted with the self tuning and conventional PI/PID Controllers. Table 11.4 shows the fan input changes given during the various experiments that were conducted with conventional and self tuning controllers designed using several methods.

	Conventional Controller	Self Tuning Controller
Direct Synthesis PI	50 to 100 100 to 50	50 to 100 100 to 50
Ziegler Nichols PI	50 to 100 100 to 50	50 to 100 100 to 50
Ziegler Nichols PID	50 to 100 100 to 50	50 to 100 100 to 50

Table 11.4: Fan input changes in experiments conducted for Disturbance Rejection

11.7.1 PI Controller Designed by Direct Synthesis

The results of the experiments carried out for the self tuning PI controller using Direct Synthesis method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the green line) in the SBHS. The second plot shows the control effort and the third shows the fan input.

The change in the fan input introduces a small dent in the temperature. However, the controller brings the temperature back to the set point. Notice the slight change in the controller behaviour on encountering the fan input change. The time taken for stabilising back is also low.

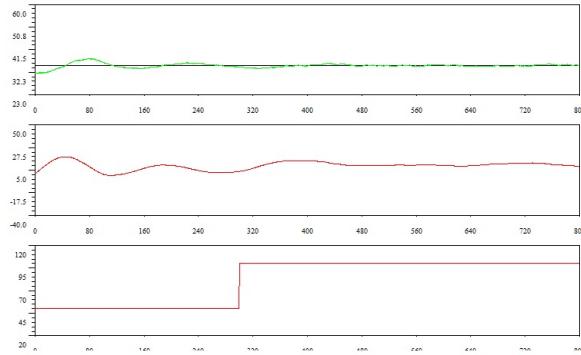


Figure 11.28: Results for fan input change from 50 to 100 for self tuning PI Controller designed using Direct Synthesis

Here, results for fan input change from 100 to 50 are also shown.

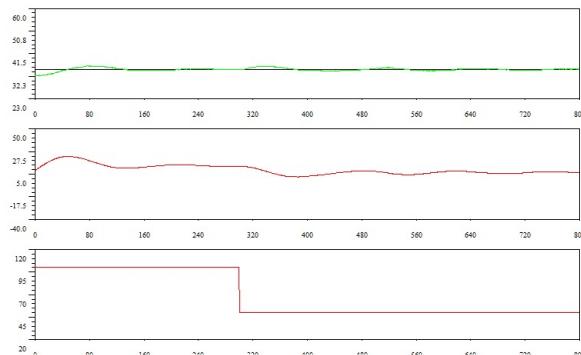


Figure 11.29: Results for fan input change from 100 to 50 for self tuning PI controller designed using Direct Synthesis

In this figure also, the temperature clearly increases a bit when the step change in the fan input is encountered. However, it quickly stabilises back and continues to be close to the set point.

From the above two results, it is clear that the self tuning controller designed with Direct Synthesis has successfully rejected the disturbance.

For comparison, results of the disturbance change for conventional PI Controller designed with direct synthesis are also shown.

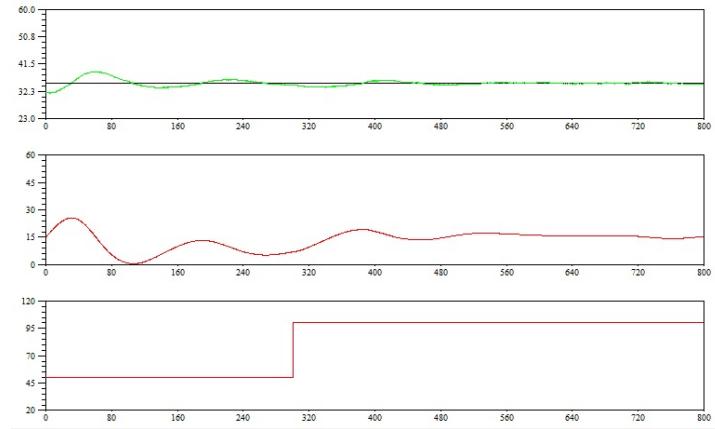


Figure 11.30: Results for the fan input change from 50 to 100 to conventional PI controller designed using Direct Synthesis

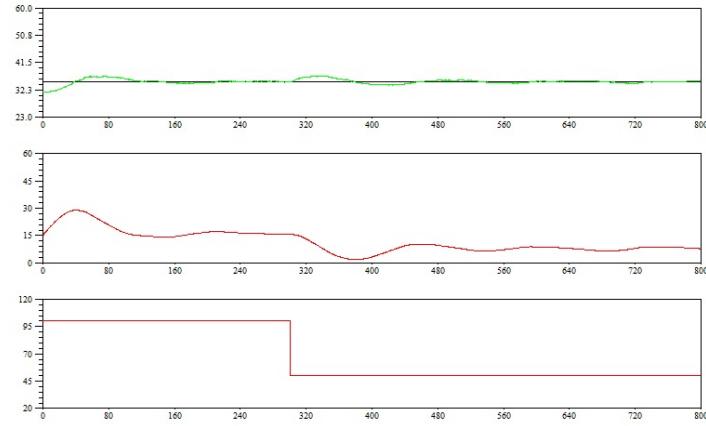


Figure 11.31: Results for the fan input change from 100 to 50 to conventional PI controller designed using Direct Synthesis

11.7.2 PI Controller using Ziegler Nichols Tuning

The results of the experiments carried out for the self tuning PI controller using Ziegler Nichols method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the green line) in the SBHS. The second plot shows the control effort and the third shows the fan input.

Even on encountering the fan input change, the temperature remains close to the set point. Notice the change in the controller behaviour on encountering the fan input change.

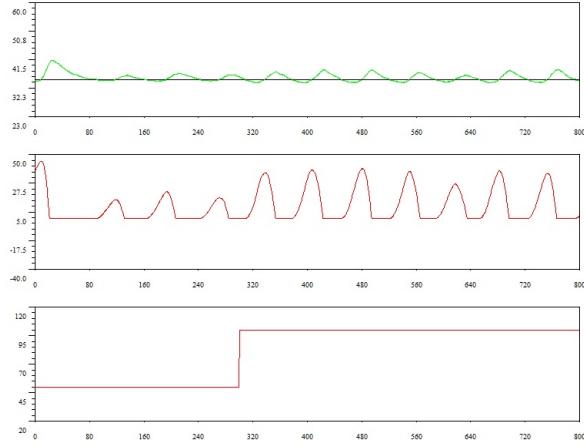


Figure 11.32: Results for Fan Input change from 50 to 100 given to Self Tuning PI Controller designed using Ziegler Nichols method

Here, result for the fan input going from 100 to 50 is also shown.

Here, a change in the control effort can be noticed. This change has been brought by the PI Controller to keep the temperature close to the set point. From the above two results, it is clear that the self tuning controller designed with direct synthesis has successfully rejected the disturbance.

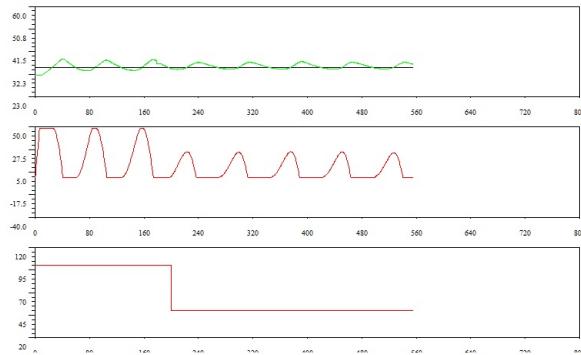


Figure 11.33: Results for fan input change from 100 to 50 given to self tuning PI controller designed using Ziegler Nichols method

For comparison, corresponding results are also shown for conventional PI controllers designed using Ziegler Nichols tuning.

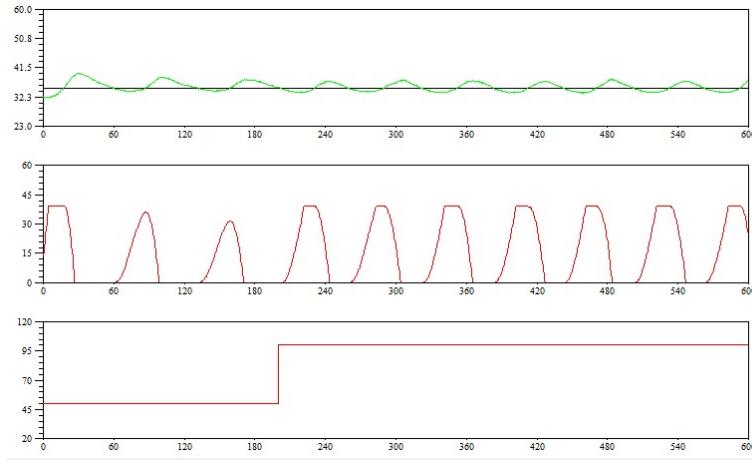


Figure 11.34: Results for the fan input change from 50 to 100 to conventional PI controller designed using Ziegler Nichols tuning

11.7.3 PID Controller using Ziegler Nichols Tuning

The results of the experiments carried out for the self tuning PID controller using Ziegler Nichols method are shown. The upper plot shows the variations of the set point temperature (the black line) and the actual temperature (the purple line) in the SBHS. The second plot shows the control effort and the third shows the fan input.

In this system also, on encountering the fan input change, the temperature remains close to the set point. Notice the change in the control effort profile when the change in the fan input is given.

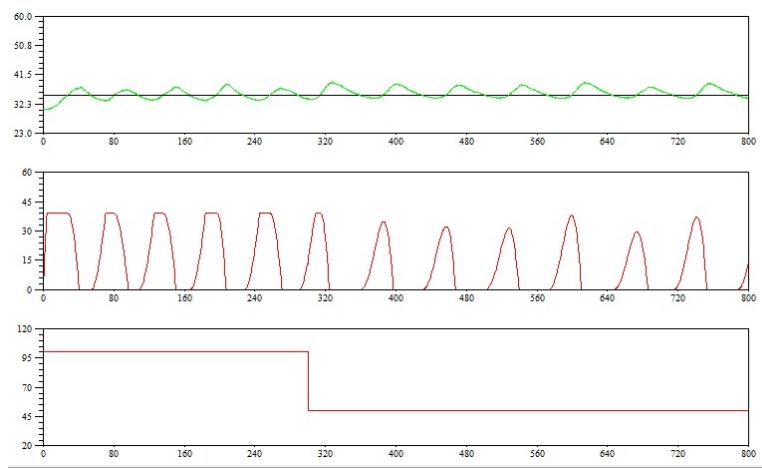


Figure 11.35: Results for the fan input change from 100 to 50 to conventional PI controller designed using Ziegler Nichols tuning

Here, result for the fan input going from 100 to 50 is also shown.

In this figure also, the temperature clearly increases a bit when the step change in the fan input is encountered. However, it quickly stabilizes back and continues to be close to the set point.

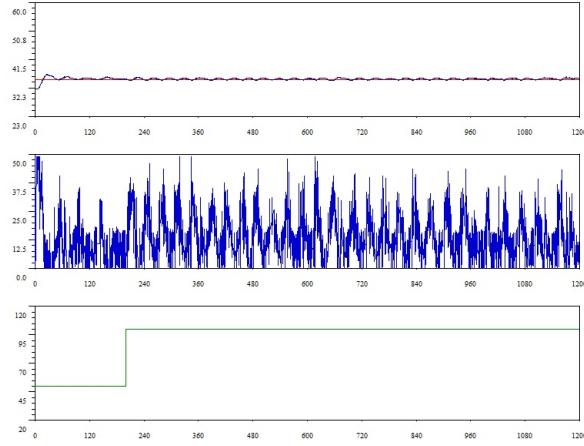


Figure 11.36: Results for Fan Input change from 50 to 100 given to Self tuning PID Controller designed using Ziegler Nichols method

For comparison, corresponding results are also shown for conventional PID controllers designed using Ziegler Nichols tuning.

11.7.4 Conclusion

We see that the self tuning controller manages to keep the temperature close to the set point temperature, even when the change in fan input is encountered. This shows that it can reject the disturbances quite nicely.

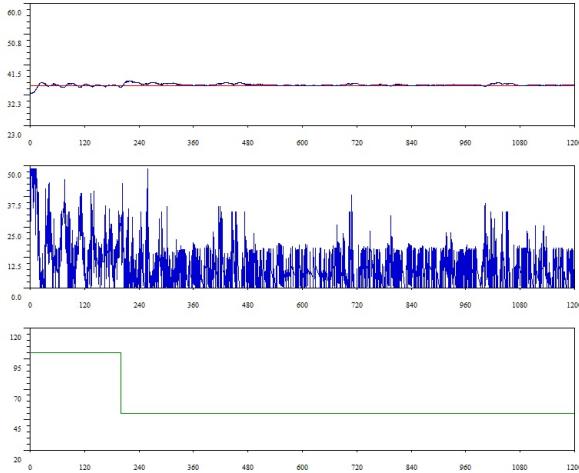


Figure 11.37: Results for fan input change from 100 to 50 given to self tuning PID controller designed using Ziegler Nichols method

11.8 Implementing Self Tuning controller on SBHS, virtually

Under the `virtual` folder locate `Self_tuning_controller` folder. You will see two folders `ConventionalTuning_Vikas` and `SelfTuning_Vikas`. These two directories will contain four more folders categorized as The necessary code is listed in the section 11.9.3 and section 11.9.5 The step by step procedure for conducting an experiment virtually, remains same as explained in section 3.5 with the following changes.

1. Change the scilab working directory to the experiment folder you want to use. For example **`virtual/Self_tuning_controller/SelfTuning_Vikas /PIControllerFanDisturbance`**
2. Use the `getd` command as `../../common_files/`
3. Execute the required .sce file.
For example, `pi_bda_tuned_dist_virtual.sce` if you want to run the PI Controller Fan disturbance experiment.

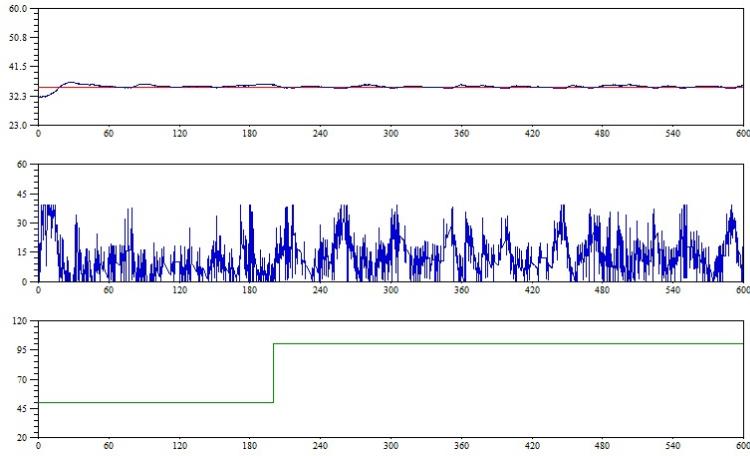


Figure 11.38: Results for the fan input change from 50 to 100 to conventional PID controller designed using Ziegler Nichols tuning

11.9 Scilab Codes

Scilab Code 11.1 ser_init.sce

```

1 mode(0)
2 global filename
3 // ** Sampling Time ** //
4 sampling_time = 1;
5 // // // // * * * // // // // //
6 m=1;
7
8 port1 = '/dev/ttyUSB0'; // For linux users
9 port2 = 'COM2'; // For windows users
10
11 res=init([port1 port2]);
12 disp(res)

```

11.9.1 Conventional Controller, local

11.9.1.1 Fan Disturbance in PI Controller

Scilab Code 11.2 pi_bda_dist.sci

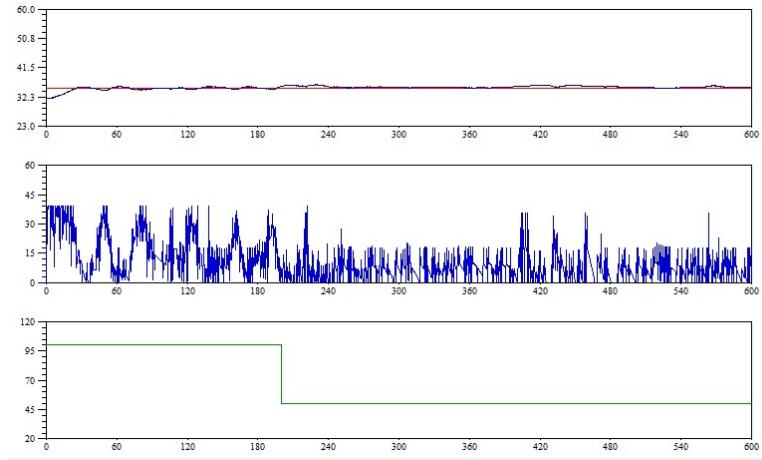


Figure 11.39: Results for the fan input change from 100 to 50 to conventional PID controller designed using Ziegler Nichols tuning

```

1 mode(0)
2 // global temp heat fan sampling_time m heatdisp
   fandisp tempdisp x name
3 function temp = pi_bda_dist(setpoint,fan,K,Ti)
4 global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name temp heat_in fan_in C0 u_old
   u_new e_old e_new
5
6 Ts = sampling_time;
7 e_new = setpoint - temp;
8
9 S0=K*(1+((Ts/Ti)));
10 S1=-K;
11 u_new = u_old + S0*e_new + S1*e_old;
12
13
14 u_old = u_new;
15 e_old = e_new;
16
17 heat = u_new;
18     temp = comm(heat,fan);

```

```

19
20     plotting([heat fan temp setpoint],[0 0 20 0],[100
21           100 40 1000])
22
23     m=m+1;
endfunction

```

11.9.1.2 Set Point Change in PI Controller

Scilab Code 11.3 pi_bda.sci

```

1 mode(0)
2 function temp = pi_bda(setpoint,fan,K,Ti)
3 global heatdisp fandisp tempdisp setpointdisp
4     sampling_time m name temp heat_in fan_in C0 u_old
5     u_new e_old e_new
6
7 Ts = sampling_time;
8 e_new = setpoint - temp;
9 S0=K*(1+((Ts/Ti)));
10 S1=-K;
11 u_new = u_old+(S0*e_new)+(S1*e_old);
12
13 u_old = u_new;
14 e_old = e_new;
15
16 heat = u_new;
17 temp = comm(heat,fan);
18
19 plotting([heat fan temp setpoint],[0 0 20 0],[100
20           100 40 1000])
21
22 m=m+1;
endfunction

```

11.9.1.3 Fan Disturbance to PID Controller

Scilab Code 11.4 pid_bda_dist.sci

```
1 mode(0)
2 function [temp] = pid_bda_dist(setpoint,fan,K,Ti,Td)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
4 e_old_old
5 global heatdisp fandisp tempdisp setpointdisp
6 sampling_time m name
7 e_new = setpoint - temp;
8
9 Ts=sampling_time;
10
11 S0=K*(1+(Ts/Ti)+(Td/Ts));
12 S1=K*(-1-((2*Td)/Ts));
13 S2=K*(Td/Ts);
14
15 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
16
17 u_old = u_new;
18 e_old_old = e_old;
19 e_old = e_new;
20
21
22 heat = u_new;
23 temp = comm(heat,fan);
24
25 plotting([heat fan temp setpoint],[0 0 20 0],[100
26 100 40 1000])
27 m=m+1;
28 endfunction
```

11.9.1.4 Set Point Change in PID Controller

Scilab Code 11.5 pid_bda.sci

```
1 mode(0)
2 function [temp] = pid_bda(setpoint,fan,K,Ti,Td)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
e_old_old
4
5 global heatdisp fandisp tempdisp setpointdisp
sampling_time m name
6
7 e_new = setpoint - temp;
8
9 Ts=sampling_time;
10
11 S0=K*(1+(Ts/Ti)+(Td/Ts));
12 S1=K*(-1-((2*Td)/Ts));
13 S2=K*(Td/Ts);
14
15 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
16
17 u_old = u_new;
18 e_old_old = e_old;
19 e_old = e_new;
20
21
22 heat = u_new;
23 temp = comm(heat,fan);
24
25 plotting([heat fan temp setpoint],[0 0 20 0],[100
100 40 1000])
26
27 m=m+1;
28 endfunction
```

11.9.2 Self Tuning Controller, local

11.9.2.1 Fan Discturbance to PI Controller

Scilab Code 11.6 pi_bda_tuned_dist.sci

```
1 mode(0)
2 // global temp heat fan sampling_time m heatdisp
3 // fandisp tempdisp x name
4 function temp = pi_bda_tuned_dist(setpoint,fan)
5 global heatdisp fandisp tempdisp setpointdisp
6 sampling_time m name temp heat_in fan_in C0 u_old
7 u_new e_old e_new
8
9
10
11
12 // the above is the ziegler nichols part
13
14
15 K = 2/(0.016*temp - 0.114);
16 Ti = (66.90 - 0.415*temp);
17
18 // the above is the direct synthesis part
19
20 e_new = setpoint - temp;
21 S0=K*(1+((Ts/Ti)));
22 S1=-K;
23 u_new = u_old+(S0*e_new)+(S1*e_old);
24
25
26 u_old = u_new;
27 e_old = e_new;
28
29 heat = u_new;
```

```

30
31     temp = comm(heat,fan);
32
33     plotting([heat fan temp setpoint],[0 0 20 0],[100
34           100 40 1000])
35
36     m=m+1;
endfunction

```

11.9.2.2 Set Point Change to PI Controller

Scilab Code 11.7 pi_bda_tuned.sci

```

1 mode(0)
2 function temp = pi_bda_tuned(setpoint,fan)
3 global heatdisp fandisp tempdisp setpointdisp
4         sampling_time m name temp heat_in fan_in C0 u_old
5         u_new e_old e_new
6
7 // L = 6;
8 // R = (0.016 * temp - 0.114) / (66.90 - 0.415 * temp);
9 // K = 0.9 / (R * L);
10 // Ti = 3 * L;
11
12
13 // The above is the Ziegler nichols part.
14
15 K = 2/(0.016*temp-0.114);
16 Ti = (66.90-0.415*temp);
17 // The above is the direct synthesis part
18
19 e_new = setpoint - temp;
20
21 S0=K*(1+((Ts/Ti)));
22 S1=-K;
23 u_new = u_old+(S0*e_new)+(S1*e_old);

```

```

24
25 u_old = u_new;
26 e_old = e_new;
27
28 heat = u_new;
29 temp = comm(heat,fan);
30
31 plotting([heat fan temp setpoint],[0 0 20 0],[100
   100 40 1000])
32
33 m=m+1;
34 endfunction

```

11.9.2.3 Fan Disturbance to PID Controller

Scilab Code 11.8 pid_bda_tuned_dist.sci

```

1 mode(0)
2
3 function [temp] = pid_bda_tuned_dist(setpoint,fan)
4 global temp heat_in fan_in C0 u_old u_new e_old e_new
   e_old_old
5
6 global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name
7
8 L = 6;
9 R = (0.016*temp - 0.114)/(66.90 - 0.415*temp);
10 K = 1.2/(R*L)
11 Ti = 2*L;
12 Td = 0.5*L;
13
14
15 e_new = setpoint - temp;
16
17 Ts=sampling_time;
18
19 S0=K*(1+(Ts/Ti)+(Td/Ts));

```

```

20 S1=K*(-1-((2*Td)/Ts));
21 S2=K*(Td/Ts);
22
23 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
24
25 u_old = u_new;
26 e_old_old = e_old;
27 e_old = e_new;
28
29
30 heat = u_new;
31 temp = comm(heat,fan);
32
33 plotting([heat fan temp setpoint],[0 0 20 0],[100
100 40 1000])
34
35 m=m+1;
36 endfunction

```

11.9.2.4 Set Point Change to PID Controller

Scilab Code 11.9 pid_bda_tuned.sci

```

1 mode(0)
2 function [temp] = pid_bda_tuned(setpoint,fan)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
e_old_old
4
5 global heatdisp fandisp tempdisp setpointdisp
sampling_time m name
6
7 L = 6;
8 R = (0.016*temp - 0.114)/(66.90 - 0.415*temp);
9 K = 1.2/(R*L)
10 Ti = 2*L;
11 Td = 0.5*L;
12
13

```

```

14 e_new = setpoint - temp;
15
16 Ts=sampling_time;
17
18 S0=K*(1+(Ts/Ti)+(Td/Ts));
19 S1=K*(-1-((2*Td)/Ts));
20 S2=K*(Td/Ts);
21
22 u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
23
24 u_old = u_new;
25 e_old_old = e_old;
26 e_old = e_new;
27
28
29 heat = u_new;
30 temp = comm(heat,fan);
31
32 plotting([heat fan temp setpoint],[0 0 20 0],[100
   100 40 1000])
33
34 m=m+1;
35 endfunction

```

11.9.3 Conventional Controller, virtual

11.9.4 Fan Disturbance in PI Controller

Scilab Code 11.10 pi_bda_dist.sci

```

1 function [stop] = pi_bda_dist(setpoint,fan,K,Ti)
2
3     global u_old u_new e_old e_new S0 S1 temp
4     e_new = setpoint - temp;
5
6     Ts=sampling_time;
7     S0=K*(1+((Ts/Ti)));
8     S1=-K;

```

```

9      u_new = u_old + S0*e_new + S1*e_old ;
10
11      heat=u_new ;
12
13      u_old = u_new ;
14      e_old = e_new ;
15
16      [ stop ,temp ] = comm(heat ,fan ) ; // N e v e r   e d i t   t h i s
17      plotting([heat fan temp setpoint],[0 0 30 0],[100
18          100 50 1000])
19
20  endfunction

```

11.9.4.1 Set Point Change in PI Controller

Scilab Code 11.11 pi_bda.sci

```

1 function [ stop ] = pi_bda( setpoint ,fan ,K, Ti )
2
3
4
5     global u_old u_new e_old e_new S0 S1 temp
6
7     e_new = setpoint - temp ;
8
9
10
11    Ts=sampling_time ;
12
13    S0=K*( 1+(( Ts / Ti )) ) ;
14    S1=-K;
15    u_new = u_old +(S0*e_new)+(S1*e_old) ;
16
17
18    heat=u_new ;
19
20

```

```

21
22     u_old = u_new;
23
24     e_old = e_new;
25
26
27
28     [stop,temp] = comm(heat,fan); // Never edit this
29         line
30
31     plotting([heat fan temp setpoint],[0 0 30 0],[100
32         100 50 1000])
33
34 endfunction

```

11.9.4.2 Fan Disturbance to PID Controller

Scilab Code 11.12 pid_bda_dist.sci

```

1 function [stop] = pid_bda_dist(setpoint,fan,K,Ti,Td)
2     global u_old u_new e_old e_old_old e_new S0 S1
3         temp
4
5     e_new = setpoint - temp;
6     Ts=sampling_time;
7
8     S0=K*(1+(Ts/Ti)+(Td/Ts));
9     S1=K*(-1-((2*Td)/Ts));
10    S2=K*(Td/Ts);
11
12    u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old
13        ;
14
15    heat=u_new;
16    u_old = u_new;

```

```

16     e_old_old = e_old ;
17     e_old = e_new ;
18
19
20     [ stop ,temp ] = comm(heat ,fan ) ; // N e v e r   e d i t   t h i s
21             l i n e
22
23     plotting([heat fan temp setpoint],[0 0 30 0],[100
24             100 50 1000])
25
26
27 endfunction

```

11.9.4.3 Set Point Change in PID Controller

Scilab Code 11.13 pid_bda.sci

```

1 function [ stop ] = pid_bda( setpoint ,fan ,K,Ti ,Td )
2     global u_old u_new e_old e_old_old e_new S0 S1
3             temp
4
5     e_new = setpoint - temp ;
6     Ts=sampling_time ;
7
8     S0=K*(1+(Ts/Ti)+(Td/Ts)) ;
9     S1=K*(-1-((2*Td)/Ts)) ;
10    S2=K*(Td/Ts) ;
11
12    u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old
13            ;
14
15    heat=u_new ;
16    u_old = u_new ;
17    e_old_old = e_old ;
18    e_old = e_new ;
19

```

```

20 [ stop ,temp ] = comm(heat ,fan ); // N e v e r   e d i t   t h i s
21   l i n e
22 plotting([ heat  fan  temp  setpoint ],[0  0  30  0 ],[100
23   100  50  1000])
24
25 endfunction

```

11.9.5 Self Tuning Controller, local

11.9.5.1 Fan Discturbance to PI Controller

Scilab Code 11.14 pi_bda_tuned_dist.sci

```

1 function [ stop ] = pi_bda_tuned_dist(setpoint ,fan )
2
3   global u_old u_new e_old e_new S0 S1 temp
4
5   K = 2/(0.016*temp -0.114) ;
6   Ti = (66.90 -0.415*temp) ;
7
8   e_new = setpoint - temp ;
9
10  Ts=sampling_time ;
11  S0=K*(1+((Ts/Ti))) ;
12  S1=-K;
13  u_new = u_old +(S0*e_new)+(S1*e_old ) ;
14
15  heat=u_new ;
16
17  u_old = u_new ;
18  e_old = e_new ;
19
20 [ stop ,temp ] = comm(heat ,fan ); // N e v e r   e d i t   t h i s
21   l i n e
22 plotting([ heat  fan  temp  setpoint ],[0  0  30  0 ],[100
23   100  50  1000])

```

```

22
23 endfunction


---



```

11.9.5.2 Set Point Change to PI Controller

Scilab Code 11.15 pi_bda_tuned.sci

```

1 function [ stop ] = pi_bda_tuned( setpoint , fan )
2
3     global u_old u_new e_old e_new S0 S1 temp
4
5     K = 2/(0.016*temp - 0.114) ;
6     Ti = (66.90 - 0.415*temp) ;
7
8     e_new = setpoint - temp ;
9
10    Ts=sampling_time ;
11    S0=K*(1+((Ts/Ti))) ;
12    S1=-K;
13    u_new = u_old +(S0*e_new)+(S1*e_old) ;
14
15    heat=u_new ;
16
17    u_old = u_new ;
18    e_old = e_new ;
19
20    [ stop , temp ] = comm(heat,fan); // Never edit this
21                                line
22                                plotting([heat fan temp setpoint],[0 0 30 0],[100
23                                100 50 1000])
24
25 endfunction

```

11.9.5.3 Fan Disturbance to PID Controller

Scilab Code 11.16 pid_bda_tuned_dist.sci

```

1 function [ stop ] = pid_bda_tuned_dist( setpoint , fan )

```

```

2
3   global u_old u_new e_old e_old_old e_new S0 S1
4   temp
5
6   L = 6;
7   R = (0.016*temp - 0.114)/(66.90 - 0.415*temp);
8   K = 1.2/(R*L)
9   Ti = 2*L;
10  Td = 0.5*L;
11
12
13  Ts=sampling_time;
14  S0=K*(1+(Ts/Ti)+(Td/Ts));
15  S1=K*(-1-((2*Td)/Ts));
16  S2=K*(Td/Ts);
17
18  u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old
19  ;
20
21  heat=u_new;
22
23  u_old = u_new;
24  e_old_old = e_old;
25  e_old = e_new;
26
27  [stop,temp] = comm(heat,fan); // Never edit this
28  line
29  plotting([heat fan temp setpoint],[0 0 30 0],[100
100 50 1000])
30
31 endfunction

```

11.9.5.4 Set Point Change to PID Controller

Scilab Code 11.17 pid_bda_tuned.sci

```

1 function [stop] = pid_bda_tuned(setpoint,fan)
```

```

2
3     global u_old u_new e_old e_old_old e_new S0 S1
      temp
4
5     L = 6;
6     R = (0.016*temp - 0.114)/(66.90 - 0.415*temp);
7     K = 1.2/(R*L)
8     Ti = 2*L;
9 // Kc and tau_i calculated
10    Td = 0.5*L;
11
12    e_new = setpoint - temp;
13
14    Ts=sampling_time;
15    S0=K*(1+(Ts/Ti)+(Td/Ts));
16    S1=K*(-1-((2*Td)/Ts));
17    S2=K*(Td/Ts);
18
19    u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old
              ;
20
21
22    heat=u_new;
23
24    u_old = u_new;
25    e_old_old = e_old;
26    e_old = e_new;
27
28    [stop,temp] = comm(heat,fan); // Never edit this
           line
29    plotting([heat fan temp setpoint],[0 0 30 0],[100
           100 50 1000])
30
31 endfunction

```

Chapter 12

Model Predictive Control in Single Board Heater System using SCILAB

This chapter presents Model Predictive Control in Single Board Heater System done by Mr. Pratik Behera.¹

12.1 Objective

- To implement Model Predictive Control (MPC) in Single Board Heater System using Scilab and perform experiments using it
- To perform experiments for various values of tuning parameters and study its effect on the system

12.2 Single Board Heater System

It is a single heater system, where in, 5cm x 2cm stainless steel blade acts as a plant, which is heated by a heating coil and cooled by a fan.

For Single Board Heater System (SBHS):

¹Copyright: Mr.Pratik Behera

- Control variable: temperature
- Manipulated variable: heater
- Disturbance variable: fan

The heater element consists of Nichrome wire - of 0.7mm diameter, wound with 20 equally spaced helical turns into a coil of 5mm x 11mm. The heater element is kept at a distance of 3.5 mm from the steel blade.

Cooling is done by a computer fan, which is placed below the stainless steel blade.

12.3 Model Predictive Control

An equivalent quadratic programming (QP) formulation for constrained DMC (as given in LQG_MPC_notes by Prof Sachin Patwardhan) is given as follows

$$\min_{U_f} \frac{1}{2} U_f(k)^T H U_f(k) + F^T U_f(k) \quad (12.1)$$

Subject to

$$A U_f(k) \leq b \quad (12.2)$$

where

$$(12.3)$$

$$A = \begin{bmatrix} I_{qm} \\ -I_{qm} \end{bmatrix}$$

$$b = \begin{bmatrix} U^H \\ -U_L \end{bmatrix}$$

Also, we have outputs and manipulated variables related to state variables by

$$x(k + 1) = \Phi x(k) + \Gamma(k) + w(k) \quad (12.4)$$

$$y(k) = Cx(k) + v(k) \quad (12.5)$$

$$(12.6)$$

ϕ is represented by matrix A in the code, Γ is represented as matrix B and C is represented as C matrix in the code.

12.4 Implementing MPC

As mentioned earlier, MPC experiments were performed on SBHS 12 remotely. For this, the scilab codes uploaded on Moodle for Process Control SBHS assignments were used. The folder containing the codes, which was used to perform MPC experiments, have been included in the attached zip file in a folder named *codes*.

12.5 Working of codes

There are three main codes, which are being used for this experiment. *mpc_init.sce* is the code which opens the xcos window, wherein, we have step block for the set-point for temperature and the fan speed. Once the values have been entered into the xcos window and the simulation is started, the *scifunc* block of xcos calls the function *mpc.sci* after every sampling time. The *mpc.sci* in turn calls *mpc_run.sci* every time it is called by *scifunc* block. The *mpc_run.sci* code optimizes manipulated variable (heater) over control horizon and returns only the first manipulated variable (heater) value. This new heater value is then sent to the heater of the SBHS to control the temperature at the set point.

12.6 Procedure to implement MPC on SBHS

1. Open the folder named *Client-Java-latest10032011* and open VirtualLab-Client.
2. After entering the details and connecting to the allotted SBHS, open Scilab 5.3.1

3. Change the current directory to this folder, where mpc codes are present.
4. Open *mpc_init.sce* and *load in scilab*
5. The xcos window opens. Description of xcos window is mentioned below in the next section.
6. Enter the required step change values, if any, in the Temperature set point and/or fan block.
7. Enter the sampling time in the clock block as 1 second
8. Start the simulation from the xcos
9. After the experiment is over, the data files can be downloaded from the Java client.

12.7 Description of xcos

When *mpc_init.sce* is executed in scilab, an xcos window opens up. The xcos window has two step input blocks. The first step input block on the left side, is for the Temperature set point and the second step input block is for the fan (disturbance variable). Also the sampling time can be entered via clock block present on the xcos.

For all the experiments done for this project, sampling time of 1 second was used (entered via clock block of xcos).

Refer to the figure below for a clear picture of the xcos.

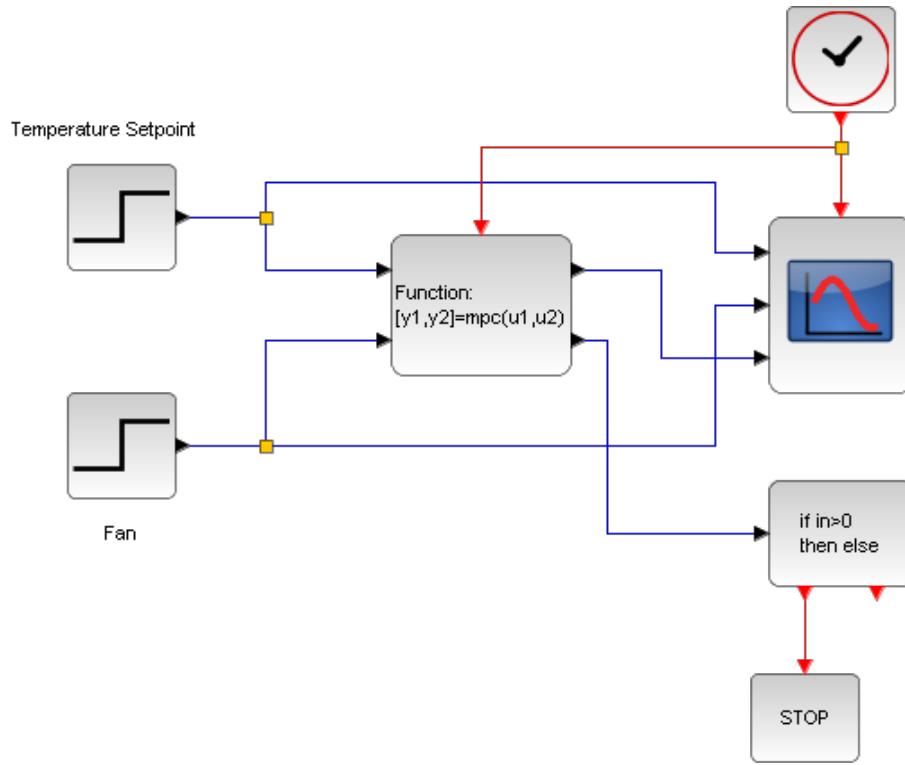


Figure 12.1: Screenshot of the xcos window with step input blocks labeled

After entering the values in the input step block, the simulation can be started. This opens up a graph, which shows the values of Temperature-set-point, fan and the actual temperature at each time instant during the simulation.

12.8 Code for MPC (*mpc_run.sci*)

The MPC code, implemented by me has been mentioned below.

```

1 function [ u_new ] = mpc_run(T, u_prev , Tset)
2 global p q xk_old
3
4 A = [0.9780 0 0 0 0 0 0 0;
5           1 0 0 0 0 0 0 0;
6           0 1 0 0 0 0 0 0;
```

```

7      0    0 1 0 0 0 0 0;
8      0    0 0 1 0 0 0 0;
9      0    0 0 0 1 0 0 0;
10     0    0 0 0 0 1 0 0;
11     0    0 0 0 0 0 1 0];
12 B = [1; 0; 0; 0; 0; 0; 0; 0];
13 C = [0 0 0 0 0 0 0.0079];
14
15 Tmax = 70;           // Maximum Temperature
16 We = 100*eye(p,p); // Error Weighting Matrix , We
17 Wu = 10*eye(q,q); // Control Weighting Matrix ,
    Wu
18 xk=A*xk_old+B*u_prev;
19
20 // Formation of Su Matrix for Quadratic term of
   optimization
21 for i = 1:1:p
22   for j = 1:1:q
23     if i <= q
24       if (i-j) >= 0
25         Su(i,j)= C*A^(i-j)*B;
26       else
27         Su(i,j)= 0;
28     end
29   else
30     if j < q
31       Su(i,j)= C*A^(i-j)*B;
32     else
33       Su(i,j) = Su(i-1,j) + C*A^(i-j)*B;
34     end
35   end
36 end
37 end
38
39 du_matrix=ones(q,q);
40
41 // Lambda Matrix for Quadratic term of optimization
42 for i = 1:1:q

```

```

43   for j = 1:1:q
44     if i == j
45       du_matrix(i,j) = 1;
46     if i >1
47       du_matrix(i,j-1) = -1;
48     end
49   else
50     du_matrix(i,j) = 0;
51   end
52 end
53
54
55 du_matrix_0 = eye(1,q); // Declaration of Lambda_0
56   vector
57
58 // Formation of Sx Matrix for Linear term of
59 // optimization
60 for i = 1:1:p
61   Sx(i,:) = C * A^i;
62 end
63
64 // Declaration of S_eta Matrix
65 S_eta = ones(1,p);
66
67 // Declaration of Set Point Vector
68 R = ones(1,p)*Tset;
69
70
71 // Temperature Prediction using information till
72 // previous instant
73 T_pred = C*xk;
74
75 // Measurement Error
76 eta = T - T_pred;
77
78 // Quadratic Term for Optimization
79 Su_t=Su';
80 du_matrix_t=du_matrix';
81 Q=2*((Su_t*We*Su)+(du_matrix_t*Wu*du_matrix));

```

```

78
79 // Linear Term in Optimization
80 R_t=R';
81 S_eta_t=S_eta';
82 du_matrix_0_t=du_matrix_0';
83 F_term1_t=(R_t-(Sx*xk)-(S_eta_t*eta))';
84 F=-2*((F_term1_t*We*Su)+((du_matrix_0_t*u_prev)']*Wu*
     du_matrix));
85
86 // Inequality Matrices and Vectors
87 A_ineq=[eye(q,q); -1*eye(q,q)];
88
89 b_ineq_term1_1=-Sx*xk-S_eta';
90 b_ineq_term1_2=(Tmax*ones(1,p))';
91 b_ineq_term1=(b_ineq_term1_1*eta+b_ineq_term1_2)';
92 b_ineq_term2_1=-Sx*xk-S_eta'*eta;
93 b_ineq_term2_2=(Tmax*ones(1,p))';
94 b_ineq_term2=-1*(b_ineq_term2_1+b_ineq_term2_2)';
95 b_ineq=[40*ones(1,q) zeros(1,q)];
96
97 me=0;
98 ci=zeros(q,1);
99 cs=40*ones(q,1);
100
101 [x,iact,iter,f]=qpsolve(Q,F,A_ineq,b_ineq');
102
103 u_new=x(1);
104 xk_old=xk;
105 endfunction

```

12.9 Other codes used

Other codes that were used for conducting this experiment are *mpc_init.sce* and *mpc.sci*. Both these codes were originally taken from Moodle (Process controls course for SBHS assignment). Please note that, both these codes were slightly modified to work with our MPC.

The only changes done in the original codes are:

- addition of global variables p,q and xk_old (p is the prediction horizon, q is the control horizon, xk_old represents the last value of an internal state)
- initialization of p, q and xk_old
- removal of some unnecessary lines (ie, lines not relevant to MPC implementation)

12.10 Experiments conducted to implement MPC

Experiments were performed as shown in table above for implementation of MPC. We carried out experiments in which both positive and negative step changes were given to Set point and Fan (disturbance variable) and the output response was obtained by application of MPC. We also have performed several experiments to study the effect of change in the values of q (control horizon) and tuning parameters - error and manipulated variable weighting factors.

The details of the experiments mentioned in this report has been tabulated in the table given in the next page. The first column of the table represents the experiment version (or number). For all the outputs and their figures, we have mentioned only their experiment version (or number) to tag them. Also note that the data files for these experiments are also named as per their experiment version number.

p and q mentioned in the table represents the prediction and control horizon respectively.

Please note: For all the above experiments and graphs, we adhered to:

- Scilab Version: 5.2.2
- SBHS number: 12 (remotely accessed)
- Sampling time: 1 second

For graphs: Until and unless mentioned, Graphic 1 represents the Temperature set point, Graphic 2 represents the Fan and Graphic 3 represents the Temperature.

Also, please note that there are two types of graphs. The first graph, containing Graphic 1, Graphic 2 and Graphic 3 were directly obtained via mscope of xcos.

The graph following this in all the experiments is the temperature and heater value graphs, which were obtained from the data (from the text file downloaded from the server after each experiment).

Expt No	Temperature Set point			Fan			(p,q)	Weighing factor (We, Wu)
	T_initial (°C)	T_final (°C)	Time (s)	F_initial	F_final	Time (s)		
1.1	35	40	250	100	150	500	(40,4)	1,1
1.2	35	40	250	100	150	500	(40,4)	10,10
1.3	35	40	250	100	150	500	(40,4)	40,40
2.1	42	37	250	150	100	500	(40,4)	1,1
2.2	42	37	250	150	100	500	(40,4)	10,10
2.3	42	37	250	150	100	500	(40,4)	40,40
3.1	35	40	250	100	150	500	(40,2)	10,10
3.2	35	40	250	100	150	500	(40,3)	10,10
3.3	35	40	250	100	150	500	(40,4)	10,10
4.1	42	37	250	150	100	500	(40,2)	10,10
4.2	42	37	250	150	100	500	(40,3)	10,10
4.3	42	37	250	150	100	500	(40,4)	10,10
5.1	35	40	250	100	150	500	(40,4)	100,2
5.2	35	40	250	100	150	500	(40,4)	2,100
5.3	35	40	250	100	150	500	(40,4)	10,100
5.1	35	40	250	100	150	500	(40,4)	100,10

Figure 12.2: Experiments performed

All the experiments mentioned in this report has been labeled as shown in this table. This table is just a summary of all the parameters that was used for the corresponding experiment. Details on the inputs and a description of the output observed for each case has been mentioned in the corresponding section of each experiment.

12.11 Sample run to implement MPC

12.12 Positive Step Change to Set Point and Fan

Let us consider experiment 1.1, wherein, a positive step change of 5° C (from 35° C to 40° C) was provided to set point at time t=250 s and a step change to fan was provided at t = 500 s, from 100 to 150.

The graph obtained has been attached below:

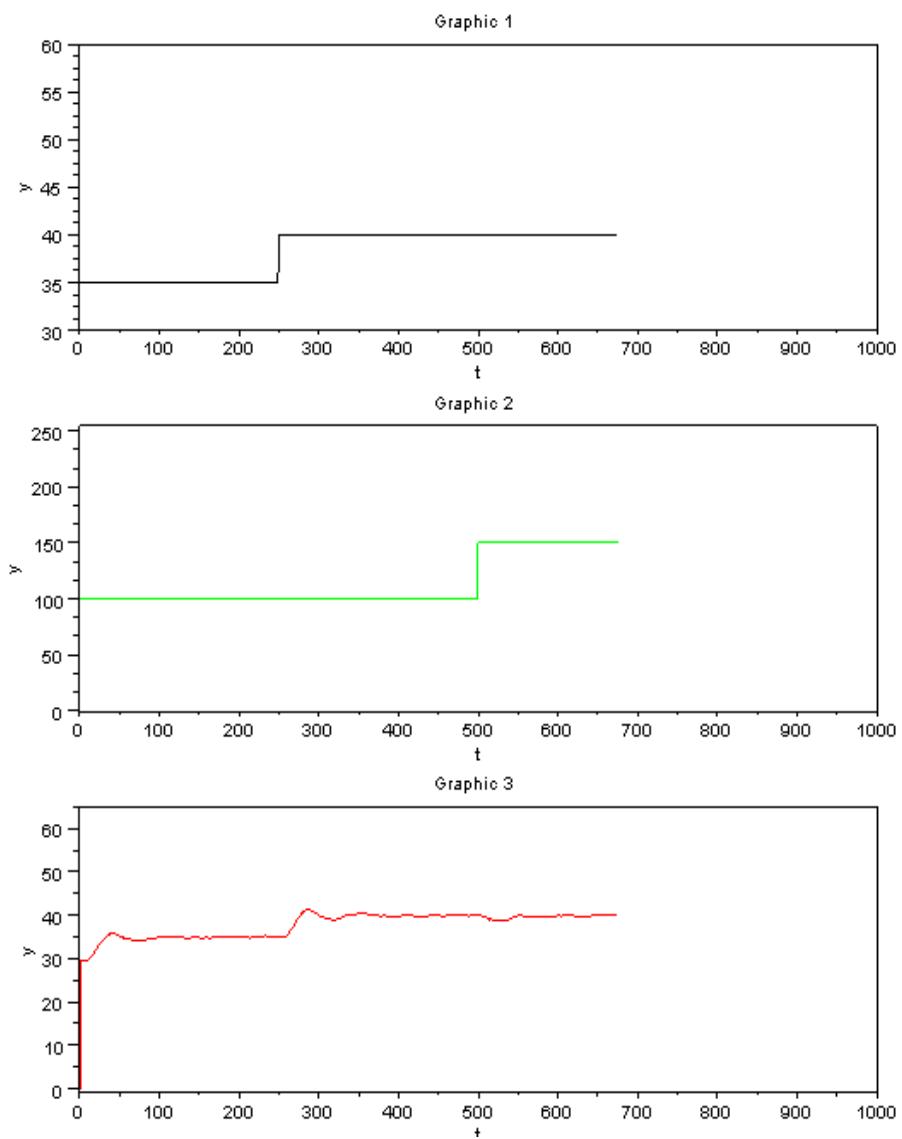


Figure 12.3: Expt 1.1

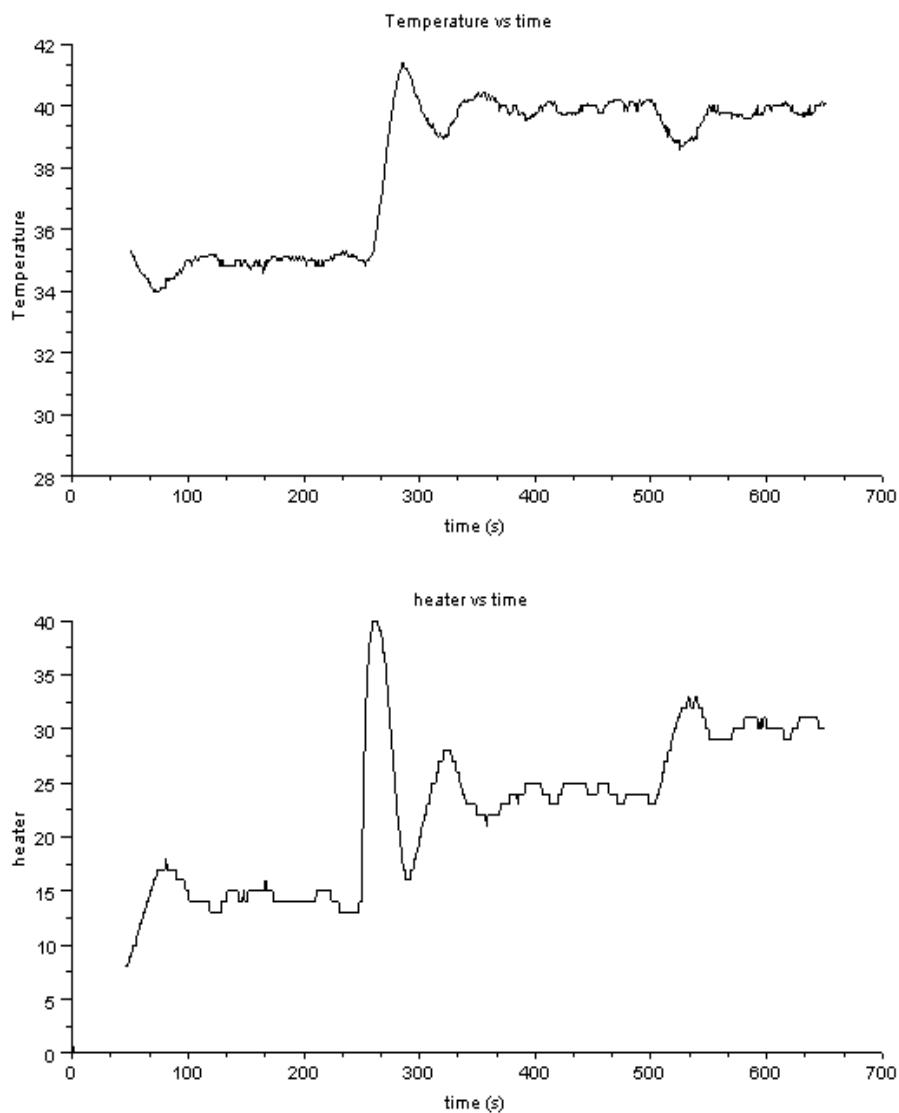


Figure 12.4: Expt 1.1

As can, be seen above, when, the temperature set point is raised to 40 from 30, at $t=250$ s, the value of the heater increases, so that it can heat up the plant upto the required set point. Similarly, when the fan speed is increased at $t=500$ s, the heater value increases yet again to maintain the same constant temperature of the SBHS blade.

12.13 Negative Step Change to Set Point and Fan

Let us consider experiment 2.1, wherein, a negative step change of 5°C (from 42°C to 37°C) was provided to set point at time t=250 s and a step change to fan was provided at t = 500 s, from 150 to 100.

The graph obtained has been attached below:

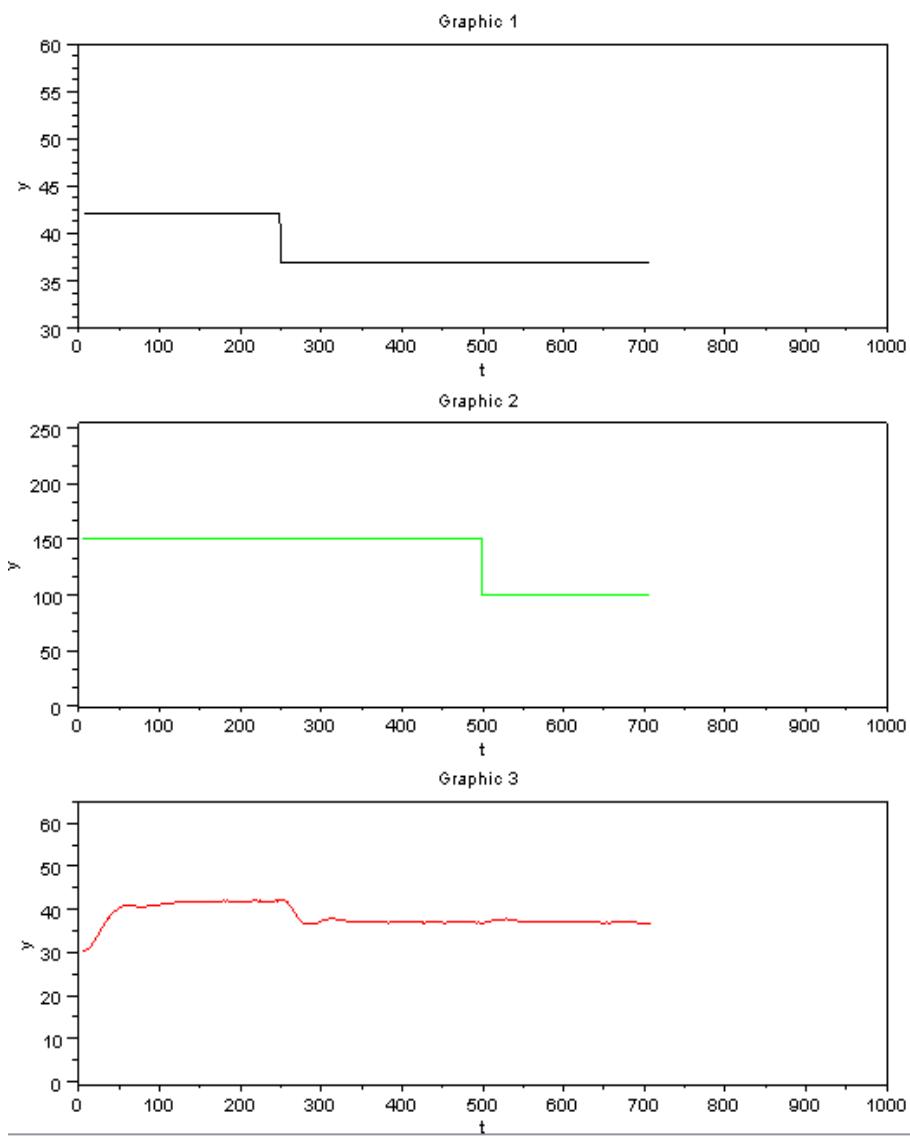


Figure 12.5: Expt 2.1

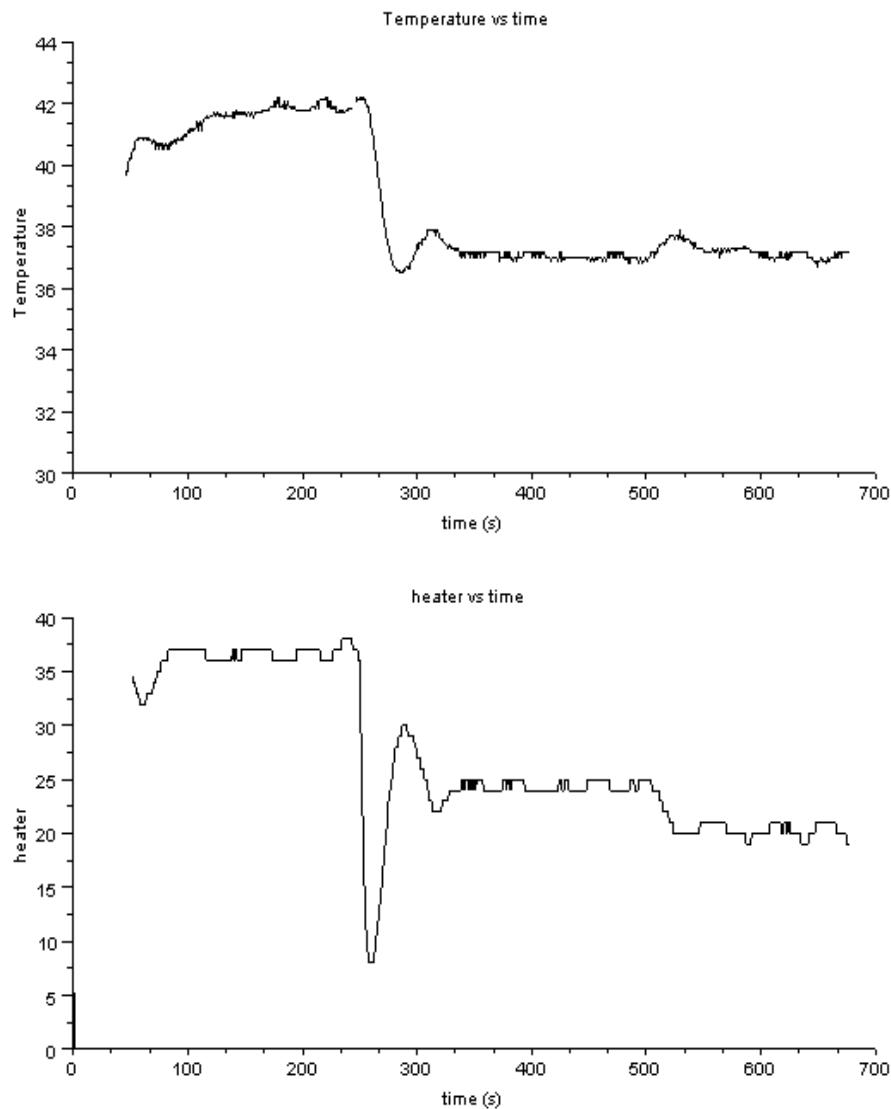


Figure 12.6: Expt 2.1

As can be seen from the graphs above, when the temperature set point drops at $t=250$ s, the value of the heater too falls, so that the plant (SBHS blade) can cool down to the required set point. Similarly, when the fan speed was decreased at $t=500$ s, the heater value decreased yet again to maintain the same constant temperature of the SBHS blade.

12.14 Effect of Tuning parameters: Weighting factors, We and Wu

We also, conducted several experiments in order the study the effect of the value of Weighting factors (both error,We and manipulated variable,Wu). We used weighting factors to be 1, 10 and 40 for both positive and negative step changes to both set point and fan (as has been summarized in Table 1). Also, experiments were done for different values of We and Wu. The results have been shown in the following graph.

12.15 For same factor of We and Wu

12.15.1 Positive Step Change and $(We, Wu) = (1,1)$ (Expt 1.1)

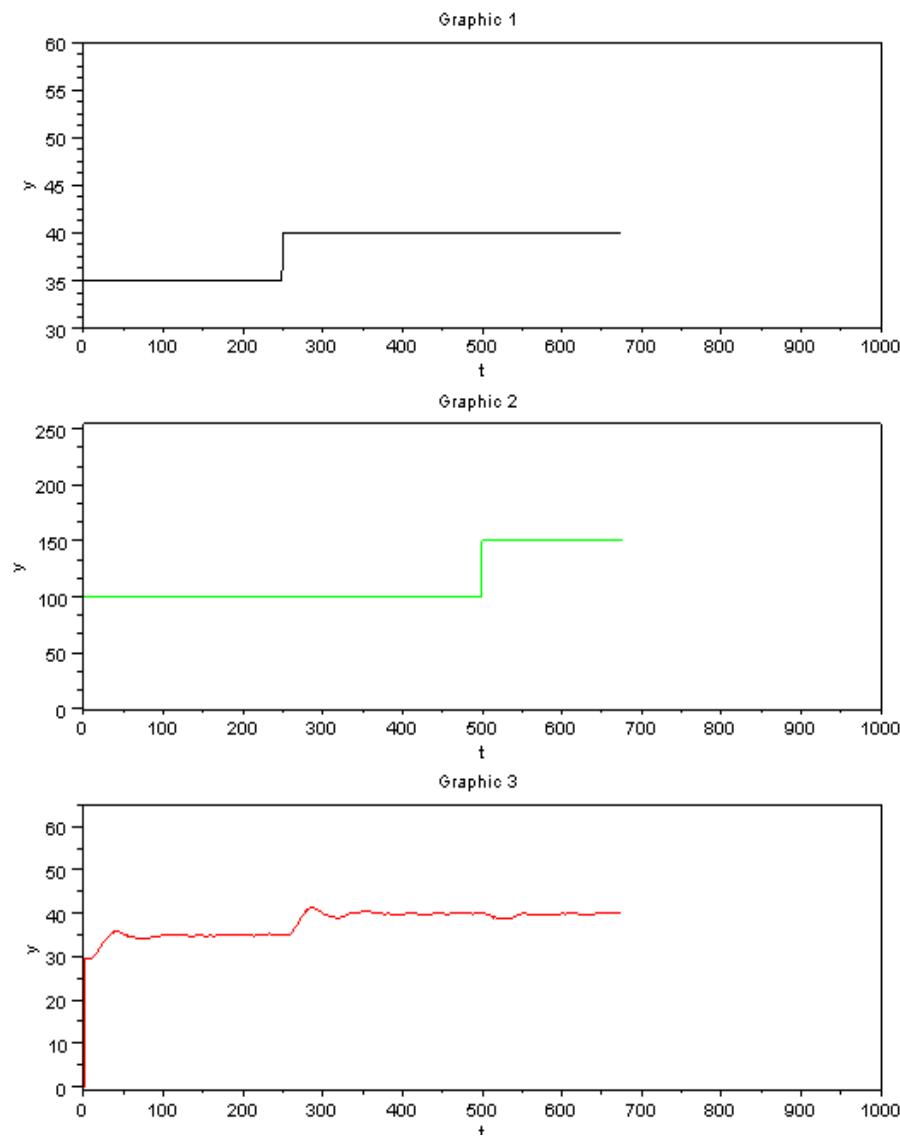


Figure 12.7: Expt 1.1

Here we can clearly see the expected output. Providing a positive step to temperature set point at 250 seconds, increased heater value as per the control effort put in by MPC. A positive step in fan at 500 seconds, decreased the temperature below its set point and hence heater value increased to take the temperature close to its setpoint.

This will be clear from the heater graph attached next.

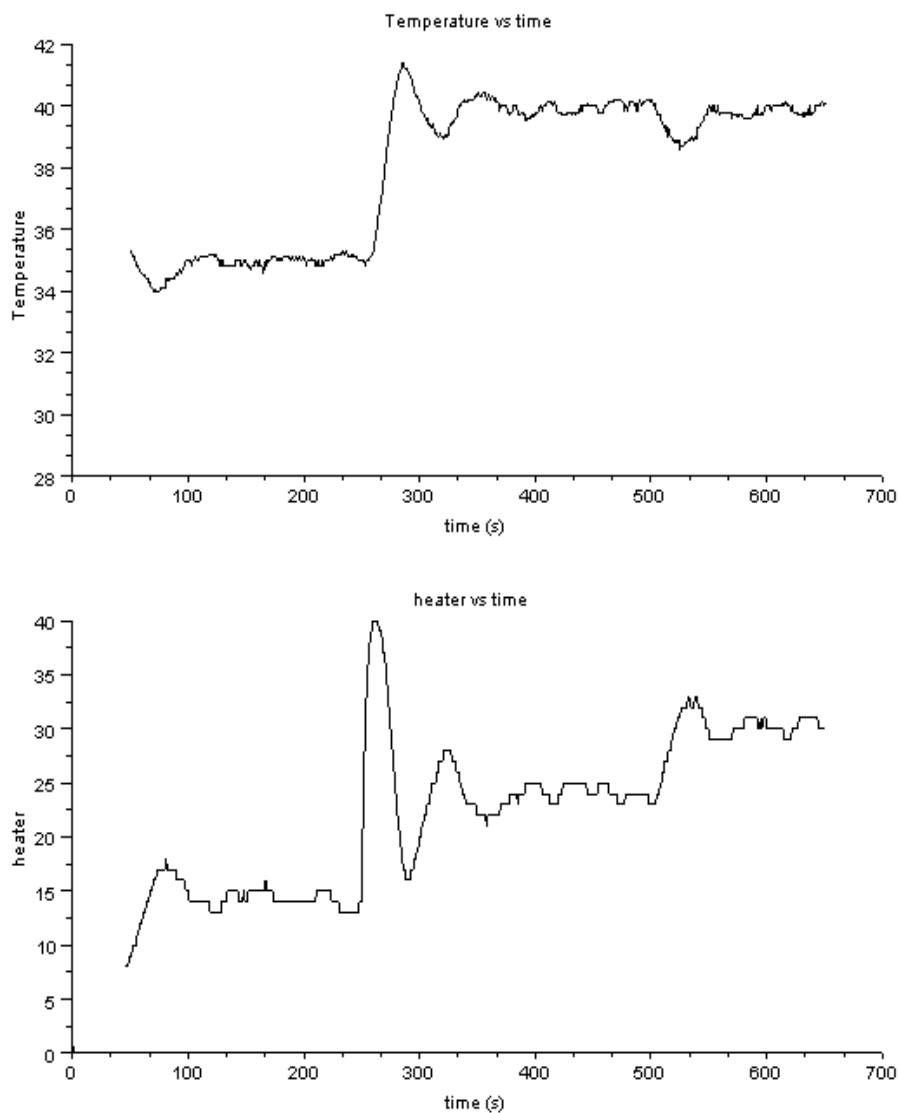


Figure 12.8: Expt 1.1

As can be clearly seen, the heater graph follows the expected trend that we talked of in the last page. Also, note that the temperature variation can be clearly seen from this graph.

This graph shows the result for the case, where we had same weighting factors for both error and manipulated variables (We and Wu). We will now see if changing

both of these is going to have any effect on the control behavior.
So, we now try an experiment with both We and Wu increased to 10.

12.15.2 Positive Step Change and $(We, Wu)=(10,10)$ (Expt 1.2)

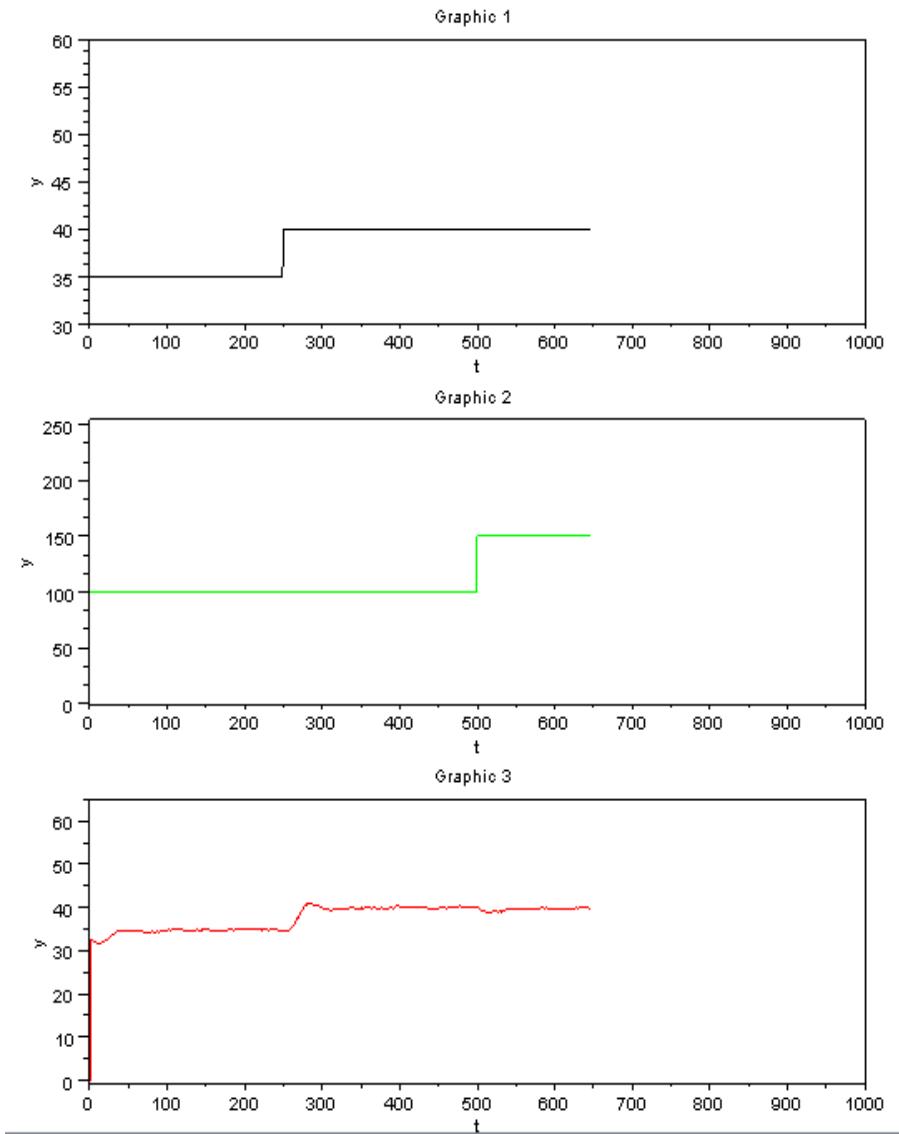


Figure 12.9: Expt 1.2

Using the same logic as has been explained in the last section, we expected to see similar temperature and heater value profiles for the positive step change in temperature set point and the fan. (Heater graph is shown in the next page along with the temperature on an expanded scale).

In this experiment, we increased We and Wu both to 10 from 1 and wish to observe if this changes the response of the plant.

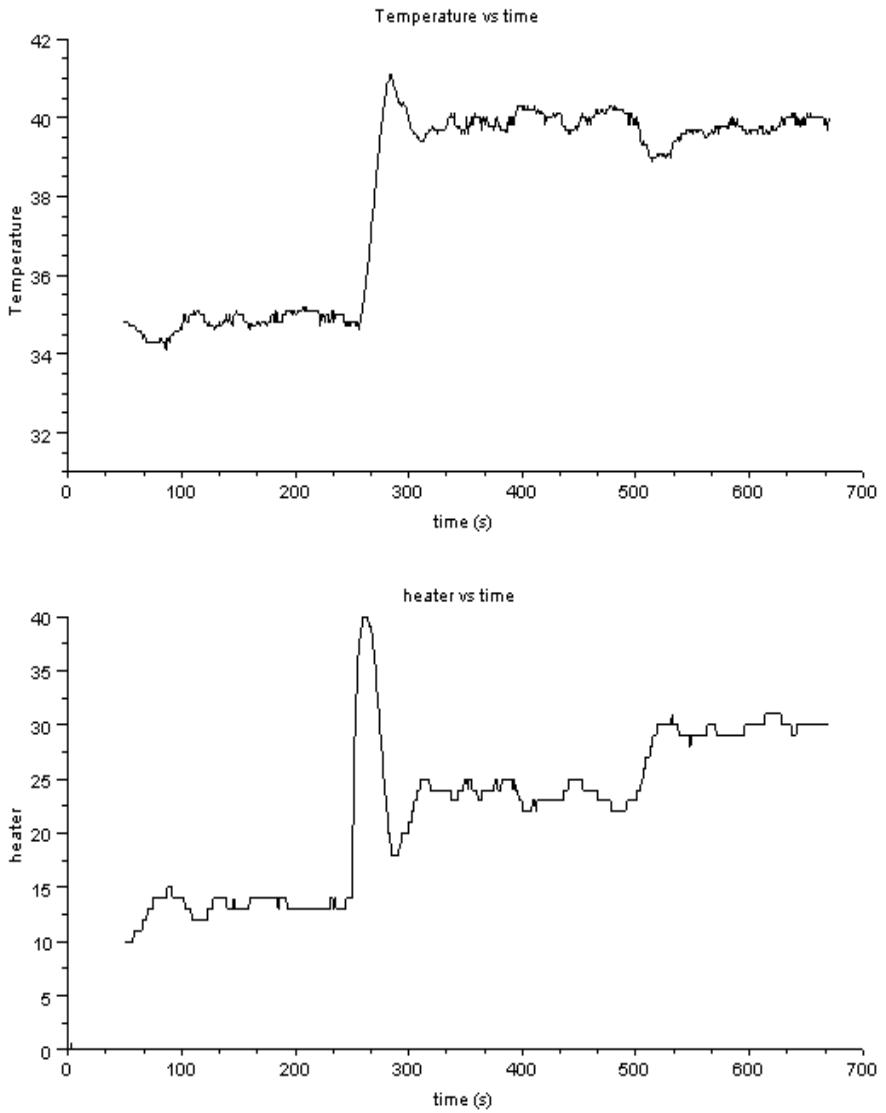


Figure 12.10: Expt 1.2

The results here are almost the same as that mentioned in the last section (where We and Wu both were 1). So, we can for the time being keep in mind that We and Wu isn't actually much affected the output. We now will carry out the experiment for even higher We and Wu (say 40) and see if it really does affect the output much.

12.15.3 Positive Step Change and $(W_e, W_u) = (40, 40)$ (Expt 1.3)

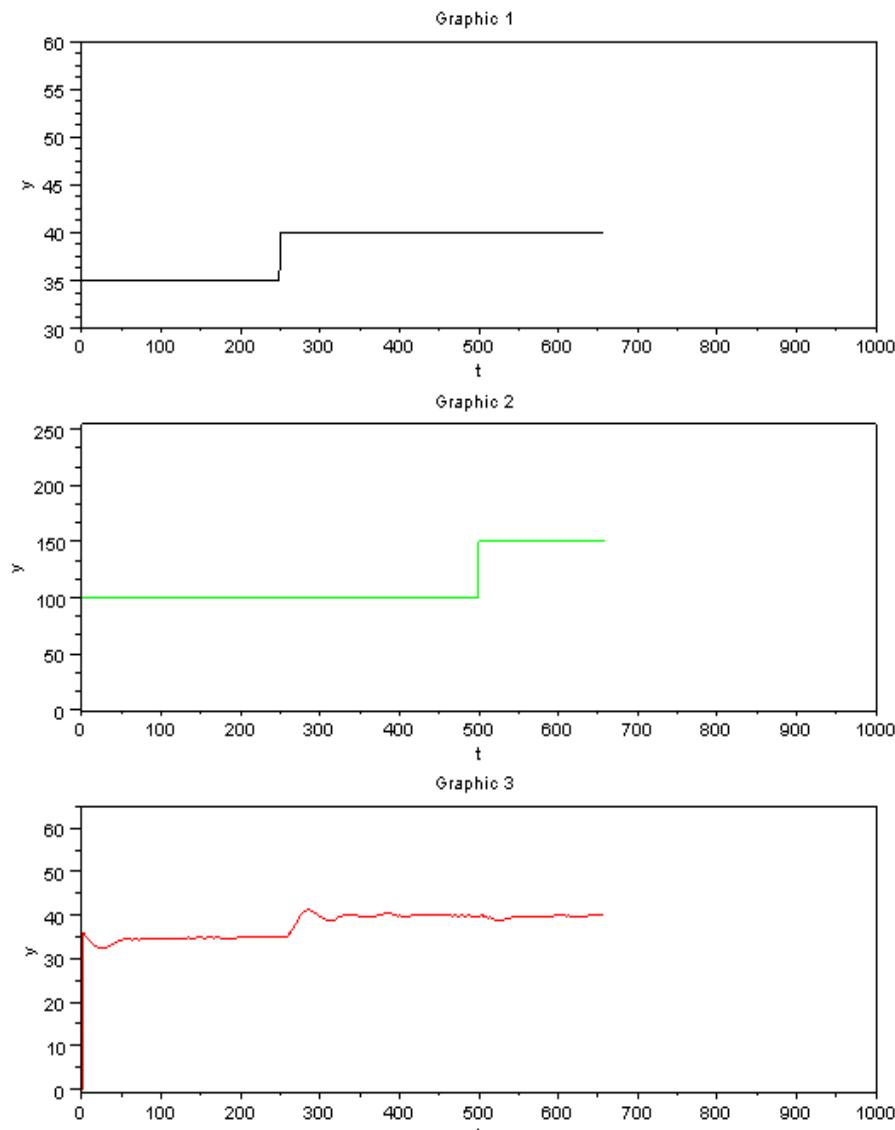


Figure 12.11: Expt 1.3

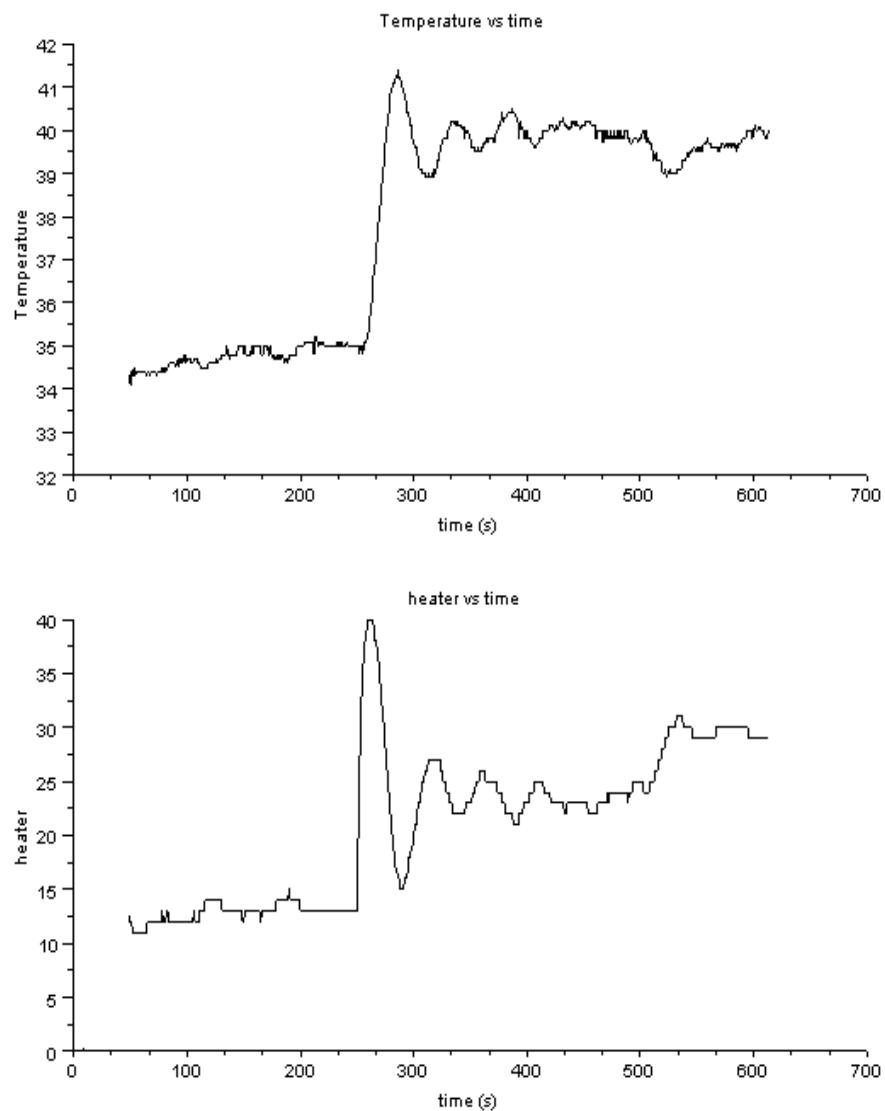


Figure 12.12: Expt 1.3

Even the results with We and Wu as 40 doesn't show much difference. They are more or less similar looking as the last two experiment's results.

12.15.4 Negative Step Change and $(We, Wu) = (1,1)$ (Expt 2.1)

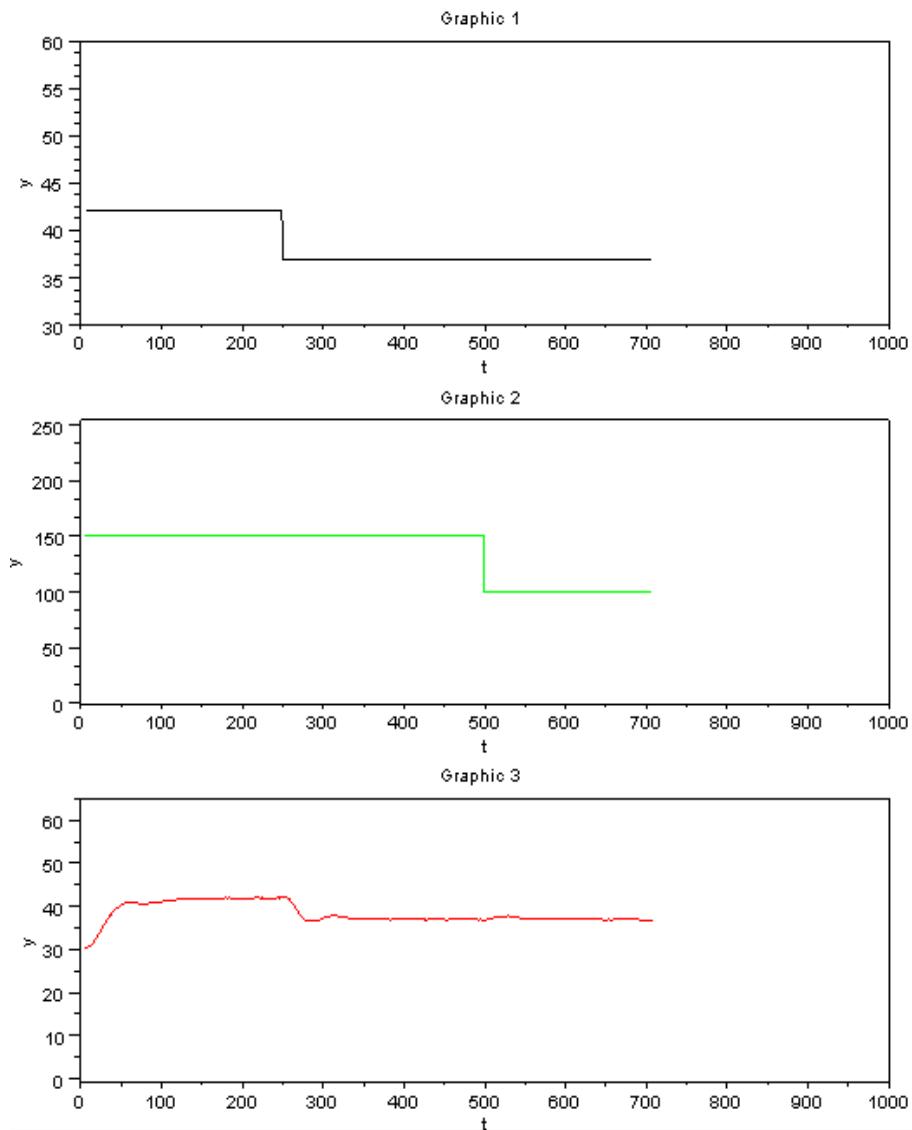


Figure 12.13: Expt 2.1

Here we expect somewhat similar results as was the case with positive step in temperature set point.

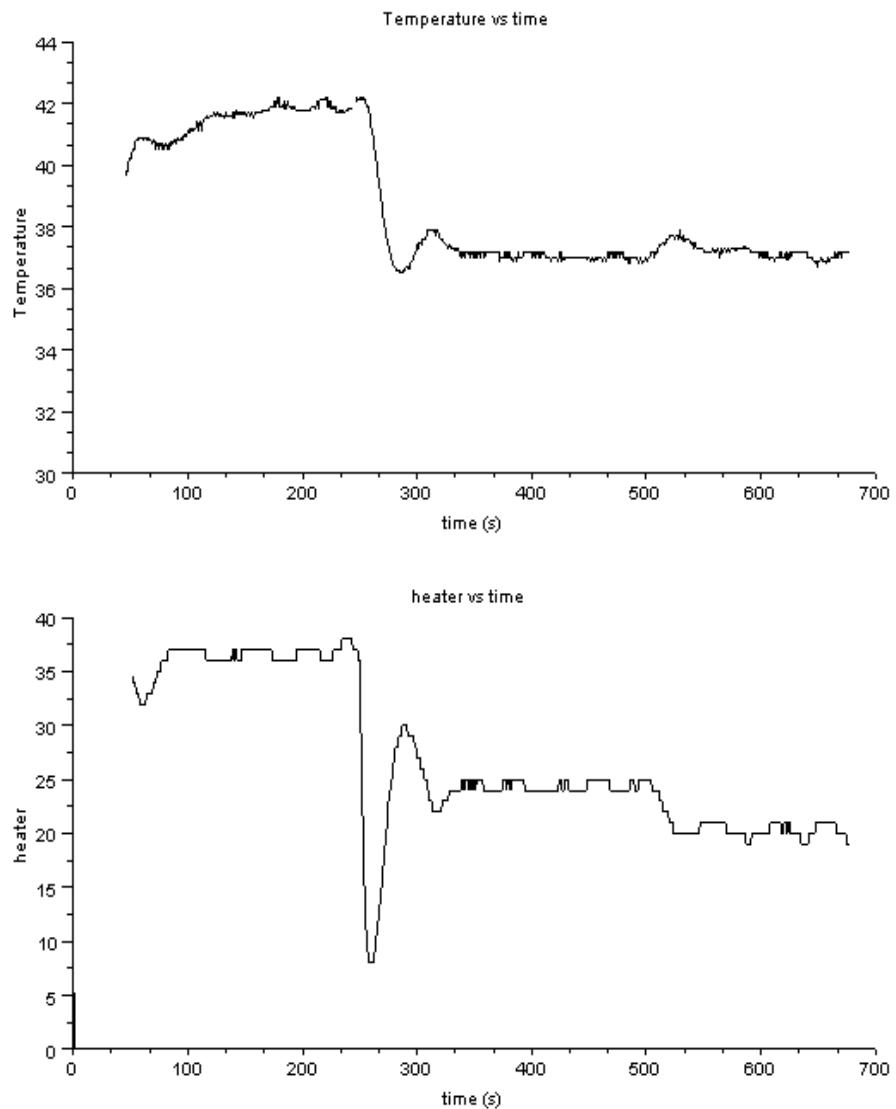


Figure 12.14: Expt 2.1

We can very clearly make out that the results follow the trends as was explained for the negative step input in the section 5.2

12.15.5 Negative Step Change and $(We, Wu)=(10,10)$ (Expt 2.2)

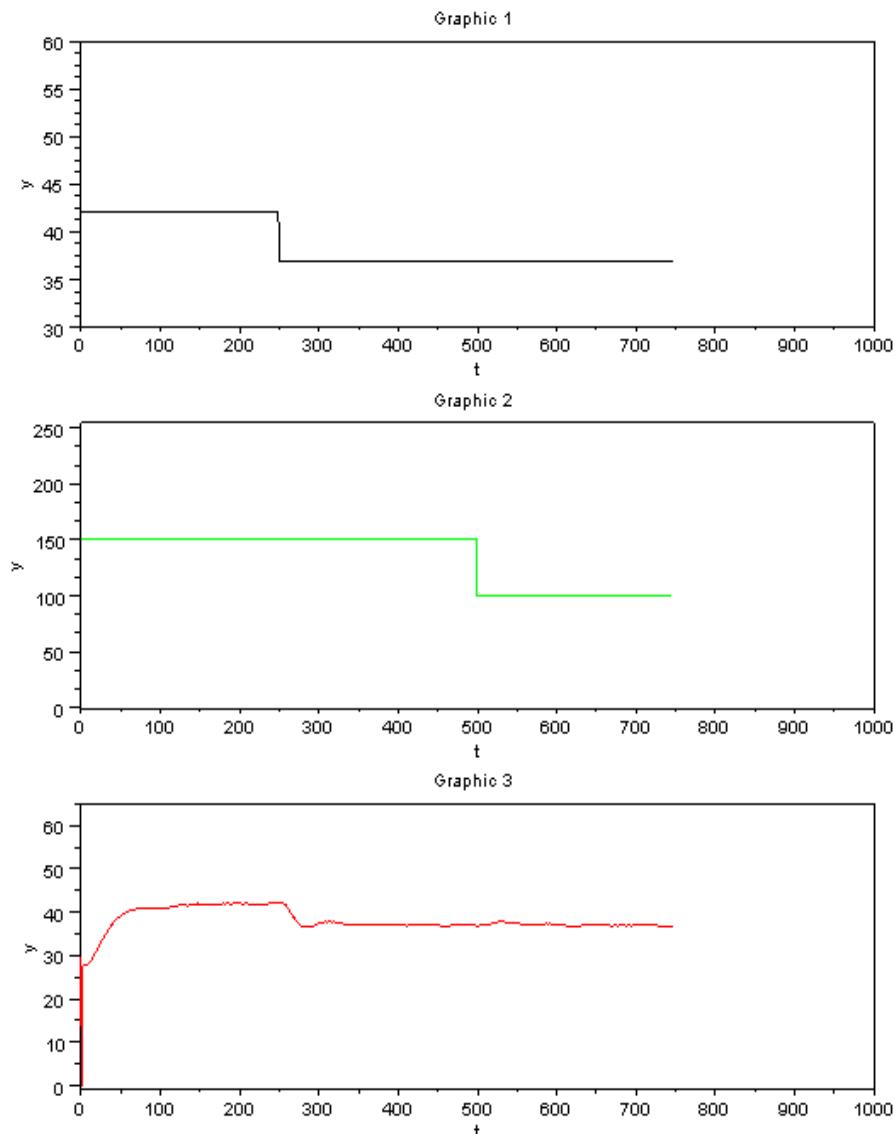


Figure 12.15: Expt 2.2

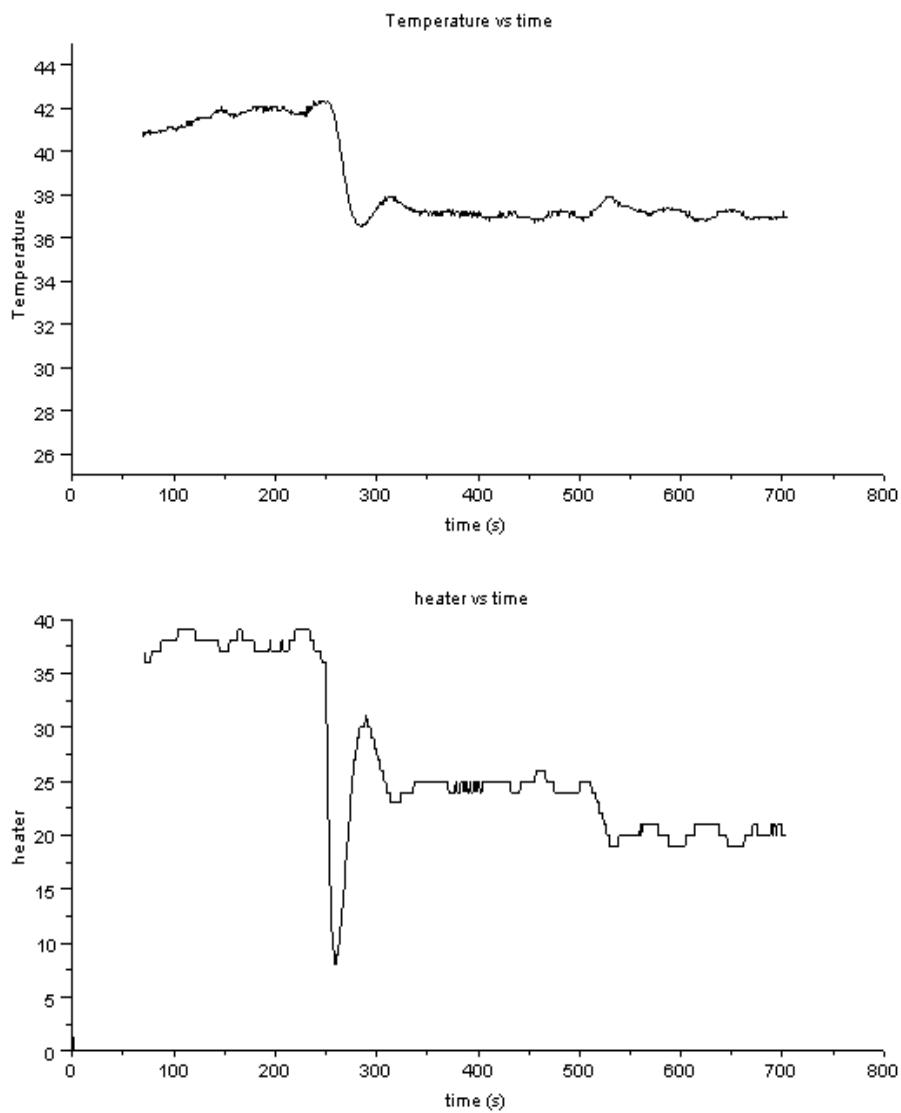


Figure 12.16: Expt 2.2

12.15.6 Negative Step Change and $(W_e, W_u) = (40, 40)$ (Expt 2.3)

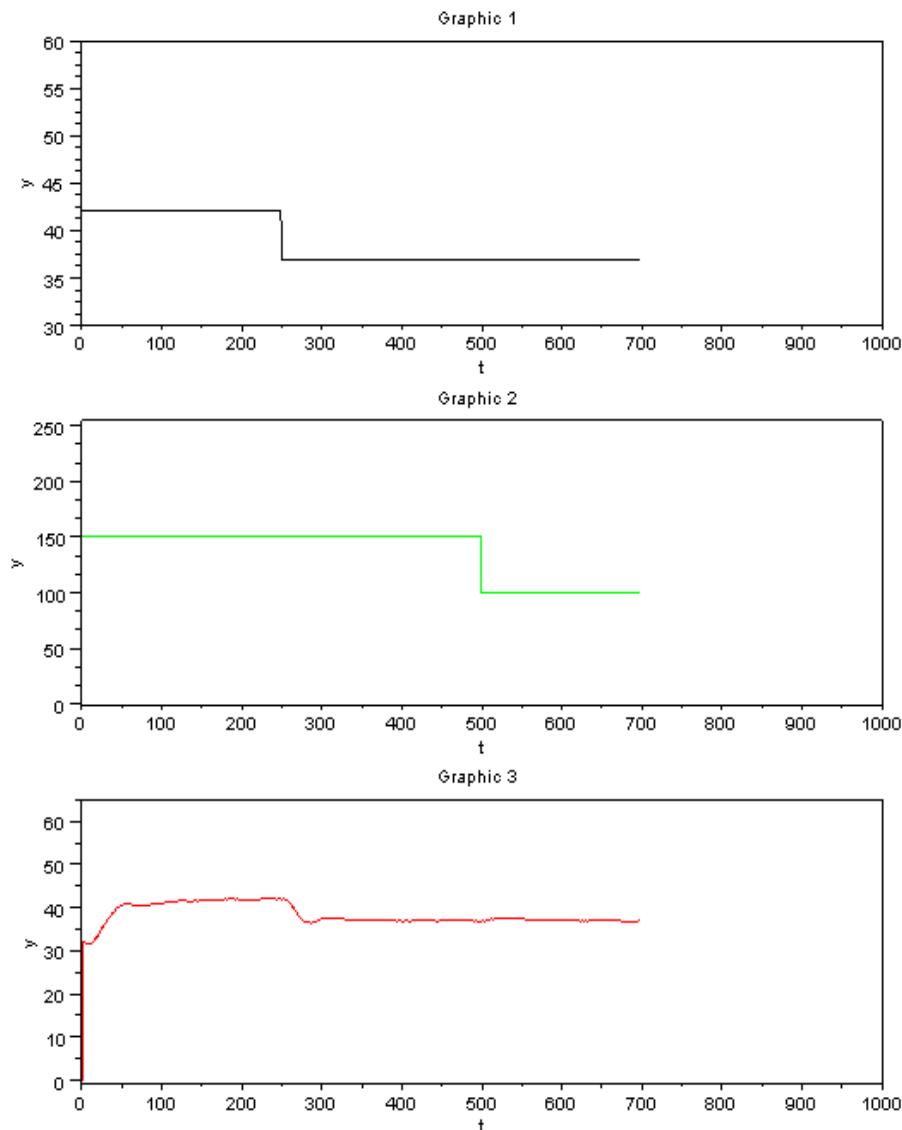


Figure 12.17: Expt 2.3

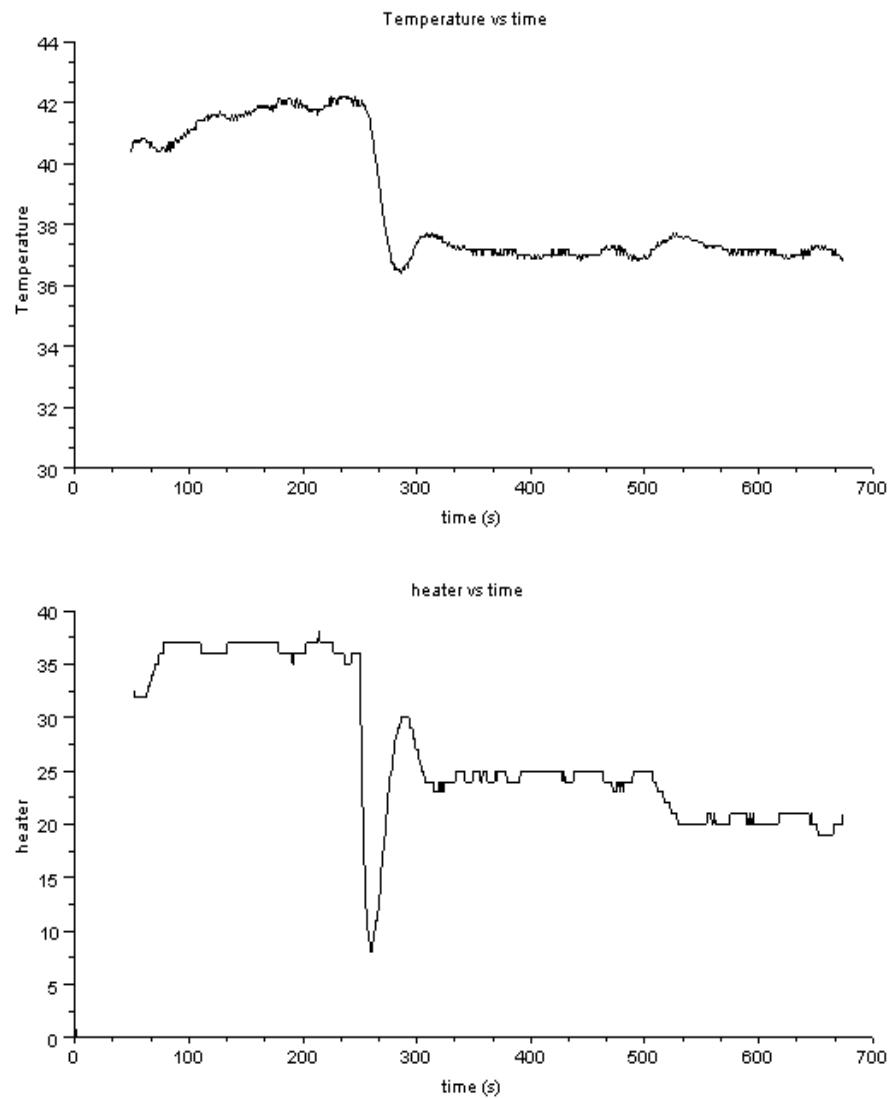


Figure 12.18: Expt 2.3

12.16 For different We and Wu factors

We very clearly see that using the same values of We and Wu is not making much difference in the control response. So, will now be trying different values for We and Wu.

12.16.1 We =100 and Wu = 2 (Expt 5.1)

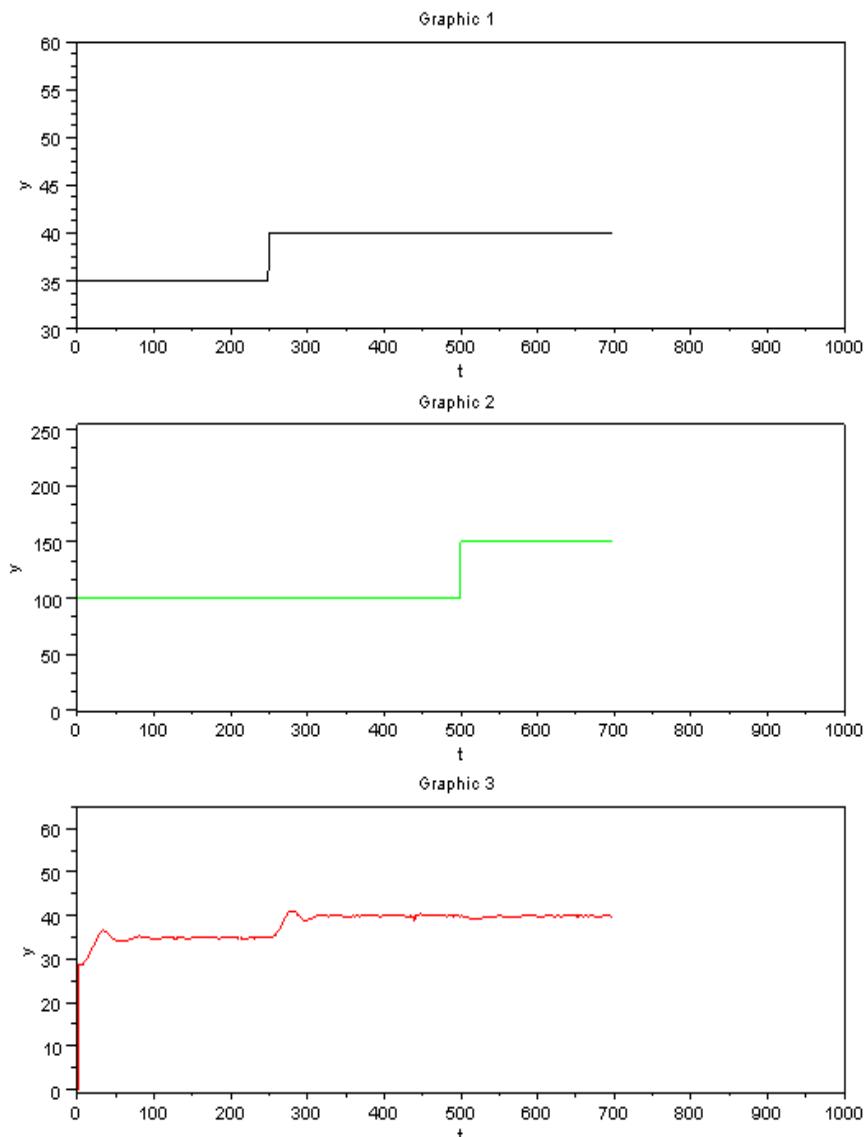


Figure 12.19: Expt 5.1

Here, we have used We as 100 and Wu as 2. The response after the positive step in temperature set point is slightly oscillatory. The temperature very well stabilizes at the required setpoint. The settling time observed is fairly low.

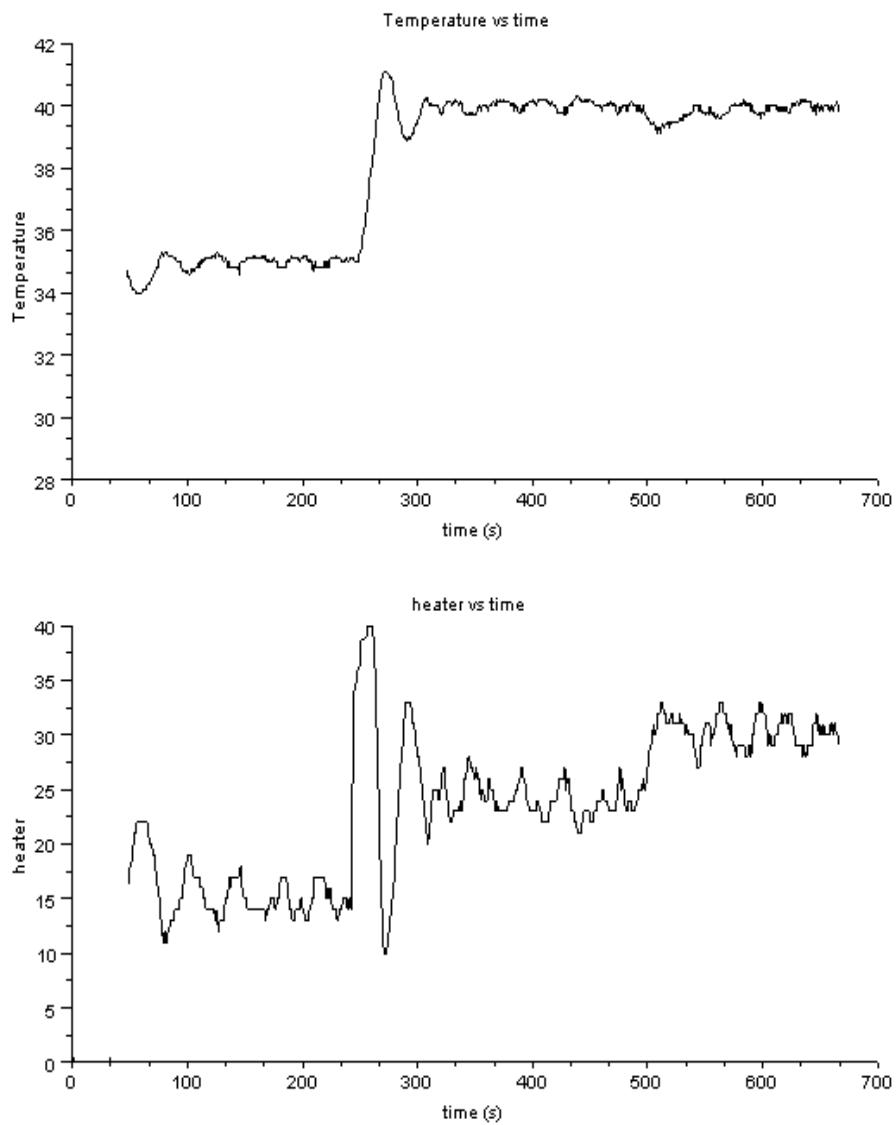


Figure 12.20: Expt 5.1

Now having seen the results of this experiment, we would like to check the possible effect of reversing the values of We and Wu . So, we conduct the next experiment, in which we have We as 2 and Wu as 100.

12.16.2 We =2 and Wu = 100 (Expt 5.2)

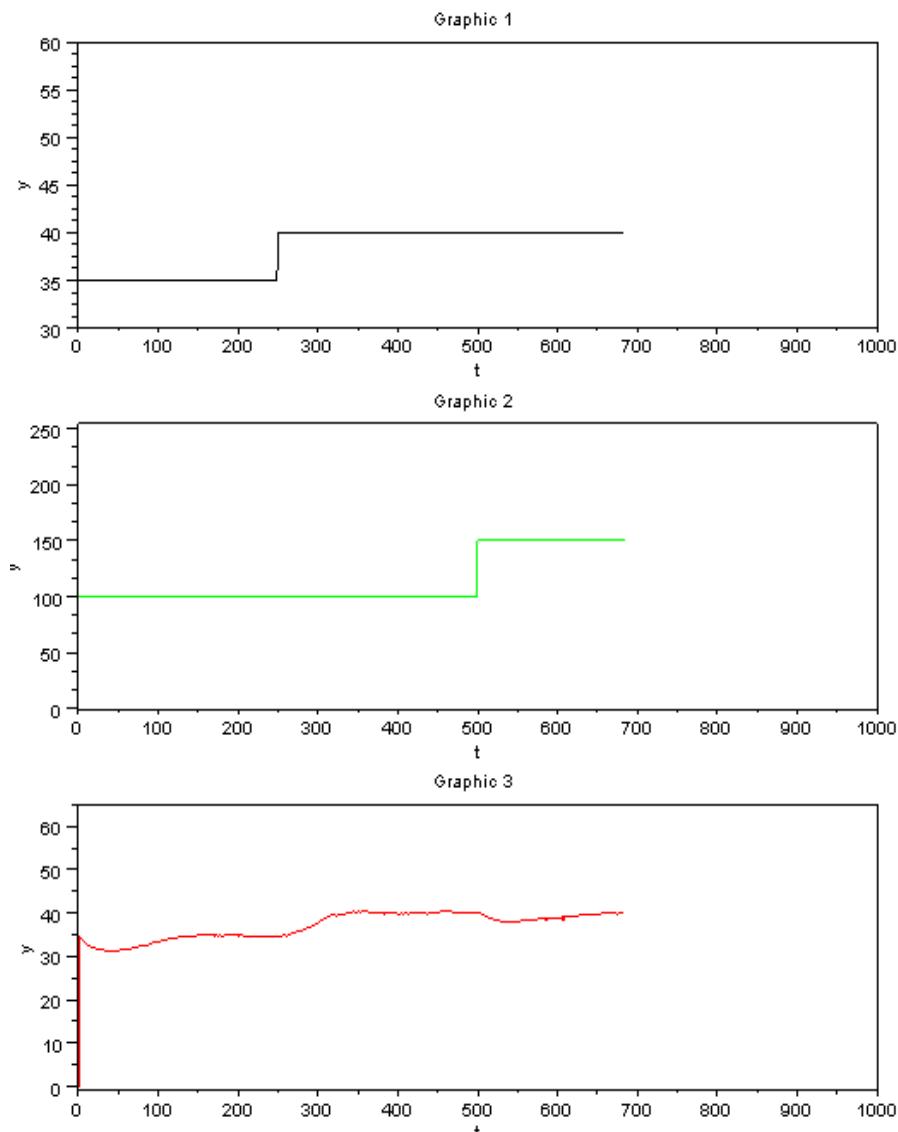


Figure 12.21: Expt 5.2

With increase in W_u , we observe that the temperature stabilizes at the required set-point, but the settling time for reaching that setpoint increases. Also, the response is not oscillatory. This result can be very clearly seen in the following temperature

and heater graph.

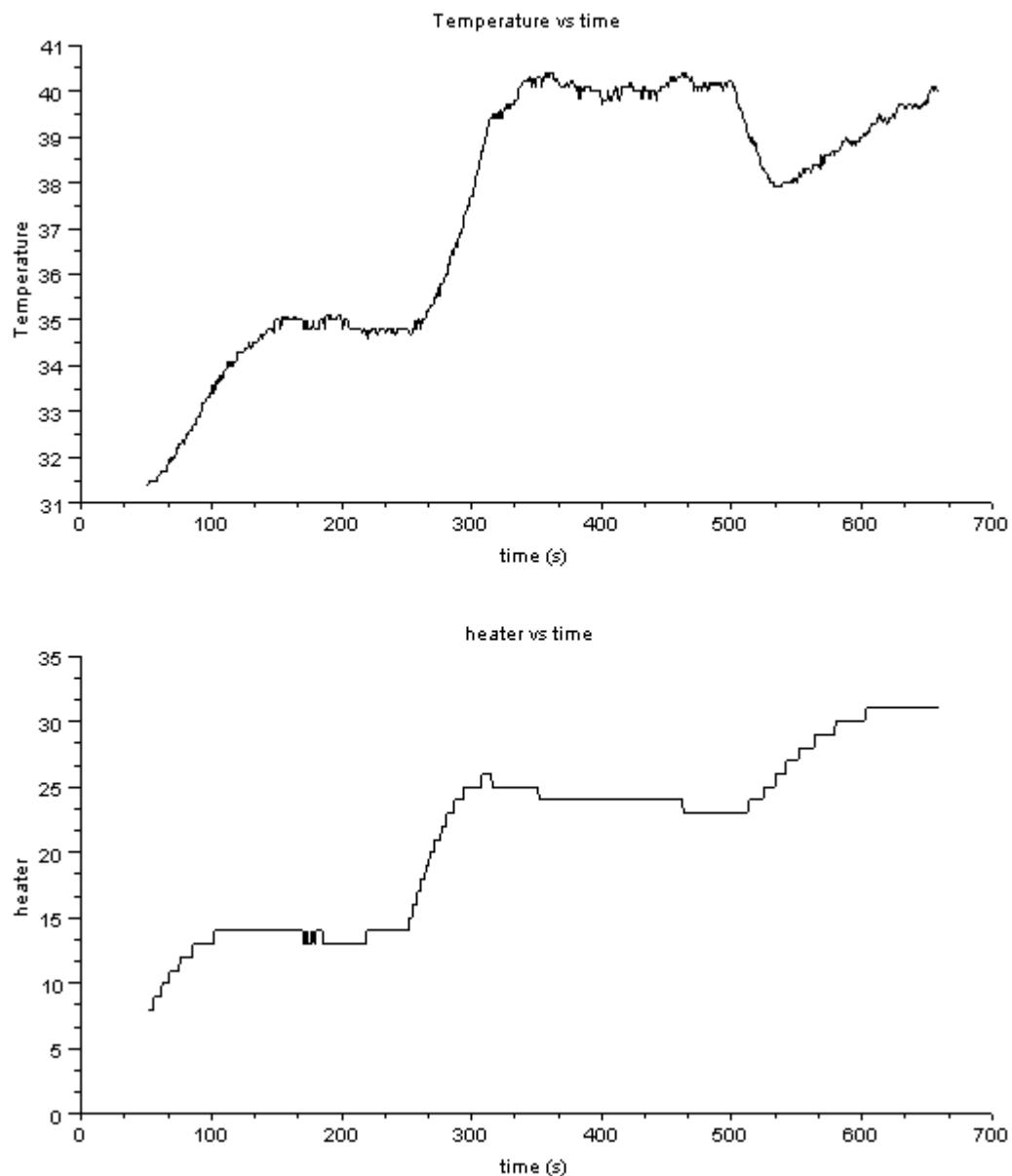


Figure 12.22: Expt 5.2

12.16.3 We =10 and Wu = 100 (Expt 5.3)

Having seen the effect of low We and high Wu (in the last section), we would like to see what happens if We is slightly increased keeping Wu the same. For this we increase the value of We to 10 and keep Wu at constant 100.

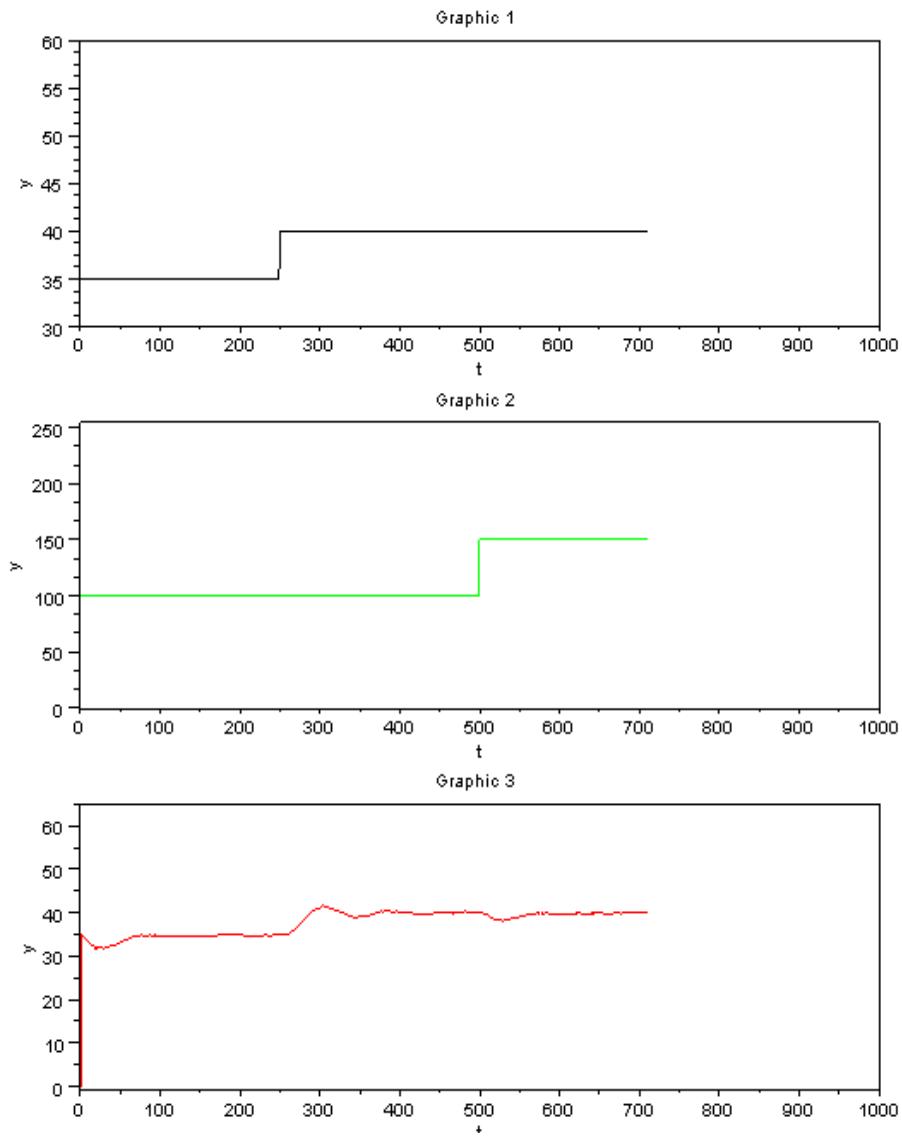


Figure 12.23: Expt 5.3

We observe that this experiments performs better than in the last section (where We was 2). It is slightly oscillatory and also, the settling time decreased much as compared to last experiment.

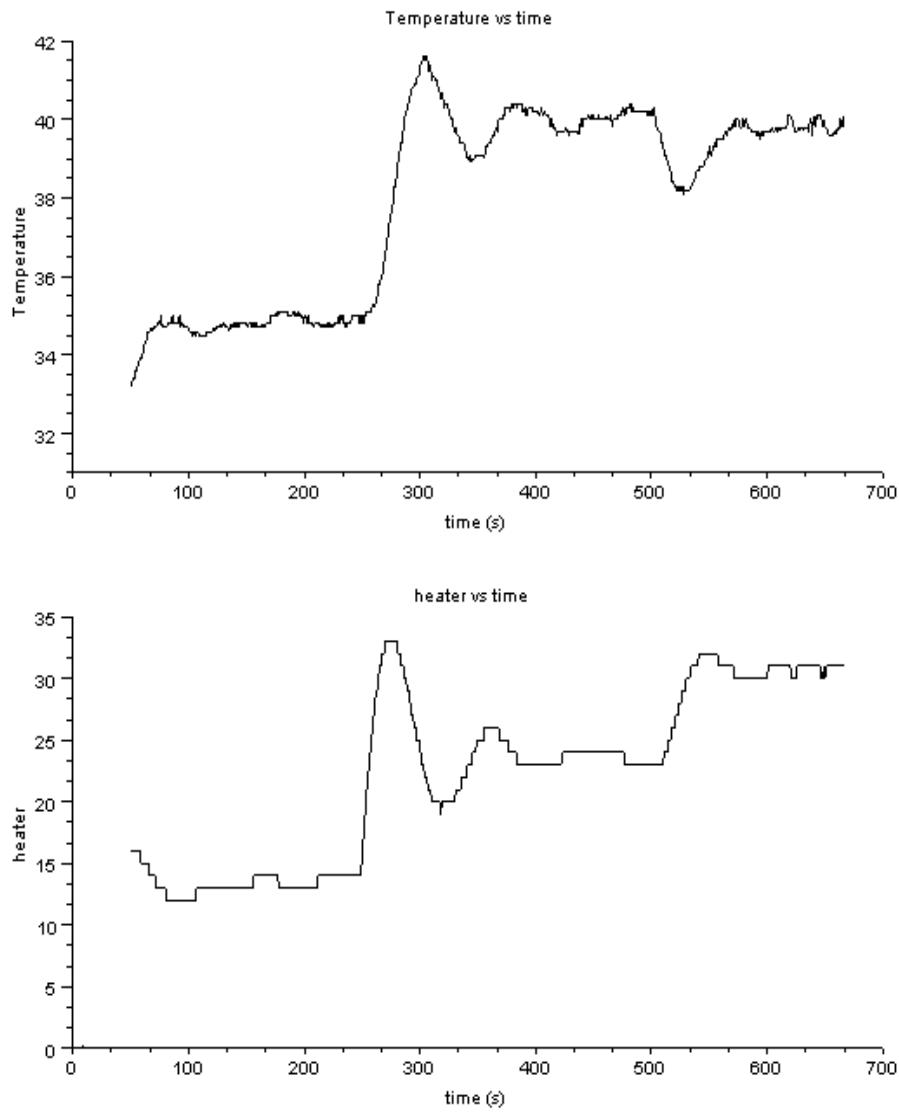


Figure 12.24: Expt 5.3

12.16.4 We =100 and Wu = 10 (Expt 5.4)

We now do a similar study for the case of Wu. We increase the value of Wu to 10, keeping We constant at 100.

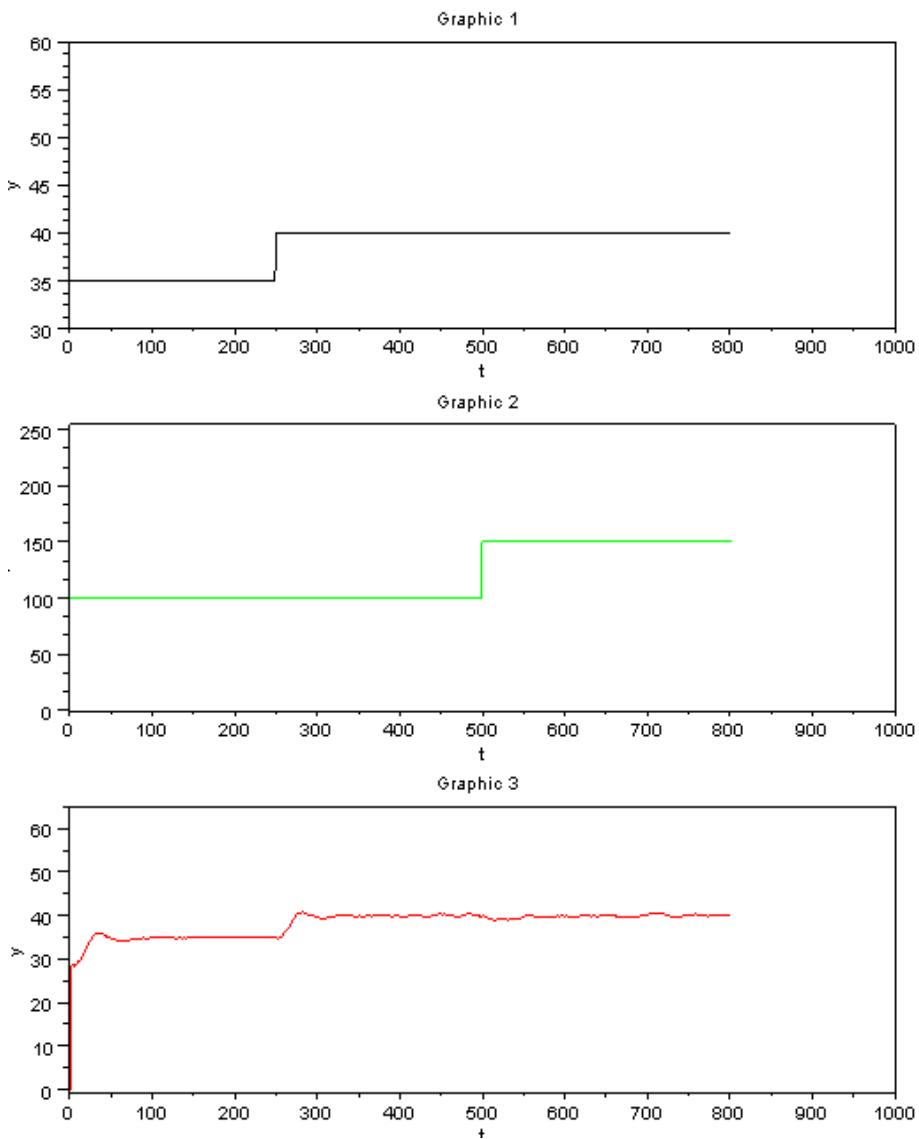


Figure 12.25: Expt 5.4

As is clear from the figure, we see slightly lesser oscillations compared to the

case when We was 100 and Wu was 2. Settling time more or less remained the same.

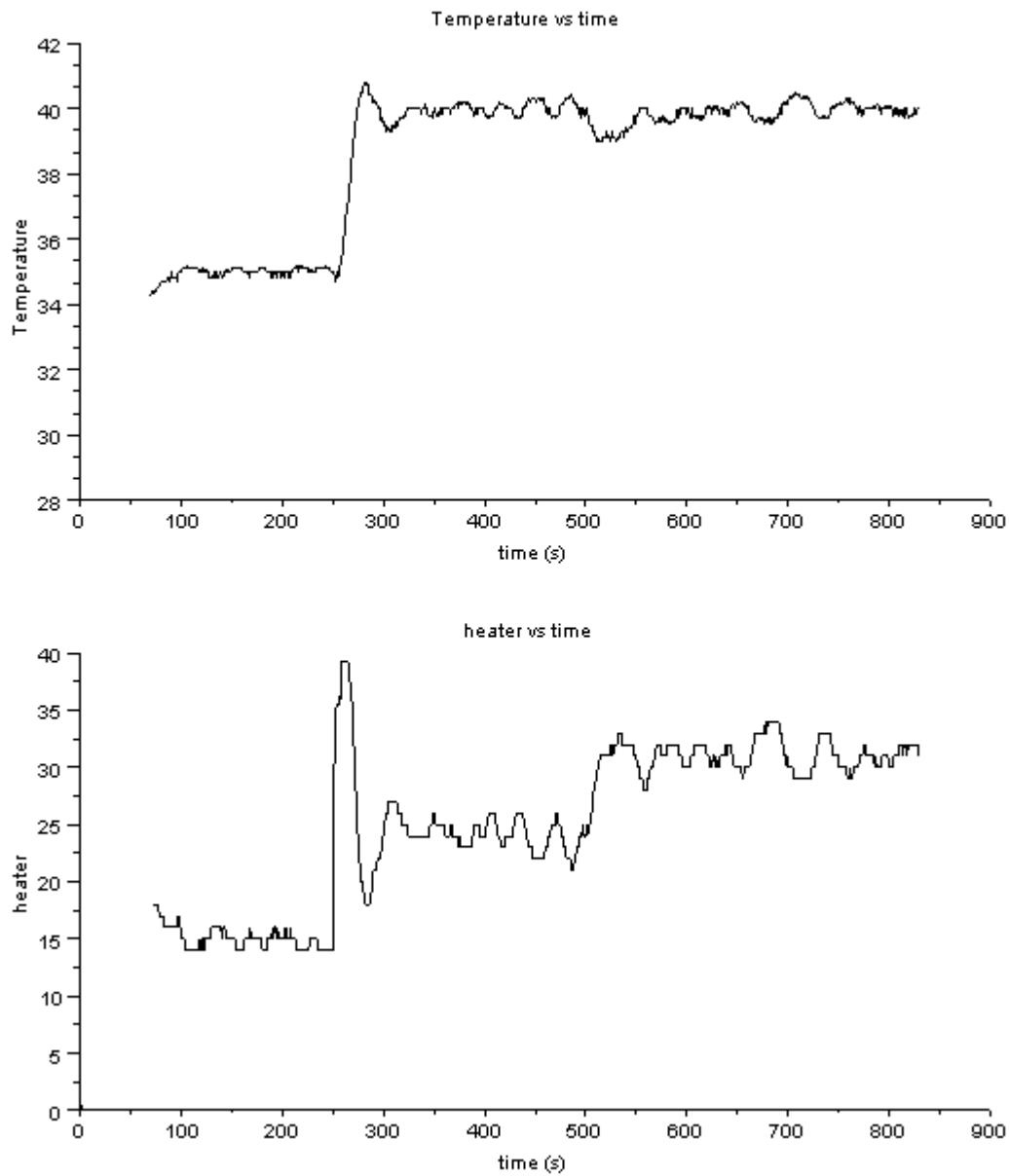


Figure 12.26: Expt 5.4

12.17 Conclusion on Weighting factor experiments

For experiments with same values of We and Wu:

- Not much difference was seen in heater value trends and temperature value trends for all the experiments performed above.
- Reason for this will be clear from the discussion on the trends mentioned below (for experiments with different values of We and Wu)

For experiments with different values of We and Wu:

- Keeping We as large (around 100) and Wu as small (2) shows better performance as compared to the case when the values are kept the other way around.
- With We very small (say around 1-2), oscillations are less, and the settling time observed was found to be more.
- With increase in We, the oscillations were observed to increase and the settling time was found to reduce and hence, better control was observed.
- So, with increase in We, any error is quickly dealt with, because with increase in We, we are actually increasing the significance of change of temperature in deciding the control action.
- With increase in Wu, oscillations reduced and the settling time was found to increase and hence, less preferred.

So, the best performance is obtained for the cases with high We and low Wu.

12.18 Effect of Control Horizon Paramter, q

We also tried to study the effect of change of control horizon (q) on the response of the SBHS to step change in Setpoint and disturbance variable. Generally the value of q (control parameter) is taken somewhere between 2 to 5. So, we performed our SBHS experiment for values of q as 2, 3 and 4 (as suggested by Mr Prashant Gupta).

Both positive and negative step change experiments for temperature set point and

disturbance variable (fan) was performed for the sake of completeness. The results obtained thereby has been mentioned in form of graphs in this section. The overall conclusion over these experiments has been mentioned in the conclusion of this part.

12.19 For positive step change in Set point and Fan speed

12.19.1 For q =2 (Expt 3.1)

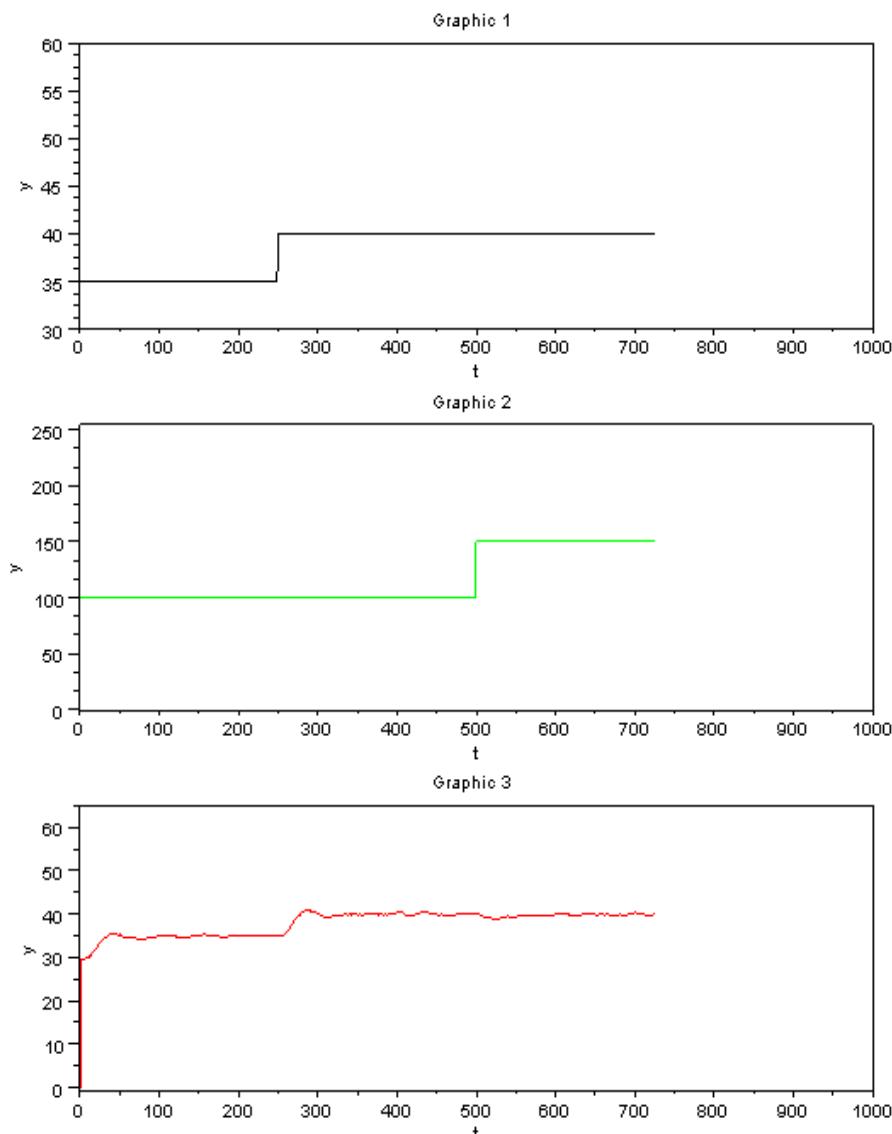


Figure 12.27: Expt 3.1

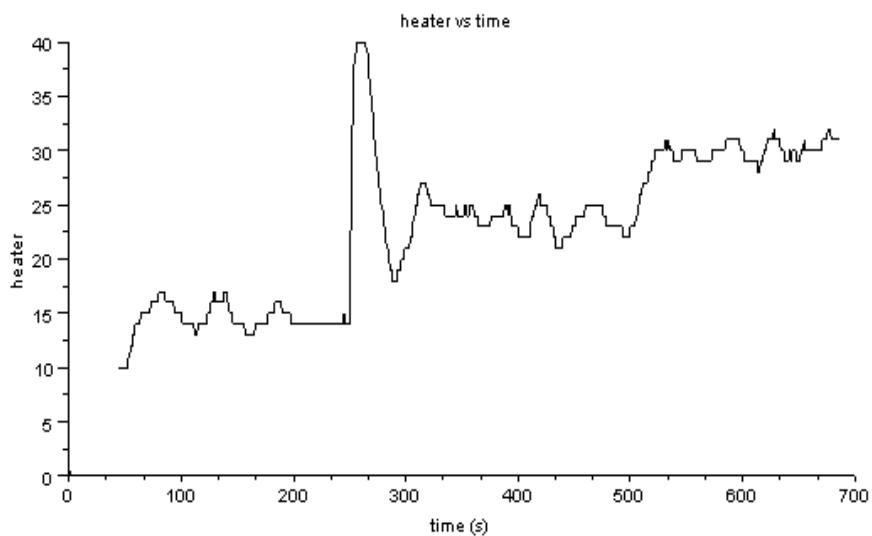
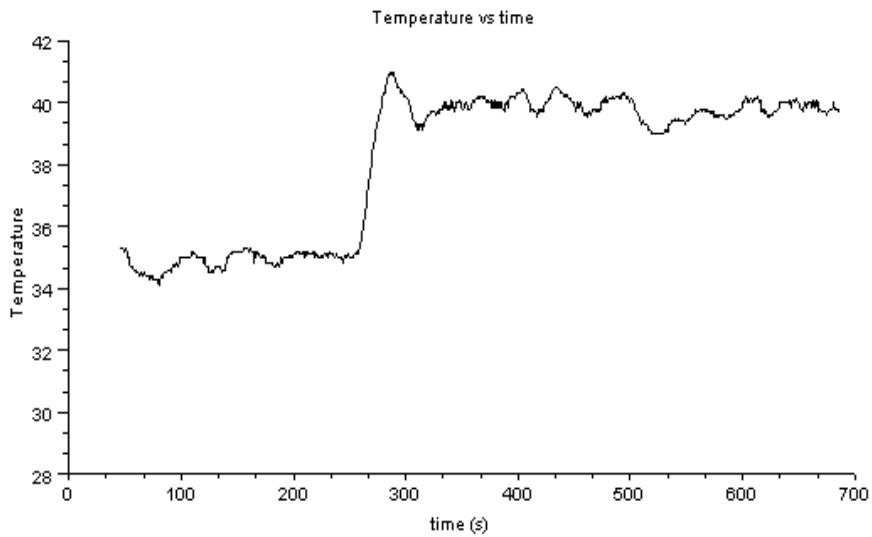


Figure 12.28: Expt 3.1

12.19.2 For $q = 3$ (Expt 3.2)

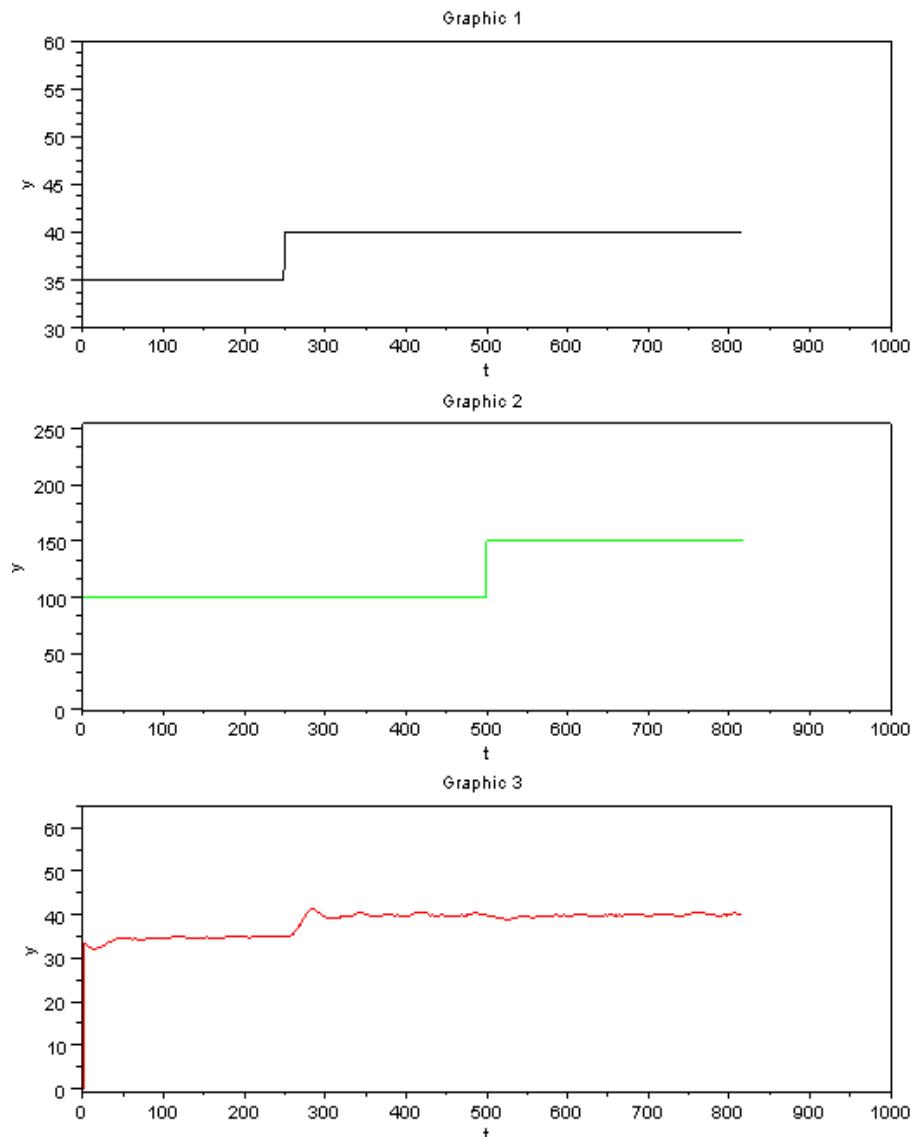


Figure 12.29: Expt 3.2

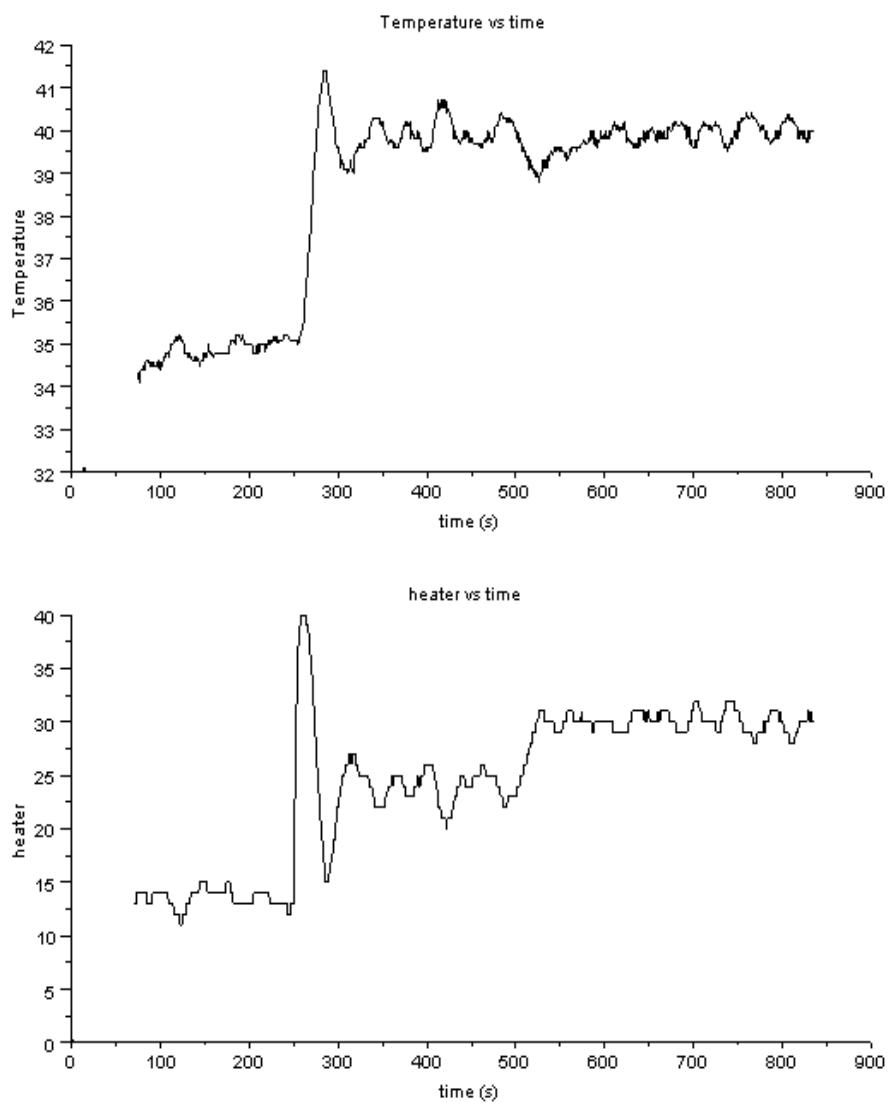


Figure 12.30: Expt 3.2

12.19.3 For $q = 4$ (Expt 3.3)

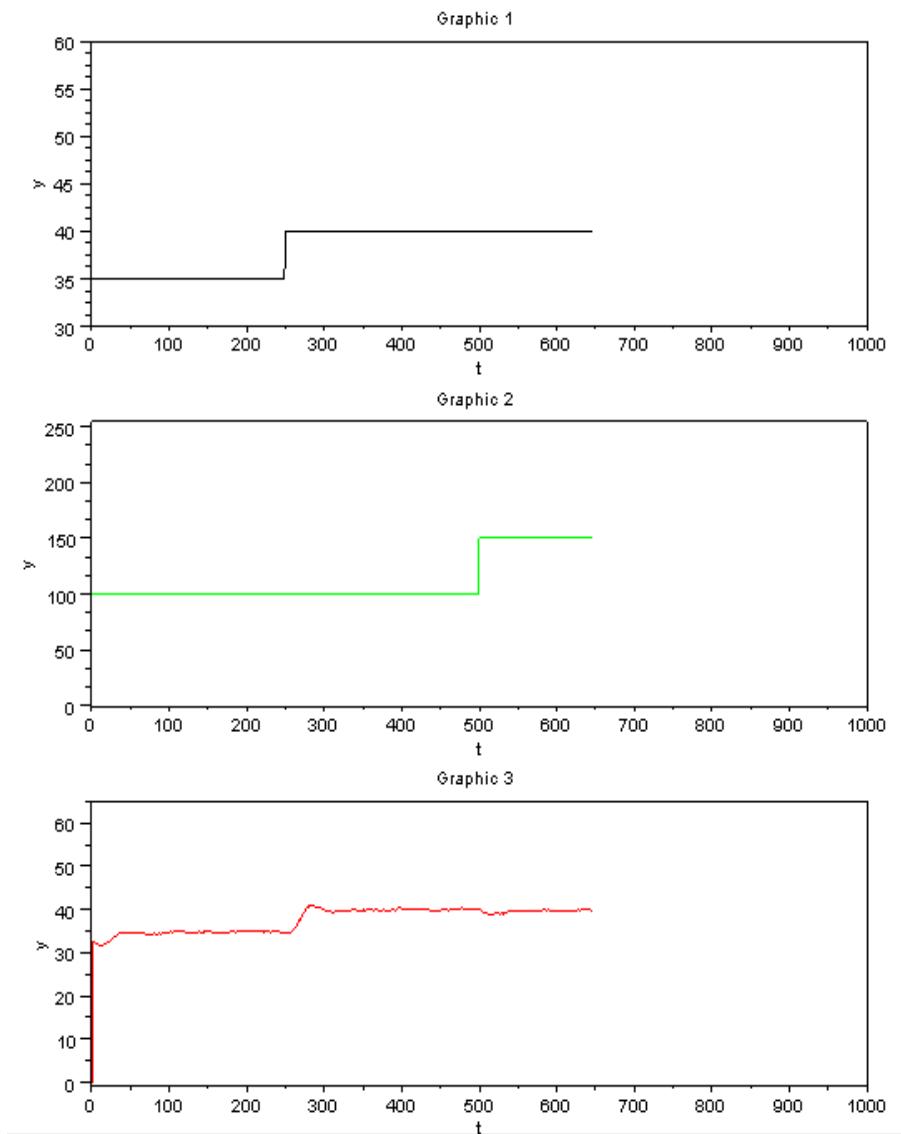


Figure 12.31: Expt 3.3

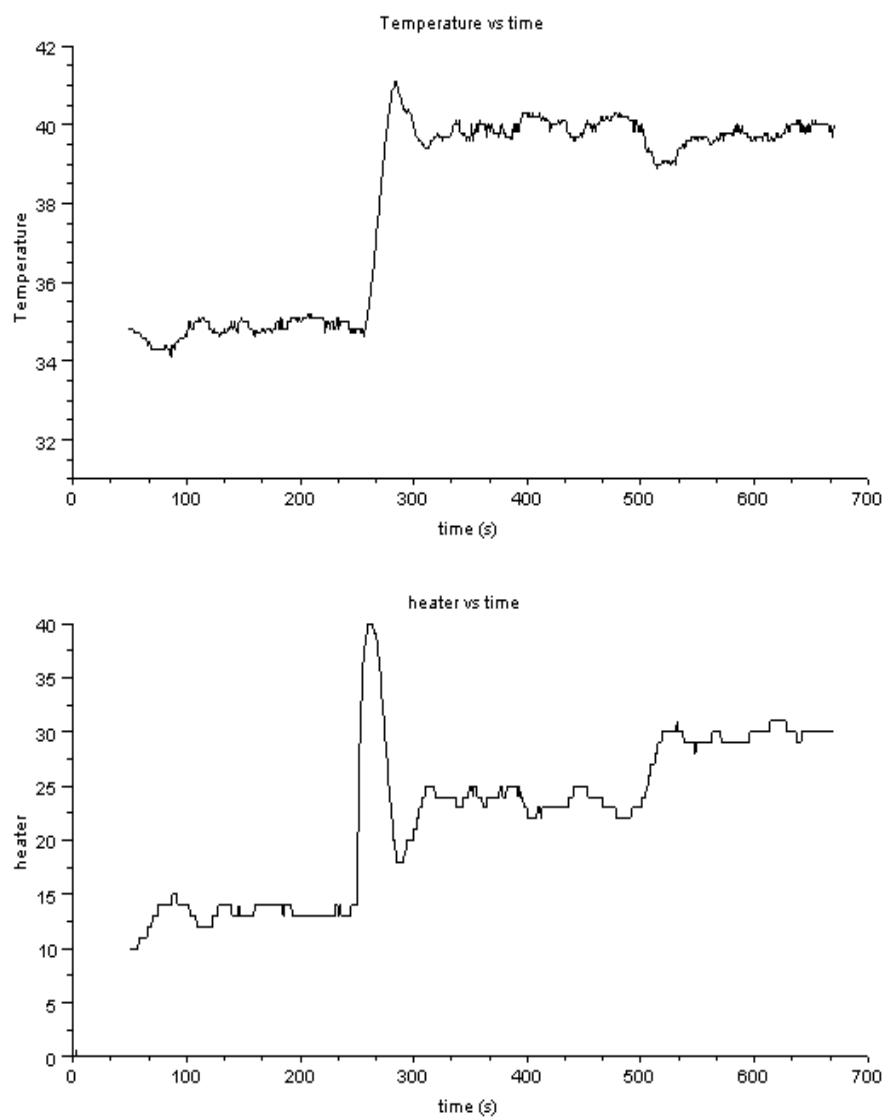


Figure 12.32: Expt 3.3

12.20 For negative step change in Set point and Fan speed

12.20.1 For q =2 (Expt 4.1)

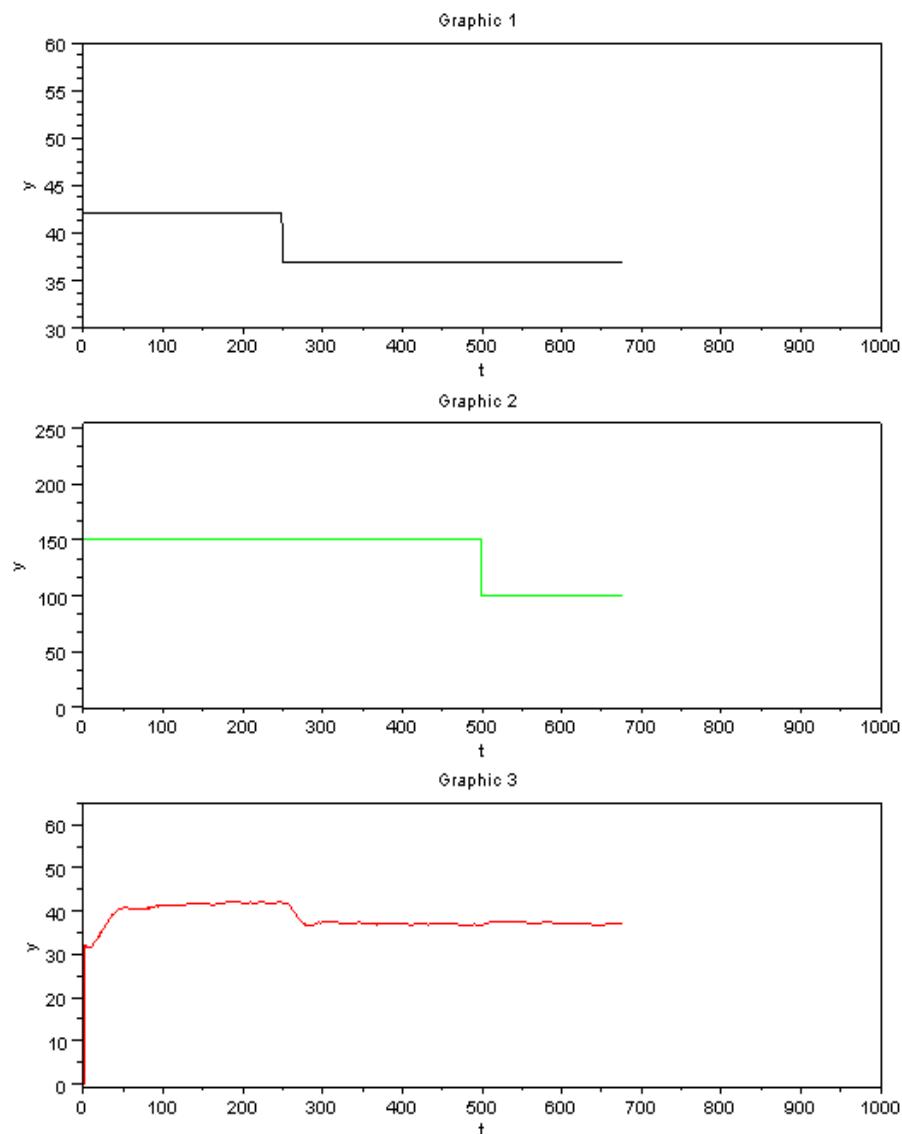


Figure 12.33: Expt 4.1

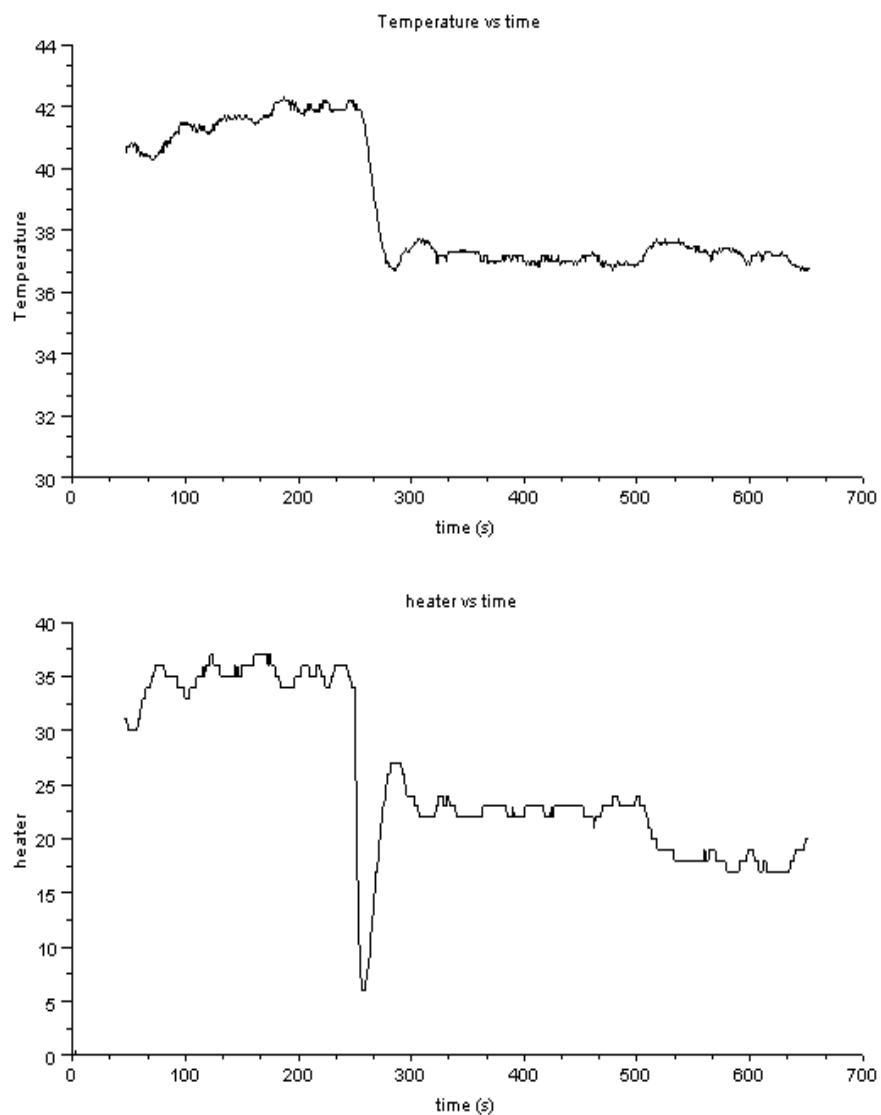


Figure 12.34: Expt 4.1

12.20.2 For $q = 3$ (Expt 4.2)

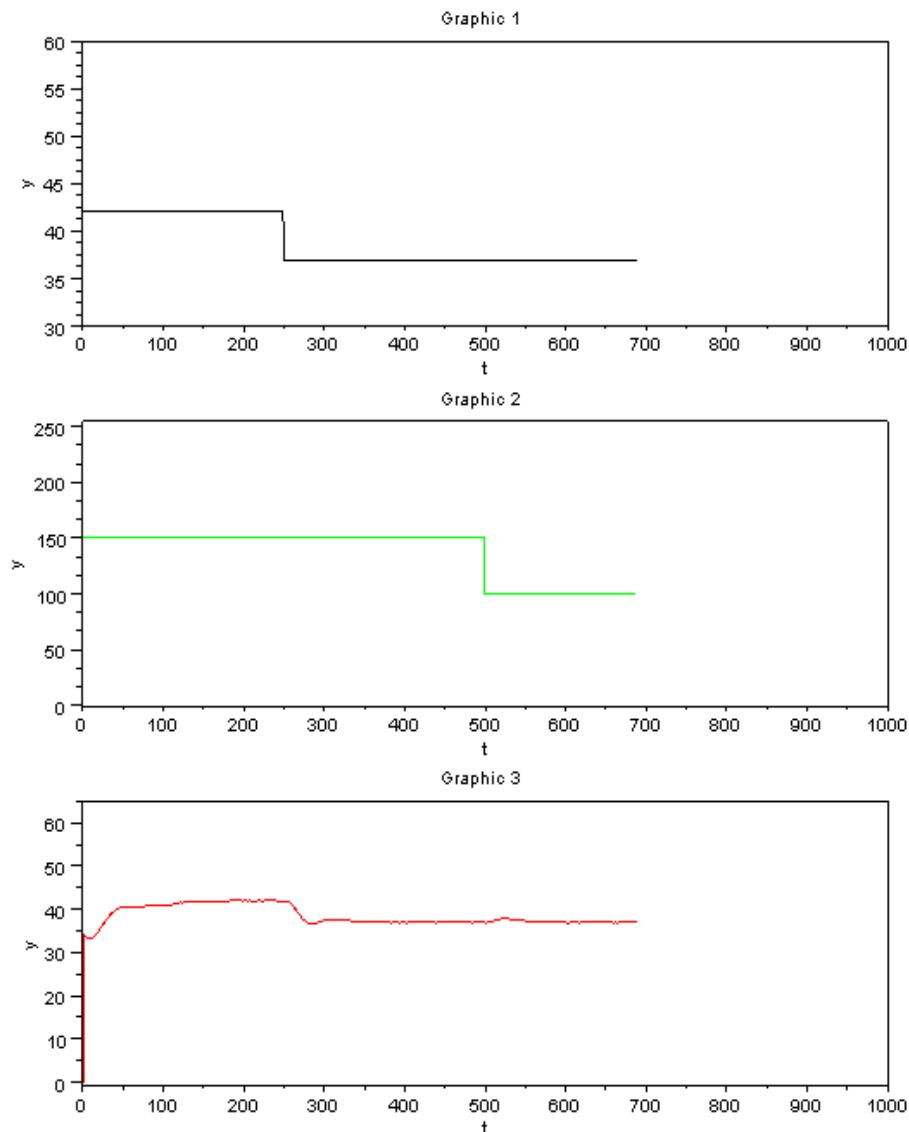


Figure 12.35: Expt 4.2

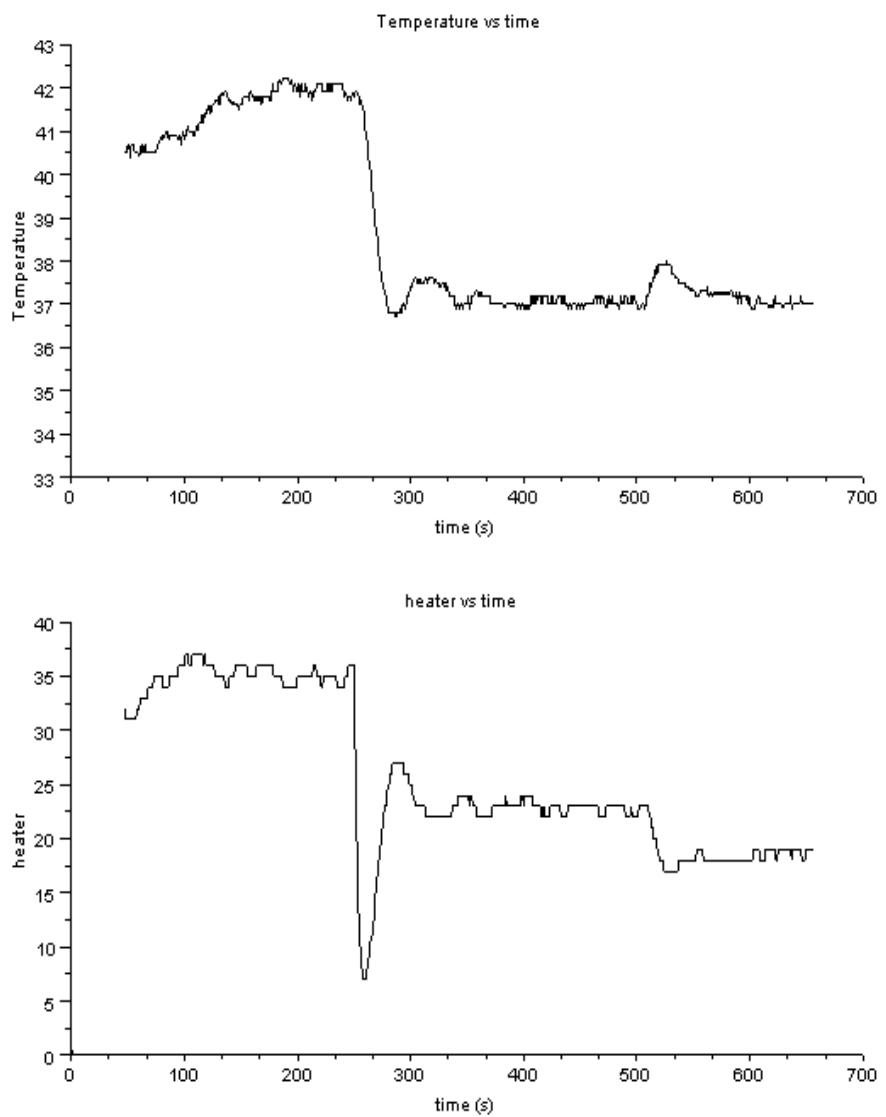


Figure 12.36: Expt 4.2

12.20.3 For q = 4 (Expt 4.3)

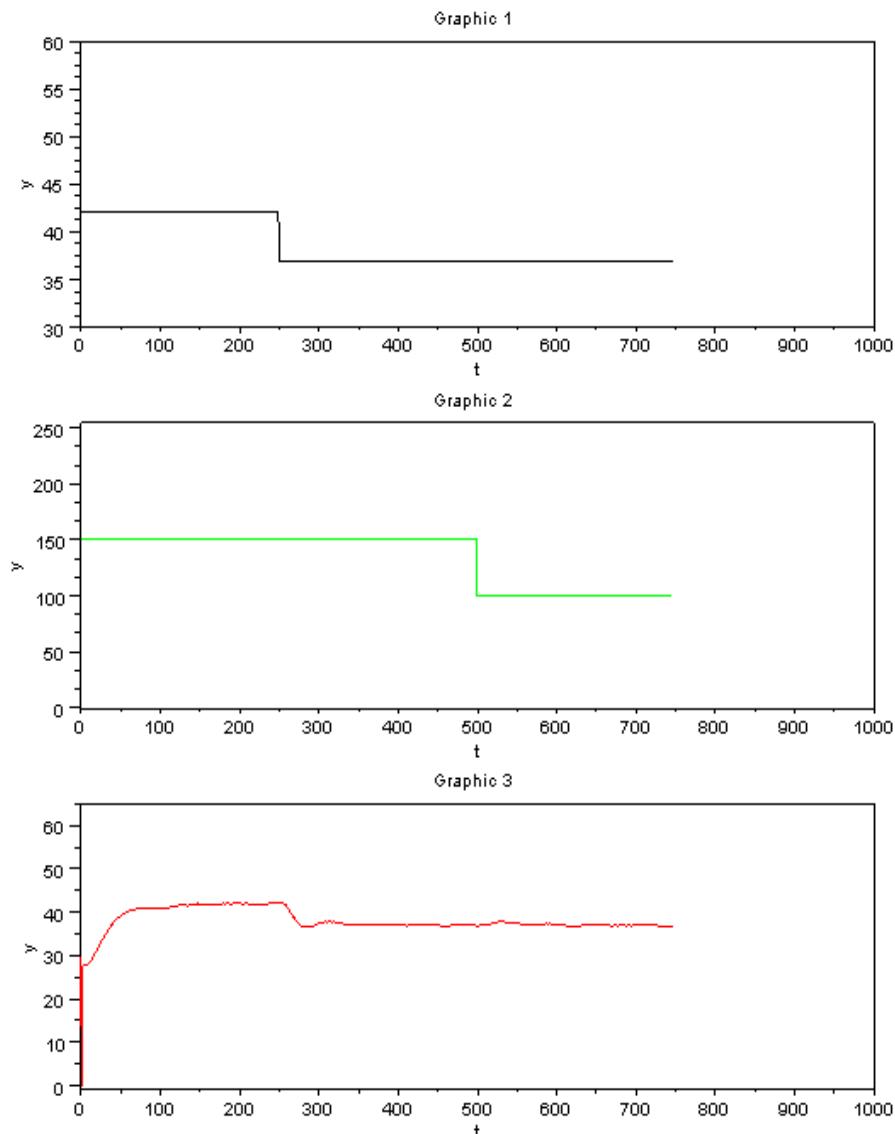


Figure 12.37: Expt 4.3

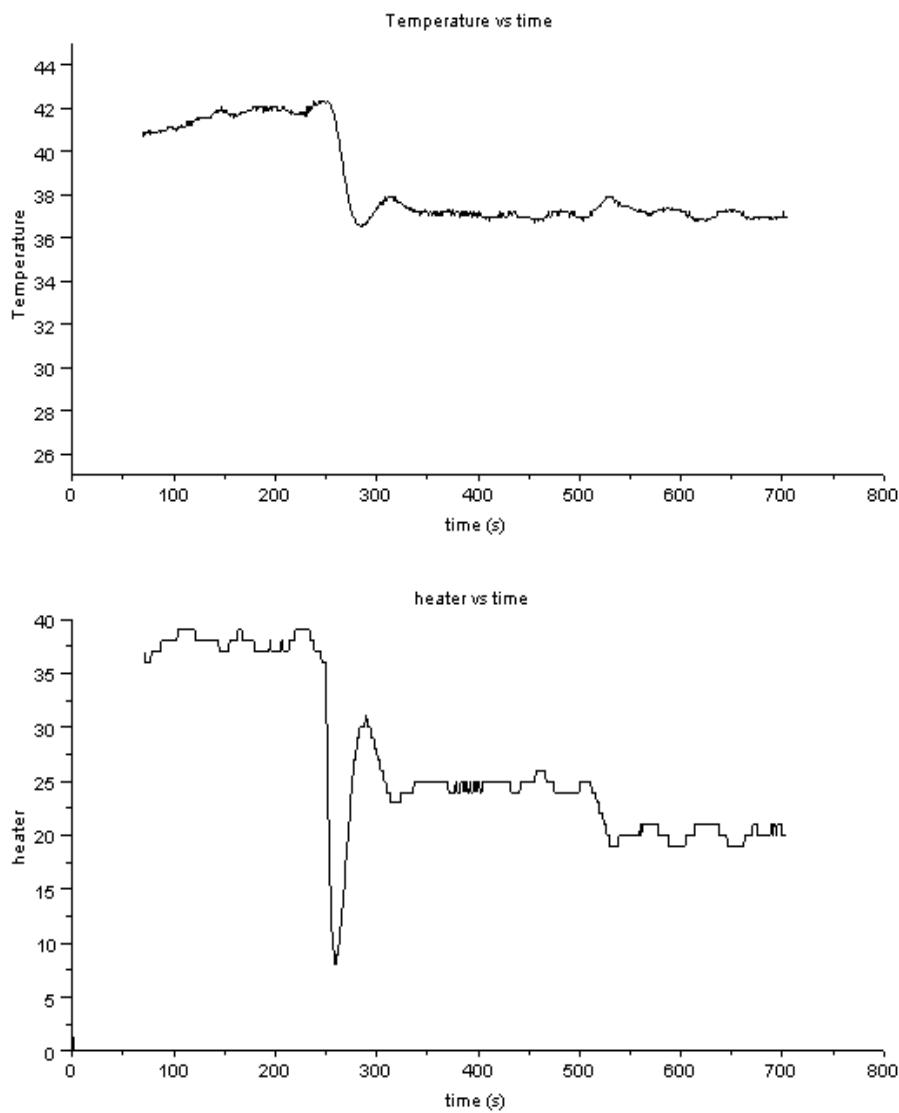


Figure 12.38: Expt 4.3

12.21 Conclusion on the effect of Control Horizon parameter

- The effect of change in q isn't very distinct in the experiments performed.

- While we are calculating the optimized value of manipulated variable at a time, the number of manipulated input moves is increasing as we are increasing the q value.
- But, only the first value of the optimized manipulated variable vector is used for control.
- Increase in q is only increasing the length of the manipulated variable vector which is to be optimized.
- Since, only the first value of manipulated variable vector is used, which itself lies in some specified range, the effect of changing q isn't very significant for SBHS.
- Also, SBHS system is a simple system with very few variables (as compared to real life industrial systems).
- Ideally, the value of q is to be maintained at 3 or 4.

12.22 Implementing Model Predictive controller on SBHS, locally

Change the working directory of scilab to the folder where the local mpc code are kept. Run the `ser_init.sce` file with the appropriate com port number. Then execute the file `mpc_init`. Launch xcos and execute the file `mpc.xcos`. The code is listed in Section 12.29

12.23 Conclusion for MPC project

The objective of this project, ie, implementing Model Predictive Control in Single Board Heater System using Scilab was successfully achieved. Several experiments were successfully performed using the developed SCILAB MPC algorithm for both positive and negative step changes in both temperature-set-point and the disturbance variable (fan).

In addition to the above objective, we also tried studying the effect of weighting factors (tuning parameter) and control horizon parameter. We observed and

concluded that increase in values of We (error weighting factor), increases oscillations and decreases settling time, while decrease in We leads to opposite effect. Wu (manipulated variable weighting factor), on the other hand has an opposite effect. It decreases oscillations and increases settling time with increase in its value. Hence, better control is obtained for high value of We and low value of Wu .

Thus, with this project, we were able to implement MPC successfully and also were able to comment on the general preferred tuning parameters (weighting factors for error and manipulated variable).

12.24 Acknowledgement

Firstly, I would like to thank Prof Moudgalya Kannan, for giving me this opportunity to undertake MPC project on SBHS. This project, which involved implementing Model Predictive Control in SBHS using SCILAB, was very interesting and provided an excellent learning opportunity. For developing the MPC algorithm, lecture notes on Model Predictive Control by Prof Sachin Patwardhan too were extremely helpful. Also, I got to learn a lot from the speaking tutorials of SCILAB and LaTeX, which had to be referred to for the completion of this project. Over and above this, it was very encouraging to see the experiments working perfectly with the developed Model Predictive Control algorithm.

I would also like to sincerely thank Mr Prashant Gupta, without whom, this project would not have been splendidly completed. I would like to thank him for the time he spent explaining the concepts, clearing the doubts and suggestions for the experiments to implement MPC.

12.25 Appendix

12.26 Appendix 1: General Information on Experiments for this Project

All the experiments for this project was performed remotely on SBHS 12, using a sampling time of 1 second. Basic codes (mpc_init.sce and mpc.sci) was taken from moodle for this course. Code for implementing MPC was written in scilab and has been mentioned in the report.

Scilab Version used: 5.2.2

SBHS number: 12 (remotely used)

Sampling time: 1 second

For graphs: Until and unless mentioned, Graphic 1 represents the Temperature set point, Graphic 2 represents the Fan and Graphic 3 represents the Temperature.

12.27 Appendix 2: Values of State Space matrices

Initially, open loop experiment was performed, and Plant Transfer function was obtained. For the open loop experiment, a step change in heater from 15 to 25 units at $t = 200$ seconds was provided (sampling time 1s). The response data was fitted to a first order transfer function with a time delay and the following was observed:

$K_p=0.37$, time constant = 45s and delay = 7s.

Using the above, we obtained the plant transfer function:

$$G_p = \frac{0.37}{1 + 45s} e^{-7s} \quad (12.7)$$

Scilab Method to calculate State Space matrices

State space matrices for a transfer function can be calculated as follows using Scilab:

```

1 s=poly(0,'s');
2 TFcont=syslin('c',[kp*(1-0.5*D)/(tau*s+1)/(1+0.5*D)]);
3 SScont=tf2ss(TFcont);

```

SScont (in the last line above), has the value of the required State Space matrices.
 (Please note: Time delays can not be directly handled in Scilab. So, for systems with delays, we will have to use alternate approach. Pade's approximation for time delay being one of the approach.)

The transfer function which we derived for our SBHS was very close to the transfer function derived by Mr Prashant Gupta. So, using the values of A, B and C which were already calculated by him previously, we obtain the following exact values:

$$A = \begin{bmatrix} 0.9780 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0079 \end{bmatrix}$$

12.28 Appendix 3: Attachments and Contact Information

Attachments

- Folder named *codes*, which contains all the codes used for the experiments. The codes in this folder should be used to reproduce MPC control experiments mentioned in this report.
- Folder named *data files*, which contains data files for all the experiments performed
- MPC Report

Contact Information

My details:

Pratik Behera (07002054)
pratik_behera@iitb.ac.in
+91 99871 54061

12.29 Scilab Code

Scilab Code 12.1 mpc.sci

```
1 function [ stop ] = mpc(Tsp , fan )
2     global fdfh fdt fncl fnclw m err_count stop p q
3         xk_old heat temp heat_old
4
5         heat = mpc_run(heat , heat_old , Tsp );
6
7         [ stop , temp ] = comm(heat , fan ); // Never edit this
8             line
9             plotting([heat fan temp Tsp],[0 40 25 0],[100 70
10                 50 1000]);
11
12         heat_old = heat ;
13
14     return
15 endfunction
```

Scilab Code 12.2 mpc_init_local.sce

```
1 // For scilab 5.1.1 or lower version users ,
2 // use scicos command to open scicos diagrams instead
3 // of xcos
4 global err_count y p q xk_old Tsp heats fan temp heat
5
6 p = 40; // prediction horizon
7 q = 4; // control horizon
8 xk_old = zeros(8,1);
9 Tsp=1;
10 heats=1;
11 fan=1;
12 temp=1;
13
14 exec ("mpc_local.sci");
15 exec ("mpc_run.sci");
```

Scilab Code 12.3 mpc_local.sci

```
1 mode(0)
2 function [temp] = mpc(Tsp,fan)
3 global temp heat_in fan_in C0 u_old u_new e_old e_new
e_old_old
4
5 global heatdisp fandisp tempdisp setpointdisp
sampling_time m name
6 // heats = 1;
7 u_new = mpc_run(temp,heats,Tsp);
8
9
10 heat = u_new;
11
12 temp = comm(heat,fan);
```

```
14     plotting([heat fan temp Tsp],[0 0 20 0],[100 100  
15             40 1000])  
16     m=m+1;  
17 endfunction
```

Bibliography

- [1] Fossee moodle. <http://www.fossee.in/moodle/>. Seen on 10 May 2011.
- [2] Spoken tutorials. http://spoken-tutorial.org/Study_Plans_Scilab. Seen on 10 May 2011.
- [3] K. M. Moudgalya. Introducing National Mission on Education through ICT. <http://www.spoken-tutorial.org/NMEICT-Intro>, 2010.
- [4] K. M. Moudgalya and Inderpreet Arora. A Virtual Laboratory for Distance Education. In *Proceedings of 2nd Int. conf. on Technology for Education, T4E*, IIT Bombay, India, 1–3 July 2010. IEEE.
- [5] Kannan M. Moudgalya. *Digital Control*. John Wiley and Sons, 2009.
- [6] Kannan M. Moudgalya. *Identification of transfer function of a single board heater system through step response experiments*. 2009.
- [7] Katsuhiko Ogata. *Modern Control Engineering*. Prentice-Hall of India, 2005.
- [8] Dale E. Seborg, Thomas F. Edgar, and Duncan A. Mellichamp. *Process Dynamics and Control*. John Wiley and Sons, 2nd edition, 2004.
- [9] Virtual labs project. Single board heater system. http://www.co-learn.in/web_sbhs. Seen on 11 May 2011.