Documentation for Single Board Heater System

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

IIT Bombay
December 20, 2014

Contents

List of Scilab Code

Chapter 1

Block diagram explanation of Single Board Heater System

Figure1.1 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 μC 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 μC 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)

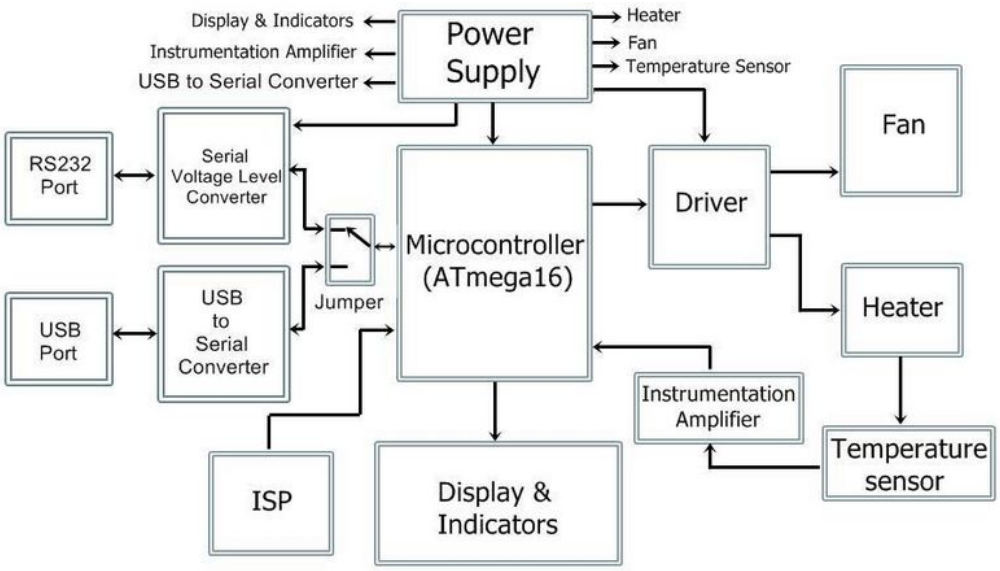

Figure 1.1: Block Diagram

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

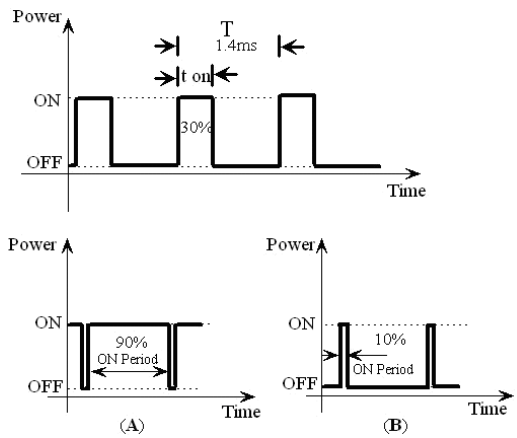

Figure 1.2: Pulse Width Modulation (A): On time is 90% of the total time period, (B): ON time is 10% of total time period

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.

$$\text{Duty cycle} = \frac{T_{ON}}{T} \tag{1.1}$$

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: 1μA/K
2. Wide range: -55°C to +150°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-Vcc(0 to 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 μs 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

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

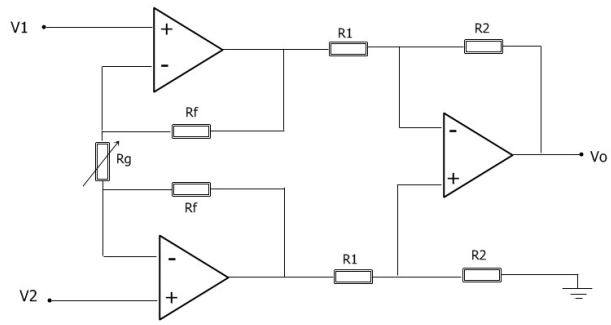

Figure 1.3: 3 Op-Amp Instrumentation Amplifier

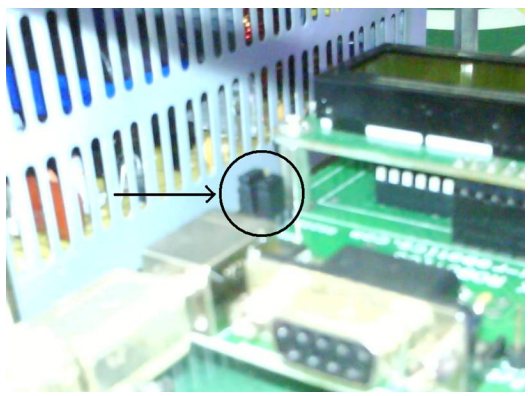

Figure 1.4: Jumper arrangement

The value of R_g is kept variable to change the overall gain of the amplifier. The signal generated by AD590 is in µA/°K. It is converted to mV/°K by taking it across a 1 KΩ resistor. The °K to°C conversion is done by subtracting 273 from the °K result. One input of the IA is fed with the mV/°K reading and the other with 273 mV. The resulting output is now in mV/°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 transreceiver.

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-

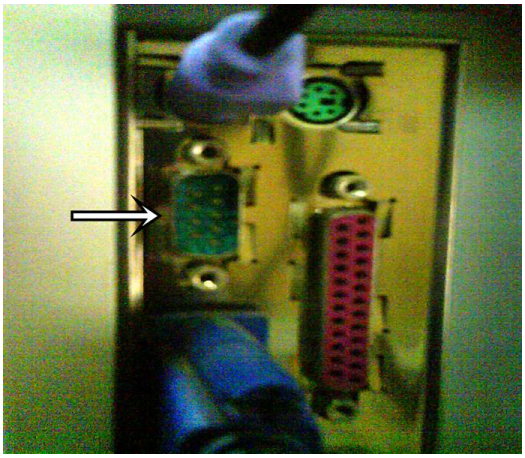

Figure 1.6: Serial port

Figure 1.7: USB communication

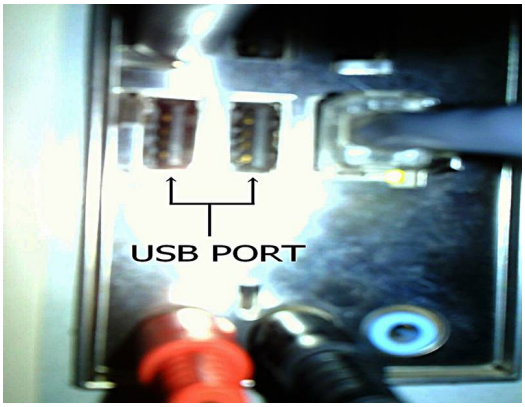

Figure 1.8: USB PORT

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.

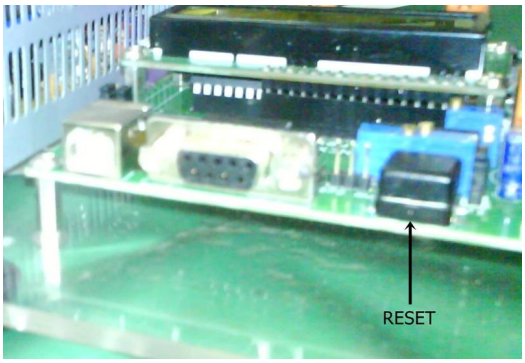

Figure 1.10: Reset

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

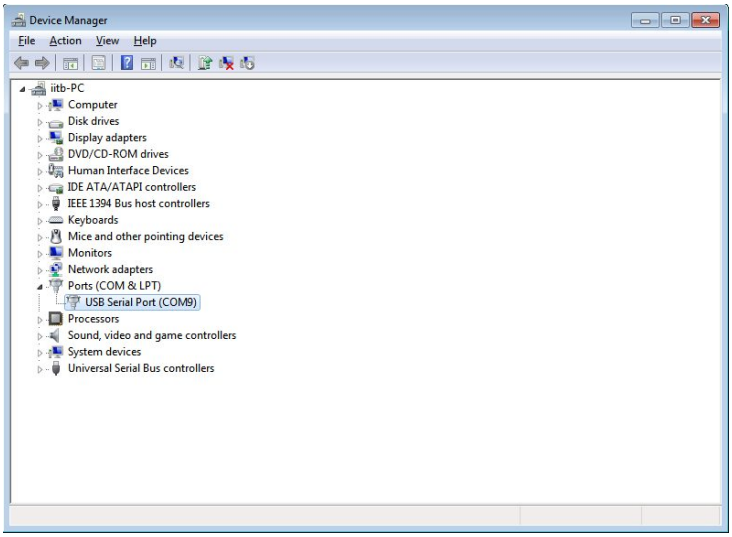

Figure 2.1: Checking Communication Port number

`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

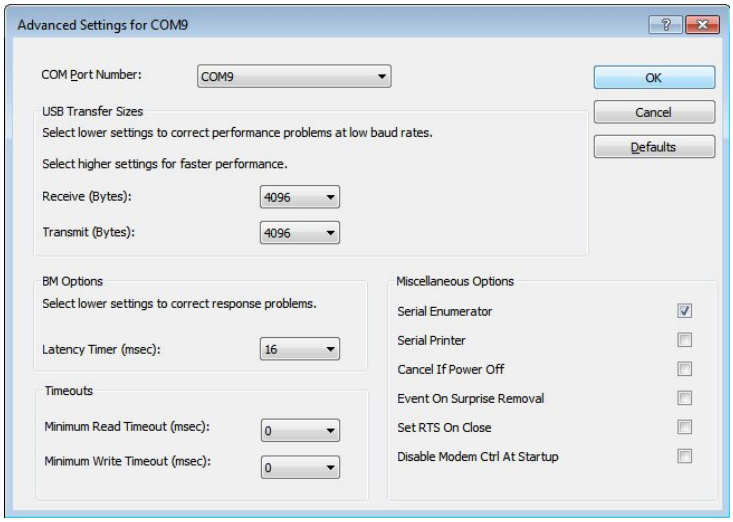

Figure 2.2: Changing Com port number

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.

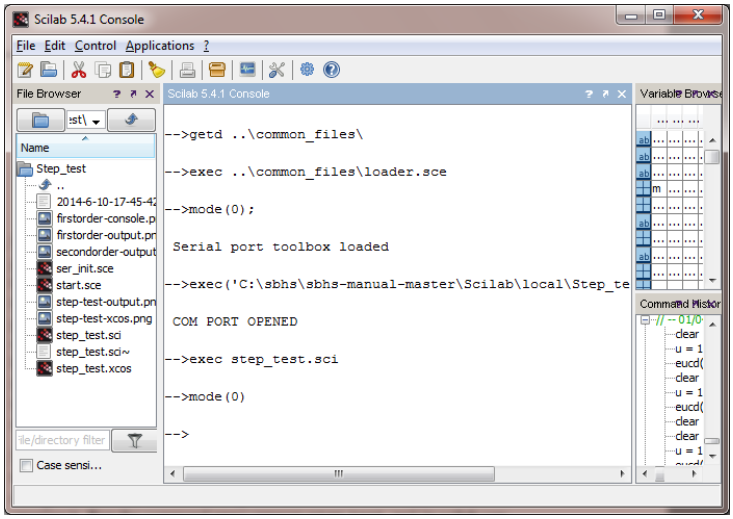

Figure 2.3: Expected responses seen on the console

All of the activities mentioned above, from `getd<space>..\common_files` untill 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-

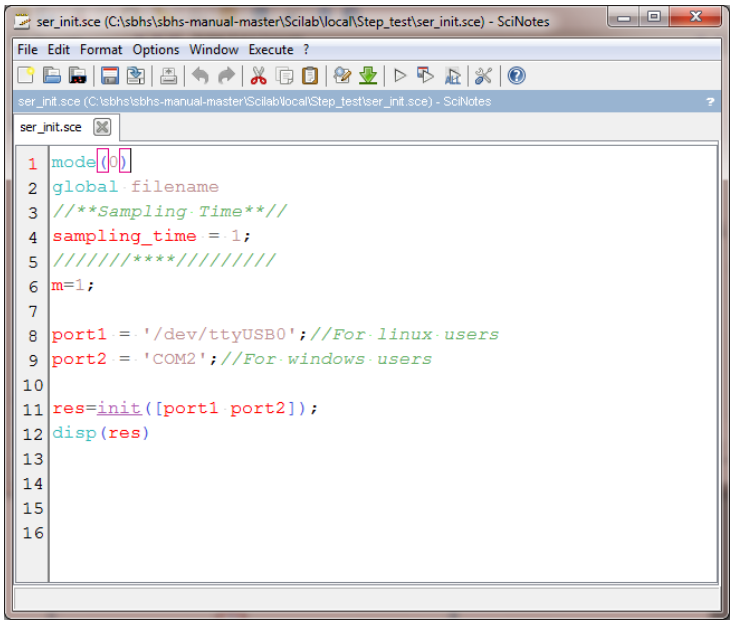

Figure 2.4: Executing script files

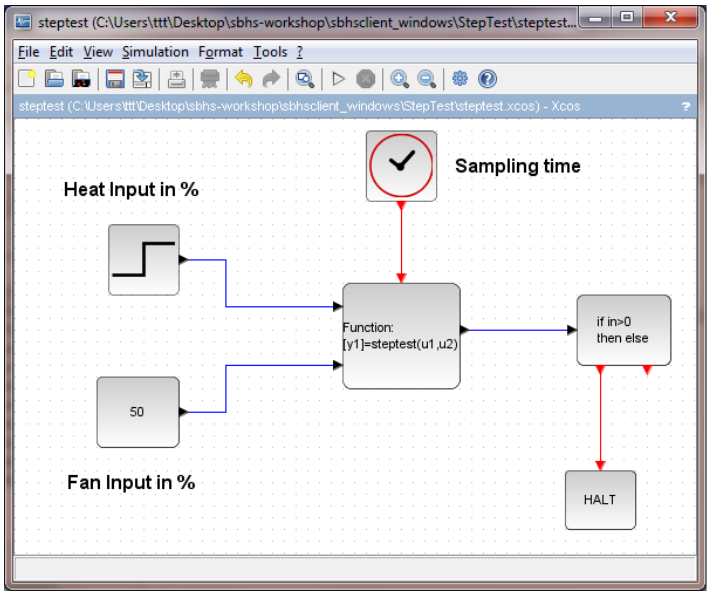

Figure 2.5: Xcos Interface

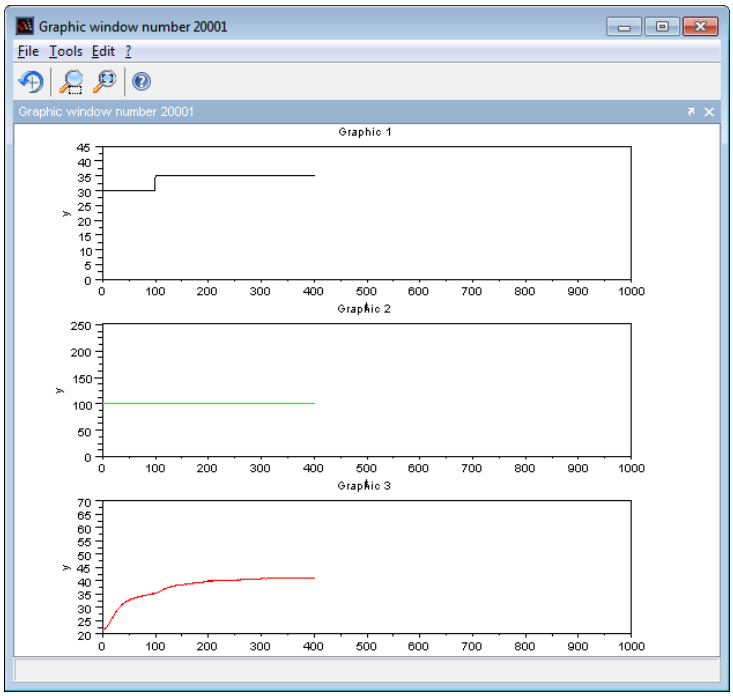

Figure 2.6: Plot obtained after executing step_test.xcos

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 permisions 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 Summary of procedure to perform a local experiment

This section sumarrizes and only lists the required commands/activities to be done in the given order to do a local experiment. It doesnot explain the expected results etc. The procedure is common for both windows as well as linux users. However, make sure you have reffered to the earlier sections of this chapter for clarity.

1. Step1: Change/ensure scilab working directory to `Step_test`. Command `pwd` can be used to know the present working directory of scilab

2. Step2: Load the functions available in `common_files` directory by executing the command `getd<space>..\common_files`

3. Step3: Load the serial communication toolbox by executing the command `exec<space>..\common_files\loader.sce`

4. Step4: Ensure correct communication port number in the `ser_init.sce` file and execute it. Execution can be done using the command `exec<space>ser_init.sce`

5. Step5: Load step test function by executing command `exec<space>step_test.sci`

6. Step6: Launch Xcos code for step test and execute it. This can be done using the command `xcos<space>step_test.xcos`

7. Step7: Execute this xcos code by clicking on the start button available on the menubar of xcos window. Let the execution run for sufficient time, until the output reaches a steadstate or untill sufficient data is collected.

Note that advance users can always make use to `start.sce` file for quickly performing the experiments.

2.4 Scilab Code under common_files

Scilab Code 2.1 `comm.sci`

```
m=1;
function [temp] = comm(heat,fan)
    global heatdisp fandisp tempdisp sampling_time m
       name handl filename

if heat<0
    heat=0
end
if heat>100
    heat=100
end
if fan<0
```

```
    fan=0
end
if fan>100
    fan=100
end

writeserial(handl,ascii(254)); //Input Heater, writeserial accepts strings; so convert 254 into its string equivalent
writeserial(handl,ascii(heat));
writeserial(handl,ascii(253)); //Input Fan
writeserial(handl,ascii(fan));
writeserial(handl,ascii(255)); //To read Temp
sleep(100);

temp = ascii(readserial(handl)); // Read serial returns a string, so convert it to its integer(ascii) equivalent
temp = temp(1) + 0.1*temp(2); // convert to temp with decimal points eg: 40.7
epoch=getdate('s');
dt=getdate();
ms=dt(10);
epoch=(epoch*1000)+ms;

A = [m,heat,fan,temp,epoch];

fdfh = file('open',filename,'unknown');

file('last', fdfh)

write(fdfh,A,'(7(f15.1,3x))');
```

```

file('close', fdfh);

m=m+1;
endfunction
```

Scilab Code 2.2 `init.sci`

```
global filename m
function status = init(port)
    global handl filename

    OS = getos();

    if OS == string('Linux')
        port_num = port(1);
        handl = openserial(port_num,"9600,n,8,0")
    else
        port_num = port(2);
        handl = openserial(port_num,"9600,n,8")
    end

if (ascii(handl) ~= [])
  status = string('COM PORT OPENED')
else
    status = string('ERROR: Check port number or USB
       connection')
end

m = 1;

dt = getdate();
year = dt(1);
month = dt(2);
day = dt(6);
hour = dt(7);
minutes = dt(8);
seconds = dt(9);
```

```


file1 = strcat(string([year month day hour minutes
    seconds]),'-');
string txt;
filename = strcat([file1, "txt"],'.');

endfunction
```

Scilab Code 2.3 `plotting.sci`

```
function [] = plotting(var,low_lim,high_lim)

    global heatdisp fandisp tempdisp setpointdisp
        sampling_time m

    timeTitle = "No. of samples with sampling time = "
        +string(sampling_time)
    if low_lim~=[] & high_lim~=[]
        heat_min = low_lim(1)
        fan_min = low_lim(2)
        temp_min = low_lim(3)
        time_min = low_lim(4)

        heat_max = high_lim(1)
        fan_max = high_lim(2)
        temp_max = high_lim(3)
        time_max = high_lim(4)

    else
        heat_min = 0
        fan_min = 0
        temp_min = 20
        time_min = 0

        heat_max = 100
        fan_max = 100
        temp_max = 100
```

```
        time_max = 1000
    end



    if length(var)==3
        heat = var(1);
        fan = var(2);
        temp = var(3);

        heatdisp=[heatdisp;heat];
        subplot(311);
        xtitle("",timeTitle,"Heat in percentage")
        plot2d(heatdisp,rect=[time_min,heat_min,
            time_max,heat_max],style=1)

        fandisp=[fandisp;fan];
        subplot(312);
        xtitle("",timeTitle,"Fan in percentage")
        plot2d(fandisp,rect=[time_min,fan_min,
            time_max,fan_max],style=2)

        tempdisp=[tempdisp;temp];
        subplot(313)
        xtitle("",timeTitle,"Temperature (deg
            celcius)")
        plot2d(tempdisp,rect=[time_min,temp_min,
            time_max,temp_max],style=5)


      elseif length(var) == 4

        heat = var(1);
        fan = var(2);
        temp = var(3);
        setpoint = var(4);

        heatdisp=[heatdisp;heat];
```

```
            subplot(311);
            xtitle("",timeTitle,"Heat in percentage")
            plot2d(heatdisp,rect=[time_min,heat_min,
                time_max,heat_max],style=1)

            fandisp=[fandisp;fan];
            subplot(312);
            xtitle("",timeTitle,"Fan in percentage")
            plot2d(fandisp,rect=[time_min,fan_min,
                time_max,fan_max],style=2)

            tempdisp=[tempdisp;temp];
            setpointdisp=[setpointdisp;setpoint]
            subplot(313)
            xtitle("",timeTitle,"Temperature (deg
                celcius)")
            plot2d(tempdisp,rect=[time_min,temp_min,
                time_max,temp_max],style=5)
            plot2d(setpointdisp,rect=[time_min,temp_min,
                time_max,temp_max],style=1)

    end
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.

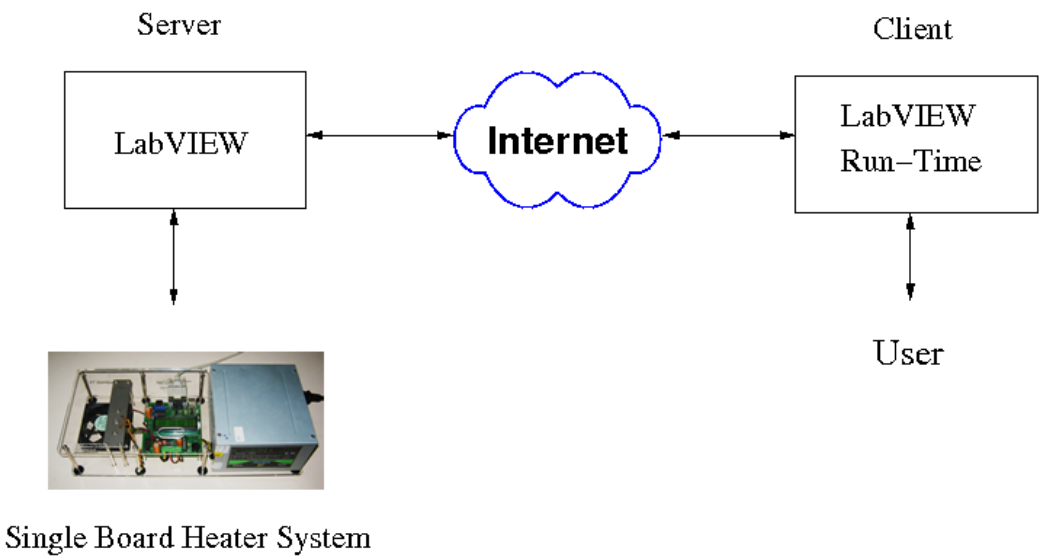

Figure 3.1: SBHS virtual laboratory with remote access using LabVIEW

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.

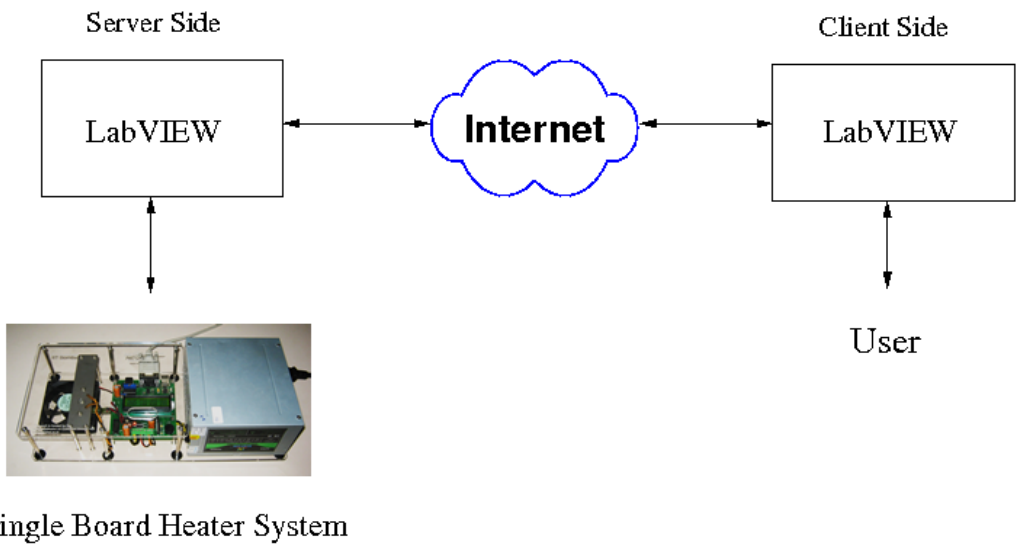

Figure 3.2: SBHS virtual laboratory with remote access and live data sharing using LabVIEW

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

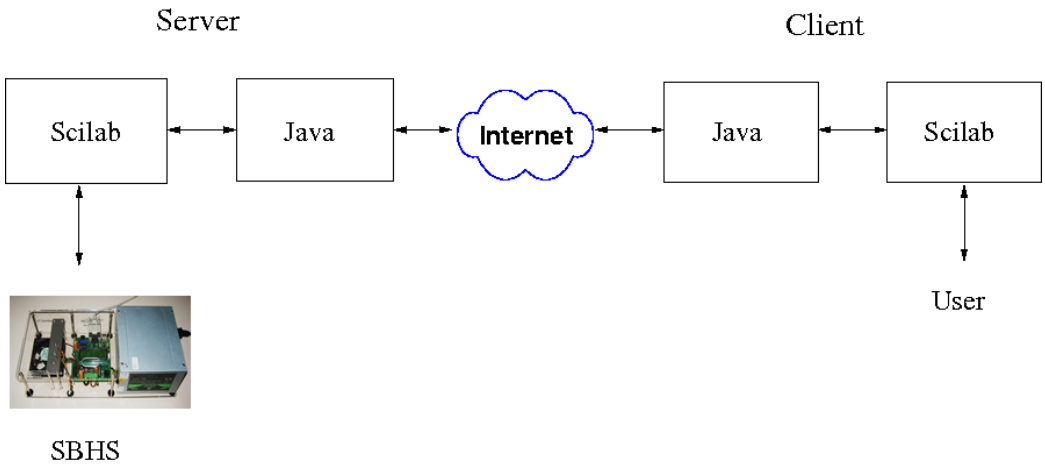

Figure 3.3: SBHS virtual laboratory using open source software

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.

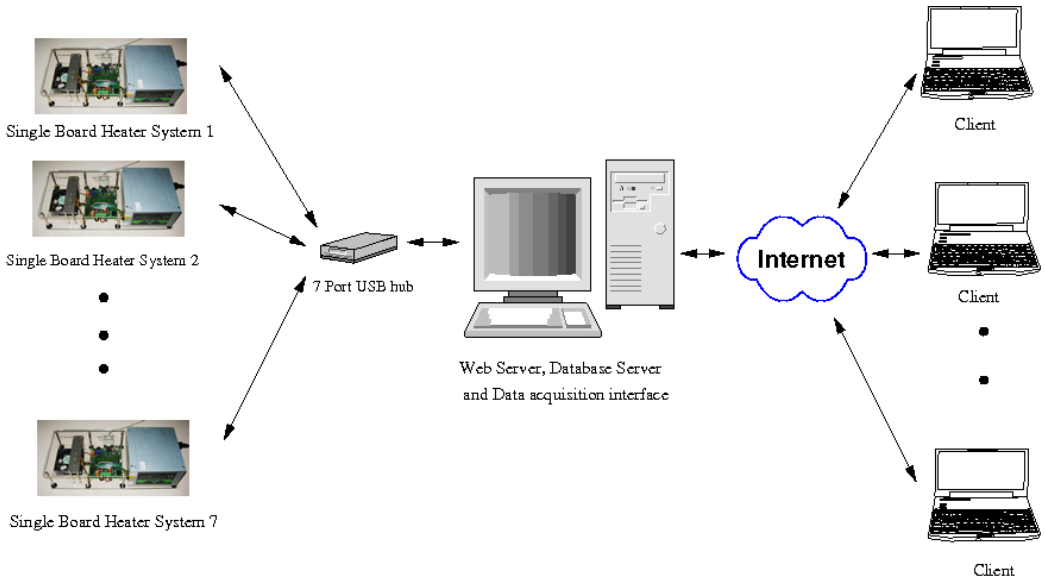

Figure 3.4: Virtual control lab hardware architecture

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 internetwork 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.

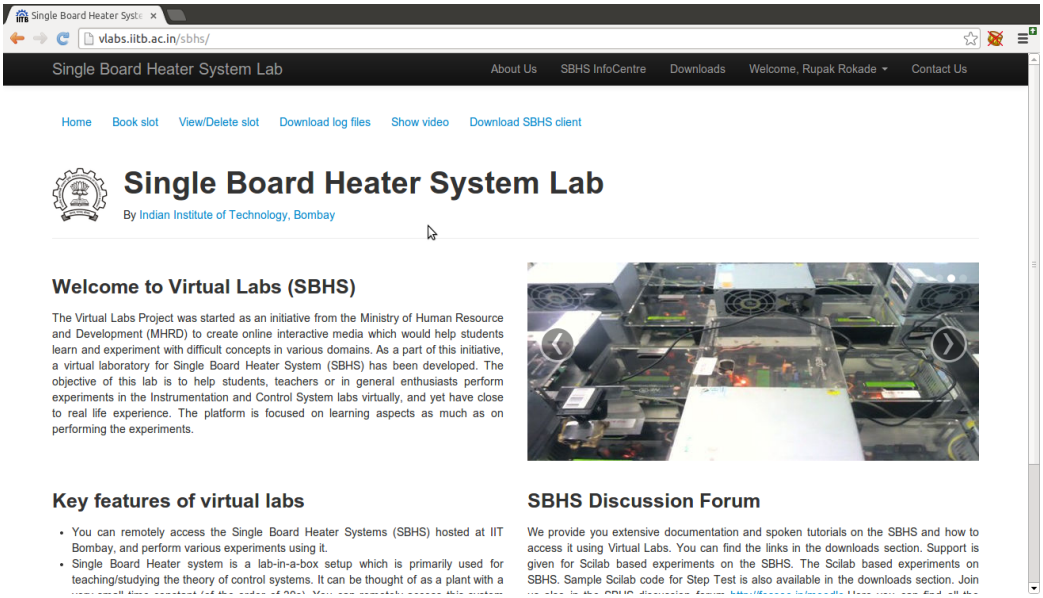

Figure 3.5: Home page of SBHS V Labs

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 remotly using your computer over

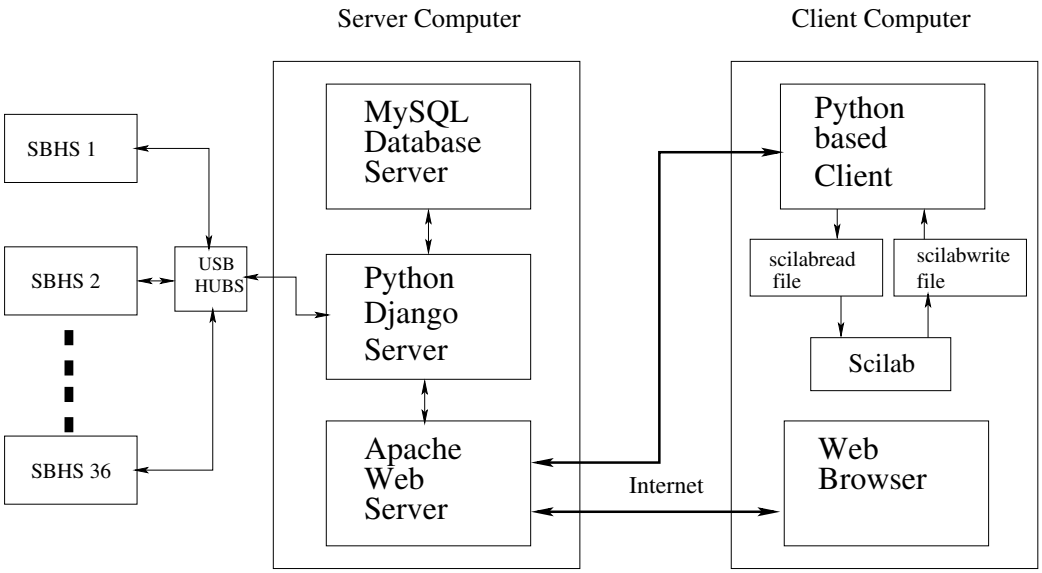

Figure 3.6: Current Architecture of SBHS Virtual Labs

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 section `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 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

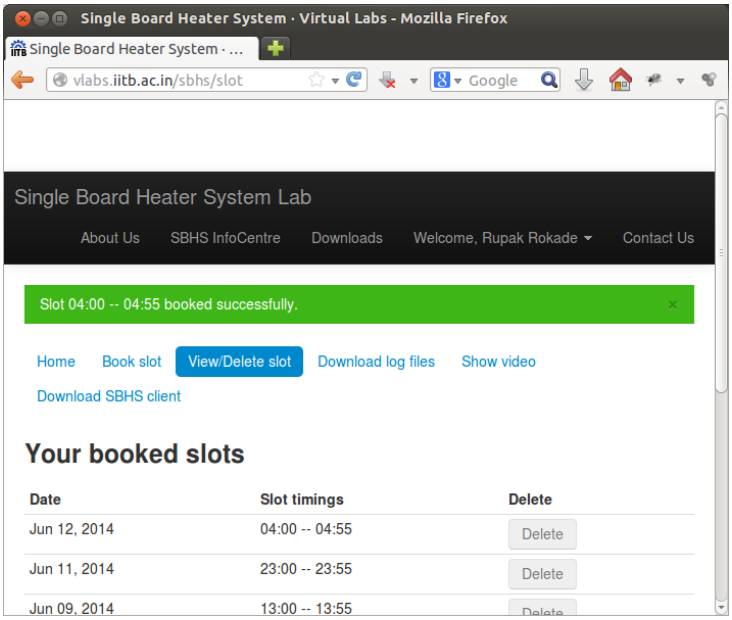

Figure 3.8: Slot booking

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 files contains various arguments whose values must be eneterd 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 ../common_files`. 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

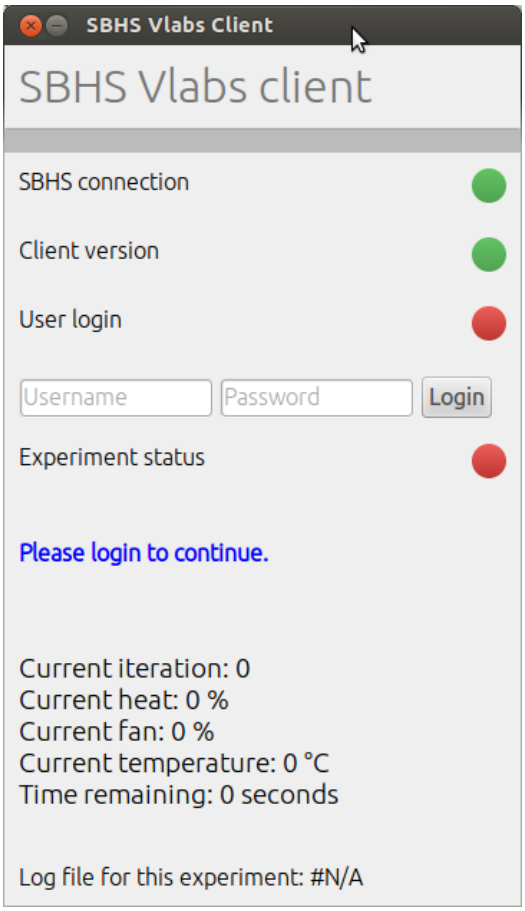

Figure 3.9: Python Client

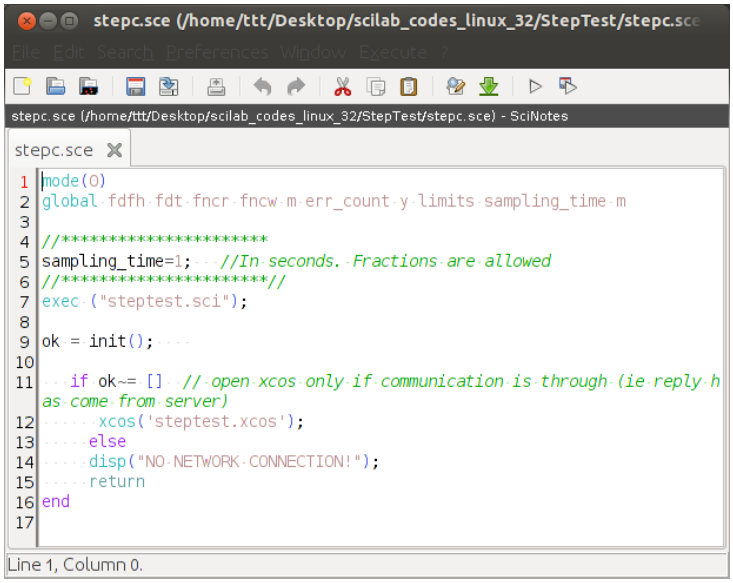

Figure 3.10: stepc.sce file

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 doesnt 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

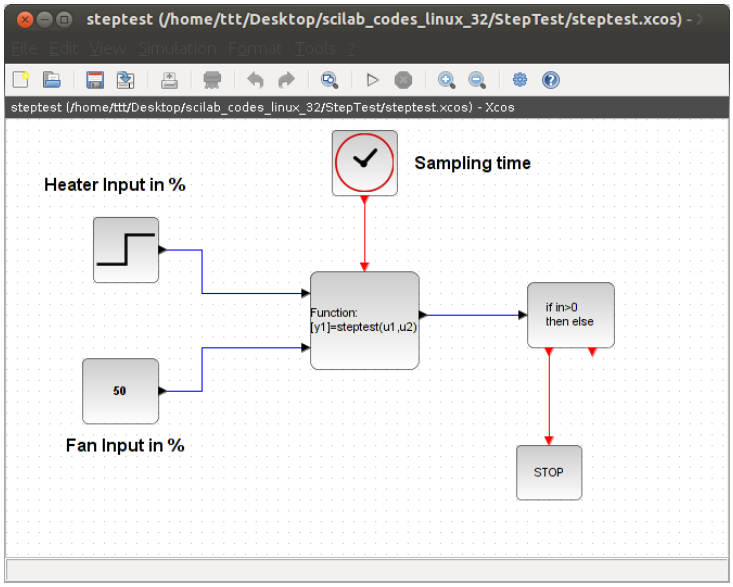

Figure 3.11: Xcos for step test

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

3.6 Summary

This section summarizes the process to perform an experiment on SBHS using virtual lab interface. This section assumes that the user has already created an account and booked a slot as explained in section 3.5.1. It also assumes that the proxy settings are already done as explained in section 3.5.2. The user should follow these steps within the booked slot time.

- Step1: Open the StepTest experiment directory
- Step2: Double-click on the file `run`. Expect the SBHS cient application to open.
- Step3: Enter the username and password inside the SBHS cient application and press `login` button. Expect the message `Ready to execute scilab code`
- Step4: Switch to the StepTest experiment directory and double-click on the file `stepc.sce`. This will launch scilab and also open the file `stepc.sce`

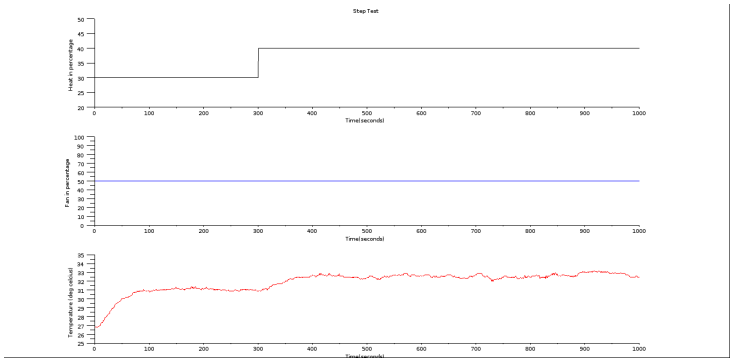

Figure 3.12: Output of Step Test

in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to StepTest and then open the `stepc.sce` file in the scilab editor.

- Step5 Switch to the scilab console and execute the command `getd ../common_files`
- Step6: Execute the file `stepc.sce`. Expect the step test xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
- Step7: Execute the step test xcos daiagram. You may change the input parameters, if required, before executing. Expect a plot window to open automatically showing three graphs.
- Step8: Stop the Xcos simulation after the experiment is completed properly.

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.

In the `step_test.xcos` file, open the heater block's parameters to apply a step change of say 10 percent to the heater at operating point of 30 percent of heater after 250 seconds. The block parameters of the step input block will have `Step time = 250`, `Initial value = 30` and `Final value = 40`. Keep the fan input constant at 50 percent. Start the experiment and let it continue until you see the temperature reach the steady state.

The step test data file will be saved in `Step_test` folder. The name of the file will be the date and time at which the experiment was conducted. A sample data file is provided in the same folder. The sample data file is named as `step-data-local.txt` and `step-data-virtual.txt`. Refer to the one de-

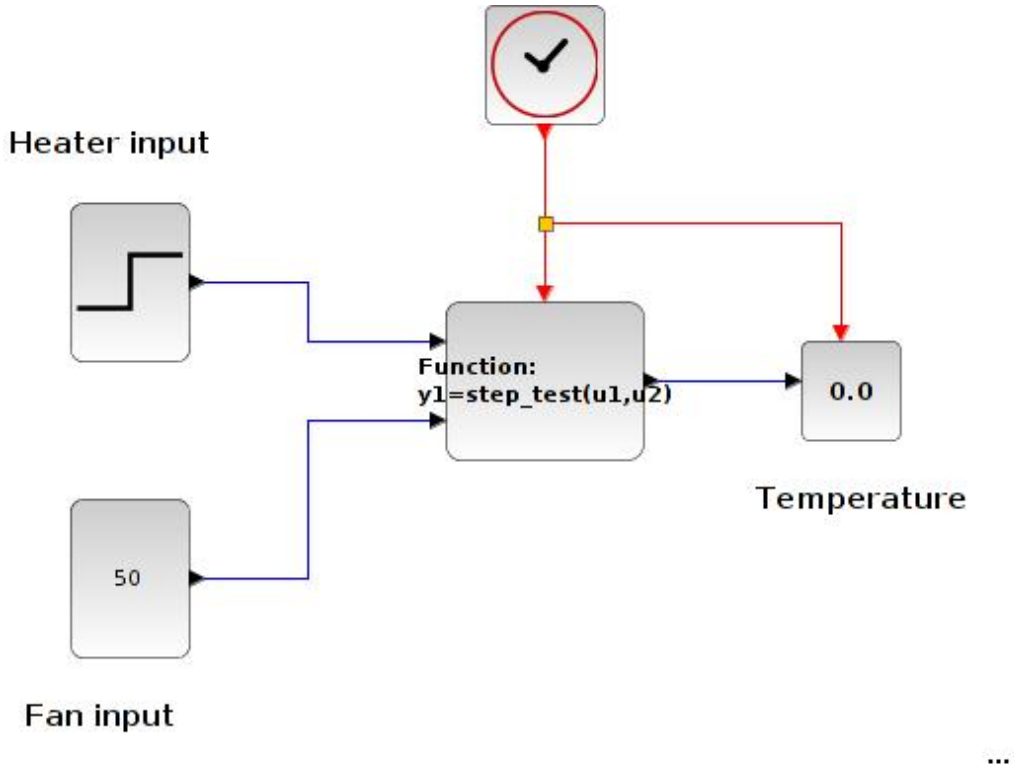

Figure 4.1: Xcos for this experiment

1.0	30.0	50.0	29.3	1412400132192.0
2.0	30.0	50.0	29.5	1412400133044.0
.				
.				
820.0	40.0	50.0	37.0	1412400950197.0
821.0	40.0	50.0	37.2	1412400951202.0

Table 4.1: Step data obtained after performing local Step Test

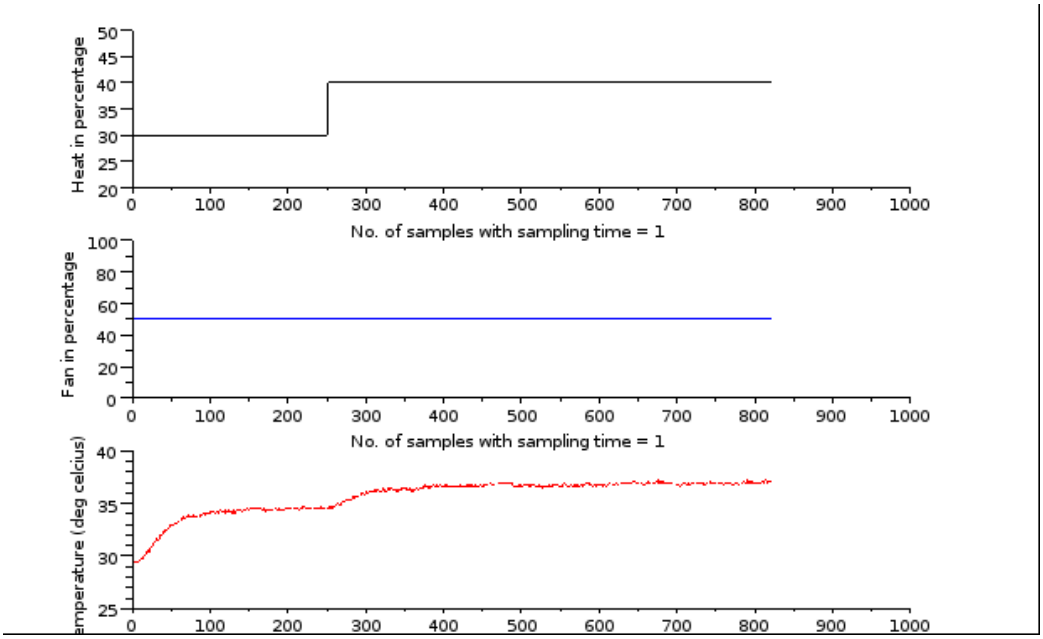

Figure 4.2: Graph shows heater current, fan speed and output temperature

pending on wheather you are performing a local or a virtual experiment. Referring to the data file thus obtained as shown in table 4.2, 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 10 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. The fifth column denotes time stamp. The virtual data file will havel four time stamp columns apart from first 3 columns. These four time stamp columns are client departure, server arrival, server departure and client arrival. These can be used for advanced control algorithms. These additional time stamps exist in virtual mode because of the presense of network delay.

4.1 Conducting Step Test on SBHS locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is exactly the same for this section. The response is as shown in figure 4.2. The output data file is as shown

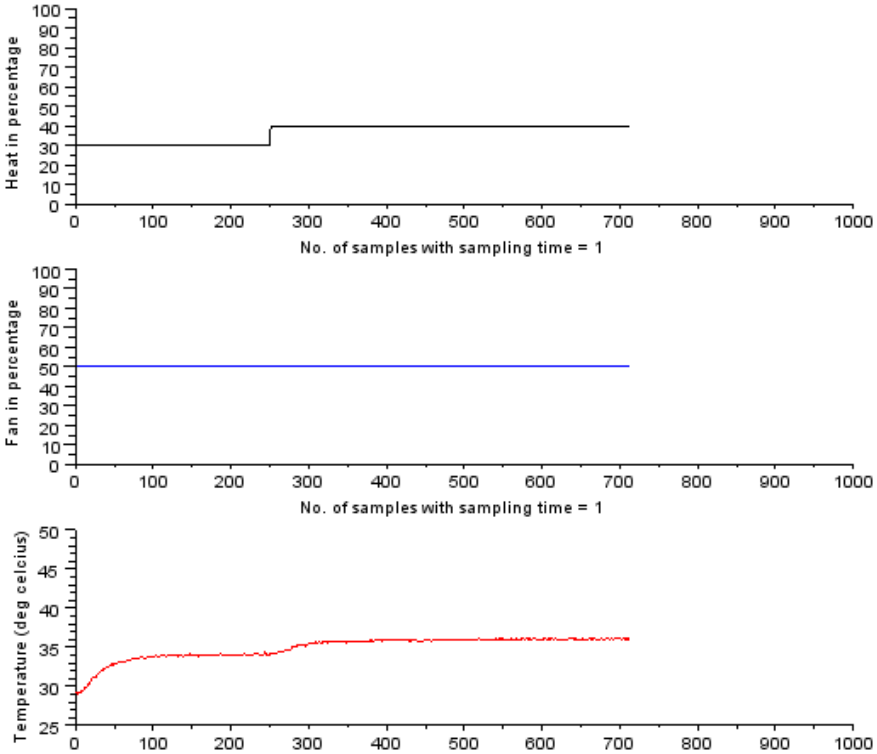

Figure 4.3: Step test Virtual experiment response

in Table 4.2

4.2 Conducting Step Test on SBHS, virtually

The detailed procedure to perform a local experiment is explained in Chapter3. A summary of the same is provided in section 3.5 It is exactly the same for this section. The virtual experiment response is shown in figure 4.3. The corresponding data file is shown in table 4.2. The time stamps shown are cut short for better viewing. This data file can be found in `StepTest` folder for virtual experiments. The name of this file is `step-data-virtual.txt`.

```
 0 0 100 29.30 14...8080 14...8955 14...8993 14...8158 0.10000E+01
1 30 50 29.00 14...9364 14...0246 14...0263 14...9442 0.10000E+01
.
.
711 40 50 36.20 14...9375 14...0280 14...0297 14...9437 0.71100E+03
712 40 50 36.10 14...0370 14...2673 14...2691 14...1834 0.71200E+03
```

Table 4.2: Step data obtained after performing virtual Step Test

4.3 Identifying First Order and Second Order Transfer Functions

In this section we shall determine the first and second order transer function model using the data obtained after performing step test experiment. Please note that this procedure is common for data obtained using both local and virtual experiments.

4.3.1 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

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

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

Rewriting the equation, we get

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

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

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

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

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

Taking the Inverse Laplace transform of equation 4.5, we get

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

From the above equation it is clear that for t=0, the value of c(t) is zero. For t= ∞, 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

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

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.

$$y(s) = G(s)u(s) \tag{4.8}$$

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

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

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

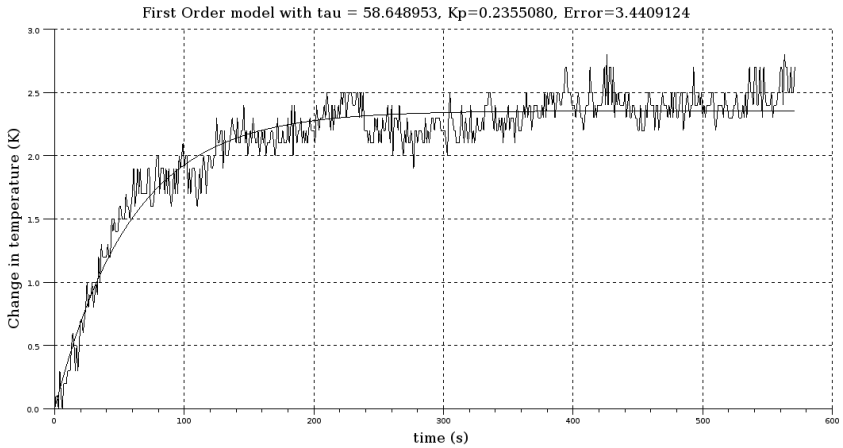

Figure 4.4: Output of the Scilab code `firstorder.sce` for data file `step-data-local.txt`

4.3.2 Procedure

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

2. Open the `Analysis` folder and then locate and open the folder `Step_Analysis`.

3. Open the `Kp-tau-order1` folder.

4. Copy the step test data file to the folder `Kp-tau-order1`.

5. Change the Scilab working directory to `Kp-tau-order1` folder under `Step_Analysis` folder.

6. Open the file `firstorder.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.

7. Save and run this code and obtain the plot as shown in figure 4.4.

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

The results presented are obtained for the data file `step-data-virtual.txt`. This data file is present under the `Step_Test` directory for local experiments.The plot thus obtained is reasonably good. See the Scilab plot to get the values of τ and K. The figure 4.4 shows a screen shot of the same. We obtain τ = 58.64, K = 0.23. 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

$$G_s(s) = \frac{0.23}{58.64s + 1} \tag{4.11}$$

4.4 Determination of Second Order Transfer Function

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

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

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

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

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:

$$A = K$$
$$B = -\frac{K\tau_1}{\tau_1 - \tau_2}$$
$$C = \frac{K\tau_2}{\tau_1 - \tau_2}$$

On substitution and inversion, we obtain

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

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.

4.4.1 Procedure

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

2. Open the `Analysis` folder and then locate and open the folder `Step_Analysis`.

3. Open the `Kp-tau-order2` folder.

4. Copy the step test data file to the folder `Kp-tau-order2`.

5. Change the Scilab working directory to `Kp-tau-order2` folder under `Step_Analysis` folder.

6. Open the file `secondorder.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.

7. Save and run this code and obtain the plot as shown in figure 4.5.

$$G_s(s) = \frac{0.235}{(57.39s + 1)(1s + 1)} \tag{4.15}$$

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

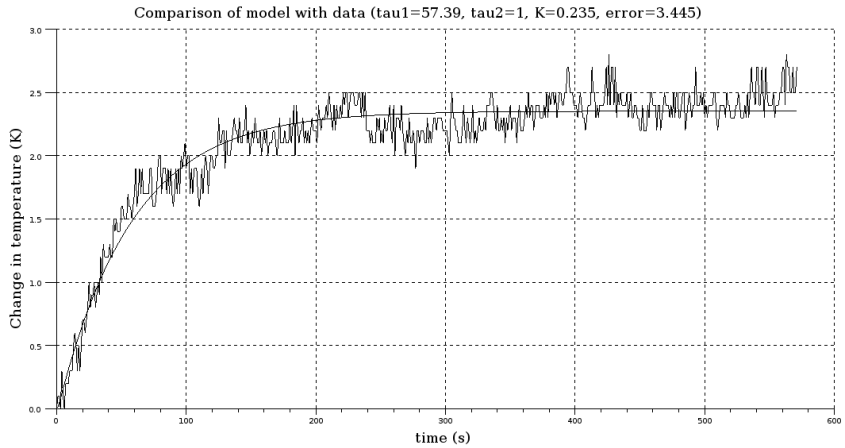

Figure 4.5: Output of the Scilab code `secondorder.sce`

4.5 Discussion

We summarize our findings now. For the first order analysis, the gain is 0.23 and the time constant τ is 58.64 seconds. For the second order analysis, the initial inflexion is well captured with the two time constants τ_1=57.39, τ_2= 1 and gain = 0.235. 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.

4.6 Scilab Code

Scilab Code 4.1 `label.sci`

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

function label(tname,tfont,labelx,labely,xyfont)
a = get("current_axes")
xtitle(tname,labelx,labely)
xgrid
```

```
t = a.title;
t.font_size = tfont; // Title font size
t.font_style = 2; // Title font style
t.text = tname;

u = a.x_label;
u.font_size = xyfont; // Label font size
u.font_style = 2; // Label font style

v = a.y_label;
v.font_size = xyfont; // Label font size
v.font_style = 2; // Label font style

// a.label_font_size = 3;

endfunction;
```

Scilab Code 4.2 `costf_1.sci`

```
function [f,g,ind] = costf_1(x,ind)
kp = x(1); tau = x(2);
y_prediction = kp * ( 1 - exp(-t/tau) );
f = (norm(y-y_prediction,2))^2;
g = numdiff(func_1,x);
endfunction

function f = func_1(x)
kp = x(1); tau = x(2);
y_prediction = kp * ( 1 - exp(-t/tau) );
f = (norm(y-y_prediction,2))^2;
endfunction
```

Scilab Code 4.3 `firstorder.sce`

```
mode(0)
filename = "step-data-local.txt"

clf
```

```
exec('costf_1.sci');
exec('label.sci');
data = fscanfMat(filename);
time = data(:, 5);
heater = int(data(:, 2));
fan = int(data(:, 3));
temp = data(:, 4);


len = length(heater);

time1 = time - time(1);
time2 = time1/1000;

len = length(heater);
heaters1 = [heater(1); heater(1:len-1)];
del_heat = heater - heaters1;
ind = find(del_heat>1);

step_instant = ind($)-1;

t = time2(step_instant:len);
t = t - t(1);
H = heater(step_instant:len);
F = fan(step_instant:len);
T = temp(step_instant:len);
T = T - T(1);
delta_u = heater(step_instant + 1)- heater(
    step_instant);

//finding Kp and Tau between Heater (H) and
    Temperature (T)
y = T;    //temperature
global('y','t');
x0 = [.3 40];
//[f,xopt,gopt] = optim(costf_1,'b',[0.1 0.1],[5 100],
    x0,'ar')
[f, xopt] = optim(costf_1,x0);
```

```
lsterr=sqrt(f);
kp = xopt(1);
tau = xopt(2);
y_prediction = kp * ( 1 - exp(-t/tau) );
plot2d(t,y_prediction);
plot2d(t,y);
title = 'First Order model with tau = ';
title = title+string(tau);
title = title+', Kp='+string(kp/delta_u);
title = title+', Error='+string(lsterr)+'';
label(title,4,'time (s)','Change in temperature (K)'
    ,4);
kp = kp/delta_u
tau
```

Scilab Code 4.4 `costf_2.sci`

```
function [f,g,ind] = costf_2(x,ind)
kp = x(1); tau1 = x(2); tau2 = x(3);
y_prediction = kp * delta_u * (1 - ...
(tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
/(tau1-tau2));
f = (norm(T-y_prediction,2))^2;
g = numdiff(func_2,x);
endfunction;
function f = func_2(x)
kp = x(1); tau1 = x(2); tau2 = x(3);
y_prediction = kp * delta_u * (1 - ...
(tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
/(tau1-tau2));
f = (norm(T-y_prediction,2))^2;
endfunction;
```

Scilab Code 4.5 `order_2_heater.sci`

```
function lsterr = order_2(t,H,T,limits,no)
x0 = [2 200 150];
// delta_u = u(2) - u(1); u = u - u(1); y = y - y(1);
```

```

delta_u = H(2)-H(1);


[f,xopt,gopt] = optim(costf_2,'b',[0 2 1],[18 300
    350],x0,'ar',200,200)
kp = xopt(1); tau1 = xopt(2); tau2 = xopt(3); lsterr =
     sqrt(f);
y_prediction = kp * delta_u * (1 - ...
(tau1*exp(-(t)/tau1)-tau2*exp(-(t)/tau2)) ...
/(tau1-tau2));
format('v',6); ord = [T y_prediction]; x = [t t t];
// xbasc();
 plot2d(t,T);

plot2d(t,y_prediction);
title = 'Comparison of model with data (tau1='
title = title+string(tau1)+', tau2='+string(tau2)
title = title+', K='+string(kp)
title = title+', error='+string(lsterr)+')'
label(title,4,'time (s)','Change in temperature (K)'
    ,4);
endfunction;
```

Scilab Code 4.6 `secondorder.sce`

```
mode(0)
filename = "step-data-local.txt";
clf
exec('costf_2.sci');
exec('label.sci');
exec ('order_2_heater.sci');


data = fscanfMat(filename);
time = data(:,5);
heater = int(data(:, 2));
fan = int(data(:, 3));
```

```
temp = data(:, 4);

// times=[time(1); time(1:$-1)];
time1 = time - time(1);
time2 = time1/1000;


// find where the step change happens

len = length(heater);
heaters1 = [heater(1); heater(1:len-1)];
del_heat = heater - heaters1;
ind = find(del_heat>1);

step_instant = ind($)-1;
t = time2(step_instant:len);
t = t - t(1);
H = heater(step_instant:len);
F = fan(step_instant:len);
T = temp(step_instant:len);
T = T - T(1);

// limits = [0,0,500,10]; no=10000; // first step
// limits = [400,0,900,26]; no=5000;// second step
lsterr = order_2(t,H,T,)
```

Scilab Code 4.7 `ser_init.sce`

```
mode(0)
global filename
//**Sampling Time**//
sampling_time = 1;
//////***/////////
m=1;

port1 = '/dev/ttyUSB0'; // For linux users
port2 = 'COM2'; // For windows users

```

```
res=init([port1 port2]);
disp(res)
```

Scilab Code 4.8 `step_test.sci`

```
mode(0)
function temp = step_test(heat,fan)
    temp = comm(heat,fan);

    plotting([heat fan temp],[20 0 25 0],[50 100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 4.9 `stepc.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits sampling_time m

// **********************
sampling_time=1;    // In seconds. Fractions are allowed
// **********************//
exec ("steptest.sci");

ok = init();

    if ok~= []  // open xcos only if communication is through (ie reply has come from server)
        xcos('steptest.xcos');
      else
      disp("NO NETWORK CONNECTION!");
      return
end
```

Scilab Code 4.10 `steptest.sci`

```
function [stop] = steptest(heat,fan)

    [stop,temp] = comm(heat,fan);// Never edit this line
    plotting([heat fan temp],[0 0 25 0],[100 100 50 1000])

endfunction
```

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 a Single Board Heater System and to identify system transfer function using ramp 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 5.1. Heater current and fan speed are the two inputs for this system. They are given in percentage of maximum. These inputs

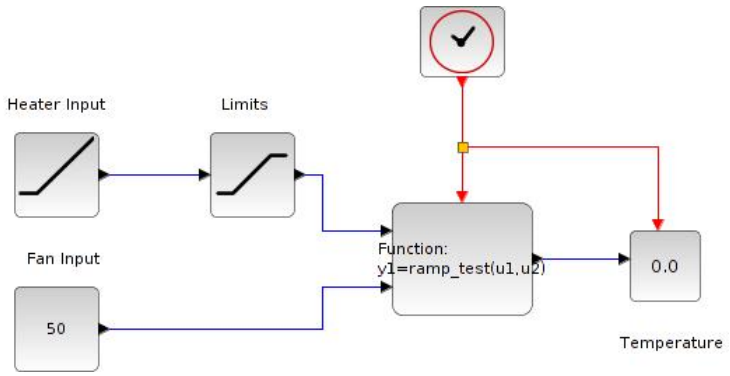

Figure 5.1: Xcos for ramp test experiment

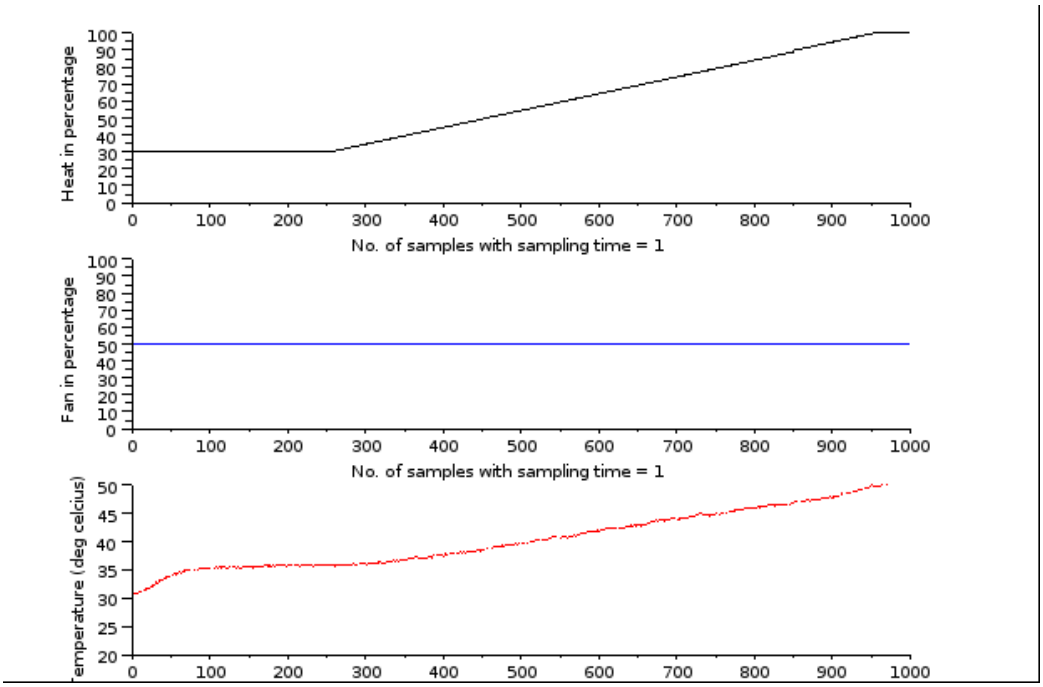

Figure 5.2: Screen shot of ramp test experiment

```
1.0      30.0      50.0       28.1   1416462726532.0
2.0       30.0        50.0         28.1   1416462727574.0
.
999.0         100.0       50.0        47.6   1416463723533.0
1000.0        100.0      50.0       47.6   1416463724533.0
```

Table 5.1: Ramp data obtained after performing local Step Test

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.

In the `ramp_test.xcos` file, open the heater block's parameters to give a ramp input to the system with some value for slope. For this experiment, we have chosen slope = 0.1. Double click on the ramp input block labled as `Heater input`. Change the following values in the respective fields: slope = 0.1, start time = 200, initial output = 20. Keep the fan constant at 100.

The ramp test data file will be saved in `Ramp_Test` folder. The name of the file will be the date and time at which the experiment was conducted. A sample data file is provided in the same folder. The sample data file is named as `ramp-data-local.txt` and `ramp-data-virtual.txt`. Refer to the one depending on wheather you are performing a local or a virtual experiment. Referring to the data file thus obtained as shown in table 6.2, 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 10 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. The fifth column denotes time stamp. The virtual data file will have four time stamp columns apart from first 3 columns. These four time stamp columns are client departure, server arrival, server departure and client arrival. These can be used for advanced control algorithms. These additional time stamps exist in virtual mode because of the presense of network delay.

5.1 Conducting Ramp Test on SBHS locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `Ramp_test`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>ramp_test.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>ramp_test.xcos`
7. Step7: Same

5.2 Conducting Ramp Test on SBHS, virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `RampTest`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the RampTest experiment directory and double-click on the file `ramptest.sce`. This will launch scilab and also open the file `ramptest.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `RampTest` and then open the `RampTest` file in the scilab editor.
5. Step5: Same

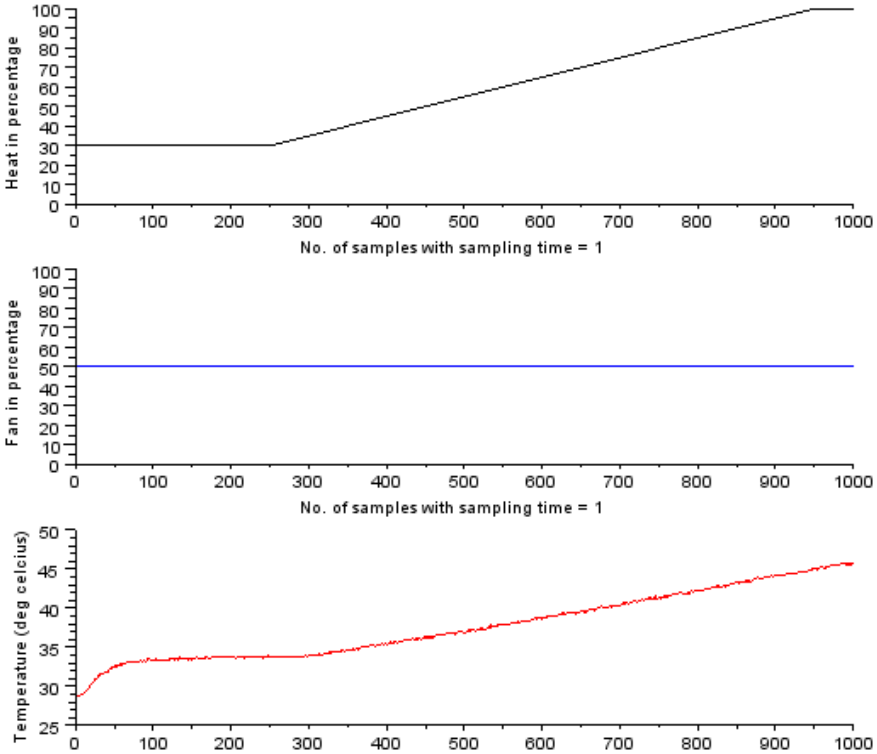

Figure 5.3: Ramp test Virtual experiment response

6. Step6: Execute the file `ramptest.sce`. Expect the ramp test xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.

7. Step7: Execute the ramptest xcos diagram.

8. Step8: Same

The virtual experiment response is shown in figure 5.3. The corresponding data file is shown in table 6.2. The time stamps shown are cut short for better viewing. This data file can be found in `RampTest` folder for virtual experiments. The name of this file is `step-data-virtual.txt`.

```
0 0 100 28.80 14...8993 14...0301 14...0318 14...9040 0.10000E+01
1 30 50 28.80 14...2861 14...4172 14...4189 14...2908 0.10000E+01
.
.
617 66 50 39.10 14...7141 14...8476 14...8494 14...7188 0.61700E+03
618 66 50 39.10 14...8120 14...9456 14...9473 14...8167 0.61800E+03
```

Table 5.2: Ramp data obtained after performing virtual Ramp Test

5.3 Identifying First Order Transfer Function

In this section we shall determine the first order transer function model using the data obtained after performing step test experiment. Please note that this procedure is common for data obtained using both local and virtual experiments.

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

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

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

Combining the previous two equations, we get

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

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

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

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

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

$$C(s) = K\upsilon \left\{ \frac{1}{s^2} - \frac{\tau}{s} + \frac{\tau^2}{\tau s + 1} \right\} \tag{5.6}$$

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

$$c(t) = K\upsilon \left\{ t - \tau + \tau e^{\frac{-t}{\tau}} \right\} \tag{5.7}$$

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

$$e(t) = r(t) - c(t) \tag{5.8}$$

$$e(t) = K\upsilon t - K\upsilon t + K\upsilon\tau - K\upsilon\tau e^{\frac{-t}{\tau}} \tag{5.9}$$

$$e(t) = K\upsilon\tau(1 - e^{\frac{-t}{\tau}}) \tag{5.10}$$

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

$$e(t) = \tau \tag{5.11}$$

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.

5.3.1 Procedure

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

2. Open the `Analysis` folder and then locate and open the folder `Ramp_Analysis`.

3. Copy the ramp test data file to this folder.

4. Change the Scilab working directory to `Ramp_Analysis`

5. Open the file `ramp_virtual.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.

6. Save and run this code and obtain the plot as shown in figure 5.4.

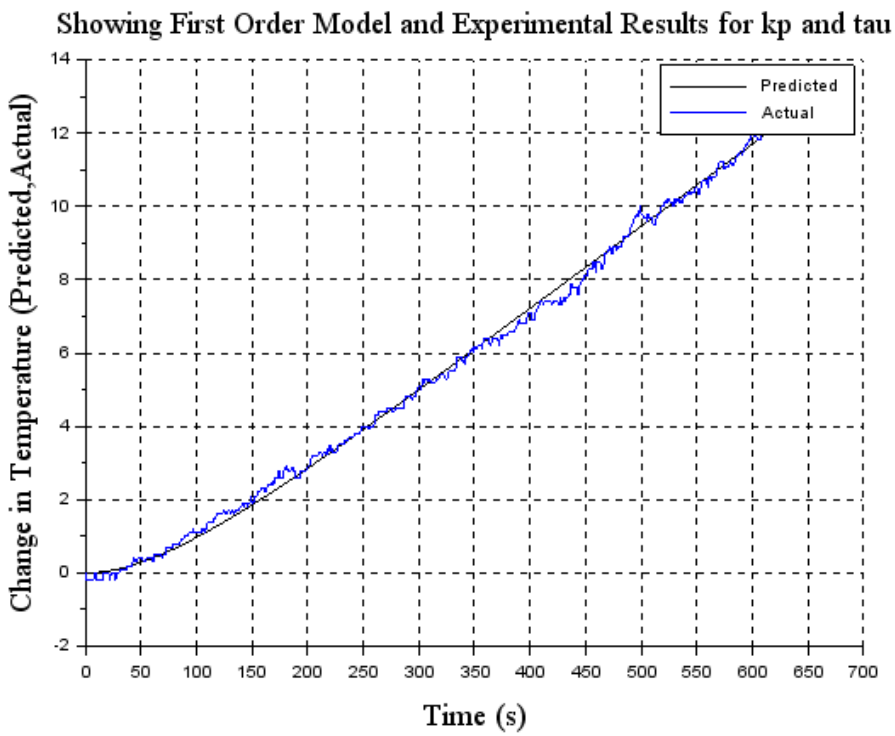

Figure 5.4: Output of the Scilab code `ramp_virtual.sce`

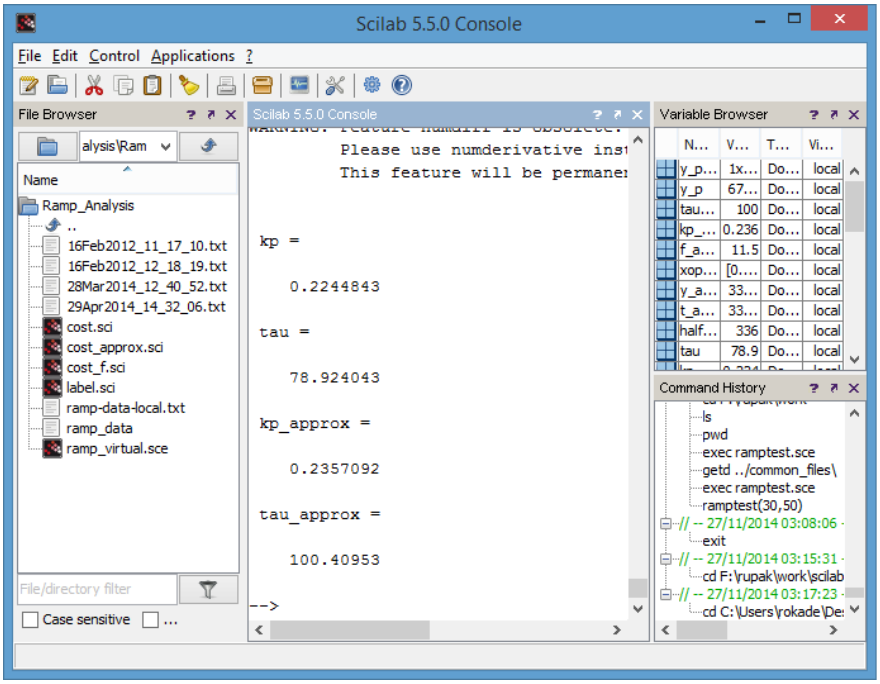

Figure 5.5: Scilab console after executing code`ramp_virtual.sce`

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

The results presented are obtained for the data file `ramp-data-virtual.txt`. This data file is present under the `Ramp_Test` directory for local experiments. The plot thus obtained is reasonably good. See the Scilab console to get the values of τ and K. It is as shown in figure 5.5 The figure 5.4 shows a screen shot of the same. We obtain τ = 78.92, K = 0.22. The transfer function obtained here is at the operating point of enterValue 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

$$G_s(s) = \frac{0.22}{78.92s + 1} \tag{5.12}$$

5.4 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 ramp 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].

The necessary codes are listed in the section 5.5.

5.5 Scilab Code

Scilab Code 5.1 `ramp_test.sci`

```
mode(0)
function temp = ramp_test(heat,fan)
    temp = comm(heat,fan);
```

```

    plotting([heat fan temp],[0 0 20 0],[100 100 50 1000])

    m=m+1;
endfunction
```

Scilab Code 5.2 `label.sci`

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

function label(tname,tfont,labelx,labely,xyfont)
a = get("current_axes")
xtitle(tname,labelx,labely)
xgrid
t = a.title;
t.font_size = tfont; // Title font size
t.font_style = 2; // Title font style
t.text = tname;

u = a.x_label;
u.font_size = xyfont; // Label font size
u.font_style = 2; // Label font style

v = a.y_label;
v.font_size = xyfont; // Label font size
v.font_style = 2; // Label font style

// a.label_font_size = 3;

endfunction;
```

Scilab Code 5.3 `cost.sci`

```
function f = func_1(x)
```

```
  k = x(1);
  tau = x(2);
  y_prediction = k*(t + tau*(exp(-t/tau) - 1));
  f = (norm(y - y_prediction,2))^2;
endfunction

function [f,g,ind1] = cost(x,ind1)
  k = x(1);
  tau = x(2);
  y_prediction = k*(t + tau*(exp(-t/tau) - 1));
  f = (norm(y - y_prediction,2))^2;
  g = numdiff(func_1,x);
endfunction
```

Scilab Code 5.4 `cost_approx.sci`

```
function f = func_approx(x)
  k = x(1);
  tau = x(2);
  y_p_approx = k*(t_approx - tau);
  f = (norm(y_approx - y_p_approx,2))^2;
endfunction

function [f,g,ind] = cost_approx(x,ind)
  k = x(1);
  tau = x(2);
  y_p_approx = k*(t_approx - tau);
  f = (norm(y_approx - y_p_approx,2))^2;
  g = numdiff(func_approx,x);
endfunction
```

Scilab Code 5.5 `ramptest.sci`

```
function [stop] = ramptest(heat,fan)

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

```
    plotting([heat fan temp],[0 0 25 0],[100 100 50 1000])

endfunction
```

Scilab Code 5.6 `ramptest.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits sampling_time m

//**********************
sampling_time=1;    //In seconds. Fractions are allowed
//**********************//
exec ("ramptest.sci");

ok = init();

    if ok~= []  // open xcos only if communication is through (ie reply has come from server)
        xcos('ramptest.xcos');
      else
      disp("NO NETWORK CONNECTION!");
      return
end
```

Scilab Code 5.7 `ramp_virtual.sce`

```
mode(-1);

//filename = "20Apr2012_15_10_35.txt";   //complete path of the saved data file
filename ="ramp-data-local.txt";
slope = 0.1; // change this to the slope that you have used in the experiment
ind1=3;
//Ramp Analysis
exec('cost_approx.sci');
```

```
exec('cost.sci');
exec('label.sci');

data = fscanfMat(filename);
time = data(:, 5);
heater = int(data(:, 2));
fan = int(data(:, 3));
temp = data(:, 4);


len = length(heater);
heaters1 = [heater(1); heater(1:$-1)];
del_heat = abs(heater - heaters1);
ind = find(del_heat >.5);

t = time(ind(2):ind($-1));
t=t/1000
H = heater(ind(2):ind($-1));
T = temp(ind(2):ind($-1));

t = t - t(1);
T = T - T(1);

y = T;
x0 = [.5 100]
global('y','t');

[f, xopt] = optim(cost,x0);
kp = xopt(1)/slope
tau = xopt(2)

len = length(t);
halfway = ceil(len/2);

t_approx = t(halfway:len);
y_approx = y(halfway:len);
global('y_approx','t_approx');

```

```
[f_approx,xopt_approx] = optim(cost_approx,x0);
kp_approx = xopt_approx(1)/slope;
tau_approx = xopt_approx(2);

// Display and Plot
disp('kp = ');
disp(kp);
disp('tau = ');
disp(tau);
disp('kp_approx = ');
disp(kp_approx);
disp('tau_approx = ');
disp(tau_approx);

y_p = kp*slope*(t + tau*(exp(-t/tau) - 1));
y_p_approx = kp_approx*slope*(t_approx - tau_approx);
y_p_approx = y_p_approx';
plot2d(t,[y_p,T]);
label('Showing First Order Model and Experimental
   Results for kp and tau',4,'Time (s)','Change in
   Temperature (Predicted,Actual)',4);
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 and Xcos as an interface for sending and receiving data. Xcos diagram is shown in figure 6.1. The heater current is varied sinusoidally. They are given in percentage of maximum. These inputs can be varied by setting the properties of the input block's properties in Xcos. 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. 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.

In the `sine_test.xcos` file, open the `sinusoid generator` block's parameters to set the value of sine magnitude and frequency. For the experiment results shown, we have chosen Magnitude = 10, Frequency = $2 * 3.14 * 0.007$. Note that the frequency is to be put in rad/sec. We keep the Phase = 0. There is also a provision to give the sine input with an offset in amplitude. This can be set using the `Offset` block. We have choosen offset of 20. The time at which the sine input

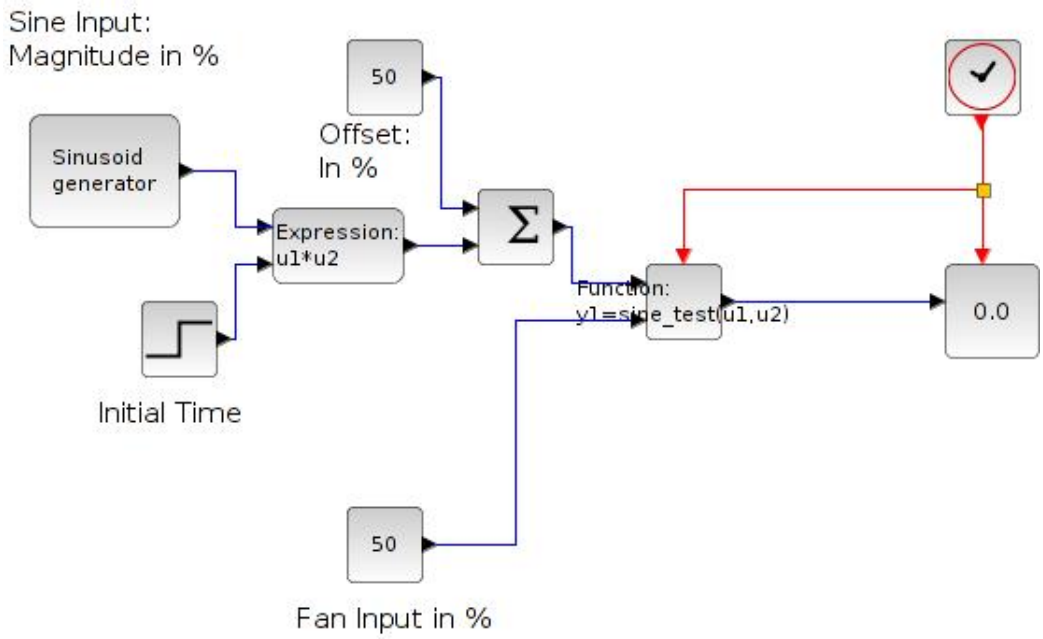

Figure 6.1: Xcos for local Sine Test

is given after the experiment is started can also be set. This can be done using the `Initial Time` block. Open the `Initial Time` block's parameters. To make the sine input appear after 200 samples of start of the experiment, keep Step time = 200, Initial Value = 0 and Final Value = 1. The initial value and final value will never change for any other value of step time.

The sine test data file will be saved in `Sine_Test` folder. The name of the file will be the date and time at which the experiment was conducted. A sample data file is provided in the same folder. The sample data file is named as `sine-data-local.txt` and `sine-data-virtual.txt`. Refer to the one de-

```
1.0      20.0        50.0       20.0    1416642805261.0
2.0        20.0        50.0         20.0    1416642806332.0
.
13093.0     28.5        50.0          26.6    1416656557353.0
13094.0      28.5        50.0         26.6    1416656558363.0
```

Table 6.1: Data obtained after application of sine input of $0.007Hz$

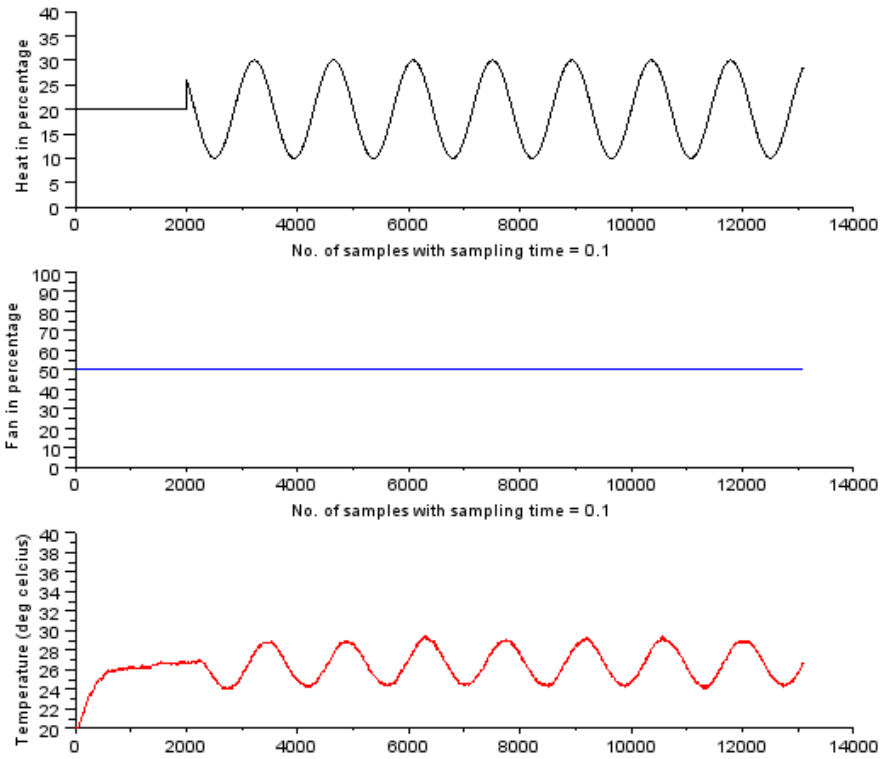

Figure 6.2: Plot for sine input 0.007Hz

pending on wheather you are performing a local or a virtual experiment. Referring to the data file thus obtained as shown in table 6.1, the first column in this table denotes samples. The second column in this table denotes heater in percentage. It starts at 20 and then varies sinosoidally. 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. The fifth column denotes time stamp. The virtual data file will have four time stamp columns apart from first 3 columns. These four time stamp columns are client departure, server arrival, server departure and client arrival. These can be used for advanced control algorithms. These additional time stamps exist in virtual mode because of the presense of network delay.

6.1 Conducting Sine Test on SBHS locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `Sine_test`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load sine test function by executing command
   `exec<space>sine_test.sci`
6. Step6: Load Xcos code for sine test using the command
   `exec<space>sine_test.xcos`
7. Step7: Same

6.2 Conducting Sine Test on SBHS, virtually

The detailed procedure to perform a local experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

```
0 0 100 28.80 14...4739 14...6427 14...6445 14...4786 0.10000E+01
1 20 50 28.90 14...5247 14...6938 14...6956 14...5294 0.10000E+01
.
.
999 18 50 31.00 14...2253 14...3973 14...3990 14...2299 0.99900E+03
1000 19 50 31.00 14...3268 14...4989 14...5007 14...3315 0.10000E+04
```

Table 6.2: Sine data obtained after performing virtual Sine Test

1. Step1: The working directory is `SineTest`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the SineTest experiment directory and double-click on the file `sinetest.sce`. This will launch scilab and also open the file `sinetest.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `SineTest` and then open the `sinetest.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `sinetest.sce`. Expect the sine test xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the sinetest xcos diagram.
8. Step8: Same

The virtual experiment response is shown in figure 6.3. The corresponding data file is shown in table 6.1. The time stamps shown are cut short for better viewing. This data file can be found in `SineTest` folder for virtual experiments. The name of this file is `sine-data-virtual.txt`.

6.3 Frequency Analysis of sine test data

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

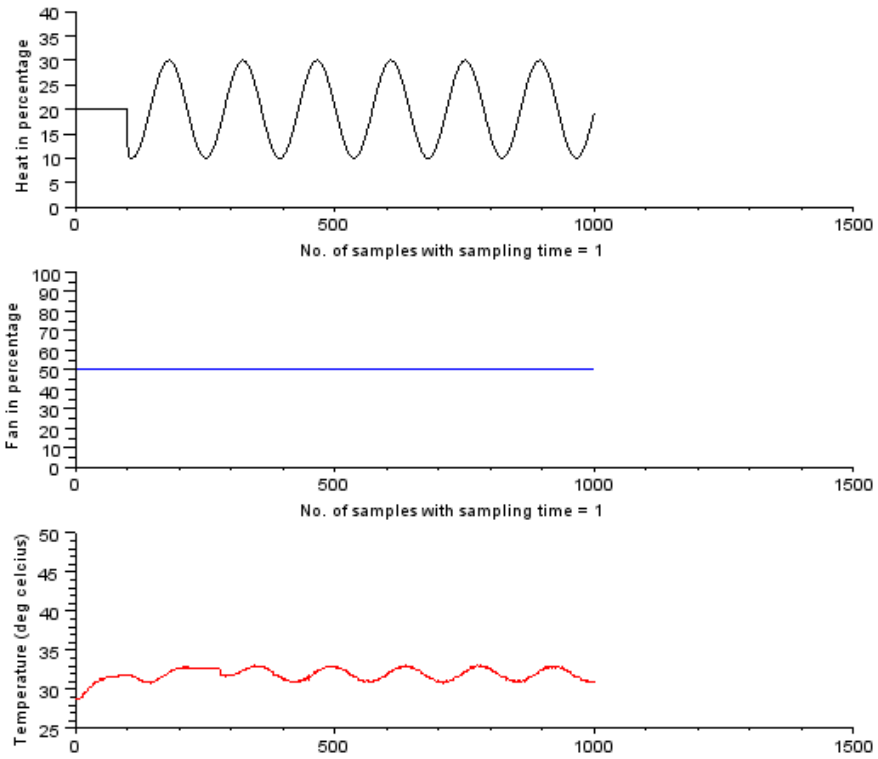

Figure 6.3: Sine test Virtual experiment response

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. Let us see the theory and procedure. Please note that this procedure is common for data obtained using both local and virtual experiments.

Consider a sinusoidal input

$$U(t) = Asin\omega t \tag{6.1}$$

The Laplace transform of the above equation yields

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

Consider the standard first order transfer function given below

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

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

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

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

Taking Laplace Inverse, we get

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

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.

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

where,

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

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.

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

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

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

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

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

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

6.3.1 Procedure

Now let us calculate amplitude ratio and phase difference.

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

2. Open the `Analysis` folder and then locate and open the folder `Sine_Analysis`.

3. Copy the sine test data file to this folder.

4. Change the Scilab working directory to `Sine_Analysis`

5. Open the file `sine-analysis.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.

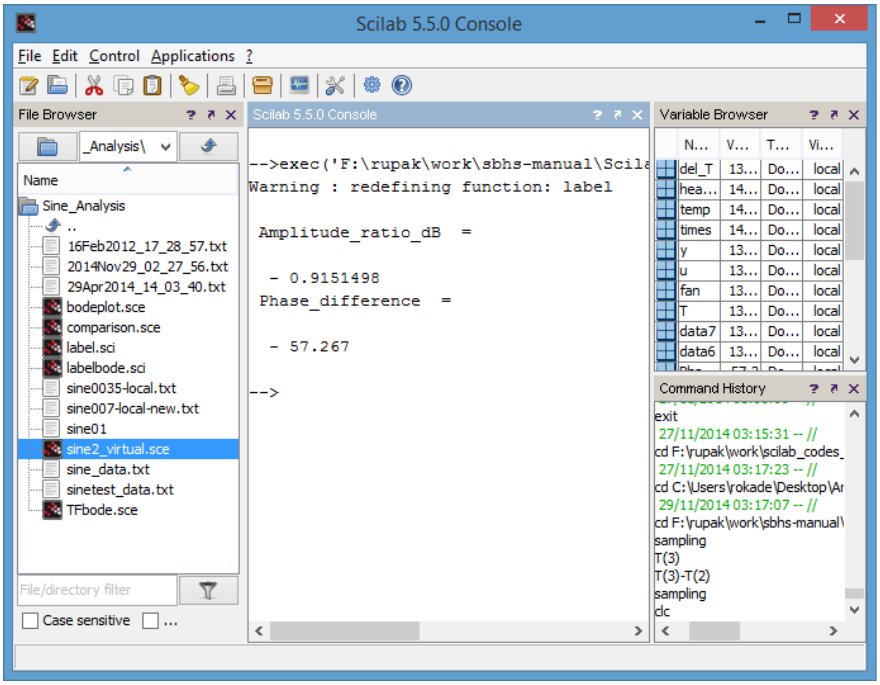

Figure 6.4: Amplitude ratio and Phase difference for local data file

6. Put the value of frequency `f` for the calculation of amplitude ratio and phase difference and execute it. Here `f` means input frequency.

7. Expect the values of amplitude ratio and phase difference on the scilab console.

The results shown are for the data file sine-data-local.txt. It could be seen from figure 6.4 that the amplitude ratio turns out to be -0.915dB and phase difference to be $-57.267°$. The plot thus obtained is shown in figure 6.5

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 frequencies into the Scilab code `TFbode.sce` and execute it to get a Bode plot of the plant which is illustrated in figure 6.6.

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 6.7. 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 6.8

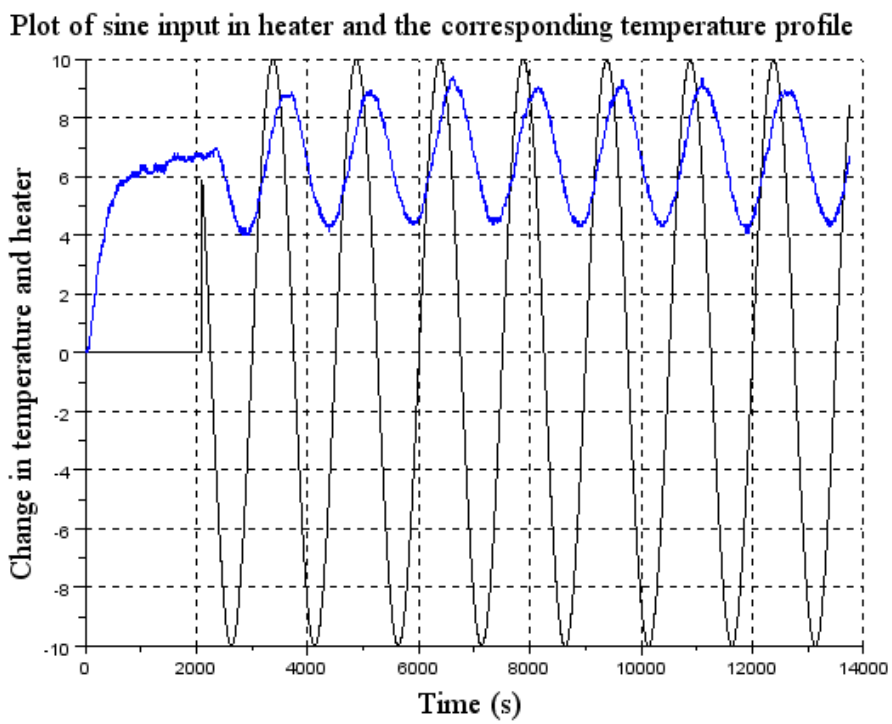

Figure 6.5: Plot of Input and Output vs time

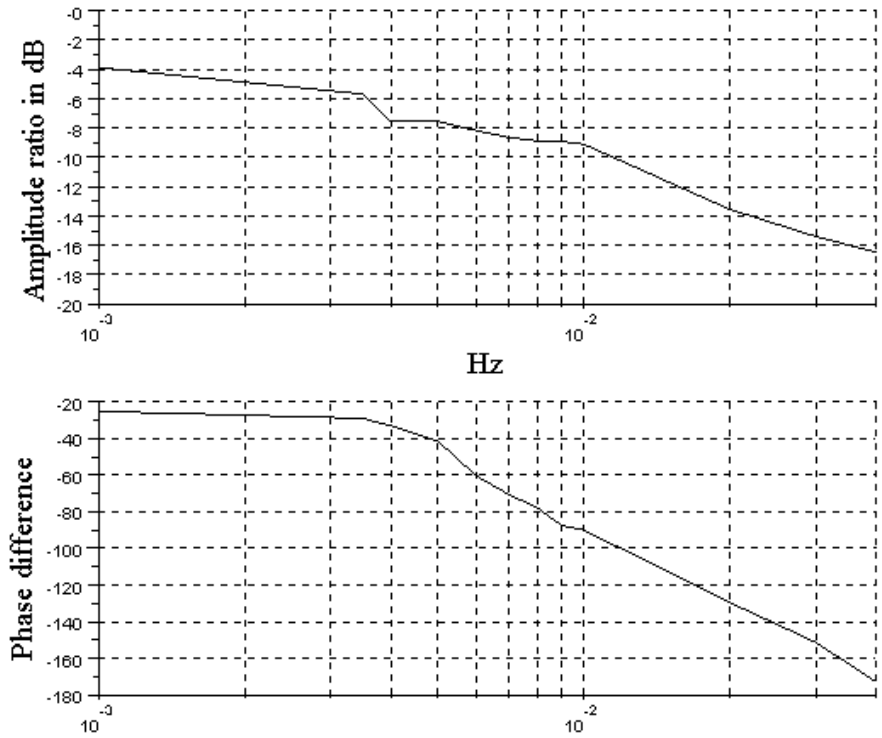

Figure 6.6: Bode plot obtained from the plant

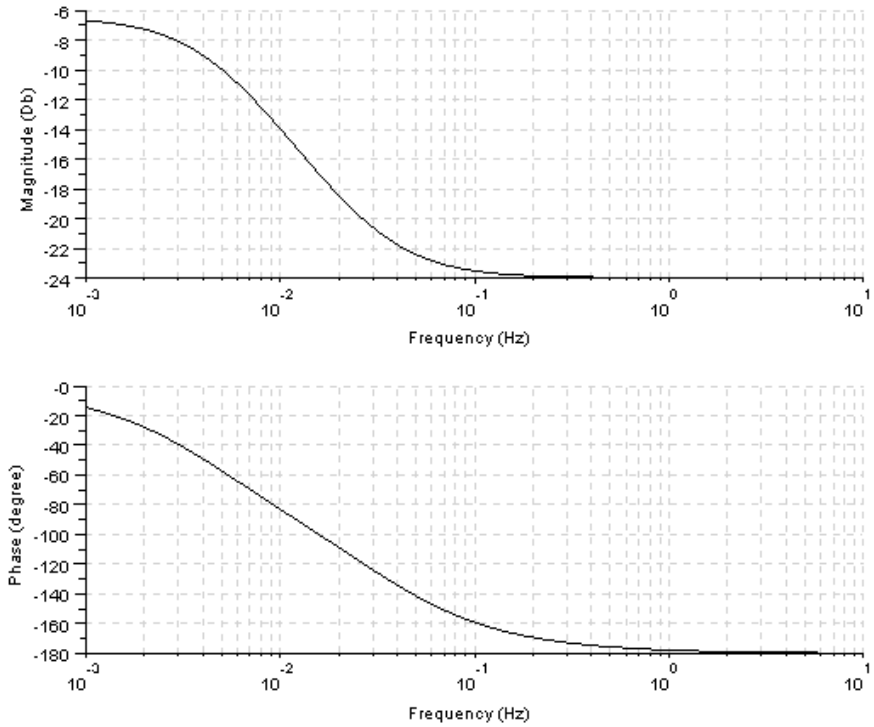

Figure 6.7: Bode plot obtained through plant's transfer function

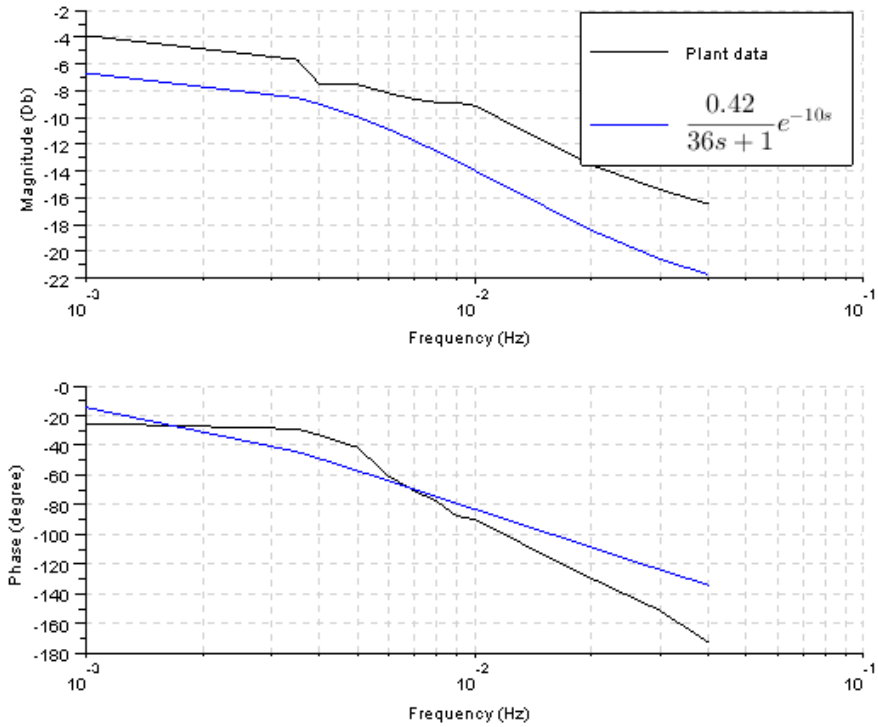

Figure 6.8: Comparison of Bode plots

6.4 Scilab Code

Scilab Code 6.1 `sine_test.sci`

```
mode(0)
function temp = sine_test(heat,fan)
    temp = comm(heat,fan);

    plotting([heat fan temp],[0 0 20 0],[100 100 40
        1000])

    m=m+1;
endfunction
```

Scilab Code 6.2 `sinetest.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits
   sampling_time m

// **********************
sampling_time=1;    // In seconds. Fractions are allowed
// **********************//
exec ("sinetest.sci");

ok = init();

   if ok~= []  // open xcos only if communication is
       through (ie reply has come from server)
       xcos('sinetest.xcos');
     else
     disp("NO NETWORK CONNECTION!");
     return
end
```

Scilab Code 6.3 `sinetest.sci`

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

```

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

endfunction
```

Scilab Code 6.4 `sine-analysis.sce`

```
mode(0);
filename= 'sine0035-local.txt';// Enter the data file name in single quotes
f=0.0035;// Enter the frequency
data6=fscanfMat(filename);
data7=data6(2:$,:);
exec('labelbode.sci');
T = data7(:,5); fan = data7(:,3);//T is time, fan is fan speed
u = data7(:,2)-data7(1,2); y = data7(:,4)-data7(1,4); // u is current, y is temperature


period=ceil(1/f);
p=length(u);
sampling = T(2)-T(1);// sampling time
sampling = sampling/1000;
index = round((period)/sampling);// calculating the duration of last cycle of waveform
times=T($-index:$);
temp = y($-index:$);// output for last cycle
heater = u($-index:$);// input for last cycle
[max_heater,pointer1] = max(heater);// determining max amplitude and index for last cycle of input (index is relative to last cycle)
[max_temp,pointer2] = max(temp);// determining max amplitude and index for last cycle of input (index is relative to last cycle)
```

```
pointer1 = pointer1 + (p-index);// conversion of index for input in terms of complete data period
pointer2 = pointer2 + (p-index);// conversion of index for output in terms of complete data period
Amplitude_ratio_dB = 20*log10(y(pointer2)/u(pointer1)) //To find gain in dB
Phase_difference = 360*f*(pointer1-pointer2)*sampling // phase difference in degrees
// Phase_difference = -((pointer1 - pointer2)/(1/f))*360


del_T = T-T(1);
del_T = del_T/1000;
plot2d(del_T,[u y]);
label('Plot of sine input in heater and the corresponding temperature profile',4,'Time (s)','Change in temperature and heater',4);
// legend(['Heater';'Temperature']);
```

Scilab Code 6.5 `label.sci`

```
// Updated (9-12-06), written by Inderpreet Arora
// Input arguments:title, xlabel, ylabel and their font sizes
function label(tname,tfont,labelx,labely,xyfont)
a = get("current_axes")
xtitle(tname,labelx,labely)
xgrid
t = a.title;
t.font_size = tfont; // Title font size
t.font_style = 2; // Title font style
t.text = tname;
u = a.x_label;
u.font_size = xyfont; // Label font size
u.font_style = 2; // Label font style
v = a.y_label;
v.font_size = xyfont; // Label font size
v.font_style = 2; // Label font style
```

```
// a.label_font_size = 3;
endfunction;
```

Scilab Code 6.6 `bodeplot.sce`

```
// bodeplot
exec('labelbode.sci');
x=[0.001,0.0035,0.004,0.005,0.006,0.007,...
0.008,0.009,0.01,0.02,0.03,0.04]; // Input frequency(Hz)
y=[-3.87,-5.67,-7.53,-7.53,-8.17,-8.64,...
-8.87,-8.90,-9.11,-13.55,-15.39,-16.47]; // Amplitude
    ratio(dB)
subplot(2,1,1);
plot2d(x,y,rect=[0.001,-20,0.04,0],logflag="ln");
xgrid();
y=[-25.2,-28.98,-33.11,-41.4,-60.48,-70.56,...
-77.76,-87.48,-90,-129.6,-151.2,-172.8]; // Phase
    difference(degree)
title = ''
label(title,4,'Hz','Amplitude ratio in dB ',4);
subplot(2,1,2);
plot2d(x,y,rect=[0.001,-180,0.04,-20],logflag="ln");
label(title,4,'','Phase difference',4);
subplot(2,1,2);
xgrid();

//s=poly(0,'s')
//h=syslin('c',(0.475/(124.827*s^2+57.26*s+1)))
//bode(h,0.001,0.04);
```

Scilab Code 6.7 `labelbode.sci`

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

function label(tname,tfont,labelx,labely,xyfont)
a = get("current_axes")
```

```
xtitle(tname,labelx,labely)
xgrid
t = a.title;
t.font_size = tfont; // Title font size
t.font_style = 2; // Title font style
t.text = tname;
u = a.x_label;
u.font_size = xyfont; //Label font size
u.font_style = 2; //Label font style
v = a.y_label;
v.font_size = xyfont; //Label font size
v.font_style = 2; //Label font style
// a.label_font_size = 3;
endfunction;
```

Scilab Code 6.8 `TFbode.sce`

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

Scilab Code 6.9 `comparison.sce`

```
s=poly(0,'s');
frq = [0.001,0.0035,0.004,0.005,0.006,0.007,...
0.008,0.009,0.01,0.02,0.03,0.04];// Input frequency(Hz)
dt=10;// delay time

tf=((0.475/(36*s+1))*((-dt/2)*s+1/(dt/2)*s+1));// transfer function using pade' approximation
h=syslin('c',tf);

[frq1,rep]=repfreq(h,frq);
[dB1,phi1]=dbphi(rep);
title = 'From actual plant data';
```

```
dB   = [-3.87,-5.67,-7.53,-7.53,-8.17,-8.64,...
 -8.87,-8.90,-9.11,-13.55,-15.39,-16.47]; // Amplitude
    ratio (dB)
phi  = [-25.2,-28.98,-33.11,-41.4,-60.48,...
 -70.56,-77.76,-87.48,-90,-129.6,-151.2,-172.8]; // Phase
     difference (degree)
bode([frq],[dB;dB1],[phi;phi1])
legend(['Plant data';'$\frac{0.42}{36s+1}e^{-10s}$'])

// transfer function using pade' approximation
```

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.

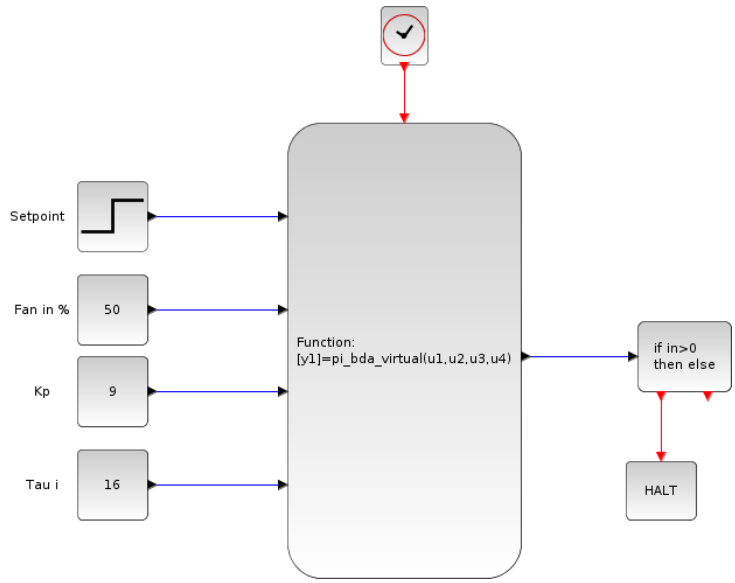

Figure 7.1: Xcos interface for this experiment

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,

$$P = K_p e(t) \tag{7.1}$$

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.

$$I = K_i \int_0^t e(t)dt \tag{7.2}$$

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.

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

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

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

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 exits 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,

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

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

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,

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

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

Finally, writing the equation for PID controller,

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

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

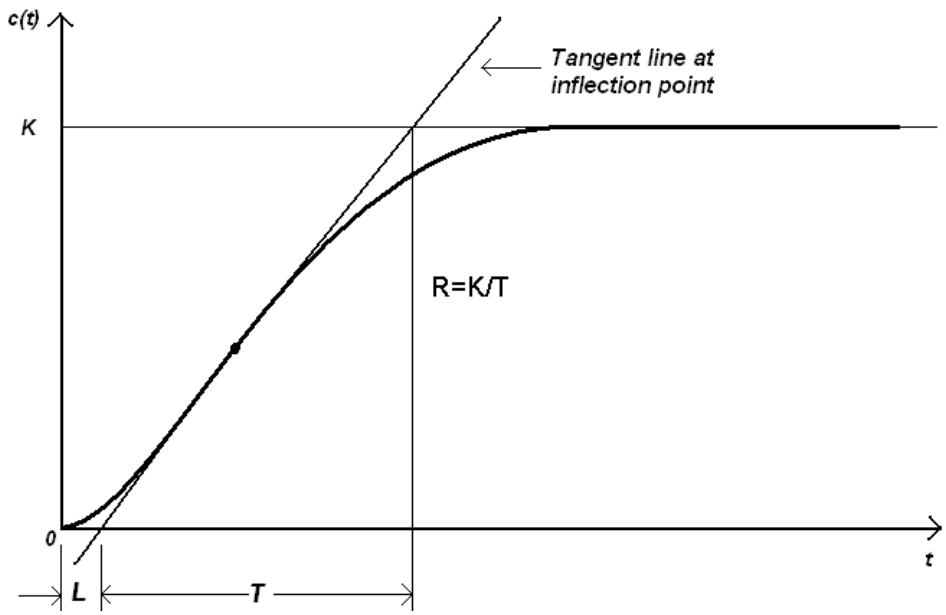

Figure 7.2: Reaction curve [5]

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,

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

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

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

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.

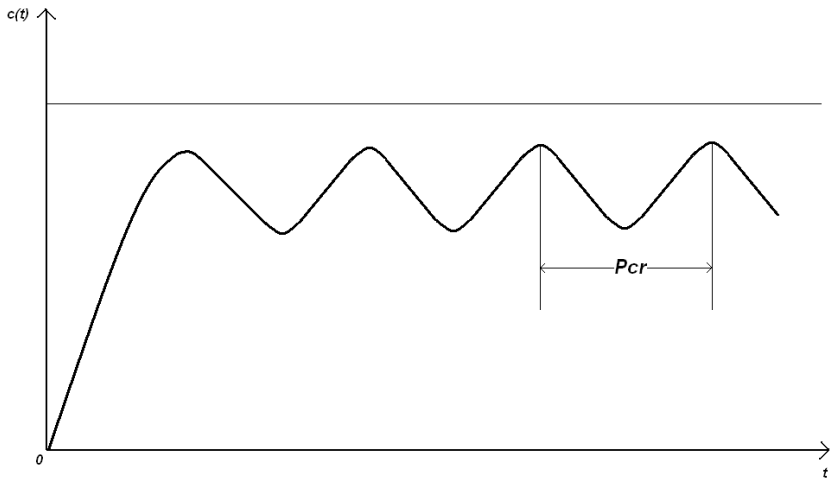

Figure 7.3: Ziegler-Nichols instability tuning method

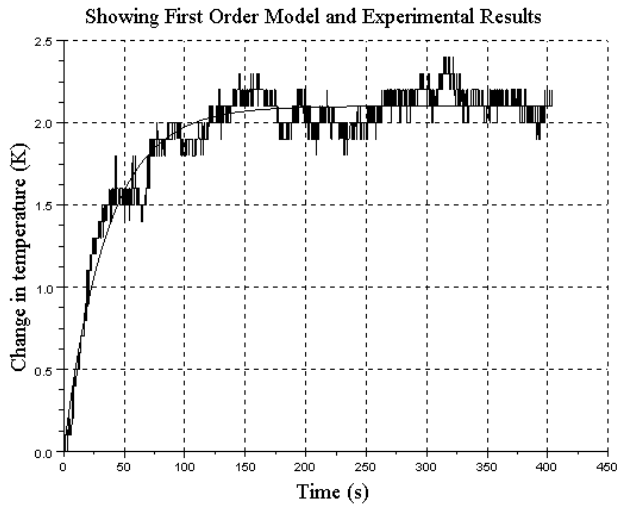

Figure 7.4: Refer to 'Step Test'experiment [6]

$$L = 6 \ \ s$$
$$T = 193 \ \ s$$

For PI

$$K = 6.031$$
$$\tau_i = 18$$

For PID

$$K = 8$$
$$\tau_i = 12$$
$$\tau_d = 3$$

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

7.3 PI Controller using Trapezoidal Approximation

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

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

On taking the Laplace transform,we obtain

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

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

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

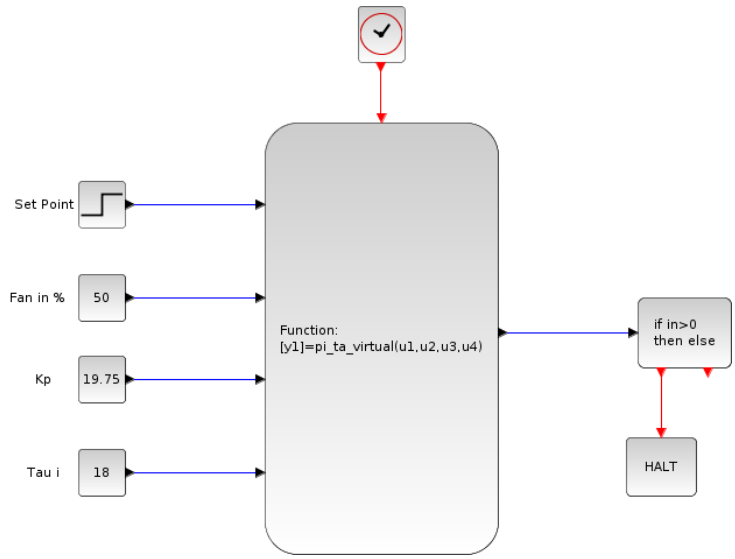

Figure 7.5: Xcos for PI controller available as `pi_ta_virtual.xcos`

On cross multiplying, we obtain

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

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

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

The PI controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) \tag{7.18}$$

where

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

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

7.3.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>pi_ta.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>pi_ta.xcos`
7. Step7: Same

The output of Xcos is shown in figure 7.6. 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.

7.3.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the PID controller experiment directory and double-click on the file `pi_ta_virtual.sce`. This will launch scilab and also open the file `pi_ta_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to

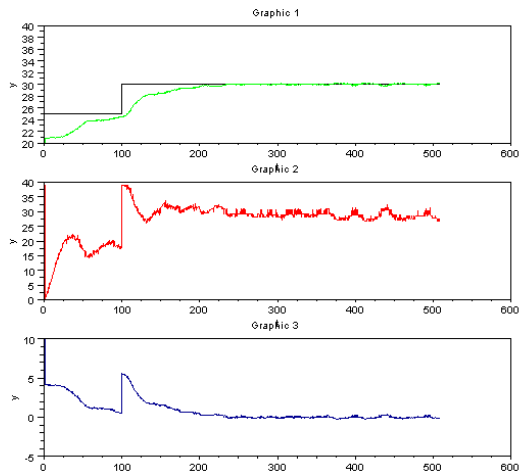

Figure 7.6: PI controller (Trapezoidal Approximation) output

`pid_controller` and then open the `pi_ta_virtual.sce` file in the scilab editor.

5. Step5: Same
6. Step6: Execute the file `pi_ta_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

7.4 Implementing PI Controller using Backward Difference Approximation

The PI controller in continuous time is given by

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

On taking the Laplace transform, we obtain

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

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

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

On cross multiplying, we get

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

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

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

The PI controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) \tag{7.26}$$

where

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

$$s_1 = -K \tag{7.28}$$

7.4.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`

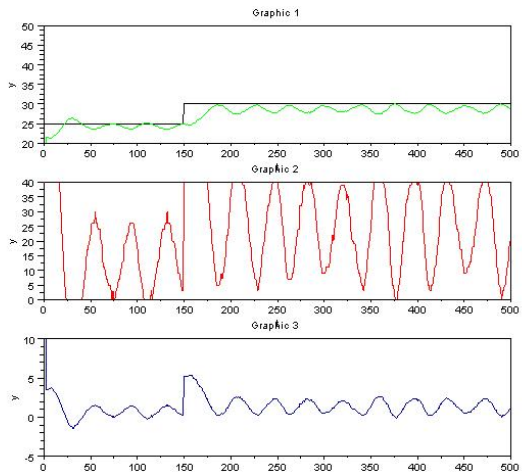

Figure 7.7: PI controller (Backward Difference Approximation) output

2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>pi_bda.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>pi_bda.xcos`
7. Step7: Same

The Xcos output is shown in figure 7.7. 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.

7.4.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.

2. Step2: Same
3. Step3: Same
4. Step4: Switch to the PID controller experiment directory and double-click on the file `pi_bda_virtual.sce`. This will launch scilab and also open the file `pi_bda_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `pid_controller` and then open the `pi_bda_virtual.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `pi_bda_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

7.5 Implementing PI Controller using Forward Difference Approximation

The PI controller in continuous time is given by

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

On taking the Laplace transform, we obtain

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

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

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

On cross multiplying, we get

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

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

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

The PI controller is usually written as

$$u(n) = u(n-1) + s_0 e(n) + s_1 e(n-1) \tag{7.34}$$

where

$$s_0 = K \tag{7.35}$$

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

7.5.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>pi_fda.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>pi_fda.xcos`
7. Step7: Same

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.

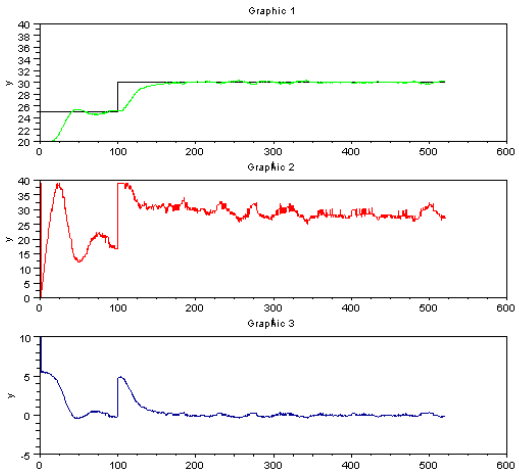

Figure 7.8: PI controller implementation (Forward Difference Approximation)

7.5.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the PID controller experiment directory and double-click on the file `pi_fda_virtual.sce`. This will launch scilab and also open the file `pi_fda_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `pid_controller` and then open the `pi_fda_virtual.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `pi_fda_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.

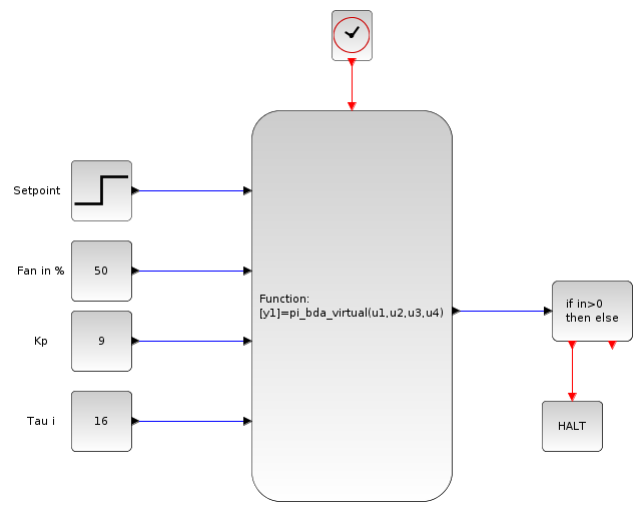

Figure 7.9: Xcos for PID controller available as `pid_bda_virtual.xcos`

7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

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

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

On taking the Laplace transform, we obtain

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

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

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

On cross multiplying, we obtain

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

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

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

The PID controller is usually written as

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

where

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

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

$$s_2 = K\left[\frac{\tau_d}{T_s}\right] \tag{7.45}$$

7.6.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`

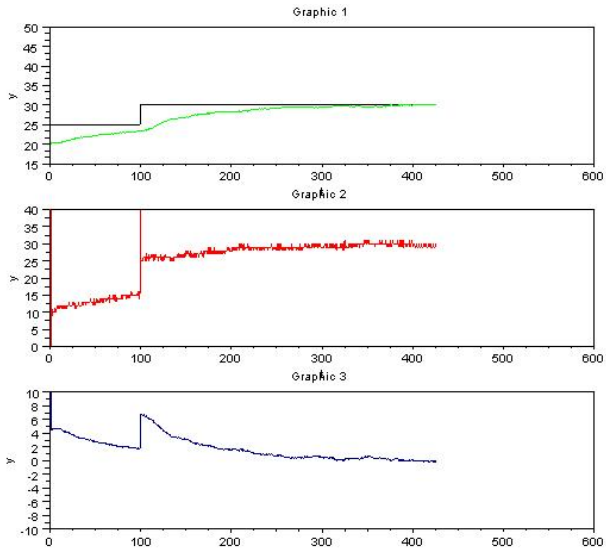

Figure 7.10: PID controller (Backward Difference Approximation) output

2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command
   `exec<space>pid_bda.sci`
6. Step6: Load Xcos code for ramp test using the command
   `exec<space>pid_bda.xcos`
7. Step7: Same

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.

7.6.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the PID controller experiment directory and double-click on the file `pid_bda_virtual.sce`. This will launch scilab and also open the file `pid_bda_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `pid_controller` and then open the `pid_bda_virtual.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `pid_bda_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

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

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

On taking the Laplace transform, we obtain

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

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

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

On cross multiplying, we obtain

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

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

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

The PID controller is usually written as

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

where

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

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

$$s_2 = K\frac{\tau_d}{T_s} \tag{7.54}$$

7.7.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`

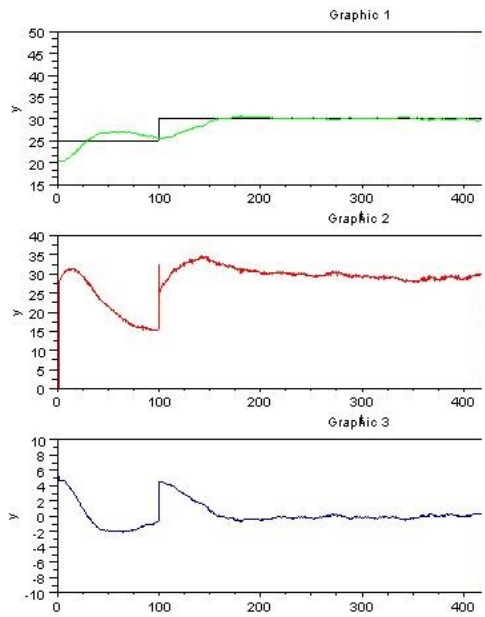

Figure 7.11: PID controller (TA - BDA) implementation

2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>pid_ta_bda.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>pid_ta_bda.xcos`
7. Step7: Same

The Xcos output is shown in figure 7.11. 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.

7.7.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the PID controller experiment directory and double-click on the file `pid_ta_bda_virtual.sce`. This will launch scilab and also open the file `pid_ta_bda_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `pid_controller` and then open the `pid_ta_bda_virtual.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `pid_ta_bda_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

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.

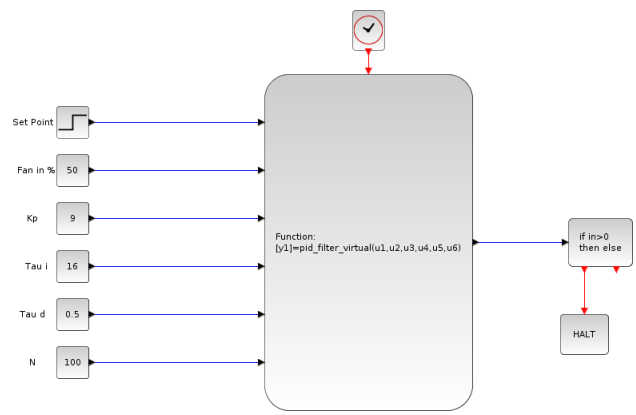

Figure 7.12: Xcos for PID controller with filtering available as `pidN_virtual.xcos`

PID filtered form is given by

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

where N is large number of the order of 100.

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

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

$$u(n) = K\left(1 + \frac{T_s}{\tau_i}\frac{1}{1 - z^{-1}} + \frac{N r_1(1 - z^{-1})}{1 + r_1 z^{-1}}\right) e(n) \tag{7.57}$$

where

$$r_1 = -\frac{\frac{\tau_d}{N}}{\frac{\tau_d}{N} + T_s} \tag{7.58}$$

On cross multiplying, we obtain

$$\begin{aligned}(1 - z^{-1})(1 + r_1 z^{-1})u(n) &= K[(1 - z^{-1})(1 + r_1 z^{-1}) \\ &+ \frac{T_s}{\tau_i}(1 + r_1 z^{-1}) + \frac{\tau_d}{T_s}(1 - z^{-1})^2]e(n)\end{aligned} \tag{7.59}$$

Simplifying and then by using shifting theorem, we obtain

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

Hence

$$\begin{aligned} u(n) &= r_1 u(n-2) - (r_1 - 1)u(n-1) \\ &+ s_0 e(n) + s_1 e(n-1) + s_2 e(n-2) \end{aligned} \tag{7.61}$$

where

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

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

$$s_2 = -K\left[r_1(1+N)\right] \tag{7.64}$$

7.8.1 Implementing locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>pid_filter.sci`

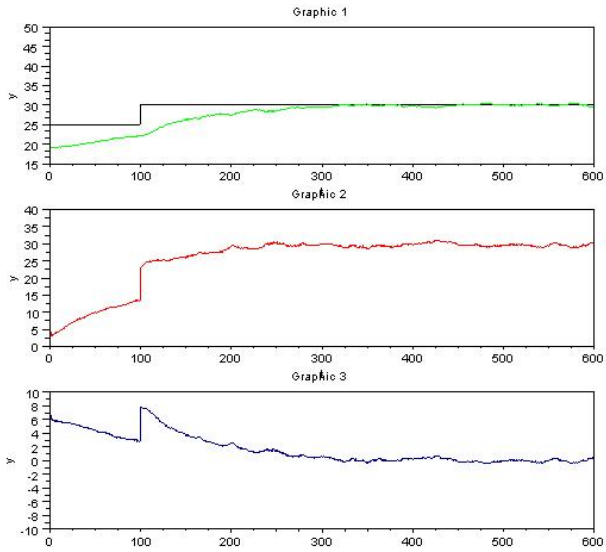

Figure 7.13: PID controller (with filtering) implementation

6. Step6: Load Xcos code for ramp test using the command `exec<space>pidN.xcos`

7. Step7: Same

The Xcos output is shown in figure 7.13. 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.

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

7.8.2 Implementing virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `pid_controller`. Open this directory.

2. Step2: Same

3. Step3: Same

4. Step4: Switch to the PID controller experiment directory and double-click on the file `pid_filter_virtual.sce`. This will launch scilab and also open the file `pid_filter_virtual.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `pid_controller` and then open the `pid_filter_virtual.sce` file in the scilab editor.

5. Step5: Same

6. Step6: Execute the file `pid_filter_virtual.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.

7. Step7: Execute the PI controller xcos diagram.

8. Step8: Same

7.9 Scilab Code

7.9.1 Scilab code for serial communication

Scilab Code 7.1 ser_init.sci used for serial communication

```
mode(0)
global filename m
//**Sampling Time**//
sampling_time = 1;
//////*****/////////
m=1;

port1 = '/dev/ttyUSB0'; // For linux users
port2 = 'COM2'; // For windows users

res=init([port1 port2]);
disp(res)
```

7.9.2 Scilab code for PI controller

Scilab Code 7.2 pi_ta.sci

```
mode(0)
function temp = pi_ta(setpoint,fan,K,Ti)
    global heatdisp fandisp tempdisp setpointdisp
        sampling_time m name temp heat_in fan_in C0
        u_old u_new e_old e_new

Ts=sampling_time;
e_new = setpoint - temp;

S0=K(1+Ts/(2*Ti));
S1=K*(-1+(Ts/(2*Ti)));
u_new = u_old+(S0*e_new)+(S1*e_old);


u_old = u_new;
e_old = e_new;

heat = u_new;
    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
        100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 7.3 pi_bda.sci

```
// global temp heat fan sampling_time m heatdisp
    fandisp tempdisp x name Ts

mode(0)
function temp = pi_bda(setpoint,fan,K,Ti)
```

```
    global heatdisp fandisp tempdisp setpointdisp
        sampling_time m name temp heat_in fan_in C0
        u_old u_new e_old e_new

Ts = sampling_time;
e_new = setpoint - temp;


S0=K(1+(Ts/Ti));
S1=-K;



u_new = u_old+ S0*e_new+ S1*e_old;


u_old = u_new;
e_old = e_new;

heat = u_new;
    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
        100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 7.4 pi_fda.sci

```
mode(0)
function temp = pi_fda(setpoint,fan,K,Ti)
    global heatdisp fandisp tempdisp setpointdisp
        sampling_time m name temp heat_in fan_in C0
        u_old u_new e_old e_new

Ts=sampling_time;
e_new = setpoint - temp;
```

```


S0=K*(1+((Ts/Ti)));
S1=-K;
u_new = u_old+(S0*e_new)+(S1*e_old);

u_old = u_new;
e_old = e_new;

heat = u_new;

    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
       100 40 1000])

    m=m+1;
endfunction
```

7.9.3 Scilab code for PID controller

Scilab Code 7.5 pid_bda.sci

```
mode(0)

function [temp] = pid_bda(setpoint,fan,K,Ti,Td)
global temp heat_in fan_in C0 u_old u_new e_old e_new
   e_old_old

global heatdisp fandisp tempdisp setpointdisp
   sampling_time m name

e_new = setpoint - temp;

Ts=sampling_time;

S0=K*(1+(Ts/Ti)+(Td/Ts));
S1=K*(-1-((2*Td)/Ts));
```

```
S2=K*(Td/Ts);

u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;

u_old = u_new;
e_old_old = e_old;
e_old = e_new;


heat = u_new;
    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
        100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 7.6 pid_ta_bda.sci

```

mode(0)

function [temp] = pid_ta_bda(setpoint,fan,K,Ti,Td)
global temp heat_in fan_in C0 u_old u_new e_old e_new
    e_old_old

global heatdisp fandisp tempdisp setpointdisp
    sampling_time m name

e_new = setpoint - temp;

Ts=sampling_time;

S0=K*(1+(Ts/(2*Ti))+(Td/Ts));
S1=K*(-1+(Ts/(2*Ti))-(2*Td/Ts));
S2=(K*Td/Ts);
u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;
```

```

u_old = u_new;
e_old_old = e_old;
e_old = e_new;


heat = u_new;

    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
        100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 7.7 pid_filter.sci

```
mode(0)

function temp = pid(setpoint,fan,K,Ti,Td,N)
    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

Ts = sampling_time;
e_new = setpoint - temp;

r1=-((Td/N)/((Td/N)+Ts));

S0=K*(1+(Ts/Ti)-(N*r1));
S1=K*((r1*(1+(Ts/Ti)+(2*N)))-1);
S2=-K*r1*(1+N);

u_new = r1*u_old_old-(r1-1)*u_old + S0*e_new + S1*
    e_old + S2*e_old_old;

u_old_old = u_old;
```

```
u_old = u_new;
e_old_old = e_old;
e_old = e_new;


heat = u_new;

    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100 100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 7.8 pid_bda_virtual.sce

```
mode(0);
// For scilab 5.1.1 or lower version users, use scicos command to open scicos diagrams instead of xcos

global fdfh fdt fncr fncw m err_count y limits sampling_time m

// ***********************
sampling_time=1;    // In seconds. Fractions are allowed
// ************************//

exec ("pid_bda_virtual.sci");

ok = init();

   if ok~= []  // open xcos only if communication is through (ie reply has come from server)
      xcos('pid_bda_virtual.xcos');
     else
     disp("NO NETWORK CONNECTION!");
     return
```

```
end
```

Scilab Code 7.9 pid_bda_virtual.sci

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

function [stop] = pid_bda_virtual(setpoint,fan,K,Ti,Td)

global temp heat C0 u_old u_new e_old e_new fdfh fdt fncr fncw m err_count stop q heatdisp fandisp tempdisp setpointdisp limits m x sampling_time e_old_old

e_new = setpoint - temp;


Ts=1;
S0=K*(1+(Ts/Ti)+(Td/Ts));
S1=K*(-1-((2*Td)/Ts));
S2=K*(Td/Ts);

u_new = u_old + S0*e_new + S1*e_old + S2*e_old_old;

heat=u_new;

u_old = u_new;
e_old_old = e_old;
e_old = e_new;

```

```
    [stop,temp] = comm(heat,fan);// Never edit this line
    plotting([heat fan temp setpoint],[0 0 30 0],[100 100 50 1000])

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.

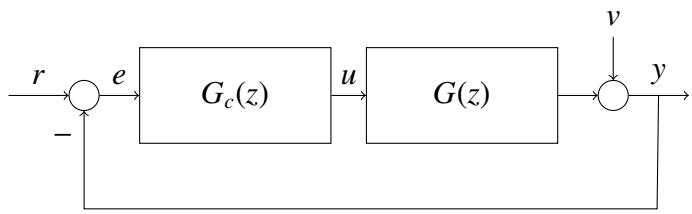

Figure 8.1: Feed back control strategy

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

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

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

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

Let,

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

Therefore,

$$y(n) = T(z)r(n) + S(z)v(n) \tag{8.4}$$

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

$$u(n) = r(n)G_f - y(n)G_b \tag{8.5}$$

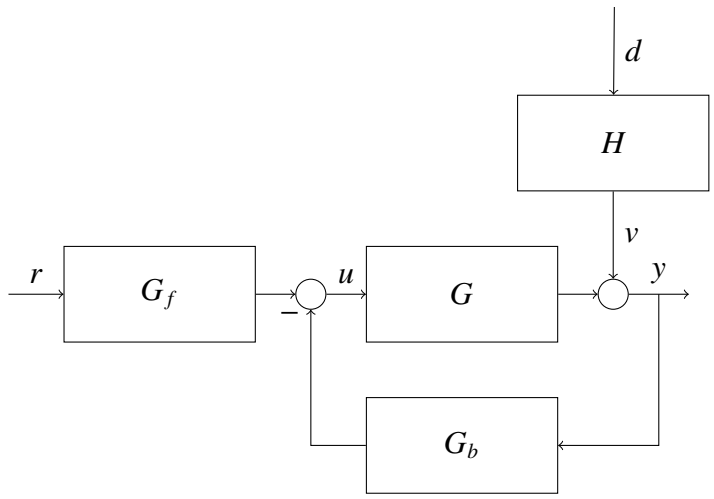

Figure 8.2: 2DOF feed back control strategy

Let

$$G_b = \frac{S_c}{R_c}, G_f = \frac{T_c}{R_c} \tag{8.6}$$

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

We get

$$R_c(z)u(n) = T_c(z)r(n) - S_c(z)y(n) \tag{8.7}$$

Consider a plant whose model is given by

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

Substituting equation 8.7 in equation 8.8, we get

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

Solving for $y(n)$,

$$\left(\frac{R_c A + z^{-k}BS_c}{R_c}\right)y(n) = z^{-k}\frac{BT_c}{R_c}r(n) + v(n) \tag{8.10}$$

This can also be written as

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

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

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

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:

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

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,

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

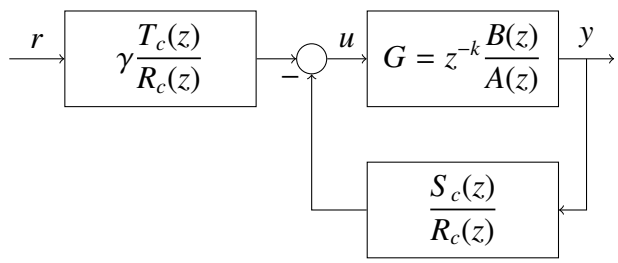

Figure 8.3: 2-DOF pole placement controller

Simplifying the block diagram shown in figure 8.3 yields

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

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

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

We can expect some cancelations between the numerator and the denominator polynomials in the LHS, thereby making $degB_r < degB$. 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

$$A = A^g A^b, B = B^g B^b \tag{8.17}$$

We also define R_c, S_c and T_c as

$$R_c = B^g R_1 \tag{8.18}$$
$$S_c = A^g S_1 \tag{8.19}$$
$$T_c = A^g T_1 \tag{8.20}$$

Hence, equation 8.16 becomes

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

After cancelling out the common factors, we obtain

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

We obtain,

$$B^b T_1 = B_r \tag{8.23}$$

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

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

$$B^b = B_r \tag{8.25}$$

The expression for γ now becomes

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

and the desired closed loop transfer function will be

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

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

$$G(s) = \frac{0.42}{35.61s + 1} \tag{8.28}$$

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

After discretization with sampling time = 1 s, we obtain

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

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

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

[1] Go 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,

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

Hence,

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

is 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 calculate 2DOF parameters using scilab

The procedure explained here is applicable to both virtual as well as local experiments. The following steps must be executed properly to calculate the controller parameters.

1. Change the directory to `2dof_controller`.
2. Execute the command `getd dc/scilab` in the scilab console.
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.

[2] All the Scilab codes are given at the end of this chapter in the section 8.6

8.4 Implementing 2DOF controller locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `2dof_controller`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load 2DOF controller function by executing command `exec<space>twodof.sci`. Make sure the correct order of control law is choosen in this file as explained in section 8.3.1
6. Step6: Load Xcos code for 2DOF controller using the command `exec<space>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.
7. Step7: Same

8.4.1 Implementing 2-DOF Controller on SBHS, Virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `2dof_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the 2dof controller experiment directory and double-click on the file `twodof.sce`. This will launch scilab and also open the file `twodof.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `2dof_controller` and then open the `twodof.sce` file in the scilab editor.

5. Step5: Same
6. Step6: Execute the file `twodof.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the PI controller xcos diagram.
8. Step8: Same

8.5 Performing pure simulation of 2DOF controller

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 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.

8.6 Scilab Code for Local Experiment

Scilab Code 8.1 `twodof_para.sce`

```
mode(0)
s=%s;
z=%z;
global Rc Sc Tc gamm
// TFcont = syslin('c',-280.14/((s-31.32)*(s+100)*(s+31.32)));
// TFcont = syslin('c',0.667/((73.5*s+1)*(1*s+1)))//second order
TFcont = syslin('c',0.668/(75.013*s+1))// first order
SScont = tf2ss(TFcont);
```

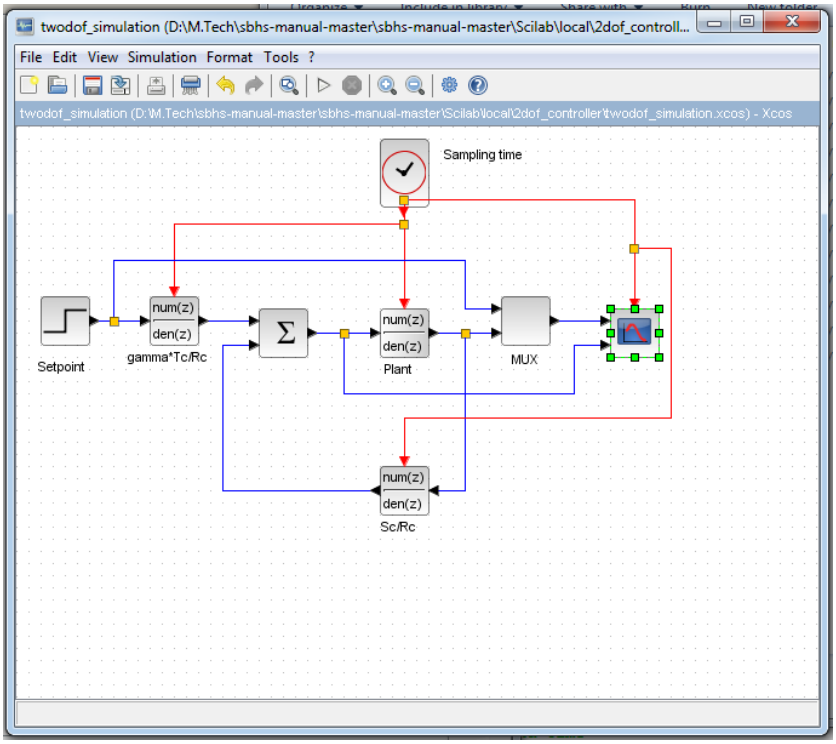

Figure 8.4: Xcos diagram for simulating 2-DOF controller

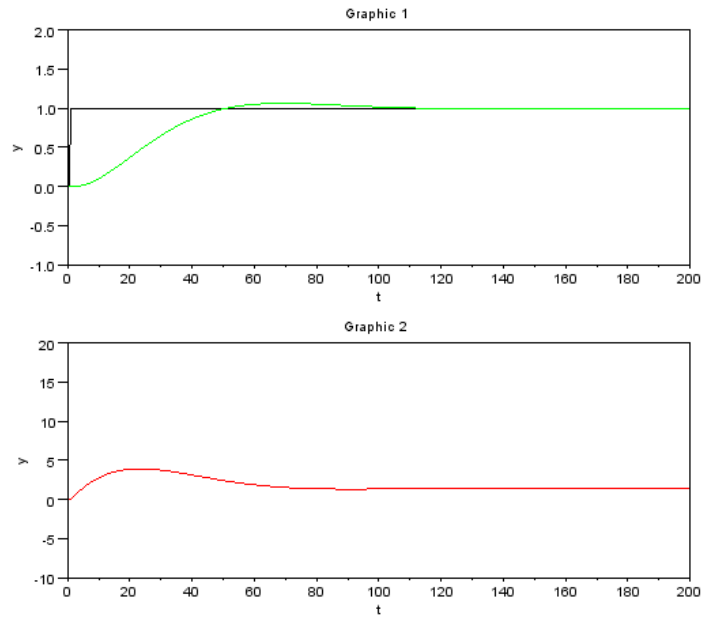

Figure 8.5: Simulation results after executing `twodof_simulation.xcos`

```
// TFdisc=ss2tf(SScont);
Ts = 0.5;
[B,A,k] = myc2d(SScont,Ts);

// polynomials are returned
[Ds,num,den] = ss2tf(SScont);
num = clean(num); den = clean(den);

// Transient specifications
rise = 35; epsilon = 0.05;
phi = desired(Ts,rise,epsilon);

// Controller design
Delta = [1 -1];
[Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi); // with integral

// Setting up simulation parameters for basic.cos
st = 0.0001; // desired change in h, in m.
```

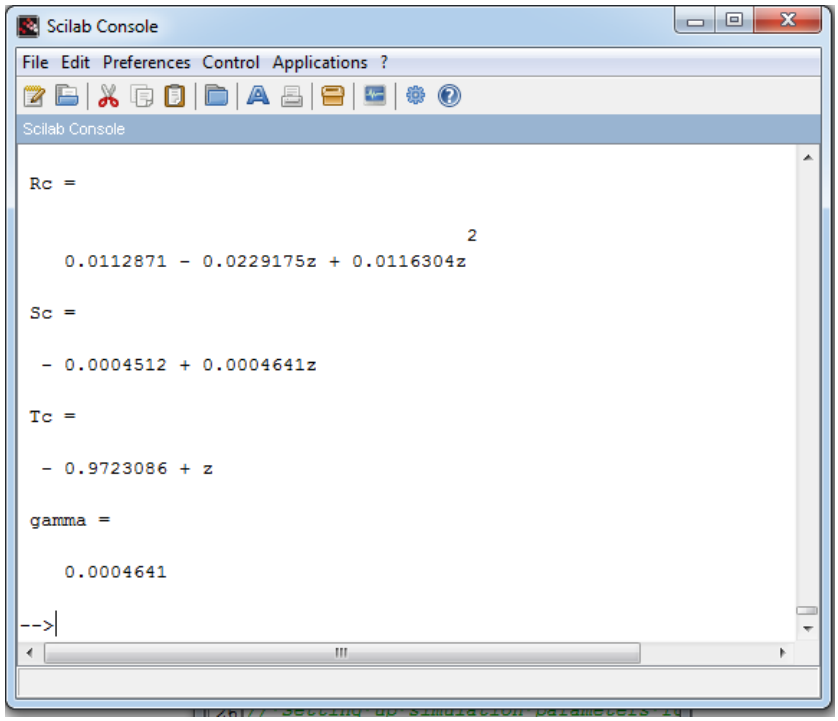

Figure 8.6: Scilab output for `twodof_para.sce`

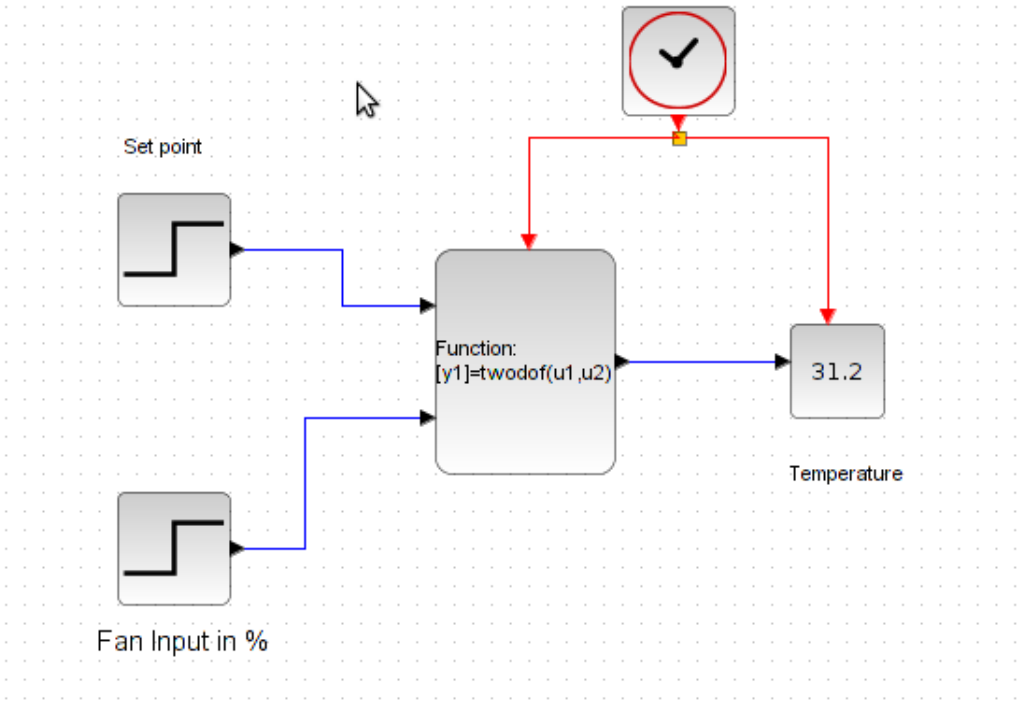

Figure 8.7: Xcos diagram for 2-DOF controller

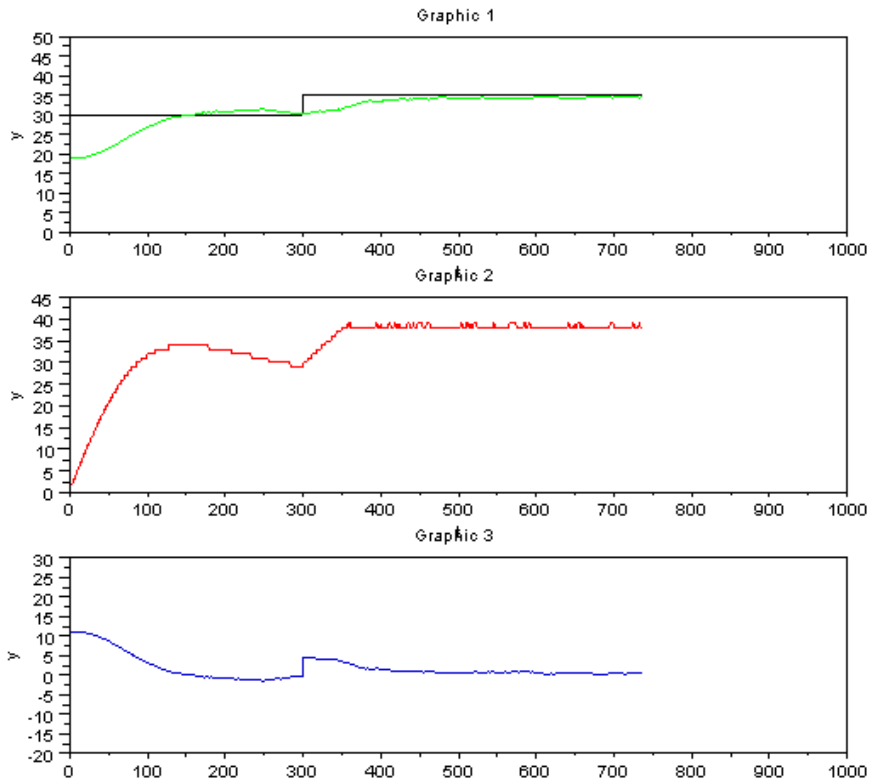

Figure 8.8: Implementation of 2-DOF controller

```
t_init = 0; // simulation start time
t_final = 0.5; // simulation end time

// Setting up simulation parameters for c_ss_cl.cos
N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;

[Tc1,Rc1] = cosfil_ip(Tc,Rc); // Tc/Rc
[Sc2,Rc2] = cosfil_ip(Sc,Rc); // Sc/Rc

[Bp] = cosfil_ip(B,1);
[Ap] = cosfil_ip(A,1);

[Tcp1,Tcp2] = cosfil_ip(Tc,1); // Tc/1
[Np,Rcp] = cosfil_ip(N,Rc); // 1/Rc
[Scp1,Scp2] = cosfil_ip(Sc,1); // Sc/1
[Cp,Dp] = cosfil_ip(C,D); // C/D

// Rc1=Rc(1); Rc2=Rc(2); Rc3=Rc(3); Rc4=Rc(4);
// Sc1=Sc(1); Sc2=Sc(2);
// Sc3=Sc(3);
// Tc1=Tc(1); Tc2=Tc(2);
// Tc3=Tc(3);
disp(Rcp,'Rc =')
disp(Scp1,'Sc =')
disp(Tcp1,'Tc =')
disp(gamm,'gamma =')



// u_old_old = 1;
// u_old = 1;
// r_old_old = 1;
// r_old = 1;
// y_old_old = 1;
// y_old = 1;
```

Scilab Code 8.2 `twodof.sci`

```
mode(0)
function [temp] = twodof(setpoint,fan)
global temp heat_in fan_in C0 u_old u_new e_old e_new
    e_old_old

global heatdisp fandisp tempdisp setpointdisp
    sampling_time m name

global temp u_old_old u_old r_old_old r_old y_old_old
    y_old u_new heat r_new y_new

Ts=sampling_time;

r_new = setpoint;
y_new = temp;
et = setpoint-temp;

// u_new = (1/Rc(1))*(gamm*Tc(1)*r_new + gamm*Tc(2)*
    r_old + r_old_old*Tc(3)*gamm - Sc(1)*y_new -Sc(2)*
    y_old - Sc(3)*y_old_old - Rc(2)*u_old - Rc(3)*
    u_old_old);// second order control law

u_new = (1/Rc(1))*(gamm*Tc(1)*r_new + gamm*Tc(2)*r_old
    -Sc(1)*y_new -Sc(2)*y_old-Rc(2)*u_old - Rc(3)*
    u_old_old);// first order control law

u_old_old = u_old;
u_old = u_new;
r_old_old=r_old;
r_old = r_new;
y_old_old=y_old;
y_old = y_new;


heat = u_new;
     temp = comm(heat,fan);

     plotting([heat fan temp setpoint],[0 0 20 0],[100
```

```
        100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 8.3 `start.sce`

```
getd ../common_files/
getd dc/scilab
exec ../common_files/loader.sce

exec ser_init.sce

exec twodof_para.sce
exec twodof.sci

xcos twodof.xcos
```

Scilab Code 8.4 `cindep.sci`

```
// Updated ----No change
// function b = cindep( S,gap)
// used in XD + YN = C. all rows except the last of
    are assumed to
// be independent. The aim is to check if the last
    row is dependent on the
// rest and if so how. The coefficients of dependence
     is sent in b
function b = cindep( S,gap)

if argn(2) == 1
        gap = 1.0e8;
end
eps = 2.2204e-016;
[rows,cols] = size(S);
if rows > cols
   ind = 0;
else
```

```
    sigma = svd(S);
    len = length(sigma);
    if (sigma(len)/sigma(1) <= (eps*max(i,cols)))
        ind = 0;                    // not independent
    else
        if or(sigma(1:len-1) ./sigma(2:len)>=gap)
            ind = 0;                // not dependent
        else
            ind = 1;                // independent
        end
    end
end
if ind
    b = [];
else
    b = S(rows,:)/S(1:rows-1,:);
    b = makezero(b,gap);
end
endfunction
```

Scilab Code 8.5 `clcoef.sci`

```
// Updated -----No change
// H. Kwakernaak, July, 1990
// Modified by Kannan Moudgalya in Nov. 1992

function [P,degP] = clcoef(Q,degQ)

[rQ,cQ] = polsize(Q,degQ);

if and(and(Q==0))
  P = zeros(rQ,cQ);
  degP = 0;
else
  P = Q; degP = degQ; rP = rQ; cP = cQ;
  j = degP+1;
  while j >= 0
  X = P(:,(j-1)*cP+1:j*cP)
```

```
    if max(sum(abs(X'))) < (1e-8)*max(sum(abs(P)))
        P = P(:,1:(j-1)*cP);
        degP = degP-1;
    else
        j = 0;
    end
    j = j-1;
  end
end
endfunction
```

Scilab Code 8.6 `colsplit.sci`

```
// colsplit
// The command
//   [P1,degP1,P2,degP2] = colsplit(P,degP,p1,p2)
// produces two polynomial matrix P1 and P2. P1 consists of the first
// p1 columns of P and P2 consists of the remaining p2 columns of P.

// H. Kwakernaak, July, 1990


function [P1,degP1,P2,degP2] = colsplit(P,degP,p1,p2)

if isempty(P)
   P1 = []; P2 = [];
   degP1 = 0; degP2 = 0;
   return;
end

[rP,cP] = polsize(P,degP);
if p1 < 0 | p1 > cP | p2 < 0 | p2 > cP | p1+p2 ~= cP
   error('colsplit: Inconsistent numbers of columns');
end
rP1 = rP; rP2 = rP; cP1 = p1; cP2 = p2;
degP1= degP; degP2 = degP;
```

```

if p1 == 0
   P1 == []; P2 = P;
elseif p2 == 0
   P1 = P; P2 = [];
else
   P1 = zeros(rP1,(degP1+1)*cP1); P2 = zeros(rP2,(degP2+1)*cP2);
   for i = 1:degP+1
       P1(:,(i-1)*cP1+1:i*cP1) = P(:,(i-1)*cP+1:(i-1)*cP+cP1);
       P2(:,(i-1)*cP2+1:i*cP2) = P(:,(i-1)*cP+cP1+1:i*cP);
   end
end
endfunction;
```

Scilab Code 8.7 `cosfil_ip.sci`

```
// Updated(31-7-07)
// Input arguments are numerator and denominator
// polynomials' co efficients in ascending
// powers of z^-1

// Scicos blocks need input polynomials
// with positive powers of z

function [nume,deno] = cosfil_ip(num,den)

[Nn,Nd] = polyno(num,'z');
[Dn,Dd] = polyno(den,'z');
nume = Nn*Dd;
deno = Nd*Dn;

endfunction;
```

Scilab Code 8.8 `indep.sci`

```
// Updated ----No change
// function b = indep(S,gap)
// determines the first row that is dependent on the
    previous rows of S.
// The coefficients of dependence is returned in b
function b = indep( S,gap)

if argn(2) == 1
        gap = 1.0e8;
        end
[rows,cols] = size(S);
ind = 1;
i = 2;
eps = 2.2204e-016;
while ind & i <= rows
      sigma = svd(S(1:i,:));
      len = length(sigma);
      if(sigma(len)/sigma(1) < (eps*max(i,cols)))
        ind =0;
      else
        shsig = [sigma(2:len);sigma(len)];
        if or( (sigma ./shsig) > gap)
           ind = 0;
        else
           ind = 1;
           i = i+1;
        end
      end

end
if ind
        b =[];

else
        c = S(i,:)/S(1:i-1,:);
        c = makezero(c,gap);
        b = [-c 1];
end
```

```
endfunction
```

Scilab Code 8.9 `left_prm.sci`

```
// function [B,degB,A,degA,Y,degY,X,degX] = ...
// left_prm(N,degN,D,degD,job,gap)
//
// does three different things according to integers that 'job' takes
// job = 1.
// this is the default. It is always done for all jobs.
//                -1                                        -1
// Given ND    , returns coprime B and A where ND    = A B
// It is enough if one sends the first four input arguments
// If gap is required to be sent, then one can send either 1 or a null
// entry for job
// job = 2.
// first solve for job = 1 and then solve XA + YB = I
// job = 3.
// used in solving XD + YN = C
// after finding coprime factorization, data are returned
//
// convention: the variable with prefix deg stand for degrees
// of the corresponding polynomial matrices
//
// input:
// N: right fraction numerator polynomial matrix
// D: right fraction denominator polynomial matrix
// N and D are not neccessarily coprime
// gap: variable used to zero entries; default value is 1.0e+8
```

```
//
// output
// b and A are left coprime num. and den. polynomial matrices
// X and Y are solutions to Aryabhatta identity, only for job = 2

function [B,degB,A,degA,Y,degY,X,degX] = left_prm(N, degN,D,degD,job,gap)
if argn(2) == 4 | argn(2) == 5
    gap = 1.0e8 ;
end
// pause
if argn(2) == 4,
job = 1; end
[F,degF] = rowjoin(D,degD,N,degN);
[Frows,Fbcols] = polsize(F,degF);          // Fbcols = block columns
Fcols = Fbcols * (degF+1) ;                // actual columns of F
T1 = []; pr =[]; degT1 = 0; T1rows = 0; shft = 0;
S=F; sel = ones(Frows,1); T1bcols =1;
abar = (Fbcols + 1):Frows;                 // a_super_bar of B-C.Chang
while isempty(T1) | T1rows < Frows - Fbcols
        Srows = Frows*T1bcols; // max actual columns of result
        [T1,T1rows,sel,pr] = ...
              t1calc(S,Srows,T1,T1rows,sel,pr,Frows, Fbcols,abar,gap);
        [T1rows,T1cols] = size(T1);
        if T1rows < Frows - Fbcols
          T1 = [T1 zeros(T1rows,Frows)];
          T1bcols = T1bcols + 1;           // max. block columns of result
          degT1 = degT1 + 1;               // degree of result
          shft = shft +Fbcols;
```

```
        S = seshft(S,F,shft);
        sel = [sel;sel(Srows-Frows+1:Srows)];
        rowvec = (T1bcols-1)*Frows+(Fbcols+1):T1bcols
            * Frows;
        abar = [abar rowvec];              // A_super_bar
            of B-C.chang
    end
end

[B,degB,A,degA] = colsplit(T1,degT1,Fbcols,Frows-
    Fbcols);
[B,degB] = clcoef(B,degB);
B = -B;
[A,degA] = clcoef(A,degA);
// pause
if job == 2
    S = S(mtlb_logical(sel),:);
                                    // columns
    [redSrows,Scols] = size(S);
    C = [eye(Fbcols,Fbcols) zeros(Fbcols,Scols-
        Fbcols)];    // append with zeros
    T2 = C/S;
    T2 = makezero(T2,gap);
    T2 = move_sci(T2,find(sel),Srows);
    [X,degX,Y,degY] = colsplit(T2,degT1,Fbcols,Frows
        - Fbcols);
    [X,degX] = clcoef(X,degX);
    [Y,degY] = clcoef(Y,degY);
elseif job == 3
    Y = S;
    degY = sel;
    X = degT1;
    degX = Fbcols;
else
    if job ~= 1
        error('Message from left_prm:no legal job
            number specified')
    end
```

```
end
endfunction
```

Scilab Code 8.10 `makezero.sci`

```
// Updated
// function B = makezero(B,gap)
// where B is a vector and gap acts as a tolerance

function B = makezero(B,gap)

if argn(2) == 1
    gap = 1.0e8;
end
temp = B(find(B));          // non zero entries of B
temp = -gsort(-abs(temp)); // absolute values sorted
    in descending order
len = length(temp);
ratio = temp(1:len-1) ./temp(2:len); // each ratio >1
min_ind = min(find(ratio>gap));
if ~isempty(min_ind)
    our_eps = temp(min_ind+1);
    zeroind = find(abs(B)<=our_eps);
    B(zeroind) = zeros(1,length(zeroind));
end
endfunction
```

Scilab Code 8.11 `move_sci.sci`

```
// function result = move_sci(b,nonred,max_sci)
// Moves matrix b to matrix result with the
    information on where to move,
// decided by the indices of nonred.
// The matrix result will have as many rows as b has
    and max number of columns.
// b is augumented with zeros to have nonred number of
     columns;
```

```
// The columns of b put into those of result as decided by nonred.

function result = move_sci(b,nonred,max_sci)
[brows,bcols] = size(b);
b = [b zeros(brows,length(nonred)-bcols)];
result = zeros(brows,max_sci);
result(:,nonred') = b;
endfunction
```

Scilab Code 8.12 `polisize.sci`

```
// Updated ---- No change
// function [rQ,cQ] = polsize(Q,degQ)
// FUNCTION polsize TO DETERMINE THE DIMENSIONS
// OF A POLYNOMIAL MATRIX
//
// H. Kwakernaak, August, 1990

function [rQ,cQ] = polsize(Q,degQ)

[rQ,cQ] = size(Q); cQ = cQ/(degQ+1);
if abs(round(cQ)-cQ) > 1e-6
   error('polsize: Degree of input inconsistent with number of columns');
else
   cQ = round(cQ);
end
endfunction
```

Scilab Code 8.13 `polyno.sci`

```
// Updated(1-8-07)
// Operations:
// Polynomial definition
// Flipping of coefficients
// Variable ------- passed as input argument (either 's' or 'z')
```

```
// Both num and den are used mostly used in scicos files ,
// to get rid of negative powers of z

// Polynomials with powers of s need to
// be flipped only

function [polynu,polyde] = polyno(zc,a)
zc = clean(zc);
polynu = poly(zc(length(zc):-1:1),a,'coeff');
  if a == 'z'
  polyde = %z^(length(zc) - 1);
  else
  polyde = 1;
  end

// Scicos(4.1) Filter block shouldn't have constant/constant
  if type(polynu)==1 & type(polyde)==1
    if a == 'z'
      polynu = %z; polyde = %z;
    else
      polynu = %s; polyde = %s;
    end;
  end;

endfunction
```

Scilab Code 8.14 `rowjoin.sci`

```
// Updated --------No change
// function [P,degP] = rowjoin(P1,degP1,P2,degP2)
// MATLAB FUNCTION rowjoin TO SUPERPOSE TWO POLYNOMIAL
// MATRICES

// H. Kwakernaak, July, 1990

function [P,degP] = rowjoin(P1,degP1,P2,degP2)
```

```

[rP1,cP1] = polsize(P1,degP1);
[rP2,cP2] = polsize(P2,degP2);
if cP1 ~= cP2
  error('rowjoin: Inconsistent numbers of columns');
end

rP = rP1+rP2; cP = cP1;
if degP1 >= degP2
   degP = degP1;
else
   degP = degP2;
end

if isempty(P1)
   P = P2;
elseif isempty(P2)
   P = P1;
else
   P = zeros(rP,(degP+1)*cP);
   P(1:rP1,1:(degP1+1)*cP1) = P1;
   P(rP1+1:rP,1:(degP2+1)*cP2) = P2;
end
endfunction
```

Scilab Code 8.15 `seshft.sci`

```
// Updated --------No change
//  function C = seshft(A,B,N)
// given A and B matrices, returns C = [<-A-> 0
//                                      0   <-B->] with B
     shifted east by N cols

function C = seshft(A,B,N)
[Arows,Acols] = size(A);
[Brows,Bcols] = size(B);
if N >= 0
   B = [zeros(Brows,N) B];
```

```
    Bcols = Bcols + N;
elseif N < 0
   A = [zeros(Arows,abs(N)) A];
    Acols = Acols +abs(N);
end
  if Acols < Bcols
        A = [A zeros(Arows,Bcols-Acols)];
  elseif Acols > Bcols
        B = [B zeros(Brows,Acols-Bcols)];
  end
  C = [A
       B];
endfunction
```

Scilab Code 8.16 `t1calc.sci`

```
// Updated
// function [T1,T1rows,sel,pr] = ...
// t1calc(S,Srows,T1,T1rows,sel,pr,Frows,Fbcols,abar,gap)
// calculates the coefficient matrix T1
// redundant row information is kept in sel: redundant rows are marked
// with zeros. The undeleted rows are marked with ones.

function [T1,T1rows,sel,pr] = t1calc(S,Srows,T1,T1rows,sel,pr,Frows,Fbcols,abar,gap)
b = 1;                                          // vector of primary red.rows

while (T1rows < Frows - Fbcols) & or(sel==1) & ~isempty(b)
      S = clean(S);
      b = indep(S(mtlb_logical(sel),:),gap); // send selected rows of S
      if ~isempty(b)
         b = clean(b);
```

```
            b = move_sci(b,find(sel),Srows);
            j = length(b);
            while ~(b(j) & or(abar==j))     // pick largest nonzero entry
                j = j-1;                    // of coeff. belonging to abar
                if ~j
                    fprintf('\nMessage from t1calc, called from left_prm\n\n')
                    error('Denominator is noninvertible')
                end
            end
            if ~or(j<pr & pmodulo(pr,Frows) == pmodulo(j,Frows)) // pr(2),pr(1)
                T1 = [T1; b];               // condition is not violated
                T1rows = T1rows +1;         // accept this vector
            end                             // else don't accept
            pr = [pr; j];                   // update prime red row info
            while j <= Srows
                sel(j) = 0;
                j = j + Frows;
            end
        end
end
endfunction
```

8.7 Scilab Code for Virtual Experiment

Scilab Code 8.17 `twodof_para.sce`

```
mode(0)
global Rc Sc Tc gamm u_old_old u_old r_old_old r_old
```

```
y_old_old y_old u_new r_new y_new
s=%s;
z=%z;
// TFcont = syslin('c',0.593/((47.21*s+1)*(1.373*s+1)));// second order
// TFcont = syslin('c',0.594/(49.19*s+1))// first order
TFcont = syslin('c',0.42/(35.61*s+1));// first order
SScont = tf2ss(TFcont);
Ts = 1;
[B,A,k] = myc2d(SScont,Ts);

// polynomials are returned
[Ds,num,den] = ss2tf(SScont);
num = clean(num); den = clean(den);

// Transient specifications
rise = 100; epsilon = 0.05;
phi = desired(Ts,rise,epsilon);

// Controller design
Delta = [1 -1];
[Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi,Delta);// with integral

// initial values
u_old_old = 0;
u_old = 0;
r_old_old = 0;
r_old = 0;
y_old_old = 0;
y_old = 0;
```

Scilab Code 8.18 `twodof.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits sampling_time m heat temp

```

```
// **********************
sampling_time=1;    // In seconds. Fractions are allowed
// **********************//
getd('dc/scilab');
exec("twodof_para.sce")
exec ("twodof.sci");

ok = init();

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

Scilab Code 8.19 `twodof.sci`

```
function [stop] = twodof(setpoint,fan)
    global temp u_old_old u_old r_old_old r_old y_old_old y_old u_new heat r_new y_new

    r_new = setpoint;
    y_new = temp;
    // u_new = (1/Rc(1))*(gamm*Tc(1)*r_new + gamm*Tc(2)*r_old + r_old_old*Tc(3)*gamm - Sc(1)*y_new -Sc(2)*y_old - Sc(3)*y_old_old - Rc(2)*u_old - Rc(3)*u_old_old);// second order control law

    u_new = (1/Rc(1))*(gamm*Tc(1)*r_new + gamm*Tc(2)*r_old-Sc(1)*y_new -Sc(2)*y_old-Rc(2)*u_old - Rc(3)*u_old_old);// first order control law
    heat = u_new;
    [stop,temp] = comm(heat,fan);// Never edit this line
    plotting([heat fan temp setpoint], [], []);

```

```
    u_old_old = u_old;
    u_old = u_new;
    r_old_old=r_old;
    r_old = r_new;
    y_old_old=y_old;
    y_old = y_new;

endfunction
```

8.8 Scilab Codes Common for both Local and Virtual Experiments

Scilab Code 8.20 `myc2d.sci`

```
// Updated(26-7-07)
// 9.2
// function [B,A,k] = myc2d(G,Ts)
// Produces numerator and denominator of discrete transfer
// function in powers of z^{-1}
// G is continuous transfer function; time delays are not allowed
// Ts is the sampling time, all in consistent time units
// User defined function
// ------

function [B,A,k] = myc2d(G,Ts)
H = ss2tf(dscr(G,Ts));
num1 = coeff(H('num'));
den1 = coeff(H('den')); // -------------
A = den1(length(den1):-1:1);
num2 = num1(length(num1):-1:1);   // flip
nonzero = mtlb_find(num1);
first_nz = nonzero(1);
B = num2(first_nz:length(num2)); // -------------
```

```
k = length(den1) - length(num1);
endfunction
```

Scilab Code 8.21 `desired.sci`

```
// Updated(26-7-07)
// 9.4
function [phi,dphi] = desired(Ts,rise,epsilon)

Nr = rise/Ts; omega = %pi/2/Nr; rho = epsilon^(omega/
    %pi);
phi = [1 -2*rho*cos(omega) rho^2]; dphi = length(phi)
    -1;
endfunction;
```

Scilab Code 8.22 `polmul.sci`

```
// Updated ------No change
// polmul
// The command
//     [C,degA] = polmul(A,degA,B,degB)
// produces the polynomial matrix C that equals the
    product A*B of the
// polynomial matrices A and B.
//
// H. Kwakernaak, July, 1990


function [C,degC] = polmul(A,degA,B,degB)
[rA,cA] = polsize(A,degA);
[rB,cB] = polsize(B,degB);
if cA ~= rB
    error('polmul: Inconsistent dimensions of input
        matrices');
end

degC = degA+degB;
C = [];
```

```
for k = 0:degA+degB
    mi = 0;
    if k-degB > mi
        mi = k-degB;
    end
    ma = degA;
    if k < ma
        ma = k;
    end
    Ck = zeros(rA,cB);
    for i = mi:ma
        Ck = Ck + A(:,i*cA+1:(i+1)*cA)*B(:,(k-i)*cB
            +1:(k-i+1)*cB);
    end
    C = [C Ck];
end
endfunction
```

Scilab Code 8.23 `polsplit3.sci`

```
// Updated(18-7-07)
// 9.11
// function [goodpoly,badpoly] = polsplit3(fac,a)
// Splits a scalar polynomial of z^{-1} into good and bad
// factors. Input is a polynomial in increasing degree of
// z^{-1}. Optional input is a, where a <= 1.
// Factors that have roots outside a circle of radius a or
// with negative roots will be called bad and the rest
// good. If a is not specified, it will be assumed as 1.

function [goodpoly,badpoly] = polsplit3(fac,a)
if argn(2) == 1, a = 1; end
if a>1 error('good polynomial also is unstable'); end
fac1 = poly(fac(length(fac):-1:1),'z','coeff');
```

```
rts = roots(fac1);
rts = rts(length(rts):-1:1);

// extract good and bad roots
badindex = mtlb_find((abs(rts)>=a-1.0e-5)|(real(rts)
    <-0.05));
badpoly = coeff(poly(rts(badindex),'z'));
goodindex = mtlb_find((abs(rts)<a-1.0e-5)&(real(rts)
    >=-0.05));
goodpoly = coeff(poly(rts(goodindex),'z'));

// scale by equating the largest terms
[m,index] = max(abs(fac));
goodbad = convol(goodpoly,badpoly);
goodbad = goodbad(length(goodbad):-1:1);
factor1 = fac(index)/goodbad(index);
goodpoly = goodpoly * factor1;
goodpoly = goodpoly(length(goodpoly):-1:1);
badpoly = badpoly(length(badpoly):-1:1);
endfunction;
```

Scilab Code 8.24 `pp_im.sci`

```
// Updated(27-7-07)
// 9.8
// function [Rc,Sc,Tc,gamma,phit] = pp_im(B,A,k,phi,
    Delta)
// Calculates 2-DOF pole placement controller.
// ------
function [Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi,Delta)

// Setting up and solving Aryabhatta identity
[Ag,Ab] = polsplit3(A); dAb = length(Ab) - 1;
[Bg,Bb] = polsplit3(B); dBb = length(Bb) - 1;

[zk,dzk] = zpowk(k);

[N,dN] = polmul(Bb,dBb,zk,dzk);
```

```
dDelta = length(Delta)-1;
[D,dD] = polmul(Ab,dAb,Delta,dDelta);
dphi = length(phi)-1;

[S1,dS1,R1,dR1] = xdync(N,dN,D,dD,phi,dphi);

// Determination of control law
Rc = convol(Bg,convol(R1,Delta)); Sc = convol(Ag,S1);
Tc = Ag; gamm = sum(phi)/sum(Bb);
endfunction;
```

Scilab Code 8.25 `xdync.sci`

```
// Updated ----No change
// function [Y,degY,X,degX,B,degB,A,degA] = xdync(N,degN,D,degD,C,degC,gap)
// given coefficient matrix in T1, primary redundant row information sel,
// solves XD + YN = C

// calling order changed on 16 April 2005. Old order:
// function [B,degB,A,degA,Y,degY,X,degX] = xdync(N,degN,D,degD,C,degC,gap)

function [Y,degY,X,degX,B,degB,A,degA] = xdync(N,degN,D,degD,C,degC,gap)
if argn(2) == 6
        gap = 1.0e+8;
end

[F,degF] = rowjoin(D,degD,N,degN);

[Frows,Fbcols] = polsize(F,degF); // Fbcols = block columns

[B,degB,A,degA,S,sel,degT1,Fbcols] = left_prm(N,degN,D,degD,3,gap);
// if issoln(D,degD,C,degC,B,degB,A,degA)
```

```
        [Crows,Ccols] = size(C);
        [Srows,Scols] = size(S);
        S = clean(S);
        S = S(mtlb_logical(sel),:);
        T2 =[];

        for i = 1:Crows,
                Saug = seshft(S,C(i,:),0);
                b = cindep(Saug);
                b = move_sci(b,find(sel),Srows);
                T2 =[T2; b];
        end

    [X,degX,Y,degY] = colsplit(T2,degT1,Fbcols,Frows-
        Fbcols);

    [X,degX] = clcoef(X,degX);
    [Y,degY] = clcoef(Y,degY);
     Y = clean(Y); X = clean(X);
endfunction
```

Scilab Code 8.26 `zpowk.sci`

```
// Updated(26-7-07)
// 9.6
// ------

function [zk,dzk] = zpowk(k)
zk = zeros(1,k+1); zk(1,k+1) = 1;
dzk = k;
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. 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.

9.1 PRBS Modelling

Similar to Chapter 4 and 5, 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 9.4. 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.

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

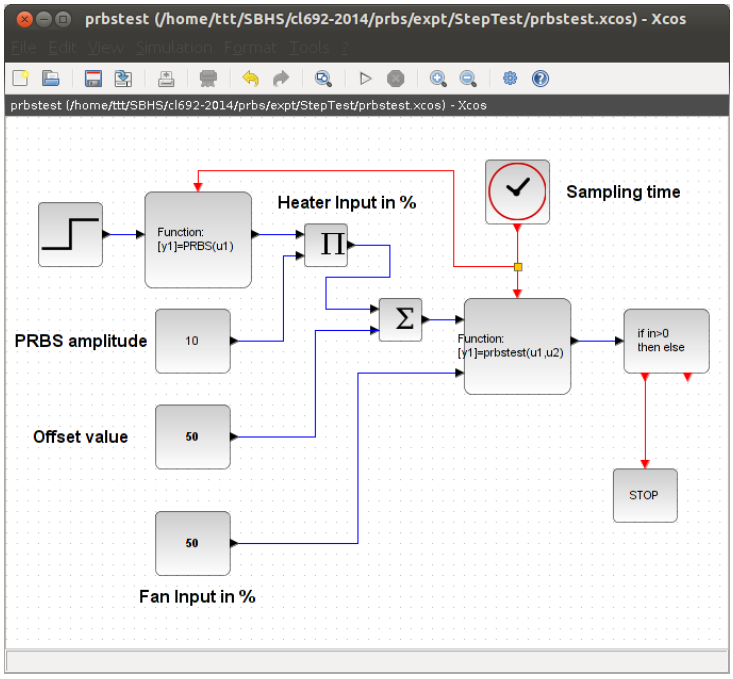

Figure 9.1: Xcos for PRBS testing experiment

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 occurance 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.

9.1.1 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

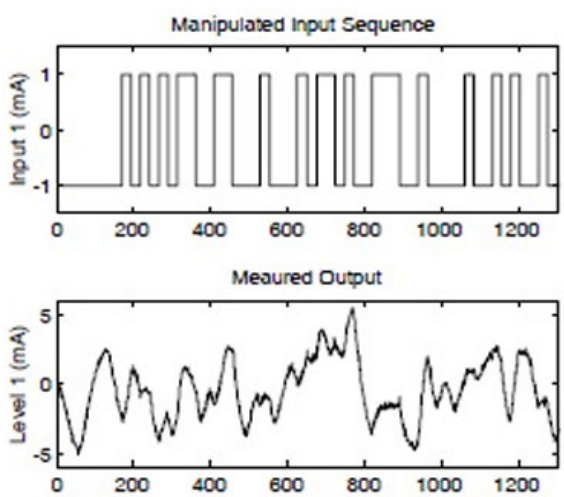

Figure 9.2: PRBS testing input and output *[Image source: CL 686 Advanced Process Control, Spring 2013-14 lecture slides. Prof. S. C. Patwardhan, IIT Bombay]*

```
1.0         50.0          50.0        36.4   1417298828422.0
2.0          50.0         50.0       36.2   1417298828525.0
.
 999.0        40.0         50.0         42.9   1417298933585.0
 1000.0      40.0          50.0          42.9   1417298933694.0
```

Table 9.1: PRBS local experiment data

9.2 Conducting PRBS Test on SBHS locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `prbs/identification`
2. Step2: Load the functions available in common files directory by executing the command `getd<space>..\..\common_files`
3. Step3: Same
4. Step4: Same
5. Step5: Load prbstest function by executing command `exec<space>prbstest.sci`. Load prbs signal generation function by executing command `exec<space>prbs.sci`
6. Step6: Load Xcos code for prbs test using the command `exec<space>prbstest.xcos`
7. Step7: Same

The response is as shown in figure 9.3. The data file thus obtained is as shown in the Table 9.1. This data file is available for reference in the directory `prbs/identification` with name `prbs-data-local.txt`

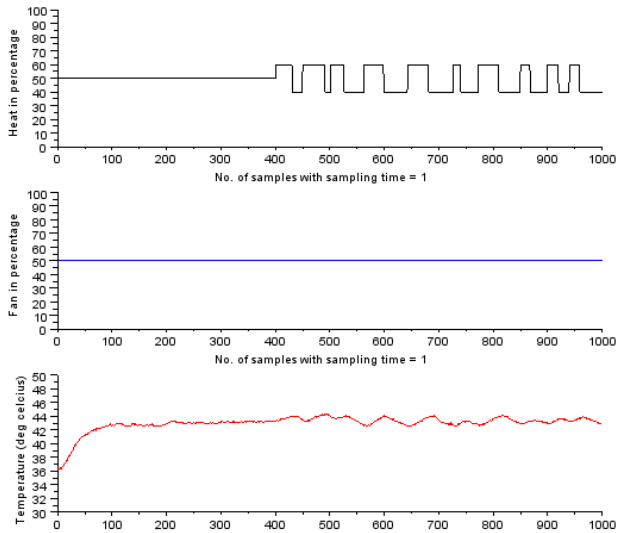

Figure 9.3: PRBS Local response

9.3 Conducting PRBS Test on SBHS, virtually

The detailed procedure to perform a local experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `prbs/identification`. Open this directory.

2. Step2: Same

3. Step3: Same

4. Step4: Switch to the working experiment directory and double-click on the file `prbstest.sce`. This will launch scilab and also open the file `prbstest.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `prbs/identification` and then open the `prbstest.sce` file in the scilab editor.

5. Step5: Load the functions available in common files directory by executing the command `getd<space>..\..\common_files`

```
0 0 100 28.40 14...1731 14...4105 14...4123 14...1763 0.10000E+01
1 50 50 28.30 14...4706 14...7078 14...7096 14...4738 0.10000E+01
.
.
983 40 50 36.70 14...6728 14...9131 14...9148 14...6759 0.98300E+03
984 40 50 36.50 14...7712 14...0115 14...0133 14...7743 0.98400E+03
```

Table 9.2: PRBS data obtained after performing virtual PRBS Test

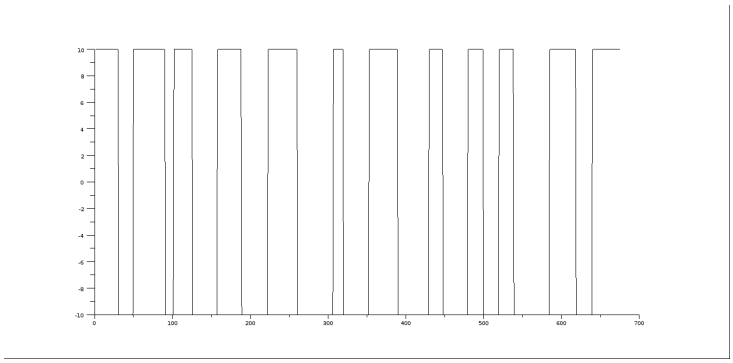

Figure 9.4: A Pseudo Random Binary Sequence

6. Step6: Execute the file `prbstest.sce`. Expect the prbs test xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.

7. Step7: Execute the prbstest xcos diagram.

8. Step8: Same

The virtual experiment response is shown in figure 9.5. The corresponding data file is shown in table 9.2. The time stamps shown are cut short for better viewing. This data file can be found in `prbs/identification` folder for virtual experiments. The name of this file is `prbs-data-virtual.txt`.

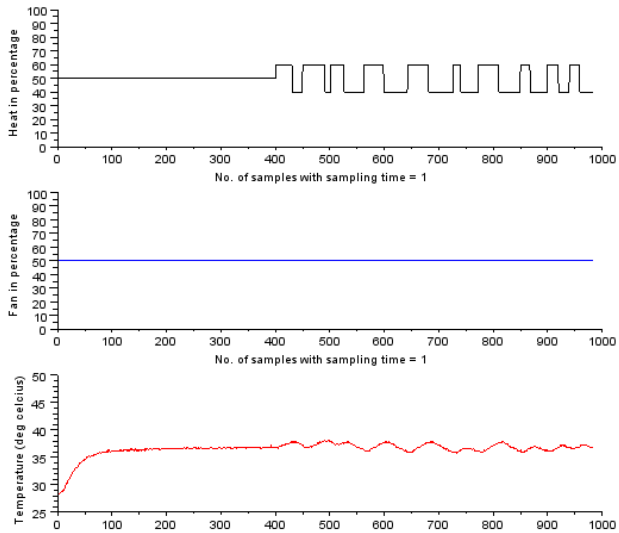

Figure 9.5: PRBS testing response for virtual experiment

9.4 Determination of Discrete Time Transfer Function models

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:

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

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

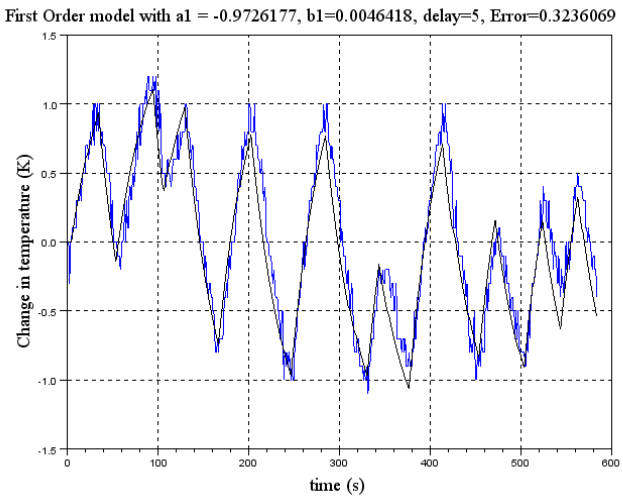

Figure 9.6: PRBS first order fit

problem, then, becomes:

$$(\hat{b_1}, \hat{b_2}, \hat{a_1}, \hat{a_2}) = \underset{b_1, b_2, a_1, a_2}{argmin} \sum_{i=0}^{N} (y(k) - \hat{y}(k))^2 \tag{9.2}$$

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

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

9.5 Determination of First order Discrete time Transfer Function

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

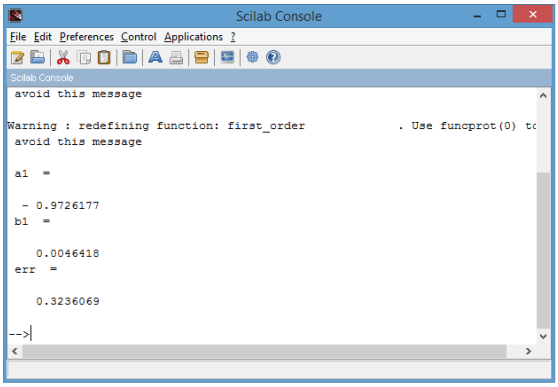

Figure 9.7: PRBS first order model

2. Open the `Analysis` folder and then locate and open the folder `Step_Analysis`.
3. Inside this folder, locate and open the folder `Discrete-order1`
4. Copy the prbs test data file to this folder.
5. Start scilab and change the working directory to `Discrete-order1`
6. Open the file `optimize.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.
7. Save and run this code and obtain the plot as shown in figure 9.6. This plot will also show the first order discrete time transfer function's coefficients a1 and b1.
8. The values are also shown on scilab console as shown in figure 9.7

The results presented are obtained for the data file `prbs-data-virtual.txt`. This data file is present under the `prbs` directory for virtual experiments.The plot thus obtained is reasonably good. See the Scilab plot to get the values of $a1$ and $b1$. The figure 9.6 shows a screen shot of the same. We obtain $a1$ = -0.97, $b1$= 0.004. The transfer function obtained here is at the operating point of 50 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 $G(z)$. We obtain

$$G(z) = \frac{0.004}{1 - 0.97z^{-1}} z^{-5} \tag{9.4}$$

9.6 Determination of Second order Discrete time Transfer Function

1. Download the Analysis folder from the sbhs website. It will be available under `downloads` section. The download will be in zip format. Extrat the downloaded zip file. You will get a folder `Analysis`.

2. Open the `Analysis` folder and then locate and open the folder `Step_Analysis`.

3. Inside this folder, locate and open the folder `Discrete-order2`

4. Copy the prbs test data file to this folder.

5. Start scilab and change the working directory to `Discrete-order2`

6. Open the file `optimize.sce` in scilab editor and enter the name of the data file (with extention) in the `filename` field.

7. Save and run this code and obtain the plot as shown in figure 9.8. This plot will also show the second order discrete time transfer function's coefficients a1, a2, b1 and b2.

8. The values are also shown on scilab console as shown in figure 9.9

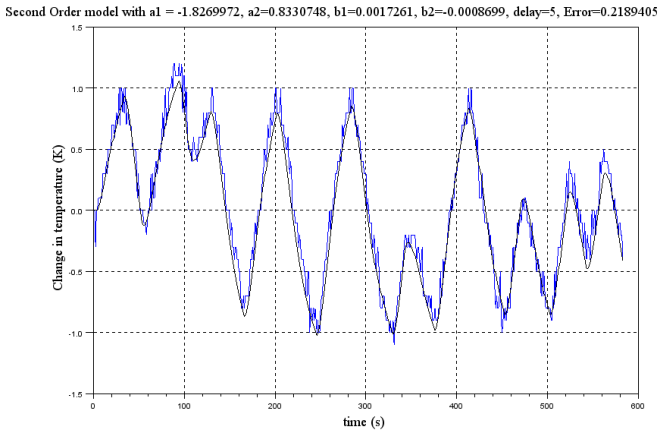

Figure 9.8: PRBS second order fit

The results presented are obtained for the data file `prbs-data-virtual.txt`. This data file is present under the `prbs` directory for virtual experiments.The plot thus obtained is reasonably good. See the Scilab plot to get the values of $a1$, $a2$, $b1$ and $b2$. The figure 9.8 shows a screen shot of the same. We obtain $a1$ = -1.82, $a2$= 0.833, $b1$ = 0.0017, $b2$= -0.00086, . The transfer function obtained here is at the operating point of 50 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 $G(z)$. We obtain

$$G(z) = \frac{0.0017 - 0.00086z^{-1}}{1 - 1.826z^{-1} + 0.833z^{-2}} z^{-5} \tag{9.5}$$

9.7 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

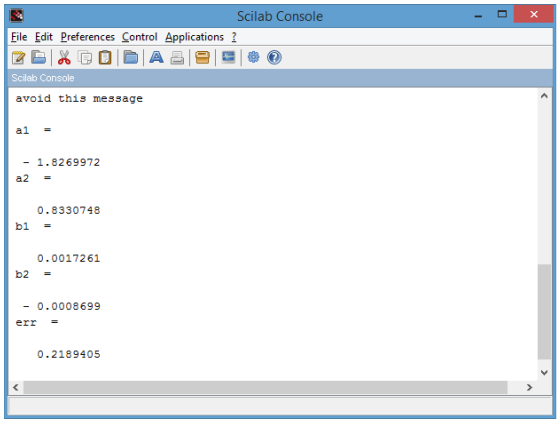

Figure 9.9: PRBS second order model

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 9.10. The controller designed is not derived for the model explained in earlier sections.

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 9.10, 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 paramenters 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

- Step1: The working directory is `prbs/controller`. Open this directory.
- Step2: Same
- Step3: Same

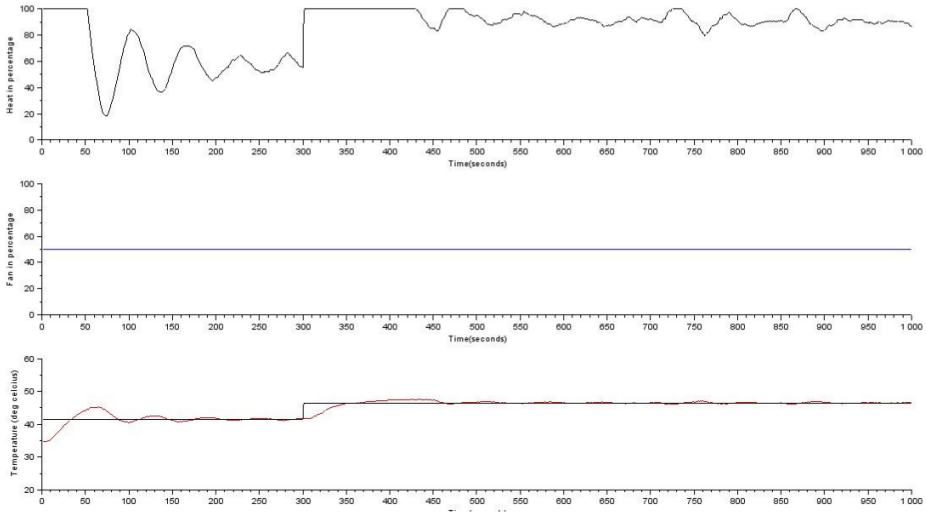

Figure 9.10: 2dof controller response

- Step4: Switch to the `controller` experiment directory and double-click on the file `prbs.sce`. This will launch scilab and also open the file `prbs.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `controller` and then open the `prbs.sce` file in the scilab editor.

- Step5: Load the functions available in common files directory by executing the command `getd<space>..\..\common_files`

- Step6: Execute the file `prbs.sce`. Expect the prbs test xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message. The values of Rc, Sc, Tc and γ are to be entered in the xcos diagram.

- Step7: Execute the prbstest xcos diagram.

- Step8: Same

9.8 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 2.3 with the following changes

1. Step1: The working directory is `prbs/controller`
2. Step2: Load the functions available in common files directory by executing the command `getd<space>..\..\common_files`
3. Step3: Same
4. Step4: Same
5. Step5: Load prbstest function by executing command `exec<space>prbs_pp.sci`.
6. Step6: Load Xcos code for prbs test using the command `exec<space>prbs_pp.xcos`. The values of Rc, Sc, Tc and γ are to be entered in the xcos diagram.
7. Step7: Same

9.9 Scilab Local codes

9.9.1 Identification codes

Scilab Code 9.1 `ser_init.sce`

```
mode(0)
global filename m
//**Sampling Time**//
sampling_time = 1;
//////***/////////
m=1;

port1 = '/dev/ttyUSB0'; // For linux users
port2 = 'COM2'; // For windows users
```

```

res=init([port1 port2]);
disp(res)
```

Scilab Code 9.2 `costfunction.sci`

```
function [f,g,ind] = costfunction(x,ind)
    global delay;
    y_prediction = second_order(u, x);
    if size(y) ~= size(y_prediction) then
        y_prediction = y_prediction';
    end
    f = (norm(y-y_prediction,2))^2;
    g = numdiff(func_1,x);
endfunction

function f = func_1(x)
    global delay;
    y_prediction = second_order(u, x);
    if size(y) ~= size(y_prediction) then
        y_prediction = y_prediction';
    end
    f = (norm(y-y_prediction,2))^2;
endfunction
```

Scilab Code 9.3 `optimize.sce`

```
mode(0);
 // Change filename here
filename = "30Apr2014_12_30_50.txt";
clf


exec('costfunction.sci');
exec('label.sci');
exec('second_order.sci');


```

```
data = fscanfMat(filename);

time = data(:, 1);

heater = int(data(:, 2));

fan = int(data(:, 3));

temp = data(:, 4);

ss_op_pt = heater(2);
for i=2:length(heater)
    if heater(i) ~= ss_op_pt then
        startTime = i;
        break
    end
end


time1 = time - time(1);
time2 = time1/1000;

baseheat = heater(5);
heater = heater(startTime:length(heater));
heater = heater - baseheat;

len = length(heater);

temp = temp(startTime:length(temp));
temp = temp - temp(1);

time = time2(startTime:length(time));
time = time - time(1);

t = time;
y = temp;
u = heater;

```

```
x0 = [0.2 0.2 0.5 0.5]; // Change initial guess here
delay = 5;        // Change delay here
global delay;
[f, xopt] = optim(costfunction, x0);

a1 = xopt(1)
a2 = xopt(2)
b1 = xopt(3)
b2 = xopt(4)

y_pred = second_order(u, xopt);

if size(y) ~= size(y_pred) then
    y_pred = y_pred';
end
err = norm(y - y_pred)/norm(y)

plot(t, y, "+");
plot(t, y_pred, "k");
// plot(t, u/10, "r");

label('Showing Second Order Model and Experimental
   Results',4,'Time (s)','Change in temperature (K)'
   ,4);
```

Scilab Code 9.4 `prbs.sci`

```
function u=PRBS(active)
if active == 0 then
u = 0;
else
global PRBSu PRBScount;
if PRBSu == [] then
PRBSu = 1;
PRBScount = 30;
end
if PRBScount == 0 then
PRBSu = -1 * PRBSu;
```

```
PRBScount = int(rand()*40)+10;
else
PRBScount = PRBScount - 1;
end
u = PRBSu;
global PRBSu PRBScount;
end
endfunction
```

Scilab Code 9.5 `prbstest.sci`

```
mode(0)
function [temp] = prbstest(heat,fan)
global heatdisp fandisp tempdisp setpointdisp
    sampling_time m name


     temp = comm(heat,fan);

     plotting([heat fan temp],[0 0 20 0],[100 100 40
        1000])

    m=m+1;
endfunction
```

Scilab Code 9.6 `second_order.sci`

```
function y = second_order(u, params)
     // Do not change anything here
     a1 = params(1);
     a2 = params(2);
     b1 = params(3);
     b2 = params(4);
     global delay;
     // End

     /// You should write your code below this line
     N=length(u);
```

```

    // Defining y vector
    // from t=0 up to t=delay-1, y=0, so in scilab y at indices i=1 up to i=delay is 0
    y(1:1:delay) = 0;
    // First nonzero output is only due to input u at t=0, i.e. in scilab input u at i=1
    y(delay+1) = b1*u(1);
    // After that y(i) can be defined as follows
    for i=delay+2:1:N
        y(i)=-a1*y(i-1)-a2*y(i-2)+b1*u(i-delay)+b2*u(i-1-delay);
    end

    // y = ?

endfunction
```

Scilab Code 9.7 `start.sce`

```
getd ../../common_files/

exec ../../common_files/loader.sce

exec ser_init.sce
exec prbs.sci
exec prbstest.sci

xcos prbstest.xcos
```

9.9.2 Controller codes

Scilab Code 9.8 `start.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits sampling_time m
```

```
global scn scd tcn tcd rcn rcd gamm

getd "dc/scilab"

// ***********************
sampling_time=1;     // In seconds. Fractions are allowed
// ************************//
exec ("prbstest-virtual.sci");
exec ("twodof_para.sce");
// exec ("sbhs_control.sci");



//[scn,scd,tcn,tcd,rcn,rcd,gamm] = sbhs_control()

ok = init();

    if ok~= []   // open xcos only if communication is through (ie reply has come from server)
        xcos('prbstest-virtual.xcos');
      else
      disp("NO NETWORK CONNECTION!");
      return
end
```

Scilab Code 9.9 `prbs_pp.sce`

```
mode(0)
function [temp] = prbs_pp(heat,fan, setpoint)
global heatdisp fandisp tempdisp setpointdisp sampling_time m name


     temp = comm(heat,fan);

     plotting([heat fan temp setpoint],[0 0 20 0],[100 100 40 1000])
```

```

    m=m+1;
endfunction
```

Scilab Code 9.10 `ser_init.sce`

```
mode(0)
global filename m
//**Sampling Time**//
sampling_time = 1;
//////***********
m=1;

port1 = '/dev/ttyUSB0'; // For linux users
port2 = 'COM2'; // For windows users

res=init([port1 port2]);
disp(res)
```

Scilab Code 9.11 `start.sce`

```
getd ../../common_files/
getd dc/scilab
exec ../../common_files/loader.sce

exec ser_init.sce
exec prbs_pp.sci

exec twodof_para.sce

xcos prbs_pp.xcos
```

Scilab Code 9.12 `twodof_para.sce`

```
mode(0)
global Rc Sc Tc gamm
global scn scd tcn tcd rcn rcd gamm
```

```
s=%s;
z=%z;


Ts = sampling_time;

// Transfer function
A = [1 -1.87 0.87];
B= [0.0020 -0.0015];
k = 3;


rise = 10;
epsilon = 0.1;
Nr = rise/Ts;

// Transient specifications
//rise = 10; epsilon = 0.05;
phi = desired(Ts,rise,epsilon);

// Controller design
Delta = [1 -1];
[Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi);// with integral

// Setting up simulation parameters for basic.cos
st = 0.0001; // desired change in h, in m.
t_init = 0; // simulation start time
t_final = 0.5; // simulation end time

// Setting up simulation parameters for c_ss_cl.cos
N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;

[Tc1,Rc1] = cosfil_ip(Tc,Rc); // Tc/Rc
[Sc2,Rc2] = cosfil_ip(Sc,Rc); // Sc/Rc

[Bp] = cosfil_ip(B,1);
[Ap] = cosfil_ip(A,1);

```

```
[Tcp1,Tcp2] = cosfil_ip(Tc,1); // Tc/1
[Np,Rcp] = cosfil_ip(N,Rc); // 1/Rc
[Scp1,Scp2] = cosfil_ip(Sc,1); // Sc/1
[Cp,Dp] = cosfil_ip(C,D); // C/D

// Rc1=Rc(1);Rc2=Rc(2);Rc3=Rc(3);Rc4=Rc(4);
// Sc1=Sc(1);Sc2=Sc(2);
// Sc3=Sc(3);
// Tc1=Tc(1);Tc2=Tc(2);
// Tc3=Tc(3);
Rcp
Scp1
Tcp1
gamm



scn = poly(Sc(length(Sc):-1:1),'z','coeff');
tcn = poly(Tc(length(Tc):-1:1),'z','coeff');
rcn = poly(Rc(length(Rc):-1:1),'z','coeff');

scd = z^(length(Sc)-1);
rcd = z^(length(Rc)-1);
tcd = z^(length(Tc)-1);
```

9.10 Scilab Virtual codes

9.10.1 Identification codes

Scilab Code 9.13 `costfunction.sci`

```
function [f,g,ind] = costfunction(x,ind)
    global delay;
    y_prediction = second_order(u, x);
    if size(y) ~= size(y_prediction) then
        y_prediction = y_prediction';
    end
```

```
    f = (norm(y-y_prediction,2))^2;
    g = numdiff(func_1,x);
endfunction

function f = func_1(x)
    global delay;
    y_prediction = second_order(u, x);
    if size(y) ~= size(y_prediction) then
        y_prediction = y_prediction';
    end
    f = (norm(y-y_prediction,2))^2;
endfunction
```

Scilab Code 9.14 `optimize.sce`

```
mode(0);
// filename = "prbs.txt"; // Change filename here
// filename = "29Apr2014_17_03_57.txt";
// filename = "30Apr2014_12_30_50.txt";
filename = "02May2014_16_23_16.txt";
clf

exec('costfunction.sci');
exec('label.sci');
exec('second_order.sci');
// data = fscanfMat(filename);
// heater1 = int(data(:, 2));
// len = length(heater1);
// heater_new = [heater1(1);heater1(1:len-1)];
// del_heater = heater1-heater_new;
// ind = find(del_heater>1);
// heater = heater1(ind(2):len);
//
// time = data(ind(2):len, 5);
//
// fan = int(data(ind(2):len, 3));
// temp = data(ind(2):len, 4);

```

```
data = fscanfMat(filename);

time = data(:, 5);

heater = int(data(:, 2));

fan = int(data(:, 3));

temp = data(:, 4);

ss_op_pt = heater(2);
for i=2:length(heater)
    if heater(i) ~= ss_op_pt then
        startTime = i;
        break
    end
end


time1 = time - time(1);
time2 = time1/1000;

baseheat = heater(5);
heater = heater(startTime:length(heater));
heater = heater - baseheat;

len = length(heater);

temp = temp(startTime:length(temp));
temp = temp - temp(1);

time = time2(startTime:length(time));
time = time - time(1);

t = time;
y = temp;
u = heater;
```

```
x0 = [0.2 0.2 0.5 0.5]; // Change initial guess here
delay = 5;        // Change delay here
global delay;
[f, xopt] = optim(costfunction, x0);

a1 = xopt(1)
a2 = xopt(2)
b1 = xopt(3)
b2 = xopt(4)

y_pred = second_order(u, xopt);

if size(y) ~= size(y_pred) then
    y_pred = y_pred';
end
err = norm(y - y_pred)/norm(y)

plot(t, y, "+");
plot(t, y_pred, "k");
// plot(t, u/10, "r");

label('Showing Second Order Model and Experimental
   Results',4,'Time (s)','Change in temperature (K)'
   ,4);
```

Scilab Code 9.15 `prbs.sci`

```
function u=PRBS(active)
if active == 0 then
u = 0;
else
global PRBSu PRBScount;
if PRBSu == [] then
PRBSu = 1;
PRBScount = 30;
end
if PRBScount == 0 then
```

```
PRBSu = -1 * PRBSu;
PRBScount = int(rand()*40)+10;
else
PRBScount = PRBScount - 1;
end
u = PRBSu;
global PRBSu PRBScount;
end
endfunction
```

Scilab Code 9.16 `prbstest.sci`

```
function [stop] = prbstest(heat,fan)

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

endfunction
```

Scilab Code 9.17 `prbstest.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits sampling_time m

// **********************
sampling_time=1;    // In seconds. Fractions are allowed
// ***********************//
exec ("prbstest.sci");
exec ("prbs.sci");

ok = init();

   if ok~= []  // open xcos only if communication is through (ie reply has come from server)
      xcos('prbstest.xcos');
     else
```

```
    disp("NO NETWORK CONNECTION!");
    return
end
```

Scilab Code 9.18 `second_order.sci`

```
function y = second_order(u, params)
    // Do not change anything here
    a1 = params(1);
    a2 = params(2);
    b1 = params(3);
    b2 = params(4);
    global delay;
    // End

    /// You should write your code below this line
    N=length(u);

    //Defining y vector
    //from t=0 up to t=delay-1, y=0, so in scilab y at indices i=1 up to i=delay is 0
    y(1:1:delay) = 0;
    //First nonzero output is only due to input u at t=0, i.e. in scilab input u at i=1
    y(delay+1) = b1*u(1);
    //After that y(i) can be defined as follows
    for i=delay+2:1:N
        y(i)=-a1*y(i-1)-a2*y(i-2)+b1*u(i-delay)+b2*u(i-1-delay);
    end

    // y = ?

endfunction
```

9.10.2 Controller codes

Scilab Code 9.19 `prbs.sce`

```
mode(0)
global fdfh fdt fncr fncw m err_count y limits
    sampling_time m

global scn scd tcn tcd rcn rcd gamm

getd "dc/scilab"

// **********************
sampling_time=1;     // In seconds. Fractions are allowed
// **********************//
exec ("prbscontrol-virtual.sci");
exec ("twodof_para.sce");
// exec ("sbhs_control.sci");



//[scn,scd,tcn,tcd,rcn,rcd,gamm] = sbhs_control()

ok = init();

    if ok~= []   // open xcos only if communication is
        through (ie reply has come from server)
        xcos('prbscontrol-virtual.xcos');
      else
      disp("NO NETWORK CONNECTION!");
      return
end
```

Scilab Code 9.20 `prbscontrol-virtual.sci`

```
function [stop,temp] = prbstest(heat,fan,setp)
    global scn scd tcn tcd rcn rcd gamm

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

```

```
endfunction
```

Scilab Code 9.21 twodof_para.sce

```
mode(0)
global Rc Sc Tc gamm
global scn scd tcn tcd rcn rcd gamm
s=%s;
z=%z;
//// TFcont = syslin('c', -280.14/((s-31.32)*(s+100)*(s+31.32)));
// TFcont = syslin('c',0.593/((47.21*s+1)*(1.373*s+1))) // second order
//// TFcont = syslin('c',0.594/(49.19*s+1)) // first order
// SScont = tf2ss(TFcont);
//// TFdisc=ss2tf(SScont);
// Ts = 1;
// [B,A,k] = myc2d(SScont,Ts);
//
//// polynomials are returned
// [Ds,num,den] = ss2tf(SScont);
// num = clean(num); den = clean(den);

Ts = sampling_time;

// Transfer function for Part - B
A = [1 -1.9529968 0.9531269];
B= [0.0057384 -0.0057355];
k = 3;

// Transfer function for Part - A
// B = [0.0043779 -0.0043266];
// A = [1 -1.9444137 0.9447818];
// k = 5;

rise = 10;
epsilon = 0.1;
```

```
Nr = rise/Ts;

// Transient specifications
//rise = 10; epsilon = 0.05;
phi = desired(Ts,rise,epsilon);

// Controller design
Delta = [1 -1];
[Rc,Sc,Tc,gamm] = pp_im(B,A,k,phi);// with integral

// Setting up simulation parameters for basic.cos
st = 0.0001; // desired change in h, in m.
t_init = 0; // simulation start time
t_final = 0.5; // simulation end time

// Setting up simulation parameters for c_ss_cl.cos
N_var = 0; xInitial = [0 0 0]; N = 1; C = 0; D = 1;

[Tc1,Rc1] = cosfil_ip(Tc,Rc); // Tc/Rc
[Sc2,Rc2] = cosfil_ip(Sc,Rc); // Sc/Rc

[Bp] = cosfil_ip(B,1);
[Ap] = cosfil_ip(A,1);

[Tcp1,Tcp2] = cosfil_ip(Tc,1); // Tc/1
[Np,Rcp] = cosfil_ip(N,Rc); // 1/Rc
[Scp1,Scp2] = cosfil_ip(Sc,1); // Sc/1
[Cp,Dp] = cosfil_ip(C,D); // C/D

//Rc1=Rc(1);Rc2=Rc(2);Rc3=Rc(3);Rc4=Rc(4);
//Sc1=Sc(1);Sc2=Sc(2);
//Sc3=Sc(3);
//Tc1=Tc(1);Tc2=Tc(2);
//Tc3=Tc(3);
Rcp
Scp1
Tcp1
gamm
```

```
scn = poly(Sc(length(Sc):-1:1),'z','coeff');
tcn = poly(Tc(length(Tc):-1:1),'z','coeff');
rcn = poly(Rc(length(Rc):-1:1),'z','coeff');

scd = z^(length(Sc)-1);
rcd = z^(length(Rc)-1);
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

$$y(n) = G(z)u(n) + \xi(n) \tag{10.1}$$

where:
y(n) = plant output

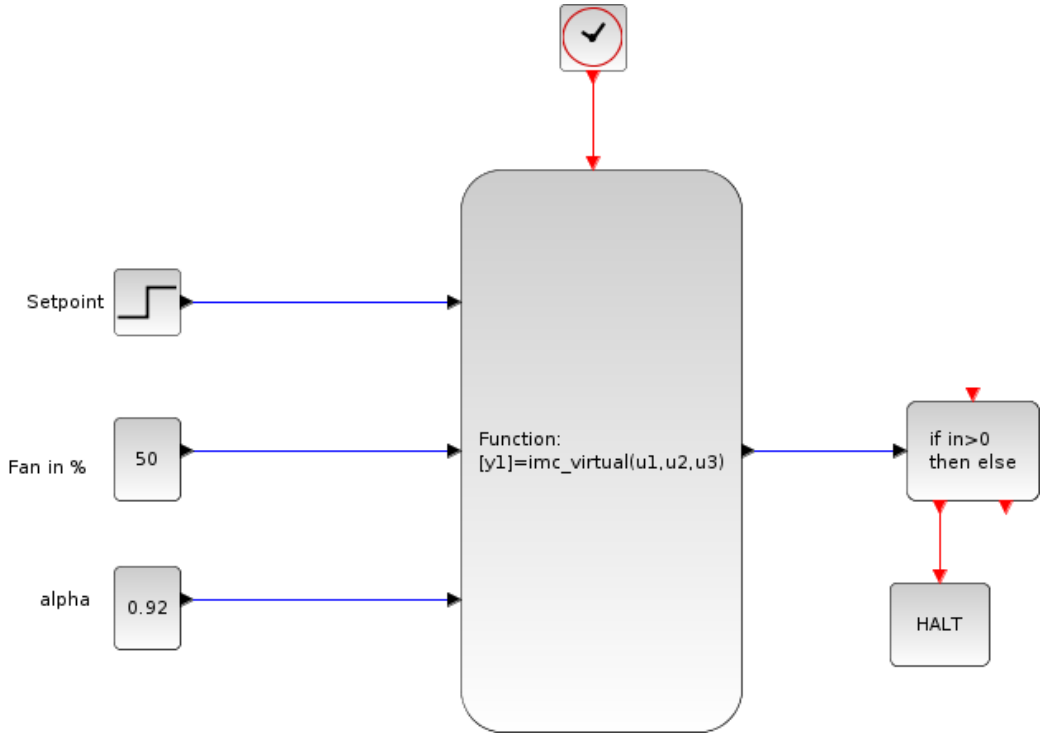

Figure 10.1: Xcos interface for this experiment

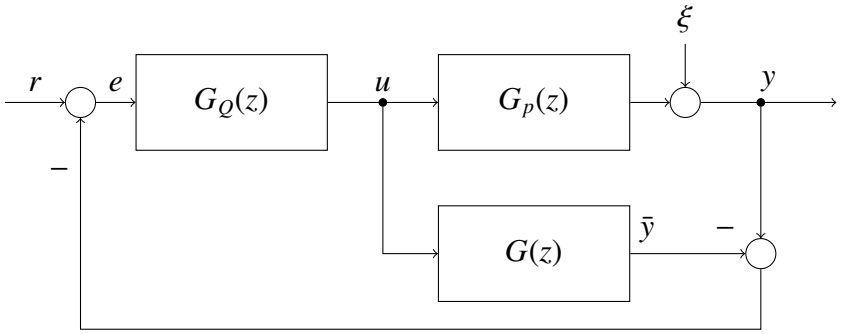

Figure 10.2: IMC feedback configuration

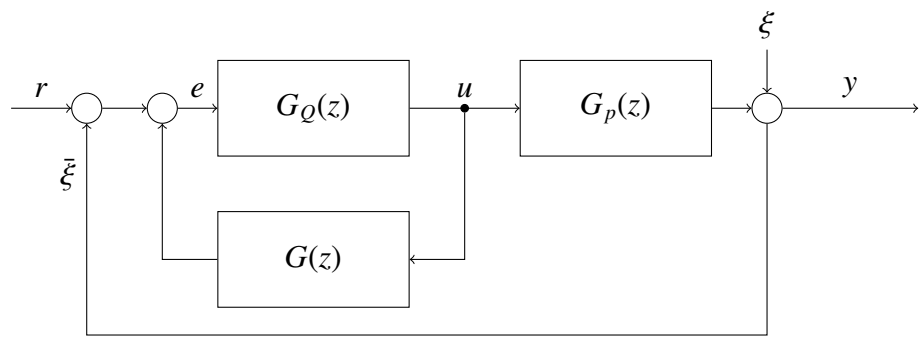

Figure 10.3: Feedback configuration

u(n) = plant input
ξ(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-

$$G = Z^{-1}\frac{0.01163}{1 - 0.9723Z^{-1}} \tag{10.2}$$

Inverting delay free plant, we get

$$\frac{A}{B} = \frac{1 - 0.9723Z^{-1}}{0.01163} \tag{10.3}$$

Comparing plant model with equation

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

We get,

$$B^g = 0.01163 \tag{10.5}$$
$$B^- = 1 \tag{10.6}$$
$$B^{nm+} = 1 \tag{10.7}$$
$$A = 1 - 0.9723Z^{-1} \tag{10.8}$$

For the stable system, internal model controller is given by

$$G_Q = \frac{A}{B^g B_s^- B_r^{nm+}} G_f \tag{10.9}$$
$$G_Q = \frac{1 - 0.9723Z^{-1}}{0.01163}\frac{1-\alpha}{1-\alpha Z^{-1}} \tag{10.10}$$

Now,

$$G_c = \frac{G_Q}{1 - GG_Q} \tag{10.11}$$
$$\frac{u}{e} = \frac{\frac{1-0.9723Z^{-1}}{0.01163}\frac{1-\alpha}{1-\alpha Z^{-1}}}{1 - Z^{-1}\frac{0.01163}{1-0.9723Z^{-1}}\frac{1-0.9723Z^{-1}}{0.01163}\frac{1-\alpha}{1-\alpha Z^{-1}}} \tag{10.12}$$

After simplifying, we get

$$\frac{u}{e} = \frac{1-\alpha}{0.01163}\frac{1 - 0.9723Z^{-1}}{1 - Z^{-1}} \tag{10.13}$$
$$\frac{u}{e} = b\frac{1 - 0.9723Z^{-1}}{1 - Z^{-1}} \tag{10.14}$$

where,

$$b = \frac{1-\alpha}{0.01163} \tag{10.15}$$

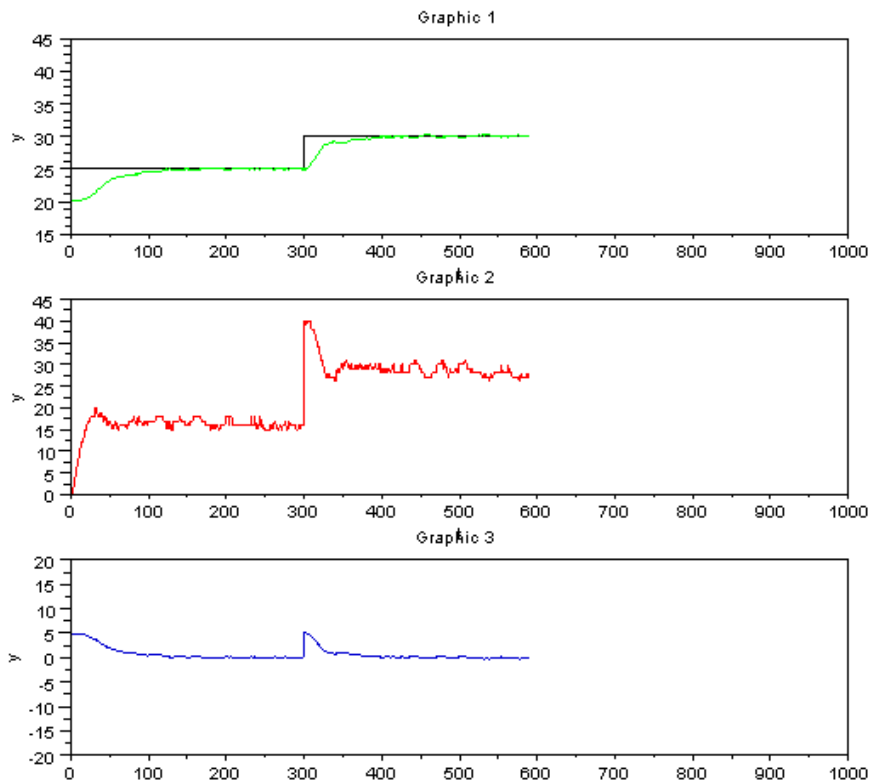

Figure 10.4: Experimental results with IMC for $\alpha = 0.92$

Hence,

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

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.

The same experiment result for $\alpha = 0.85$ is as shown in fig 10.5. 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.

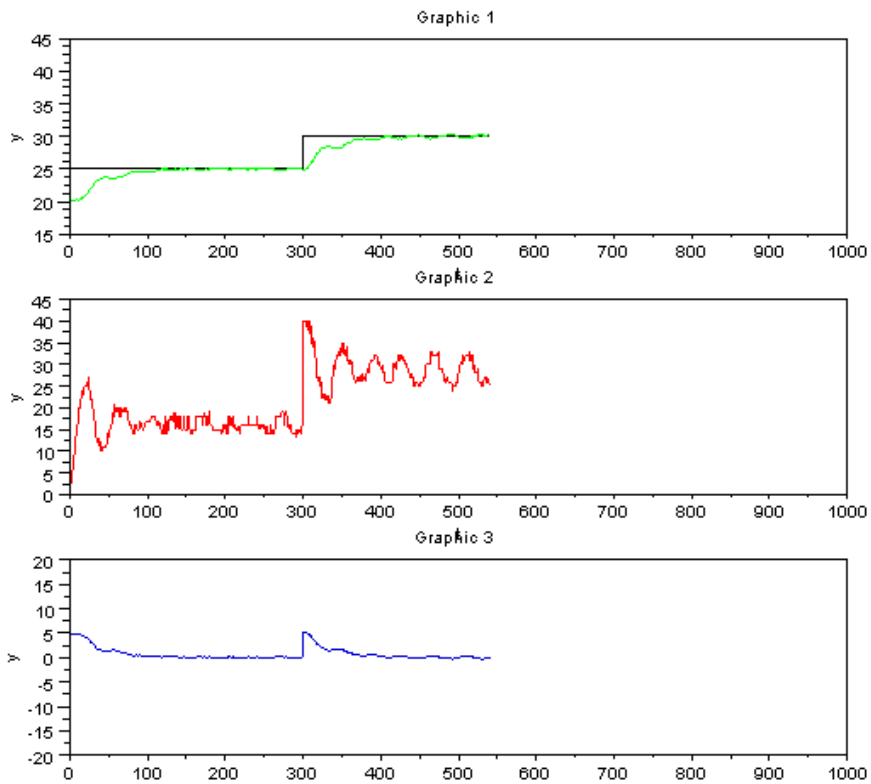

Figure 10.5: Experimental results with IMC for $\alpha = 0.85$

10.2.1 Implementing IMC locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `imc_controller`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load ramp test function by executing command `exec<space>imc.sci`
6. Step6: Load Xcos code for ramp test using the command `exec<space>imc.xcos`
7. Step7: Same

10.2.2 Implementing IMC virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `imc_controller`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the IMC experiment directory and double-click on the file `imc.sce`. This will launch scilab and also open the file `imc.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `imc_controller` and then open the `imc.sce` file in the scilab editor.
5. Step5: Same

6. Step6: Execute the file `imc.sce`. Expect the IMC controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.

7. Step7: Execute the IMC controller xcos diagram.

8. Step8: Same

10.3 Scilab Code

Scilab Code 10.1 `ser_init.sce`

```
mode(0)
global filename m
//**Sampling Time**//
sampling_time = 1;
//////***/////////
m=1;

port1 = '/dev/ttyUSB0'; // For linux users
port2 = 'COM2'; // For windows users

res=init([port1 port2]);
disp(res)
```

Scilab Code 10.2 `imc.sci`

```
mode(0)
function [temp] = imc(setpoint,fan,alpha)
global temp heat_in fan_in C0 u_old u_new e_old e_new
    e_old_old

global heatdisp fandisp tempdisp setpointdisp
    sampling_time m name

e_new = setpoint - temp;
b=((1-alpha)/0.01163);

```

```
u_new = u_old + b*(e_new - (0.9723*e_old));


u_old = u_new;
e_old = e_new;


heat = u_new;

    temp = comm(heat,fan);

    plotting([heat fan temp setpoint],[0 0 20 0],[100
       100 40 1000])

    m=m+1;
endfunction
```

Scilab Code 10.3 `imc_virtual.sce`

```
mode(0);
// For scilab 5.1.1 or lower version users, use scicos
    command to open scicos diagrams instead of xcos

global fdfh fdt fncr fncw m err_count y limits
   sampling_time m

// **********************
sampling_time=1;    // In seconds. Fractions are allowed
// **********************//

exec ("imc_virtual.sci");

ok = init();

   if ok~= []  // open xcos only if communication is
       through (ie reply has come from server)
       xcos('imc.xcos');
     else
```

```
    disp("NO NETWORK CONNECTION!");
    return
end
```

Scilab Code 10.4 `imc_virtual.sci`

```
mode(0)


function [stop] = imc_virtual(setpoint,fan,alpha)

global temp heat C0 u_old u_new e_old e_new fdfh fdt
   fncr fncw m err_count stop q heatdisp fandisp
   tempdisp setpointdisp limits m x sampling_time
   e_old_old

e_new = setpoint - temp;

b=((1-alpha)/0.01163);
u_new = u_old + b*(e_new - (0.9723*e_old));


heat=u_new;
u_old = u_new;
e_old = e_new;


    [stop,temp] = comm(heat,fan);// Never edit this
       line
    plotting([heat fan temp setpoint],[0 0 30 0],[100
       100 50 1000])

endfunction
```

Chapter 11

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]

11.1 MPC theory

An equivalent quadratic programming (QP) formulation for constrained DMC (as given in LQG_MPC_notes by Prof Sachin Patwardhan) is given as follows

$$min_{U_f} \frac{1}{2} U_f(k)^T H U_f(k) + F^T U_f(k) \tag{11.1}$$

Subject to

$$AU_f(k) \leq b \tag{11.2}$$

where

$$\tag{11.3}$$

[1]Copyright: Mr.Pratik Behera

$$A = \begin{bmatrix} I_{qm} \\ -I_{qm} \end{bmatrix}$$

$$b = \begin{bmatrix} U^H \\ -U_L \end{bmatrix}$$

Also, we have outputs and manipulated variables related to state variables by

$$x(k+1) = \Phi x(k) + \Gamma(k) + w(k) \tag{11.4}$$

$$y(k) = Cx(k) + v(k) \tag{11.5}$$

$$\tag{11.6}$$

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.

11.2 Implementing MPC

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.

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.

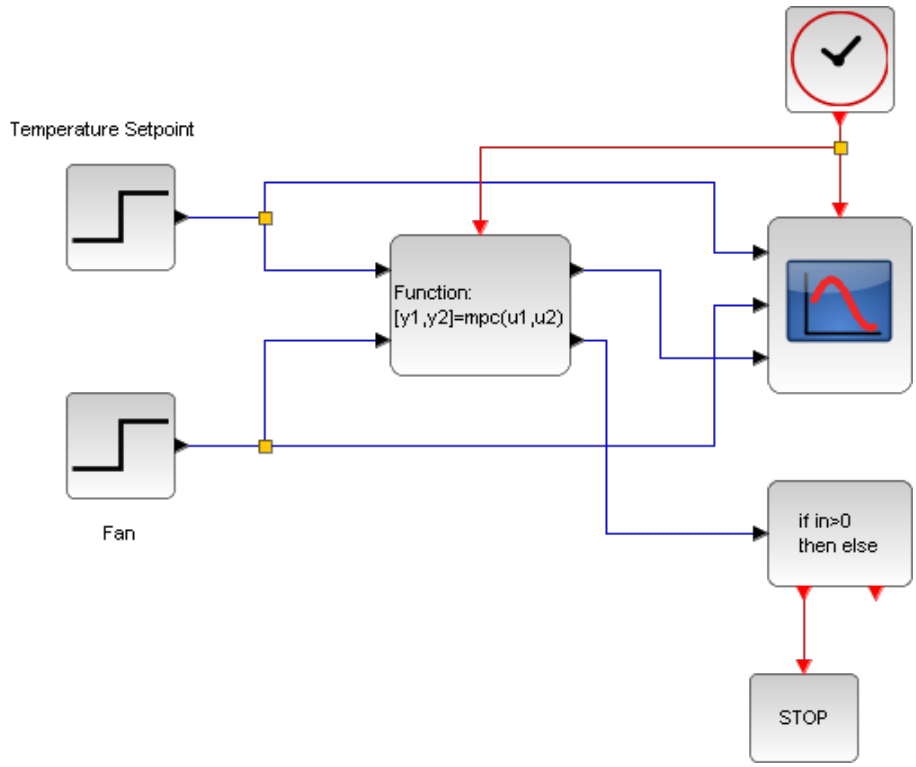

Figure 11.1: Screenshot of the xcos window with step input blocks labeled

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.

Other codes that were used other than `mpc_run.sci` 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)

11.3 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 11.2: 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.

11.4 Sample run to implement MPC

11.4.1 Positive Step Change to Set Point and Fan

Let us consider an experiment, 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.

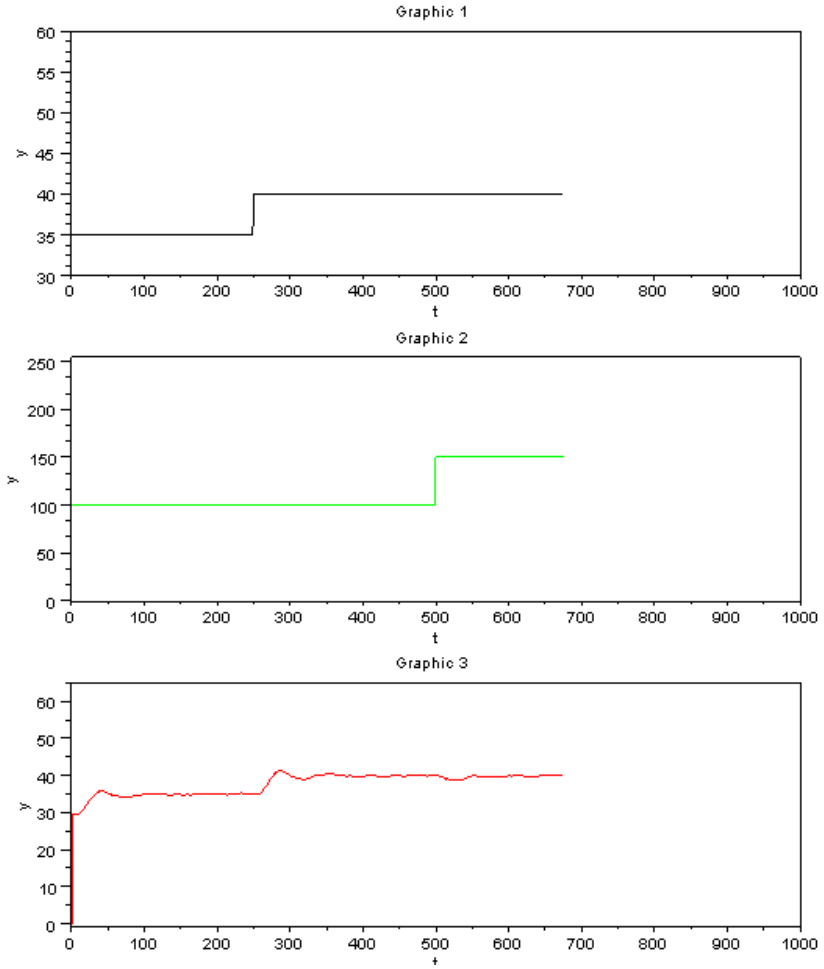

Figure 11.3: Xcos output for positive Step Change to Set Point and Fan

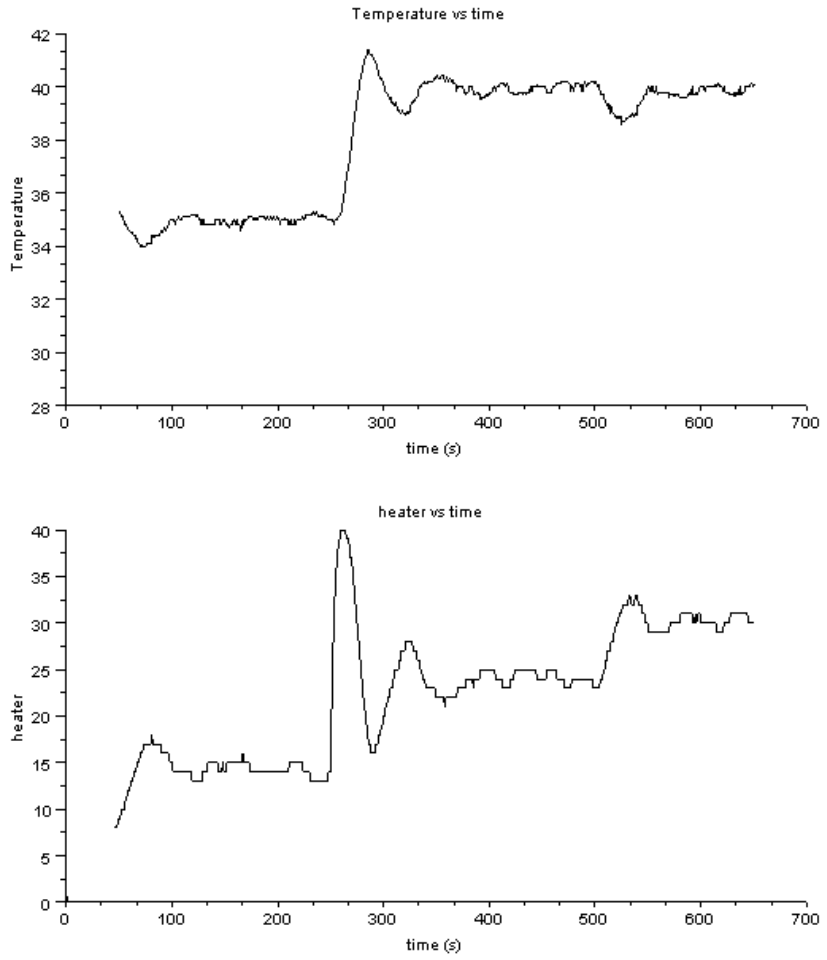

Figure 11.4: Temperature and Heater plot

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=500s, the heater value increases yet again to maintain the same constant temperature of the SBHS blade.

11.4.2 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.

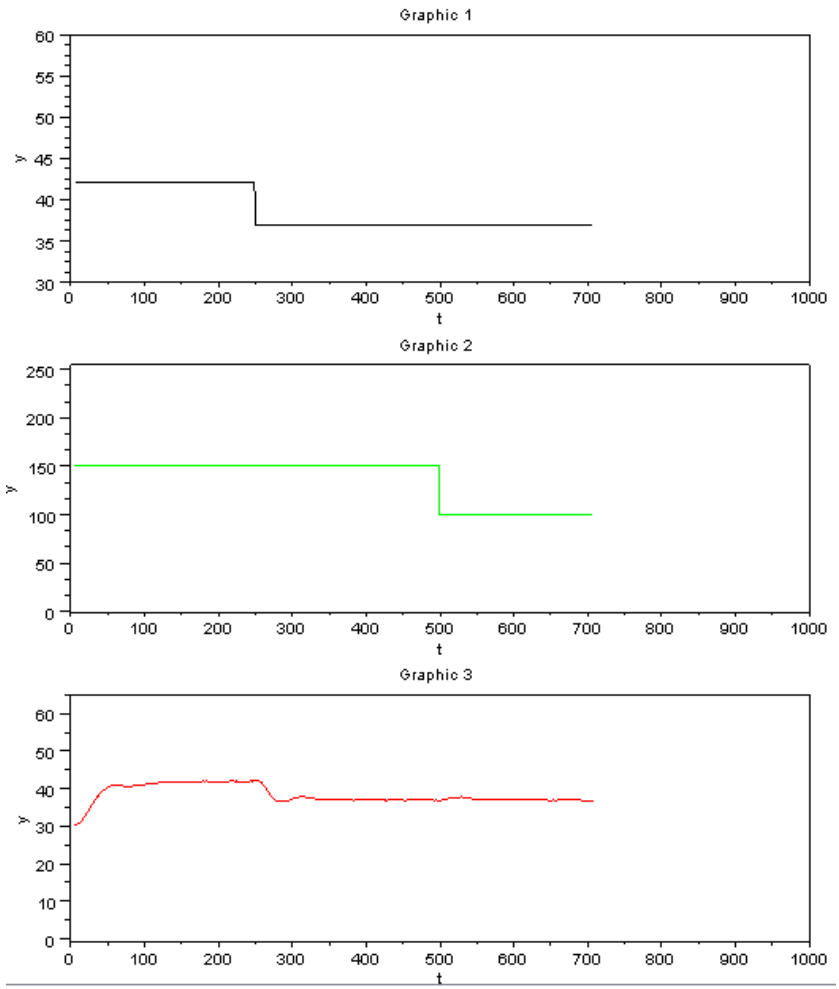

Figure 11.5: Xcos output for negative Step Change to Set Point and Fan

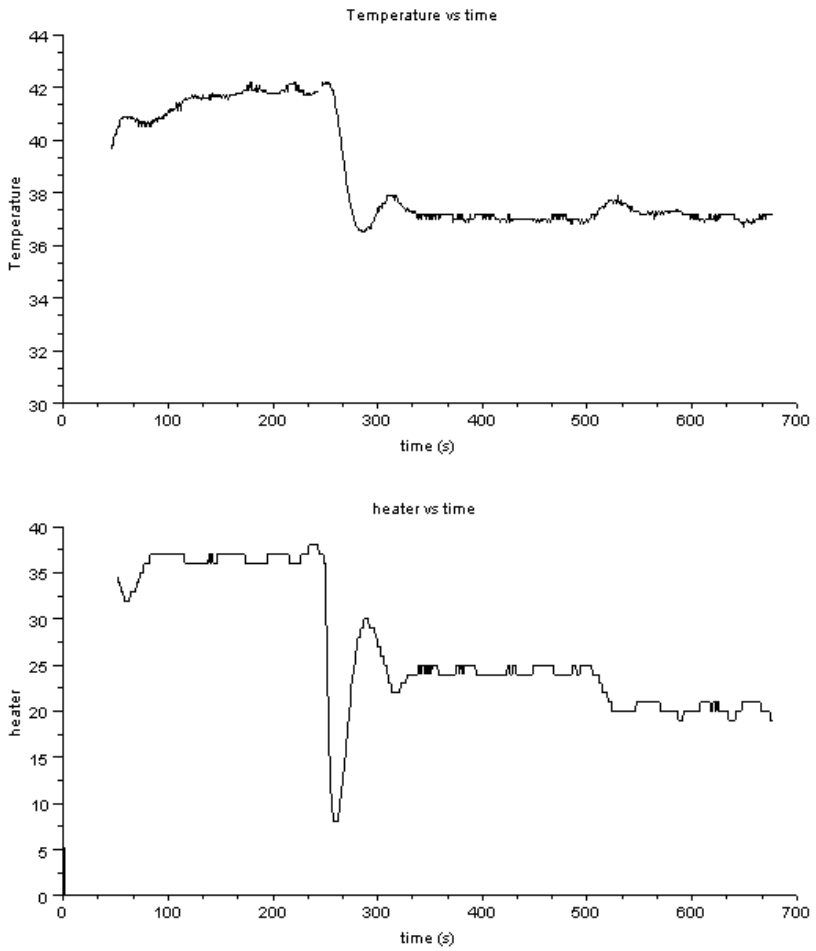

Figure 11.6: Temperature and Heater plot

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=500s, the heater value decreased yet again to maintain the same constant temperature of the SBHS blade.

11.5 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.

11.5.1 Positive Step Change and (We, Wu)=(1,1)

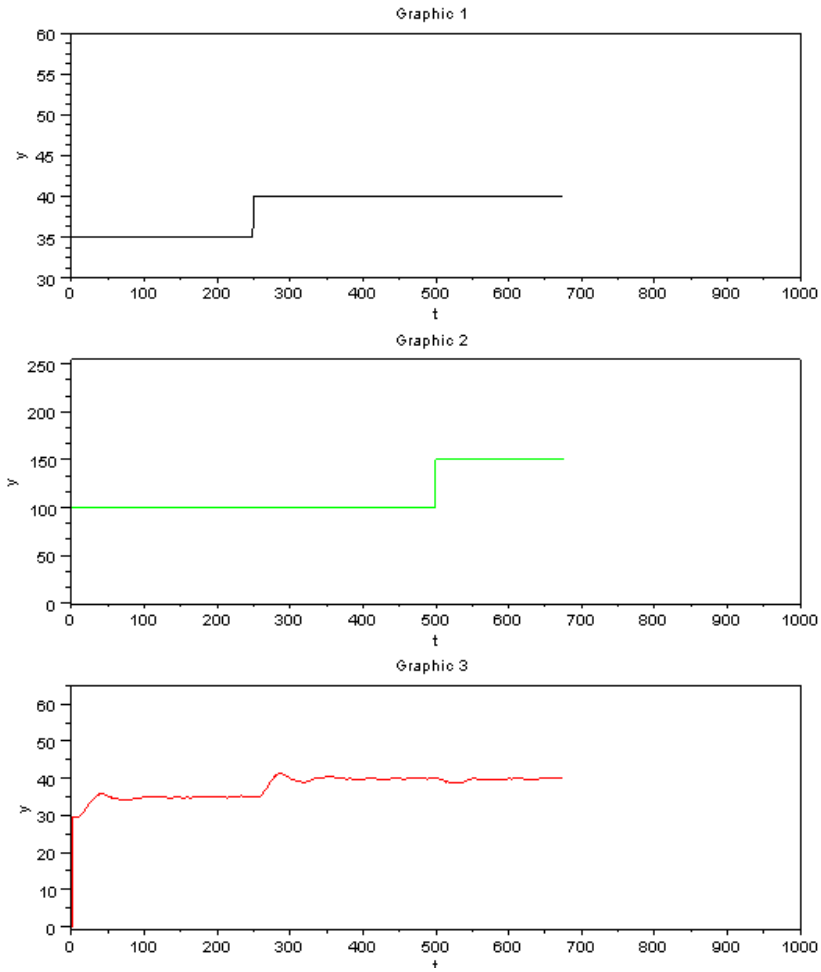

Figure 11.7: Xcos output for positive Step Change and (We, Wu)=(1,1)

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.

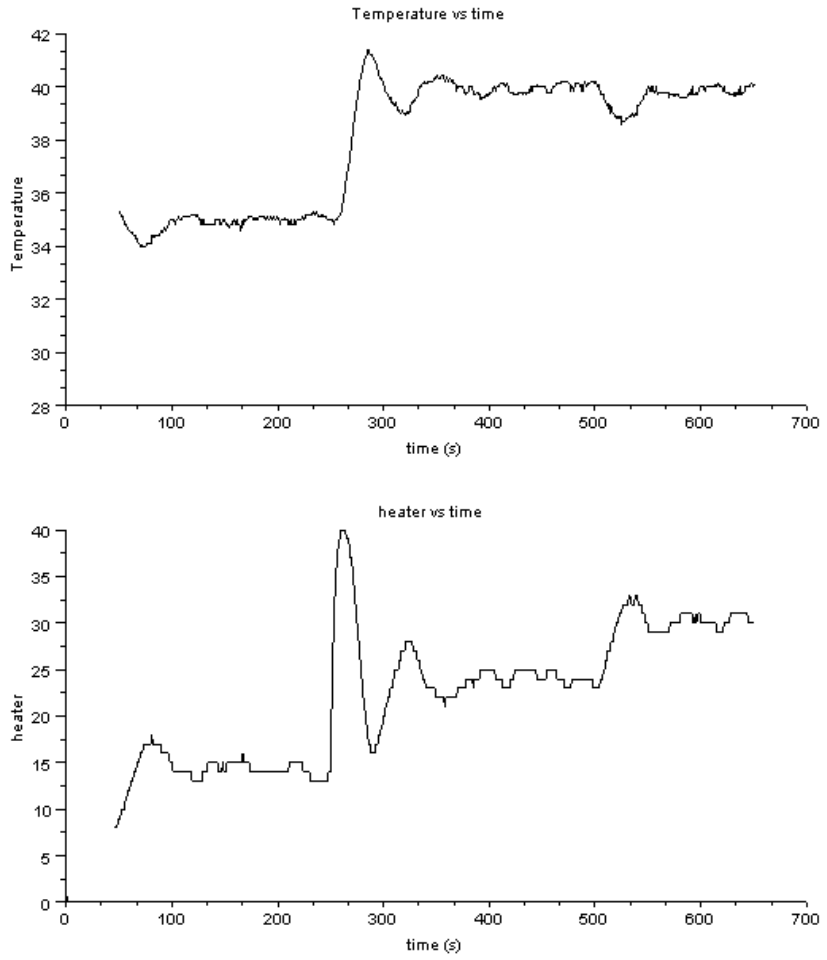

Figure 11.8: Temperature and Heater plot

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.

11.5.2 Positive Step Change and (We, Wu)=(10,10)

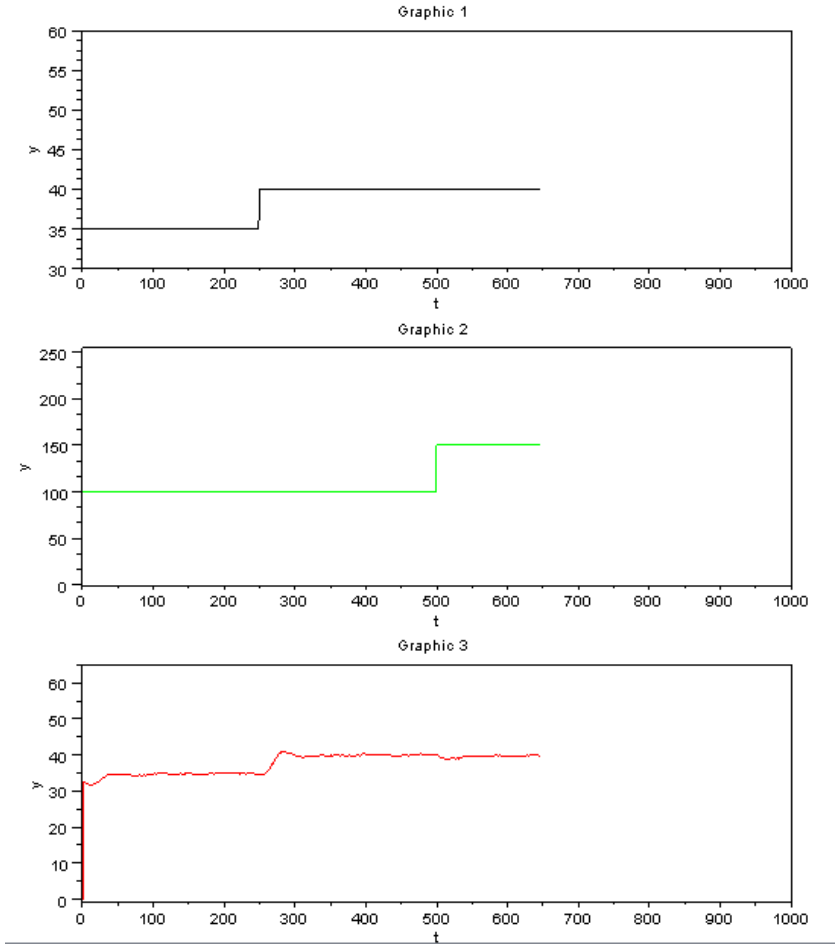

Figure 11.9: Xcos output for positive Step Change and (We, Wu)=(10,10)

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.

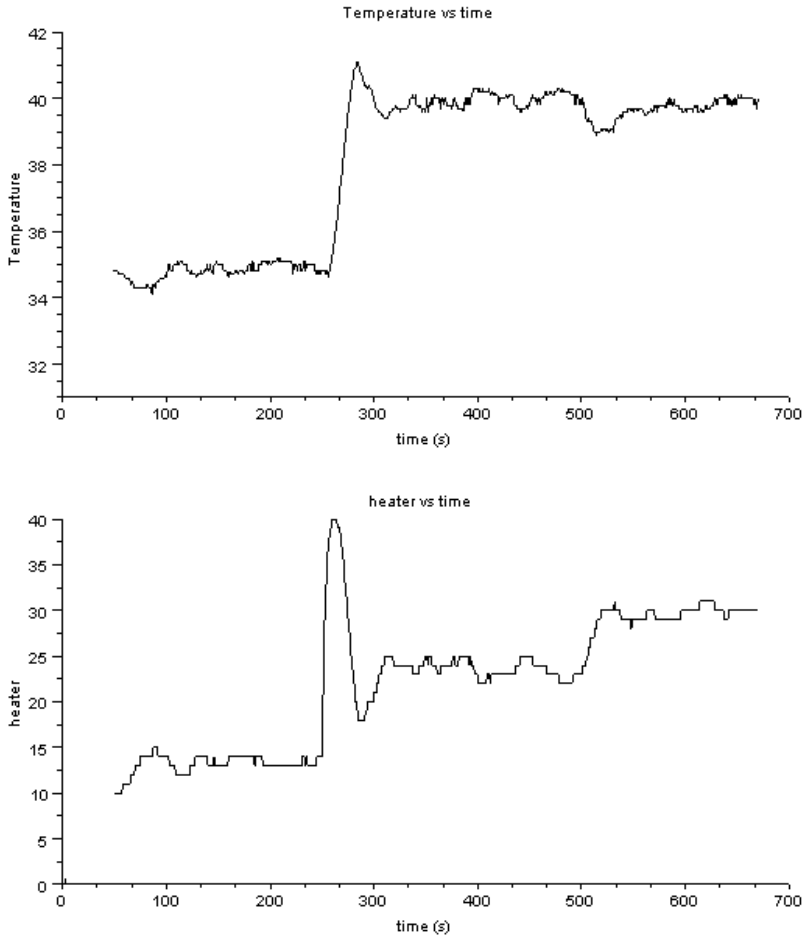

Figure 11.10: Temperature and heater plot

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.

11.5.3 Positive Step Change and (We, Wu)=(40,40)

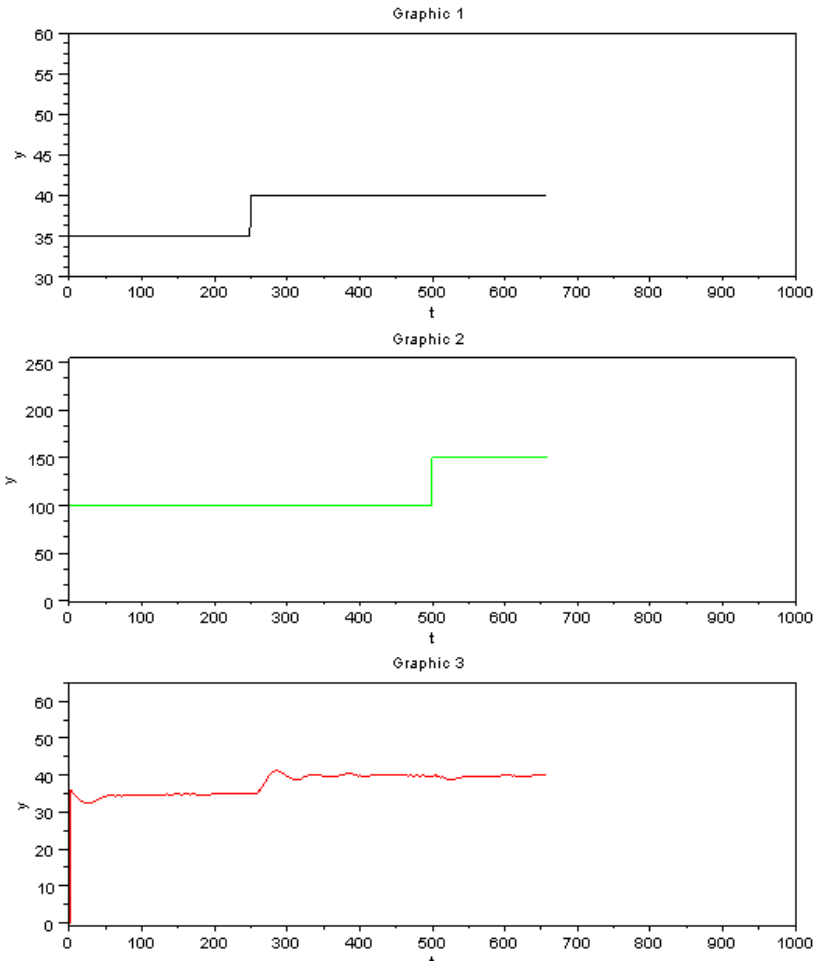

Figure 11.11: Xcos for positive Step Change and (We, Wu)=(40,40)

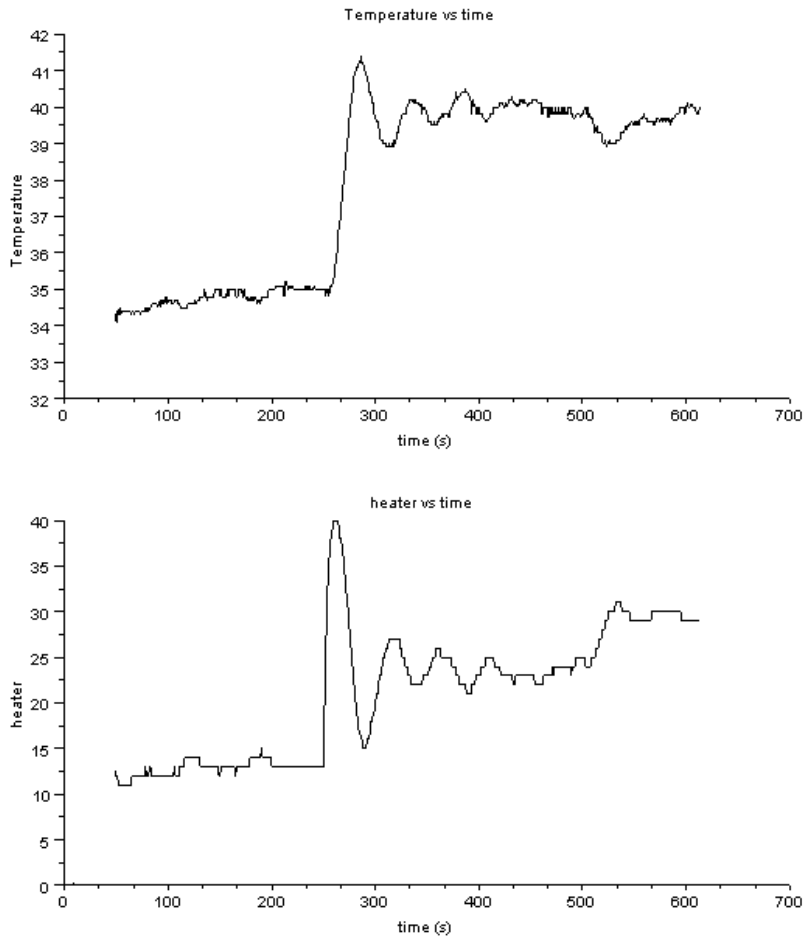

Figure 11.12: Temperature and heater plot

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.

11.5.4 Negative Step Change and (We,Wu)=(1,1)

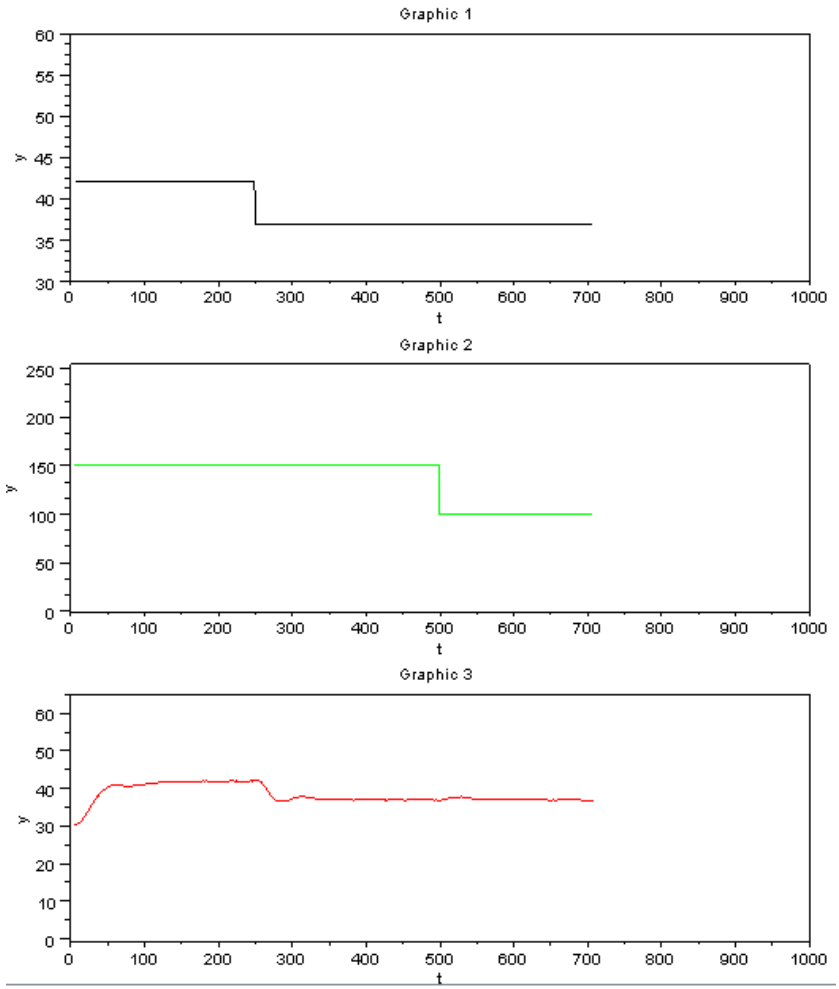

Figure 11.13: Xcos for negative Step Change and (We, Wu)=(1,1)

Here we expect somewhat similar results as was the case with positive step in temperature set point.

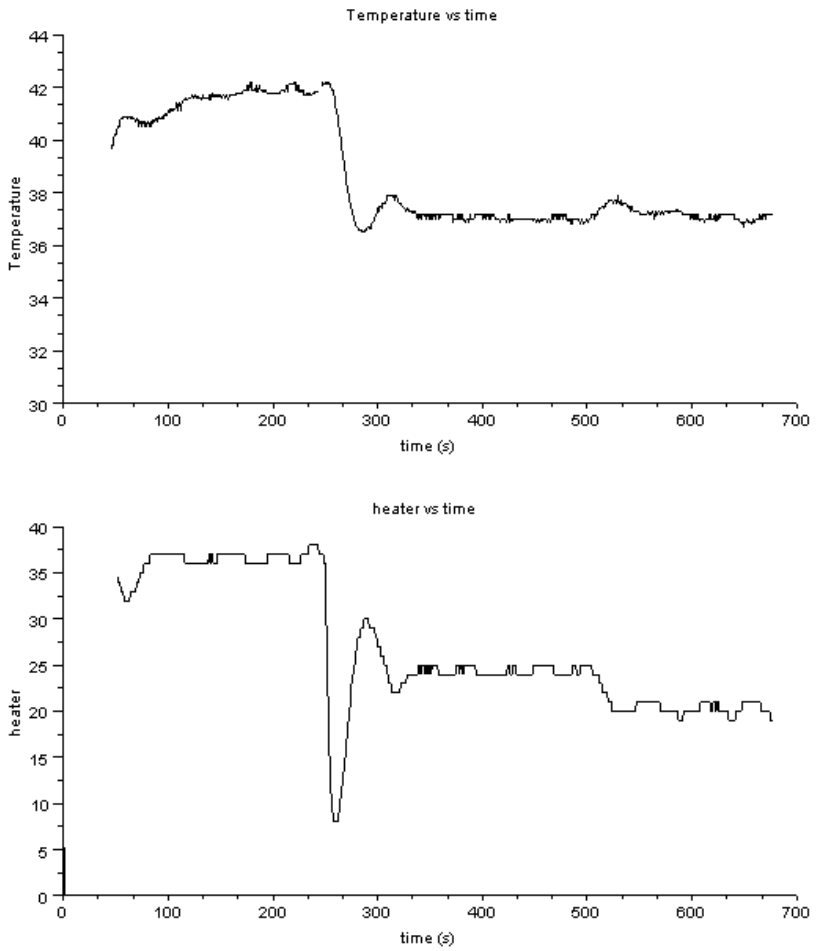

Figure 11.14: Temperature and heater plot

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

11.5.5 Negative Step Change and (We, Wu)=(10,10)

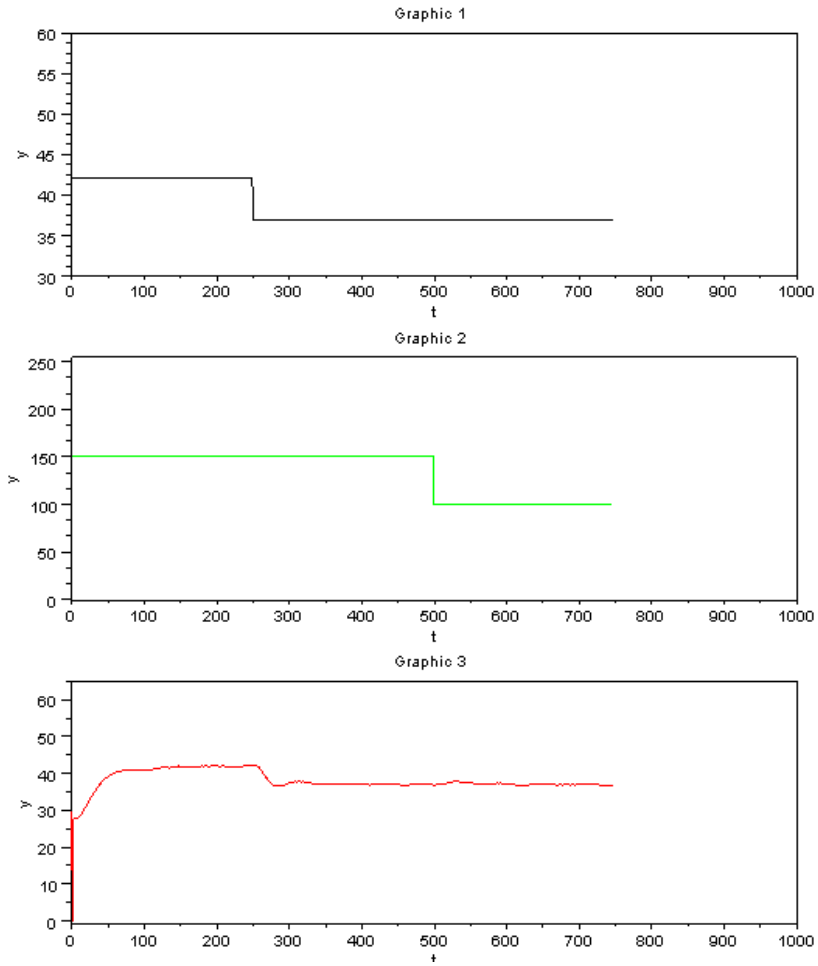

Figure 11.15: Xcos for negative Step Change and (We, Wu)=(10,10)

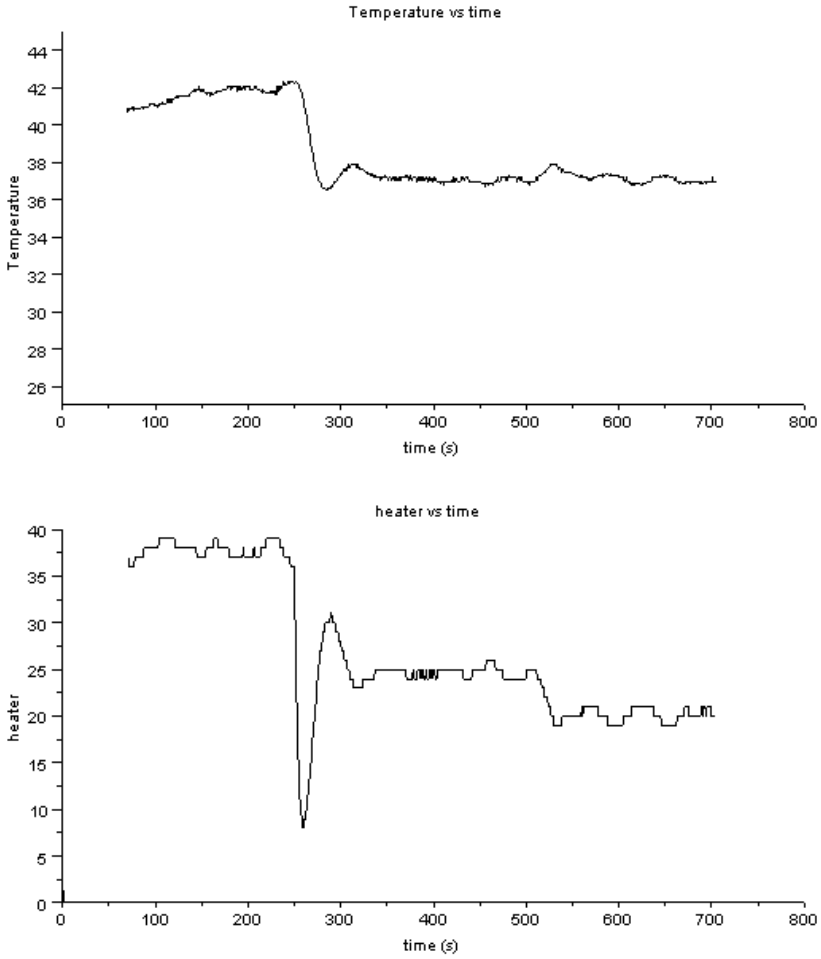

Figure 11.16: Temperature and heater plot

11.5.6 Negative Step Change and (We, Wu)=(40,40)

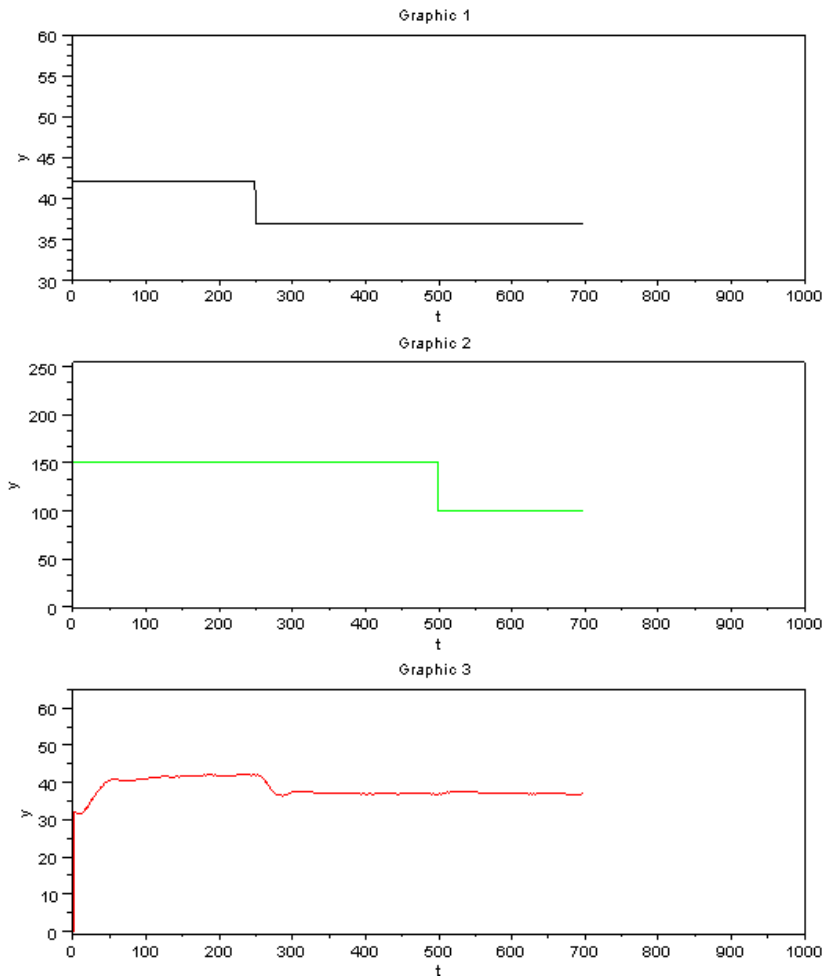

Figure 11.17: Xcos for negative Step Change and (We, Wu)=(40,40)

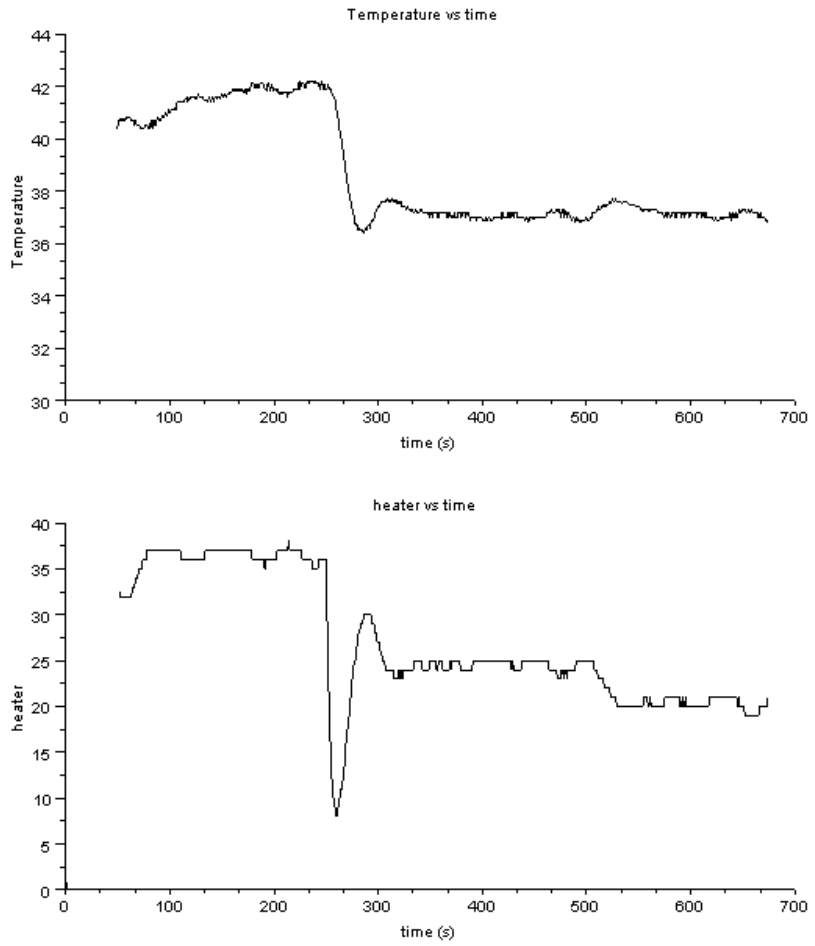

Figure 11.18: Temperature and heater plot

11.6 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.

11.6.1 We =100 and Wu = 2

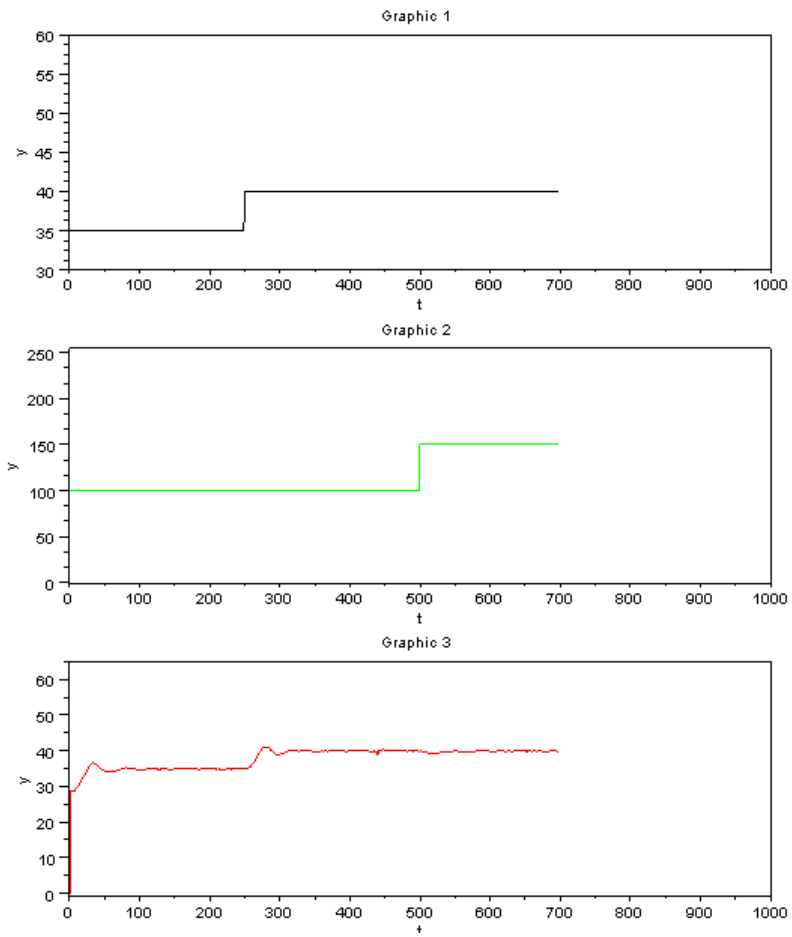

Figure 11.19: Output for We =100 and Wu = 2

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 stabilzes at the required setpoint. The settling time observed is fairly low.

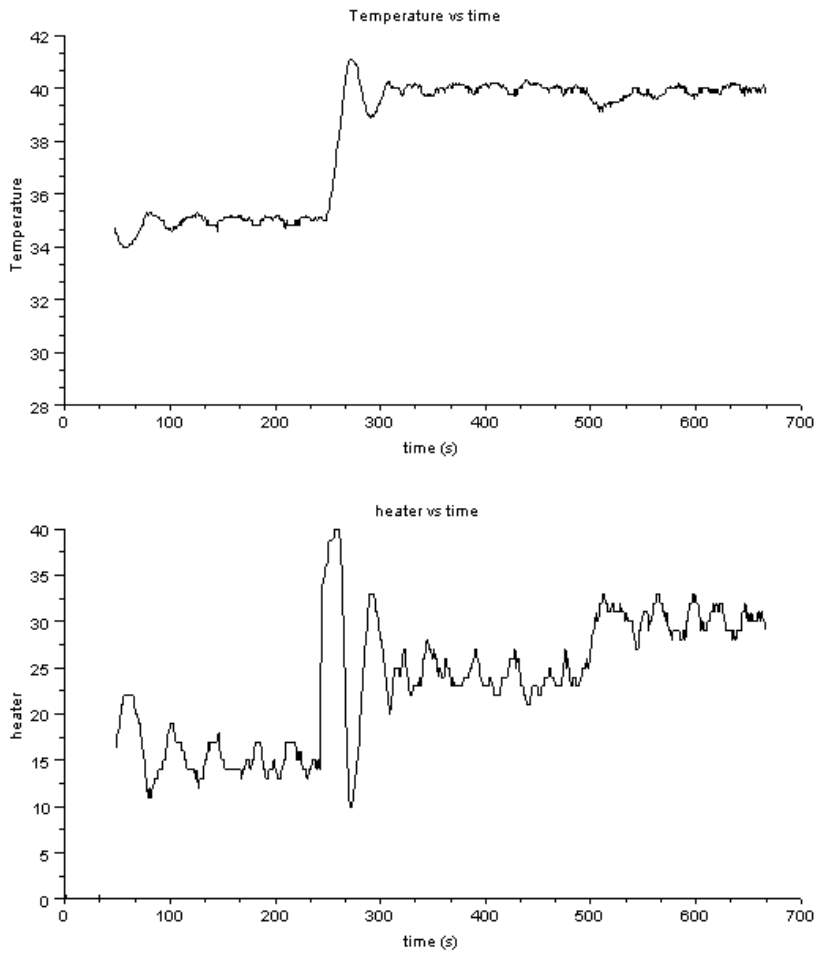

Figure 11.20: Temperature and heater plot

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.

11.6.2 We = 2 and Wu = 100

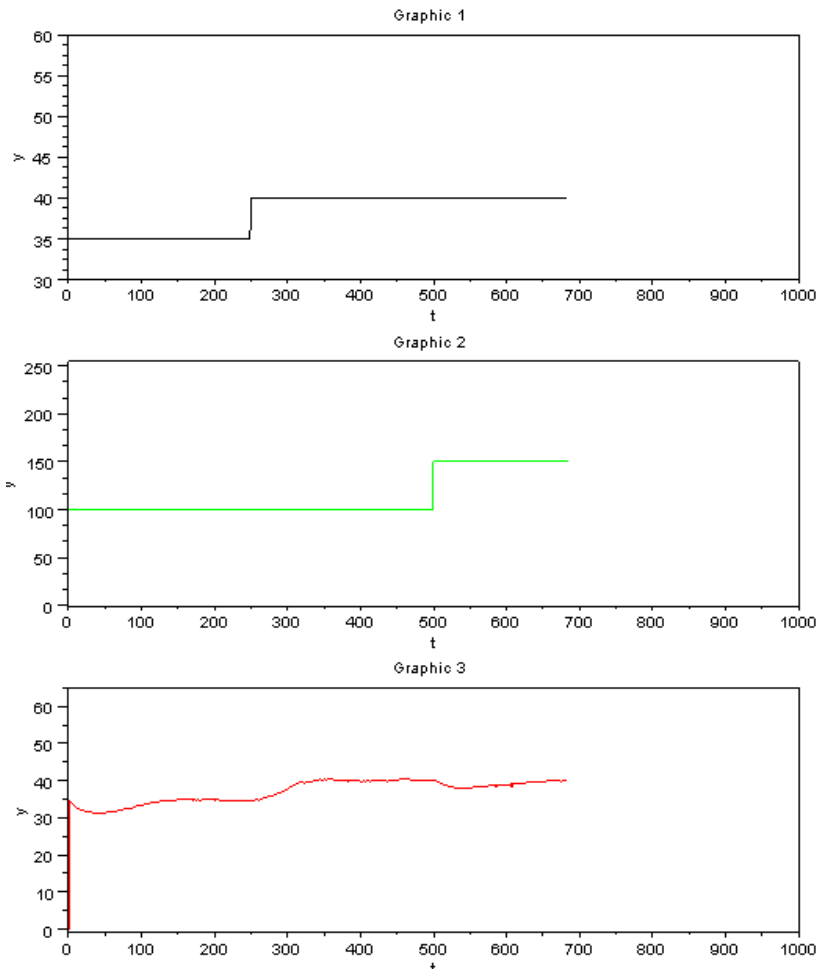

Figure 11.21: Output for We = 2 and Wu = 100

With increase in Wu, we observe that the temperature stabilzes at the required stepoint, 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.

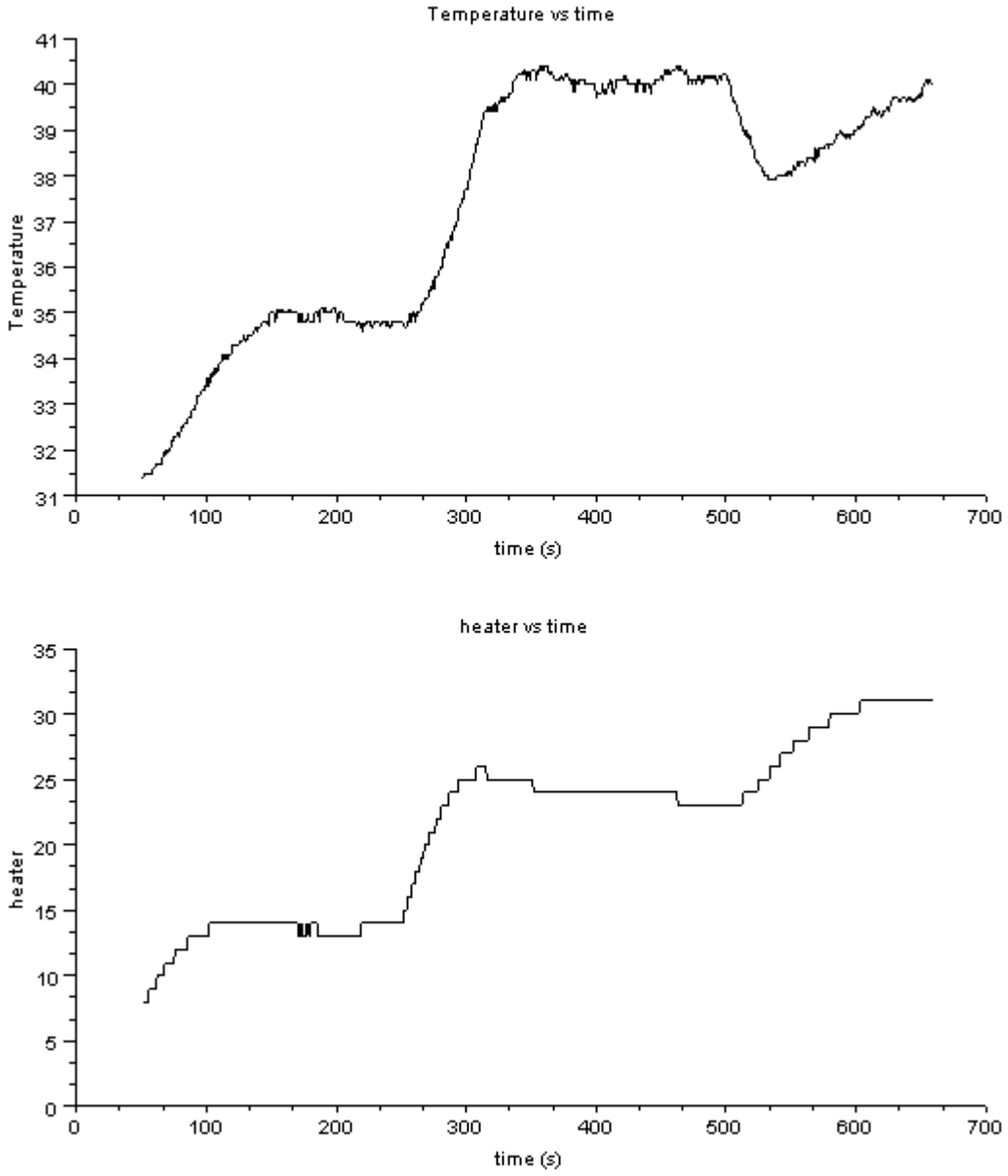

Figure 11.22: Temperature and heater plot

11.6.3 We =10 and Wu = 100

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.

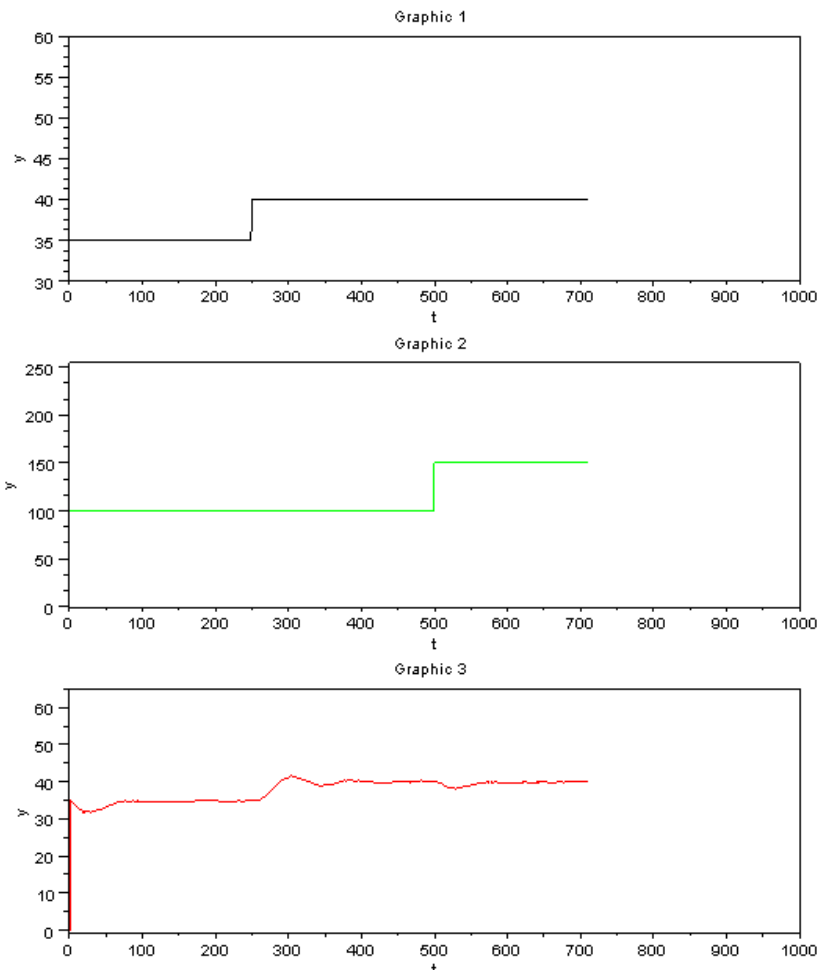

Figure 11.23: Output for We =10 and Wu = 100

We observe that this experiments performs better than in the last section (where We was 2). It is slighly oscillatory and also, the settling time decreased much as compared to last experiment.

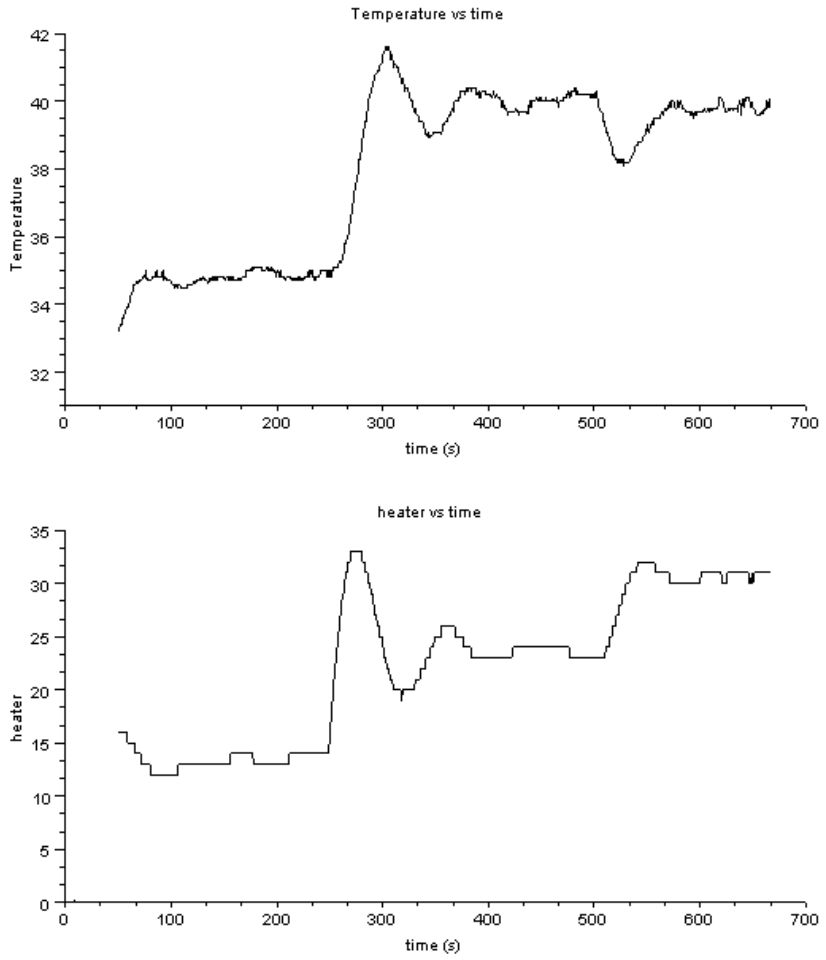

Figure 11.24: Temperature and heater plot

11.6.4 We =100 and Wu = 10

We now do a similar study for the case of Wu. We increase the value of Wu to 10, keeping We constant at 100.

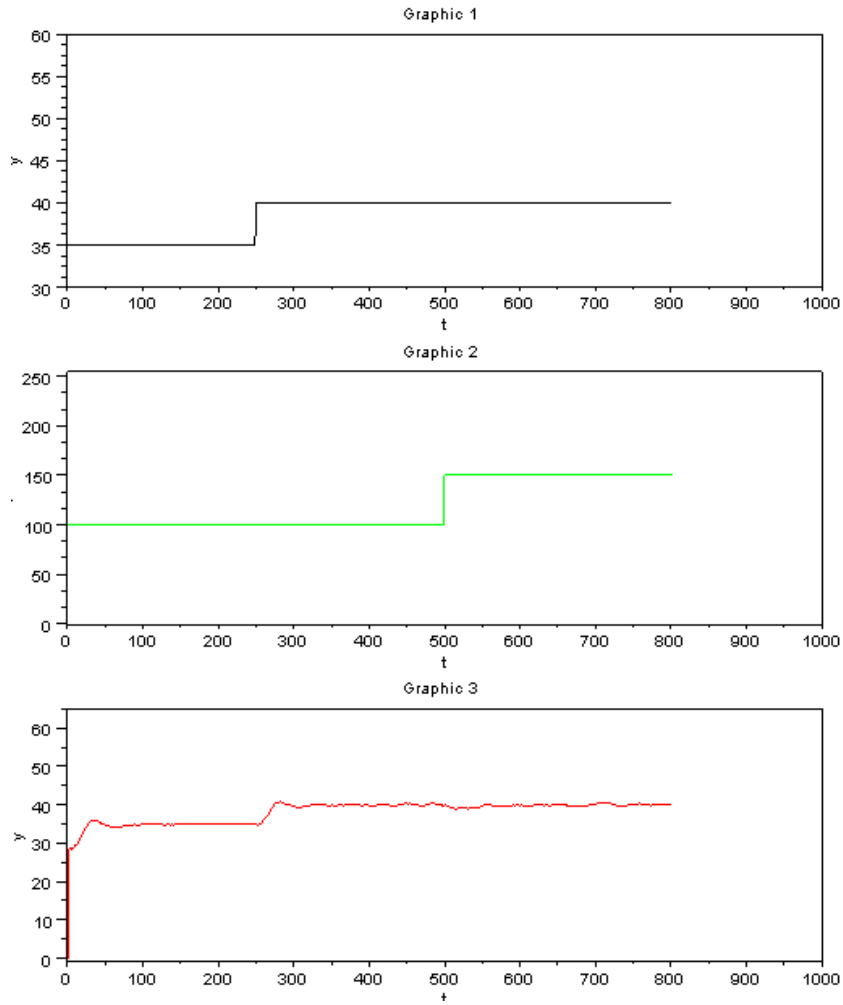

Figure 11.25: Output for We =100 and Wu = 10

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.

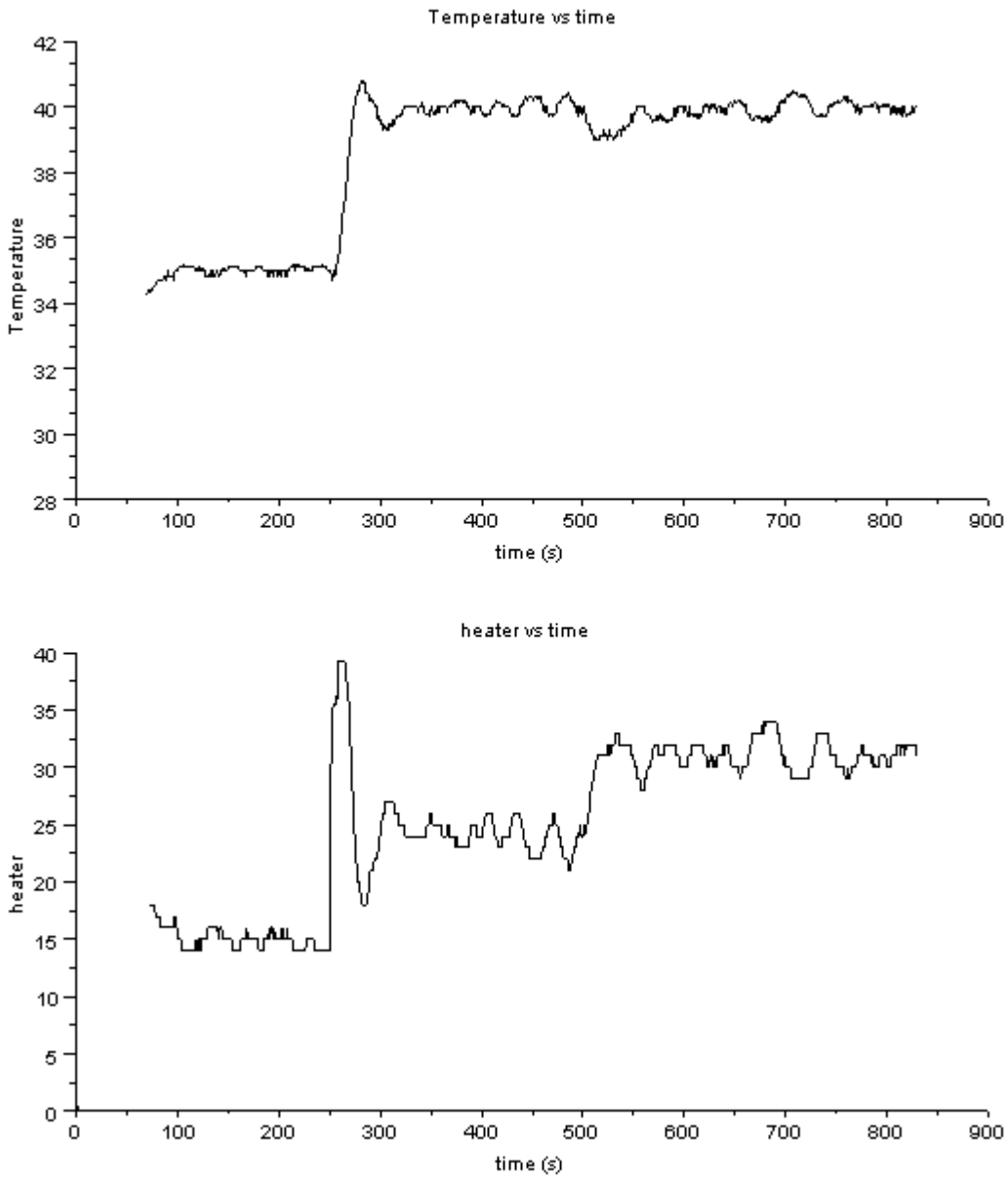

Figure 11.26: Temperature and heater plot

11.6.5 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.

11.7 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.

11.7.1 For positive step change in Set point and Fan speed

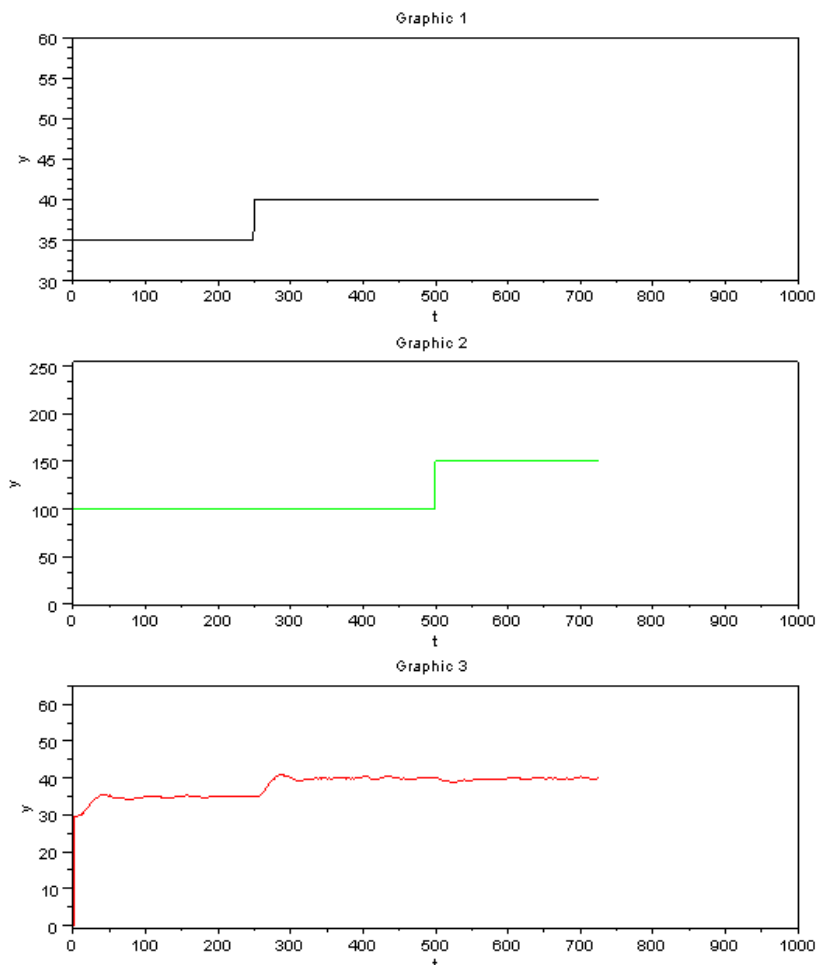

Figure 11.27: Xcos putput for positive step change with q=2

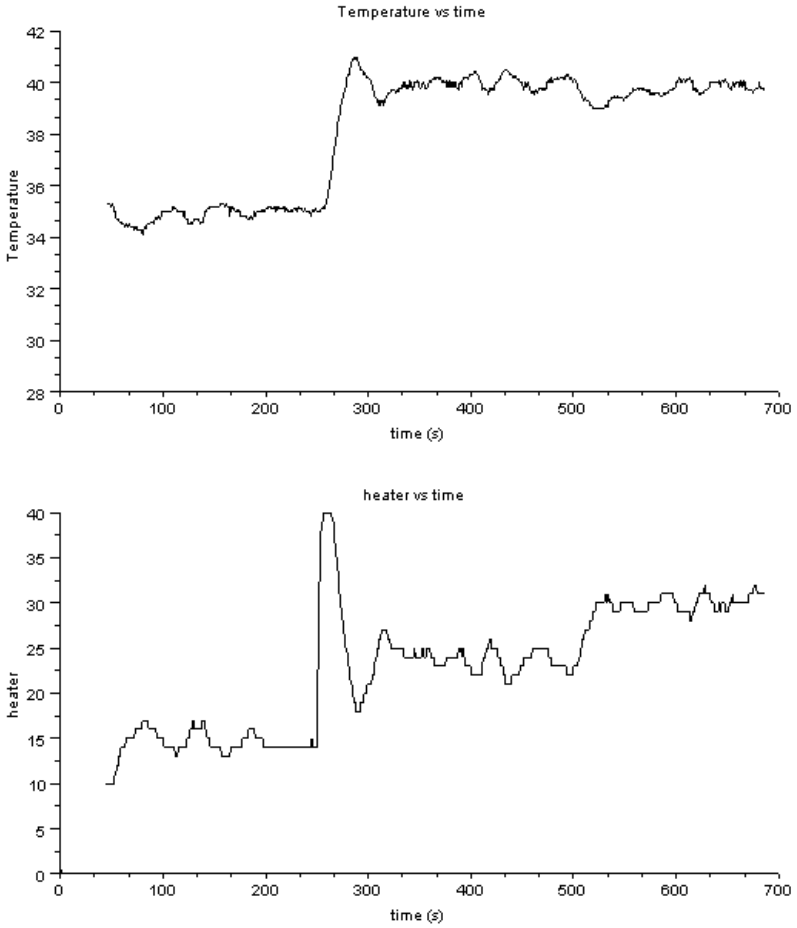

Figure 11.28: Temperature and heater plot

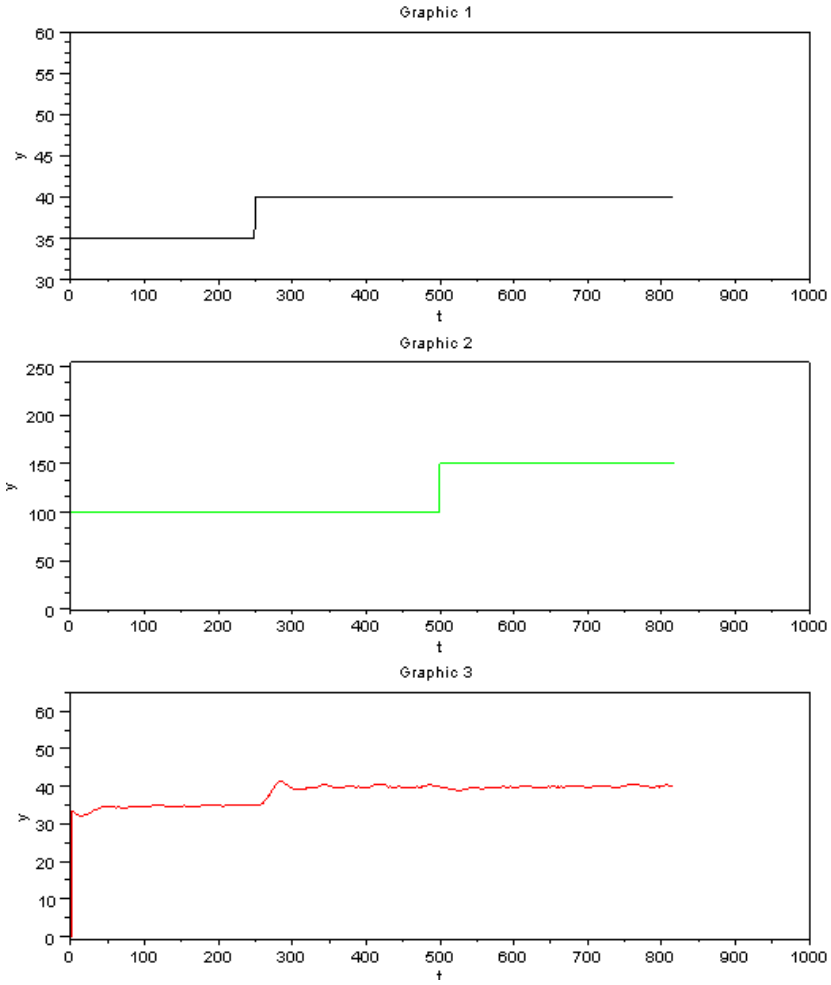

Figure 11.29: Xcos putput for positive step change with q = 3

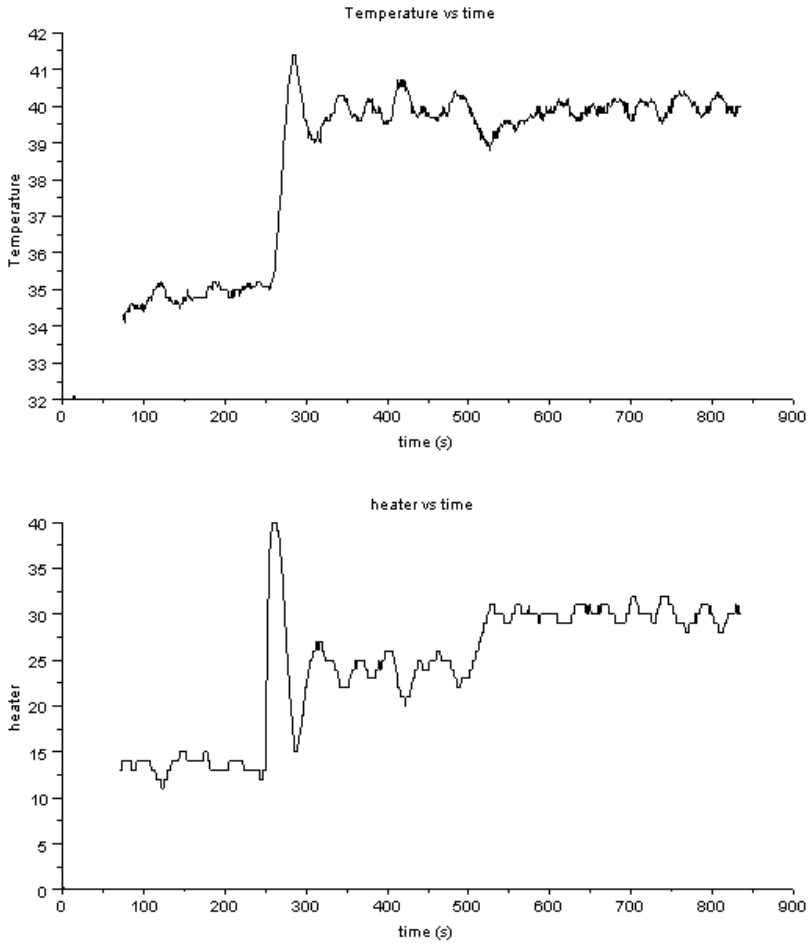

Figure 11.30: Temperature and heater plot

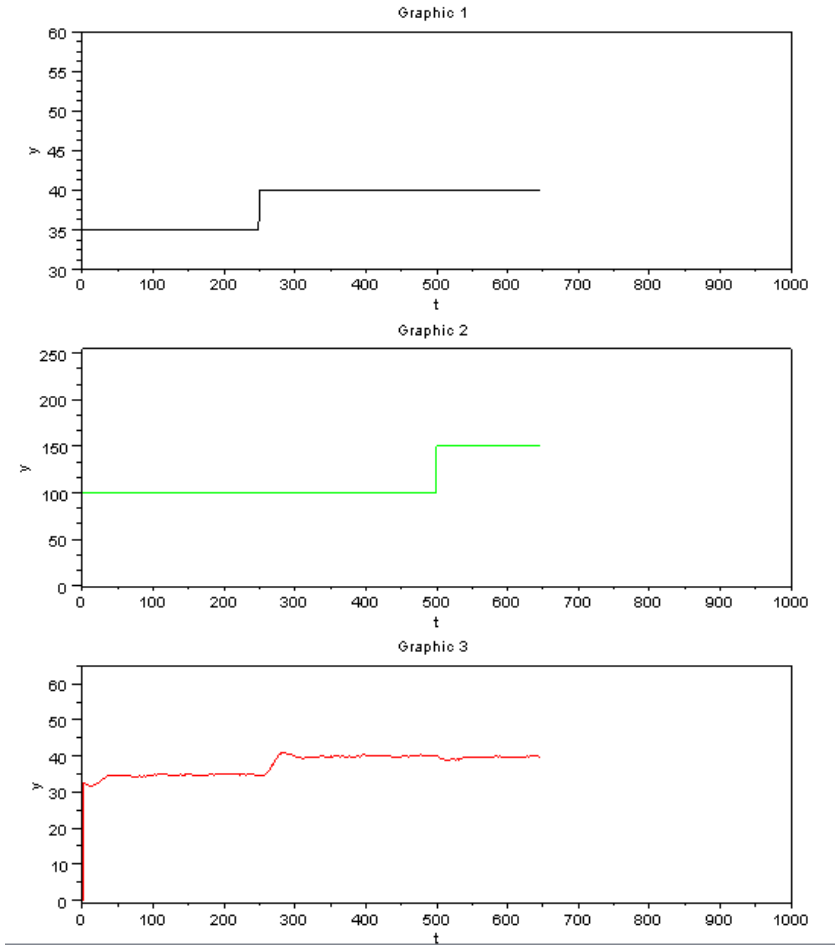

Figure 11.31: Xcos putput for positive step change with q = 4

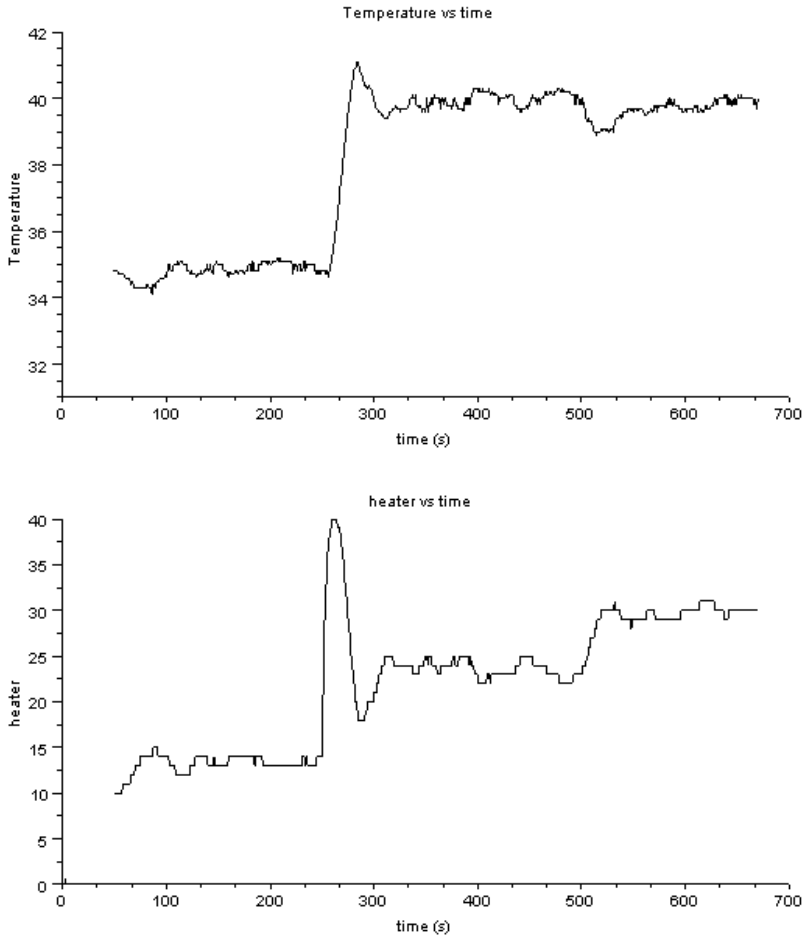

Figure 11.32: Temperature and heater plot

11.7.2 For negative step change in Set point and Fan speed

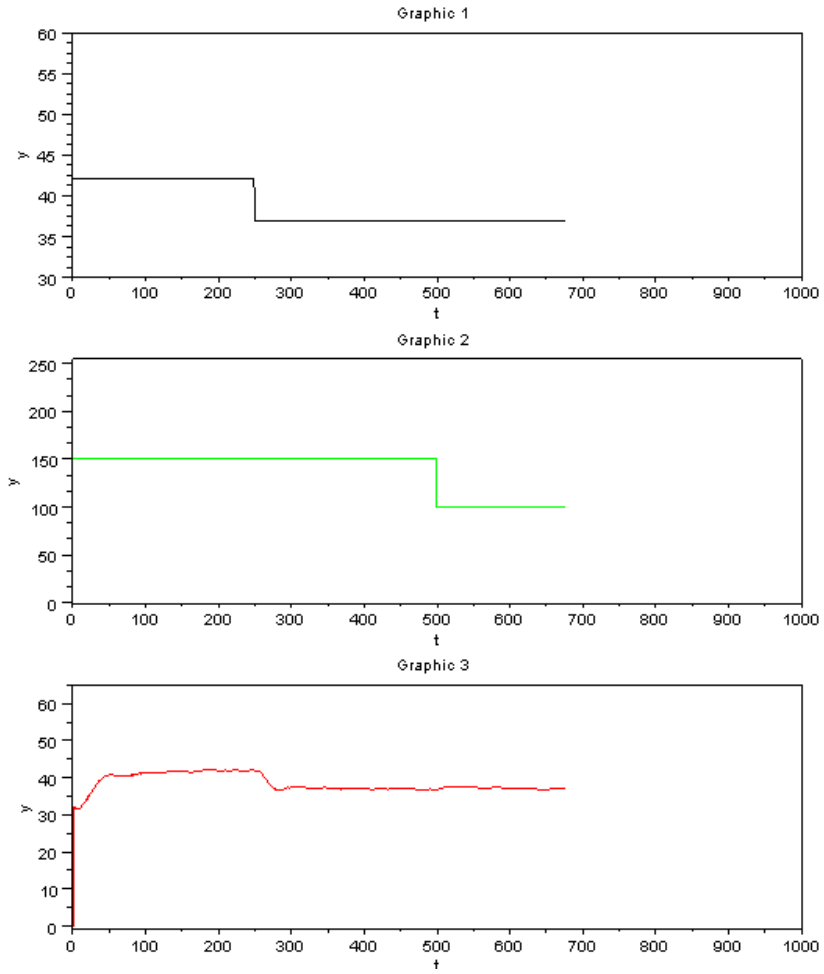

Figure 11.33: Xcos putput for negative step change with q = 2

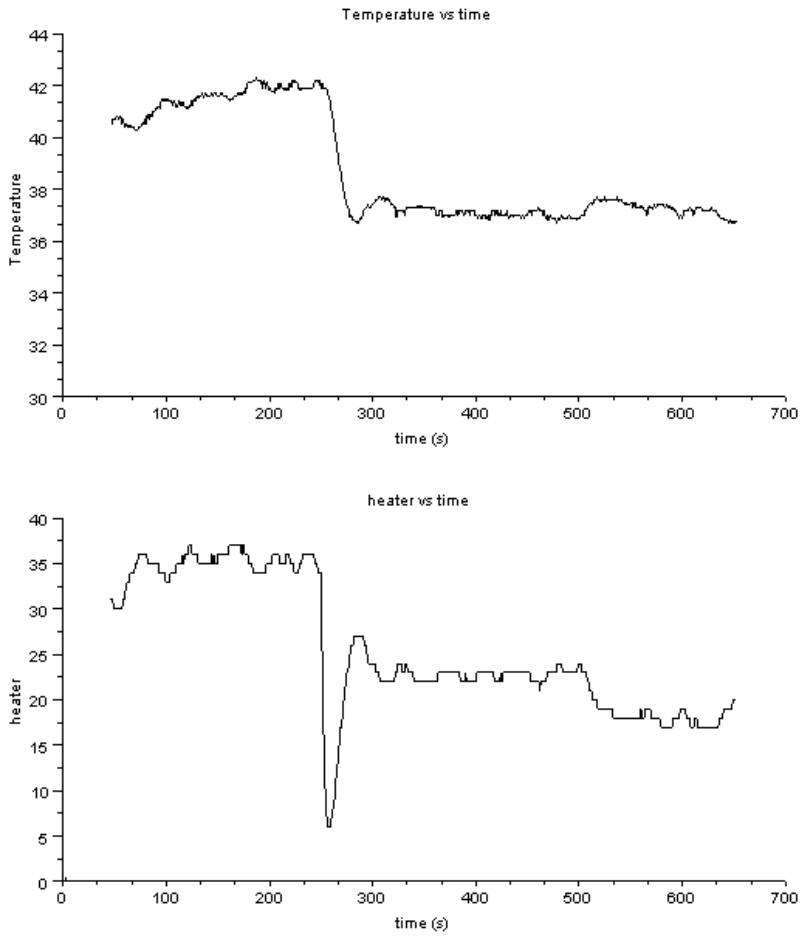

Figure 11.34: Temperature and heater plot

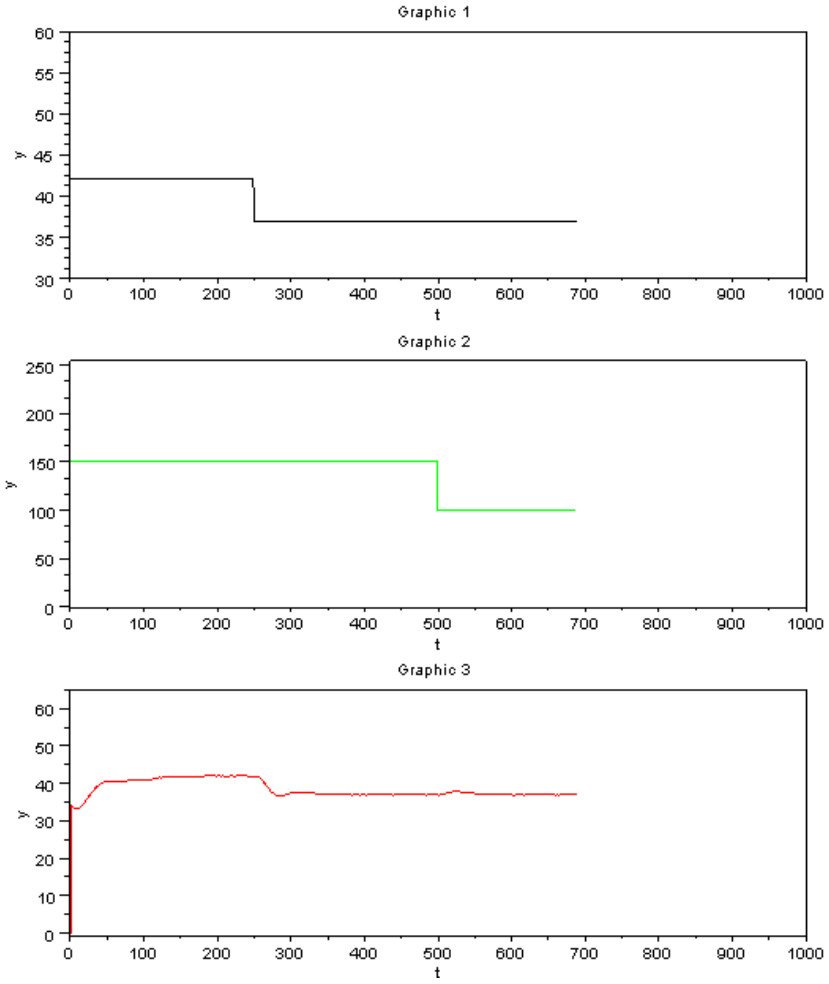

Figure 11.35: Xcos putput for negative step change with q = 3

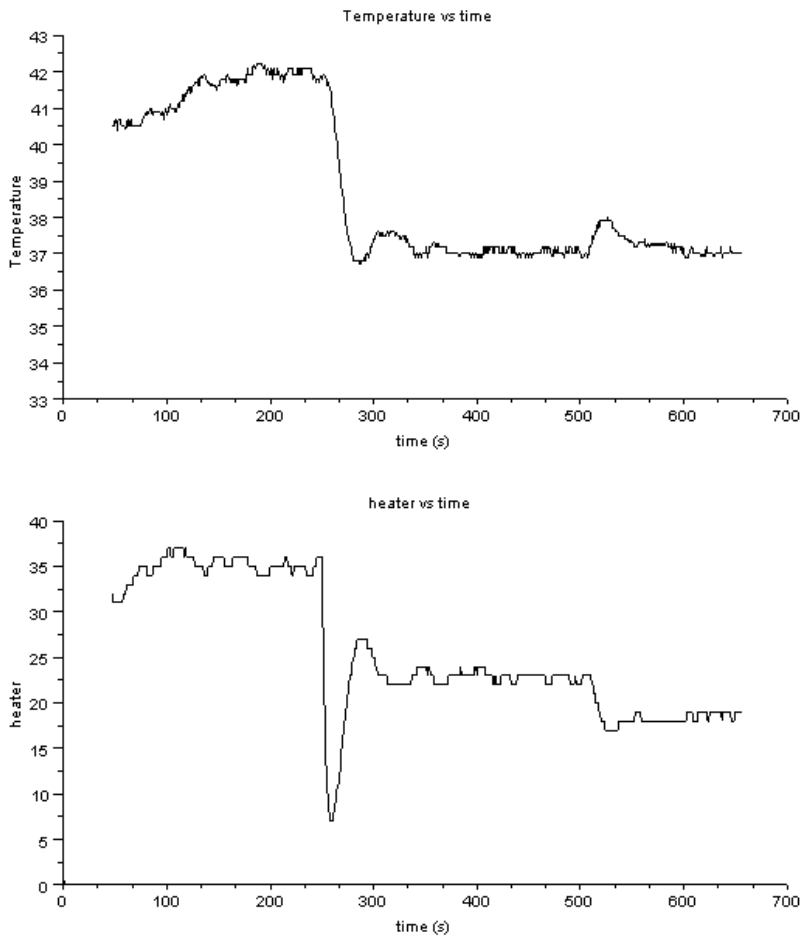

Figure 11.36: Temperature and heater plot

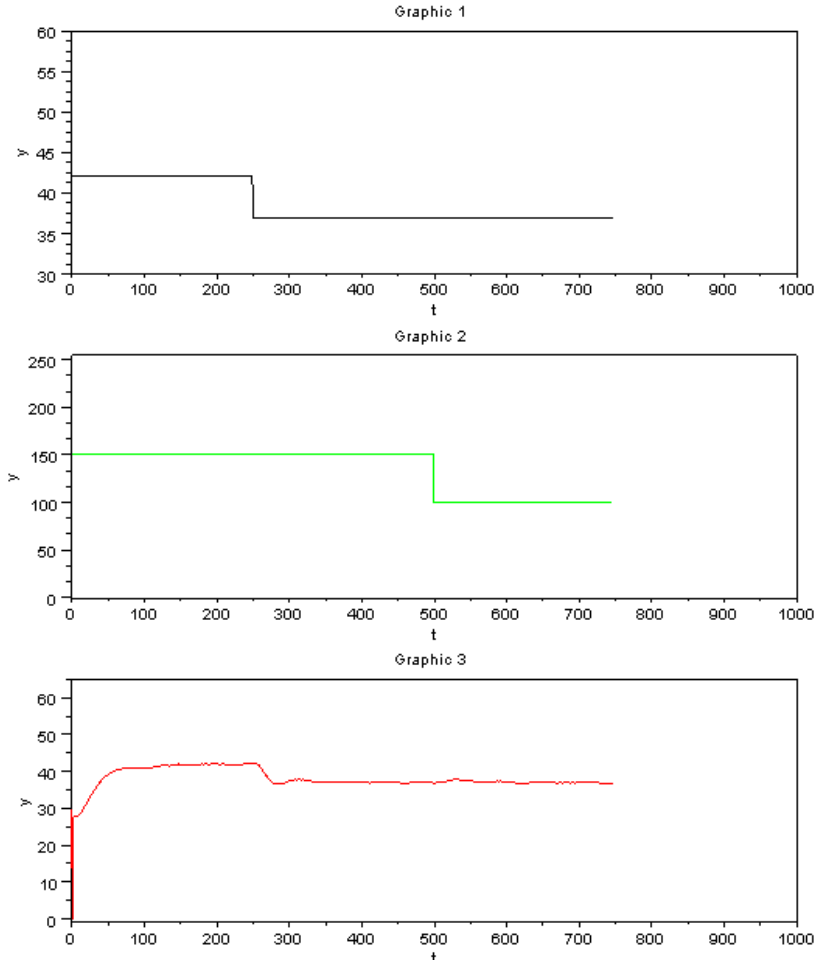

Figure 11.37: Xcos putput for negative step change with q = 4

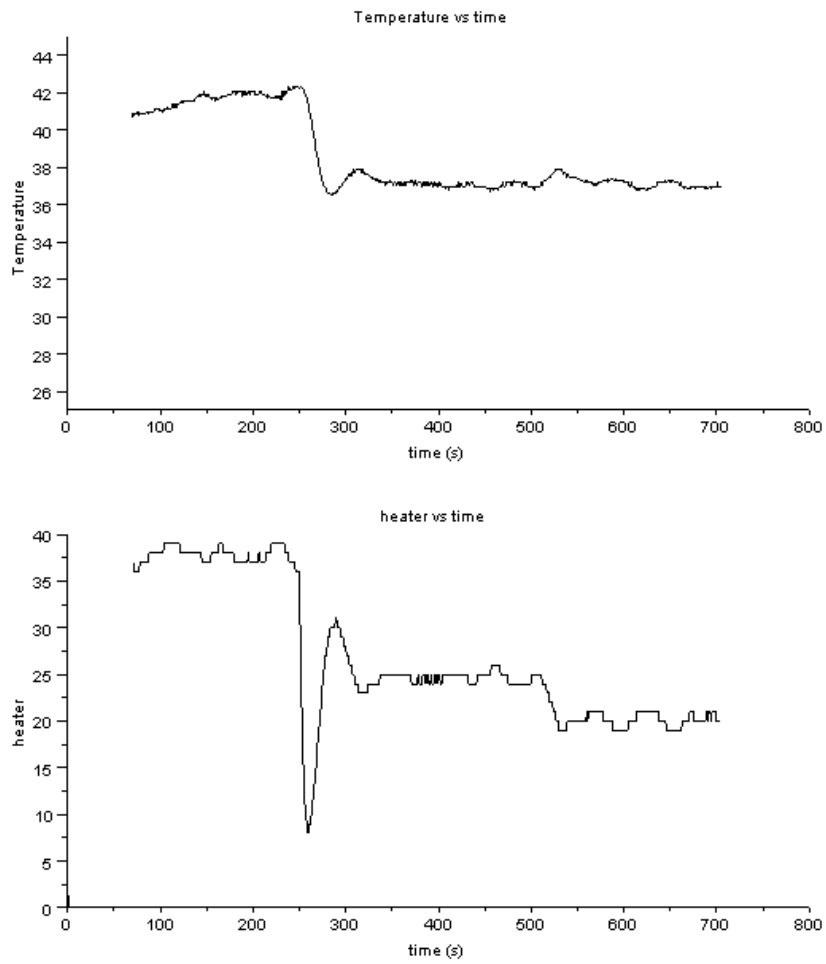

Figure 11.38: Temperature and heater plot

11.7.3 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.

11.8 Implementing MPC locally

The detailed procedure to perform a local experiment is explained in Chapter2. A summary of the same is provided in section 2.3 It is same for this section with following changes.

1. Step1: The working directory is `mpc`
2. Step2: Same
3. Step3: Same
4. Step4: Same
5. Step5: Load mpc function by executing command
   `exec<space>mpc_init_local.sci`
6. Step6: Load Xcos code for mpc experiment using the command
   `xcos<space>mpc.xcos`
7. Step7: Same

11.9 Implementing MPC virtually

The detailed procedure to perform a virtual experiment is explained in Chapter3. A summary of the same is provided in section 3.5. It is same for this section with following changes.

1. Step1: The working directory is `mpc`. Open this directory.
2. Step2: Same
3. Step3: Same
4. Step4: Switch to the MPC experiment directory and double-click on the file `mpc.sce`. This will launch scilab and also open the file `mpc.sce` in the scilab editor. Linux users will have to launch scilab manually. They also have to change the working directory to `mpc` and then open the `mpc.sce` file in the scilab editor.
5. Step5: Same
6. Step6: Execute the file `mpc.sce`. Expect the PI controller xcos diagram to open automatically. If this doesnt happen, check the scilab console for error message.
7. Step7: Execute the MPC controller xcos diagram.
8. Step8: Same

11.10 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).

11.11 Acknowledgement

Firstly, I (Pratik Behera)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.

11.12 General Information on Experiments for this Project

All the experiments for this project was performed remotely on SBHS, 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.

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:
Kp=0.37, time constant = 45s and delay = 7s.

Using the above, we obtained the plant transfer function:

$$G_p = \frac{0.37}{1 + 45s} e^{-7s} \tag{11.7}$$

Scilab Method to calculate State Space matrices
State space matrices for a transfer function can be calculated as follows using Scilab:

```
s=poly(0,'s');
TFcont=syslin('c',[kp*(1-0.5*D)/(tau*s+1)/(1+0.5*D)]);
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 dervied for our SBHS was very close to the transfer function derived in an earlier attempt. So, using the values of A, B and C which were already calculated by him previously, we obtain the following exact values:

$$A = \begin{bmatrix} 0.9780 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0079 \end{bmatrix}$$

11.13 Scilab Code

Scilab Code 11.1 `mpc.sci`

```
function [stop] = mpc(Tsp,fan)
    global fdfh fdt fncr fncw m err_count stop p q
       xk_old heat temp heat_old

    heat = mpc_run(temp,heat_old,Tsp);

    [stop,temp] = comm(heat,fan);// Never edit this
       line
    plotting([heat fan temp Tsp],[0 40 25 0],[100 70
       50 1000]);

    heat_old = heat;

return
endfunction
```

Scilab Code 11.2 `mpc_init_local.sce`

```
// For scilab 5.1.1 or lower version users,
//use scicos command to open scicos diagrams instead
    of xcos
```

```

global err_count y p q xk_old Tsp heats fan temp heat

p = 40; // prediction horizon
q = 4;  // control horizon
xk_old = zeros(8,1);
Tsp=1;
heats=1;
fan=1;
temp=1;

exec ("mpc_local.sci");
exec("mpc_run.sci");
```

Scilab Code 11.3 `mpc_local.sci`

```
mode(0)
function [temp] = mpc(Tsp,fan)
global temp heat_in fan_in C0 u_old u_new e_old e_new
    e_old_old

global heatdisp fandisp tempdisp setpointdisp
    sampling_time m name
// heats = 1;
u_new = mpc_run(temp,heats,Tsp);


heat = u_new;

     temp = comm(heat,fan);

     plotting([heat fan temp Tsp],[0 0 20 0],[100 100
        40 1000])

    m=m+1;
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.