UNIVERSITY OF WARWICK

School of Engineering

Programming with MATLAB Laboratory Worksheet – 2hrs

Follow the exercises in this worksheet to construct a MATLAB program that will emulate aspects of the Temperature Sensing Instrumentation Model as specified in your **ES2A7_MATLAB_Introduction** sheet. This exercise will use basic MATLAB programming structures. PLEASE READ AND FOLLOW THIS SHEET CAREFULLY.

The symbol -✎- appearing at any point in the instructions below requires you to record the response of the action in your lab-book.

Command Line Programming

Boot up MATLAB from the Desktop icon.

Open the 'Help' window by clicking on '?'

(**Note:** It can be helpful to click on the 'Index' tab in Help, and enter any command name or operator to find out what it does.)

In the main MATLAB window, use the **Current Directory** window to set a new folder labelled **ES2A7_MATLAB2** in your personal folder space. Your Current Directory should be set to work within that folder.

MATLAB commands can be entered and executed directly in MATLAB's Command Window. We will experiment with this before writing MATLAB files.

Typing directly into the Command Window, simulate 30 successive reads of the temperature sensor by entering:

>> **tdata1 = rand(1,30)*200;** (*Check Help index – 'random numbers'*)

(*Explain the command shown above -✎-*)

Left-click the **Workspace** tab in the Directory Window. Right-click on the **tdata1** entry in the **Workspace** window, and select **Open Selection**. (*Note the response -✎-*).

Click the cursor back into the Command Window and press the ↑ arrow on the keyboard, until the above command reappears in the Command Window. (This saves you re-typing it). Remove the semicolon and enter it again.

>> **tdata1 = rand(1,30)*200** (*-✎- What does the semicolon do?*)

View a plot of this input selection by entering:

>> **plot(tdata1)**

Minimise the plot Figure window. In the Command Window, enter:

>> **tdata2 = rand(1,30)*200;**

View **tdata2** in the Array Editor.

Now enter:

>> plot(tdata2)

Restore Figure 1. (*Note what happened to the plot in Figure 1 -✎-*)

Now enter:

>> hold on (*Check Help index – 'hold'*)

followed by:

>> plot(tdata1, 'g') (*Check Help index – 'plot[1] [2] [3]'*)

Restore Figure 1. (*Note the effect of 'hold' and the use of a 'linespec' in above -✎-*)

From the Figure 1 file menu, click **Save As…**, and in the 'File name:' box enter '**TestPlot1**'.

Click on the 'Current Directory' tab under the Workspace window, and you should see your plot now saved as a **TestPlot1.fig** in your **ES2A7_MATLAB2** folder.

We could continue checking out a great many MATLAB functions by using command line operations in this way, but it makes sense to begin to construct proper file-based MATLAB programs (known as **M-Files**). However, it is worth remembering that – even when writing or debugging M-Files – it can sometimes be useful to test individual elements by trying them directly in the Command Window first.

M-File Programming

1. Creating a Program

MATLAB commands can be entered and executed directly in MATLAB's Command Window. However, application programs are created via **M-files**.

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose. The first step in creating a program is to open an editing window. To create a new M-file, enter the word ***edit*** at the MATLAB command prompt. To edit an existing M-file, type ***edit*** followed by the filename:

>> edit tempsensor.m

Click 'Yes' to the prompt "Do you want to create it?" (*Note the result -✎-*)

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

With the Current Directory window tab open in the left-hand window, use **File -> Save As…** to save **tempsensor.m** in your new folder.

In the Command Window, use the ***which*** function to see if your M-file program is on the MATLAB path:

>> which tempsensor -✎-

Place the MATLAB cursor in the Editor window and use comment lines to add a header to the program, e.g.

% tempsensor.m - A program to act as a temperature sensor instrumentation

% model, and to demonstrate the basic program control structures used in

% MATLAB.

% Author: Your Name, University of Warwick

% Date: --/--/----

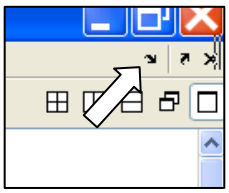
% Ver: 1.0

Note that one benefit of the MATLAB editor is that – if you keep typing a long line of text - it wraps a continuing comment line and automatically adds '**%**' to the next line. Click the 'save' icon on the editor window.

Move the MATLAB cursor to the 'Command Window' and enter:

>> **help tempsensor**

(*Note what this does -✎- . This help information can be accessed for any MATLAB M-File, including the vast array of library files, provided the file is on the MATLAB path*).



Docking the Editor Window – You may find it convenient for normal use to 'dock' the Editor window in the main MATLAB window. This enables you to conveniently see what code you have, and also (immediately underneath in the Command Window) see what it is actually doing. To do this click on the curved arrow in the top right corner of the Editor Window. You should see that the Editor window is now in the top right corner of the main MATLAB window, immediately above the Command window.

2. Adding Directories to the Search Path

The **help tempsensor** command found the M-File called **tempsensor** because the folder **ES2A7_MATLAB2** is currently open. However, if you close MATLAB and re-open it at a later date you may not be able to run files stored in this folder if it is not included in the MATLAB path. In the Command Window enter:

>> **path** (*Note the response -✎-*)

You should see a large range of folders on the MATLAB path, but it probably doesn't include the folder **ES2A7_MATLAB2**. (*Check Help index – 'addpath' to add current directory – note that 'Set Path' can be found on the main File menu in the MATLAB window*).

Once you have permanently added folder **ES2A7_MATLAB2** to the MATLAB path, check that this has been successful by entering in the Command Window:

>> **path** (*Note the response -✎-*)

(*Note: MATLAB searches the path folders in the order shown in the Command Window. Arranging the more commonly used folders to be near the start of the list could improve programming speed in some cases*).

3. Designing Program Features of the 'tempsensor' Application

We will start by displaying information to the user of the **tempsensor** program that will inform them of what the program is. In the Editor window, leave an empty line below the header comments and add:

disp('tempsensor.m')

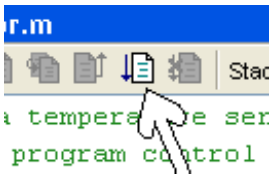
disp('A program to act as a temperature sensor instrumentation model')

(*Check Help index – 'disp'*)

We could have written this as a single line, but note that the Editor displays a fine vertical line that limits the recommended viewing width.

The program can be tested by clicking on the 'run' icon in the Editor.

This should cause the above statements to display in the Command Window.



3.1 Adding a User Input

Now we will use a simple user input to simulate a temperature sensor level being input one value at a time. In Editor, add the lines:

disp(' ')

%Adding a user input to simulate sensor read

temp = input('Enter sensor temperature value > '); (*Check Help index – 'input'*)

Note the use of the comment to describe that section of the program. This can be very helpful for later understanding and debugging.

Run the program again and note that you are prompted (in the Command Window) to enter a value. Enter a number against this prompt, then check the Workspace tab to see that this variable and its value are now stored in the workspace.

3.2 Using the 'if-then-else' Structure

Initially we will use **'isempty'** to see if no value at all is entered against the prompt.

(*Check Help index – 'isempty'*)

Add the following to your program, run it, and press 'enter' against the prompt without actually entering any value.

```
if isempty(temp)  %Checking input argument validity
  disp('No input value entered.');
  temp = input('Enter sensor temperature value > ');
end
```

This gives the user a 'second chance' to enter a correct value, but if a value is still not entered the program will continue. Not much use if we actually need a 'temp' value.

3.3 Using 'while' and 'for' loops with 'return'

It could be more sensible to use a 'while' loop that will keep prompting the user until a valid entry is given. (*Check Help index – 'while'*). REPLACE the above 'if' statement with:

```
while isempty(temp)  %Checking input argument validity
  disp('No input value entered.');
  temp = input('Enter sensor temperature value > ');
end
```

Test this by continually entering no value at the prompt. Terminate by either entering a value for 'temp' or pressing **Ctrl-c** at the keyboard.

Rather than using an interminable loop – if it is absolutely necessary to record a temperature value in order for the program to do anything useful - it may be better to give the user a fixed number of opportunities to enter a value, then terminate the program formally. We will do this by setting up a '**for**' loop and initiating a loop index '**i**'. (Note that, unlike 'C', we cannot create and initialise the index variable in the 'for' loop condition statement). REPLACE the above 'while' loop with:

```
i=0;
for i = 0:5
  if isempty(temp)
    disp('No input value entered.');
    temp = input('Enter sensor temperature value > ');
  else
     break
  end %end if
  i=i+1;  %increment loop index
  if i == 6
     disp('NO VALID ENTRY - PROGRAM TERMINATED')
     return
  end %end if
end %end for
```

Run this version of your program to check how many times you are offered the prompt before the program terminates. (*Check Help index – 'return'. What is the difference between 'return', 'break', and 'continue'* -✎-).

3.4 Using 'switch', 'case', and 'otherwise' to determine required output

The introduction sheet you were given for this laboratory exercise, describes temperature ranges that require either an 'alert', or an 'emergency warning' message. Use a 'switch' construct to check for and respond to temperatures in this range.

(*Check Help index – 'switch'*). Add the following to your program:

```
%Checking for Warning message requirements
temp2=num2str(temp); %Convert current temp value to string for display
switch temp
   case {90, 91, 92, 93, 94, 95}
      disp(['ALERT - LOW Process temperature = ',temp2])
   case {105, 106, 107, 108, 109, 110}
      disp(['ALERT - HIGH Process temperature = ',temp2])
   otherwise
      if temp < 90
         disp('WARNING - Process temperature is BELOW 90 deg F')
      elseif temp > 110
         disp('WARNING - Process temperature is ABOVE 110 deg F')
      end %end if
end %end switch
```

(*Check Help index – 'num2str'- why do we need to convert 'temp' with this before displaying it in the 'disp' statement?)*)

Now run the program several times, entering values in the range 0 to 200 (but including the 'alert' ranges of 90 to 95 and 105 to 110), and check that you get the required responses.

3.5 The use of Functions to allow repetitive actions

Assume we prefer to display the temperature readings in degrees Centigrade rather than Fahrenheit. MODIFY the start of your 'Warnings' code section as follows:

```
%Checking for Warning message requirements
tempC = tempconvert(temp); %Call to 'tempconvert' function
temp2=num2str(tempC); %Convert current temp value to string for display
switch temp
   case {90, 91, 92, 93, 94, 95}
      disp(['ALERT - LOW Process temperature = ',temp2,' deg C'])
   case {105, 106, 107, 108, 109, 110}
      disp(['ALERT - HIGH Process temperature = ',temp2,' deg C'])
   otherwise
        ----- continued as above ----
```

(*Check Help index – 'function [1] [2]'*)

The command **tempC = tempconvert(temp);** will pass the parameter **temp** to a function that will be saved as a separate M-file. The variable **tempC** is the parameter that is returned by the function **tempconvert** (i.e. 'tempC' is the centigrade value of the ^{o}F temp submitted).

Open a new file in MATLAB Editor window, add the code below, save as 'tempconvert.m'.

```
%--------------------------------------------------------------------%
function tempC = tempconvert(temp)
%Function to convert degrees F to degrees C
tempC = (temp-32)*5/9;

% end of Function
%--------------------------------------------------------------------%
```

This is a very simple function that can be called from any point in the program.

You have now completed the objective of using the basic programming structures in MATLAB.

One final point. It is normally a good idea to use the keyword '**clear all**' and '**close all**' at the start of your program, so that you are not running your program with previous data still in the Workspace, and you do not have any previous Figure windows in place.

(*Check Help index – 'clear [1] [2][3] [4]'*)

4.0 Self-Assessment – Now see if you can use the above knowledge to complete this task

You may want to try something a bit more ambitious with the programming structures used above. With your current **tempsensor.m** program open in the Editor window, use **File > Save As…** from the main MATLAB menu and save a second version of this file called **tempsensor2.m**.

Instead of using single user inputs, try setting up a loop that creates contiguous temperature readings from 85 ^{o}F to 115 ^{o}F inclusive, and progressively displays each reading on a plot. (You can remove all your section on 'Checking input argument validity').

Before this loop (in the program file), create and hold a plot of this temperature range as shown in Figure 1 below.

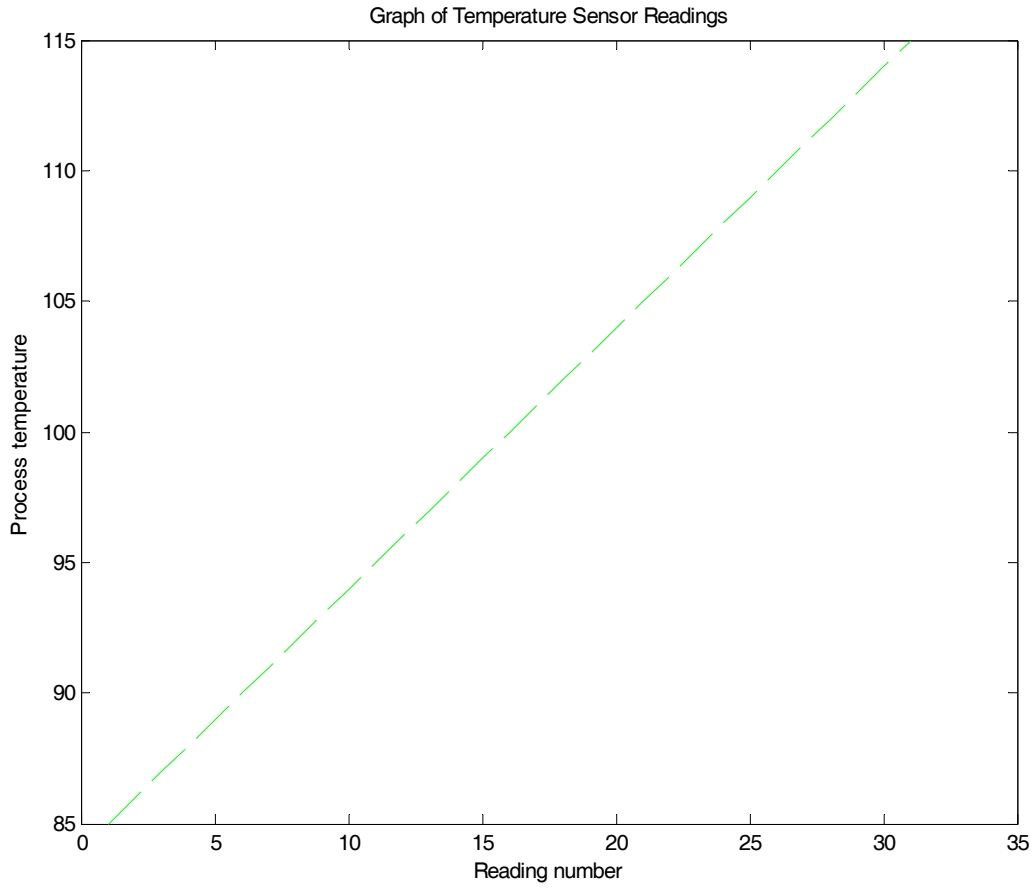


Figure 1: Base plot of 'recorded' temperature sensor range

Then for each iteration of the loop, add a red marker cross onto the above plot. Your Command Window can still display warning messages as before. Figure 2 shows how the plot should look after the first three readings have been plotted.

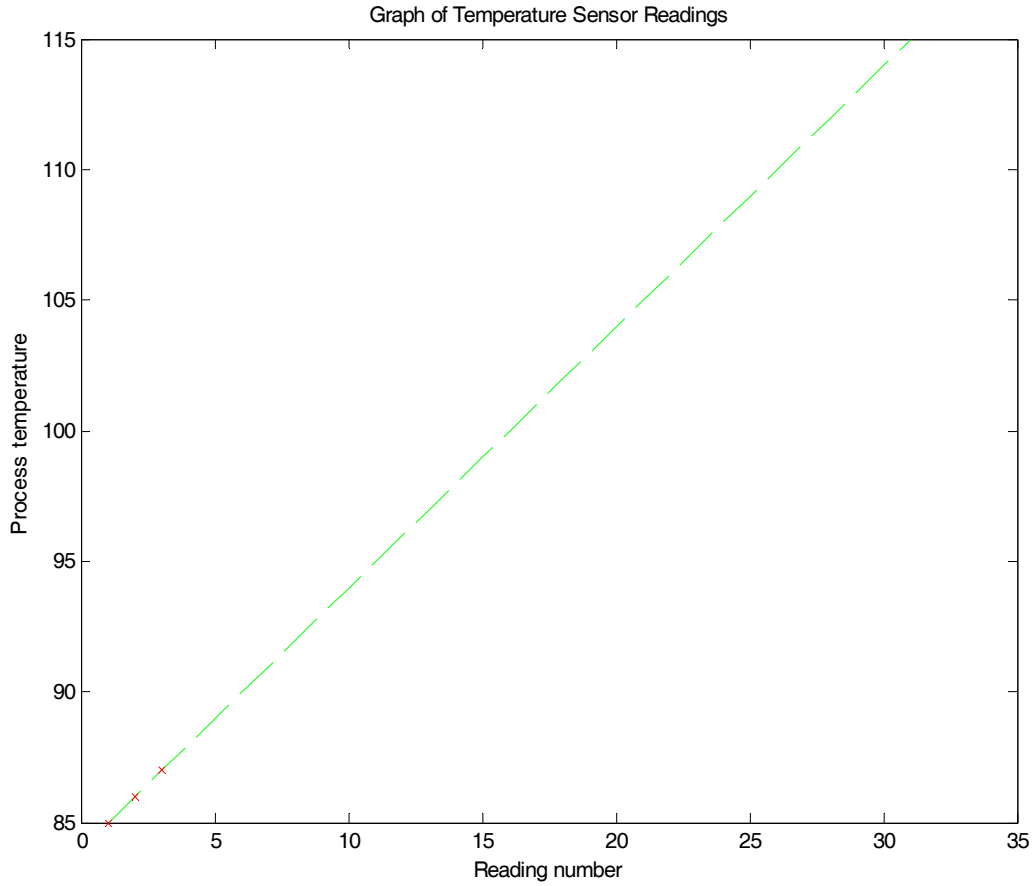


Figure 2: Temperature Plot after three 'recorded' readings

Use the Help index to help you find ways of setting this up. In order that the plot can be seen in real time, you may want to use a timer function after each plot command to plot the next point. Try,

```
figure(1)
t = timer;
set(t, {'StartDelay', 'Period'}, {2, 1});
t.TimerFcn = 'disp(''Processing...'')'
start(t);
wait(t);
```

'**figure(1)'** should make sure that the figure is open while you start the timer delay, and you can mess around with the numbers in the **{ }** brackets if you want to change the timing.