UNIVERSITY OF ARIZONA
DEPARTMENT OF CHEMICAL AND ENVIRONMENTAL ENGINEERING
CHEE 413 – PROCESS DYNAMICS AND CONTROL
SPRING 2012

MATLAB TUTORIAL FOR PROCESS CONTROL LAB

I. Graphical User Interface (GUI)

A uicontrol, (short for user interface control) object is a graphical input device that *MATLAB* adds to the graphical window it opens when you ask it to create a plot. They can also be used to open pop-up windows. It provides a method of entering values or communicating with *MATLAB* while a script is running without using *MATLAB*'s command prompt.

A uicontrol object is created with a command of format:

```
handle = uicontrol('PropertyName',PropertyValue,...);
```

where '`handle`' is a name you choose to make it easy to refer to the object you've created. The object is assigned properties by including 'PropertyName'/PropertyValue pairs inside the parentheses. The order in which the pairs are entered does not matter, but you always start with the name of the property you are defining and follow it with the value you want that property to have. For example,

```
h = uicontrol('Style', 'pushbutton', 'String', 'Clear', 'Position',...
   [20 150 100 70], 'Callback', 'cla');
```

creates an object with the handle name '`h`' and it will be a pushbutton object with the word '`Clear`' printed on the button. The location and size of the button are indicated by the four-value array, `[20 150 100 70]`, which indicates the lower left corner of the button will be located 20 units to the right and 150 units above the lower left corner of the window you are putting the button in. The button will have a width of 100 units and a height of 70 units. You can think of the array as: `[left bottom width height]`. The default units are pixels, but can be changed using the '`Units`' property.

Depending on the type of control object you choose to use, you may need to define a callback action--in the example above, '`cla`' would be a string corresponding to the name of another m file or it can be the handle for a function that you want executed when the control is activated; some controls require a callback function to make them useful, but many do not.

A uincontrol object of type '`pushbutton`' is Boolean--its default initial value is 'false' until it is clicked, at which point it becomes 'true'. To read the value, use:

```
get(h,'Value');
```

If you would like to change the state of the object from 'false' to 'true' without clicking on it, you can use a '`set`' command:

```
set(h,'Value','1');
```

If you had created an object of type '`edit`',

```
hEdit = uicontrol('Style', 'edit');
```

which is a box in which you can type text, you would read the text displayed in the box with

```
get(hEdit,'String');
```

and you could change the text using

```
set(hEdit,'String','New Text');
```

MATLAB includes a Graphical User Interface Design Environment (GUIDE) that can be used to create elaborate and nice looking GUIs. Feel free to use GUIDE if you want to learn how it works, but there is a substantial learning curve and it is more complicated than what is required for this class. You will probably accomplish the goals of this lab activity more quickly just learning to create a few uicontrol objects without GUIDE.

If the script is short and executes quickly, your plot command might not have time to complete before the next iteration of your loop executes--if your plot does not appear complete, add a pause command (`pause(0.3)`) after your plot creation command to slow things down.

To get started with the most useful help files in *MATLAB*, enter 'uicontrol' in the search field of the help browser.

II. Hardware Communications - MCC data acquisition board

Communication between *MATLAB* and MCC hardware is handled by the Data Acquisition Toolbox which is part of the full licensed version of *MATLAB*. However, the Data Acquisition Toolbox contains commands specific to hardware from several different manufacturers and *MATLAB* does not fully unpack the files that are specific to a manufacturer until you direct it to. In order to tell *MATLAB* you will be communicating to hardware manufactured by the Measurement Computing Corporation (MCC) you must enter a onetime initialization command.

1) Run *MATLAB* as administrator. If you are running Windows XP, being logged in as an admin is probably sufficient, but if your machine is running Windows Vista or 7, even if you are logged in as an admin, you will have to exit *MATLAB*, right-click on your *MATLAB* icon, and select 'Run as Administrator' from the menu.
2) On the *MATLAB* command line, enter:
   daqregister('C:\Program Files*MATLAB*\R2010a\toolbox\daq\daq\private\mwmcc.dll')
   You should receive a confirmation message indicating success. Please note, the file pathway listed above is dependent on the specific version you are running. For example, if you are running release 2009b, then you would have to replace the 'R2010a' listed above with 'R2009b'.

A) *InstaCal*

You must have the appropriate hardware driver installed before your PC can communicate to the data acquisition board. For MCC devices, all drivers are included within *InstaCal* which can be downloaded for free from the MCC website (http://www.mccdaq.com/software.aspx).

Plug your USB hardware into the computer before starting *InstaCal*. If it is the first time a particular board has been plugged into the machine, you may have to wait a minute or two for the computer to install the new hardware; wait until you get a message indicating the new hardware has been successfully installed.

When you open *InstaCal*, it will automatically scan for any MCC hardware connected to the computer. If it detects any hardware, it will add it to the board list and assign it a board number. You can change the assigned number if desired. Make note of the number because it will be required within *MATLAB* when you open a communications channel.

When *InstaCal* detects the USB-1208FS used to communicate with the two-tanks hardware, it configures the board with 4 channels of differential analog input--you will need to right-click on the board and change its configuration to have 8 channels of single-ended input. You will have to do this configuration change every time *InstaCal* detects the board. The configuration is not saved until you exit *InstaCal*, so exit before starting *MATLAB*.

InstaCal includes the ability to create a virtual board you can use to practice reading voltages with *MATLAB*. Inside *InstaCal*, right click on the 'PC Board List' and select 'Add Board…' Under the 'ISA' tab, scroll to the bottom of the list and select 'DEMO-BOARD'--you have to scroll to the bottom of the list to see it. When it appears on your board list, you can right-click on the demo board and assign unique output types to each channel. These channels can be read inside *MATLAB* just as if you had a physical board connected to the computer.

B) Reading Voltage

In *MATLAB*, to open communication with the board, we create an analog input object with

```
AI = analoginput('mcc',1);
```

where '`AI`' is an easy-to-remember handle you create and the numeral 1 indicates we will be communicating with board number 1, as was set by *InstaCal*. The reserved term `'mcc'` tells *MATLAB* what type of hardware the object will be communicating with.

Now that we have established communication with board 1, we have to define which channels on the board we want the object to refer to with

```
addchannel(AI,0:1);
```

which associates channels 0 and 1 with our object '`AI`'. Because we are reading pressure gauges with these channels, we will be treating the results from both channels similarly and it probably

makes senses to use one object for both pressure gauges. However, if we anticipated wanting to treat the results from the two channels differently, we could create separate objects for each channel. For example,

```
AI1 = analoginput('mcc',1);
AI2 = analoginput('mcc',1);
addchannel(AI1, 0);
addchannel(AI2, 1);
```

Once the object is created, we can read from the specified channels using

```
start(AI);
```

which tells the data acquisition engine to retrieve the signals from the hardware. It is usually a good idea to follow a start command with

```
wait(AI,3);
```

which tells the script to pause until the data acquisition engine is done acquiring the data. If it takes longer than the specified time (3 seconds), then it gives an error message and moves on. If you skip the wait command the script will continue on, but if you have any additional commands that tell the data acquisition engine to communicate with the same hardware, they will be skipped if the engine has not completed its current task.

After the data acquisition engine has acquired the data, use a '`getdata`' command to make the data accessible to *MATLAB*.

```
Measured_Voltage = getdata(AI);
```

will assign the read data to a variable named '`Measured_Voltage`'. By default, the data acquisition engine will retrieve 1000 voltage values every time you issue a '`start`' command. If the object '`AI`' is reading from two channels, then '`Measured_Voltage`' will be a two column array with 1000 rows. If you are reading a pressure from a gauge to determine the depth of water in the tank, 1000 readings may seem excessive, but the analog signal from the pressure gauge is typically very noisy, so it would be a good idea to use the average of all 1000 values as your measurement.

When the board reads a voltage, it converts the voltage to an 11-bit digital value, but when *MATLAB* gets the data from the data acquisition engine, it converts the digital value back to voltage; if the board is connected to a 3 V source, the average value of '`Measured_Voltage`' will be 3.

C) Transmitting Voltage

Sending voltages with the data acquisition board is very similar to reading voltages. Instead of creating an analog input object, you create an analog output object, again indicating the board number that was set by *InstaCal* and assigning channels to the object.

```
AO = analogoutput('mcc',1);
addchannel(AO, 0);
```

Again, you can choose to assign both channels 0 and 1 to the same object, or you could decide to create separate objects for each channel. If you include both channels in a single object, you will be sending a 2 column array to the board whenever you instruct the board to transmit a new value. If you are controlling pump speed, you will be sending voltages for both pumps even if you only need to adjust the speed on one pump. Creating separate objects for each channel avoids this.

To transmit the voltage, first create an array of values to be transmitted. The array must have the same number of columns as the object has channels. If the array has multiple rows, each row will be transmitted in order at a rate that can be specified using '`SampleRate`', but to control pump speeds we only need to send a single value. If you are sending a single voltage to a single channel, you will need to create a 1x1 array with the value you wish to send.

```
voltage_out = [2.5]
```

Pump control is facilitated by the fact that the board transmits the last value received indefinitely--the pump will continue at the set speed until you tell it to change. Even though units are not specified, the values in the array correspond to voltages. If you transmit a '2.5', the board will transmit 2.5 volts.

To transmit the array you have created, we first queue the array using

```
putdata(AO, voltage_out)
```

and then we activate the output channel with

```
start(AO)
wait(AI,3)
```

where it is again a good idea to include the '`wait`' command to avoid a pile-up of commands to the board. If *MATLAB* tells the data acquisition engine to execute another command before the previous one has completed, the subsequent commands are ignored.

D) Cleanup

Upon completion of your script, it is good practice to end with

```
stop(AI)
clear AI
delete(AI)
```

and

```
stop(AO)
clear AO
delete(AO)
```

[does this turn off a board if it is transmitting voltage?] which purges the analog input and output objects from *MATLAB* and will avoid any conflicts if you run another script that tries to create another analog input object by the same name.

III. Hardware Communications - Mindstorms NXT©

The first time you plug the NXT brick into a PC, you may need to pause for a minute or two to allow the PC to recognize the new device. Once you receive notification that the device has been successfully installed, run *TransferMotorControlBinaryToNXT.bat* to open communications between the PC and the brick. You may have to search for the file--it was installed when you unpacked the zip file including the *RWTH – Mindstorms NXT toolbox*. Also, if you have not already installed the *NeXTTool* utility to the same directory, you will get an error message.

E) Open Communication Channel

It is good practice to end your script with a command that closes the communication channel between *MATLAB* and the NXT to avoid any conflict with whatever script you run next, but in case that did not happen, start your script with '`COM_CloseNXT all`'.

Next, open a channel to the NXT with

```
handleNXT = COM_OpenNXT('bluetooth.ini')
```

where '`handleNXT`' is an arbitrary handle you choose to make it easy to refer to the specific communication channel you have created. Use the initialization file for Bluetooth communications even though we are using a USB cable just because it is the most up-to-date initiation file available. It checks for USB connections first, because they are faster than Bluetooth, and only looks for a Bluetooth device if a USB connection is not found. Follow that command with

```
COM_SetDefaultNXT(h)
```

so that subsequent commands will be directed to the NXT without having to re-specify the communications channel. At the end of the script, include

```
COM_CloseNXT(h)
```

to close the channel and prevent unintended future conflicts.

F) Motor Movement

To issue commands for the NXT servo motors, we first create a motor control object. The object will automatically be created with structure fields corresponding to the assorted commands we might wish to give to the motor. To control the motor's behavior, we populate the appropriate structure field with the value we wish the motor to use and then send the object with all of its

structure fields to the NXT brick. The brick then puts the motor into the state specified by the object structure.

A motor control object is created with the '`NXTMotor`' command, which includes an initial term to tell the brick which of the three motor ports you want the object related to, and then you include property name/property value pairs, similar to the way '`uicontrol`' objects were created.

The command

```
moveA = NXTMotor(MOTOR_A);
```

creates a motor control object named '`moveA`' that controls a motor plugged into port A ('`MOTOR_A`' is a reserved term). This works, but the brick will rely on its default parameters to decide how to move the motor when we tell it to move. Alternatively, we could use

```
moveA = NXTMotor(MOTOR_A,'Power',50,'ActionAtTachoLimit','Coast');
```

Which also creates a motor object named '`moveA`', but now when the motor moves, it will move with 50% of its available power, and when it is done rotating the specified number of degrees (when it reaches its tachometer limit), it will coast to a stop. This command, however, will only move the motor in the forward direction. If we decide we want the motor to move in the opposite direction, we can modify the object with

```
moveA.Power = -50;
```

which will now cause the motor to move in the backwards direction at half power whenever a move command is sent to the brick.

We told the object to coast to a stop when it reaches the tachometer limit, but we have not told it what the limit is. With

```
moveA.TachoLimit  = 180;
```

we can tell the motor we want it to turn one half of a revolution (180°) after the move command has been issued, after which it will coast to a stop.

So we've created an object describing how we want the motor to be moved. When we want the command executed we enter

```
moveA.SendToNXT();
```

which sends the object structure to the brick telling it to start the motor at 50% power in the reverse direction. The motor runs until the internal angle sensor detects the motor has moved 180° degrees, at which point the brick turns off the power and the motor slows to a stop.

This is only a brief introduction. Please read the *Motor Control* section under *Tutorial: Programming Robots* in the help files included under *RWTH - Mindstorms NXT Toolbox*.

G) Reading Motor Position

After the NXT brick has executed the motor movement, it updates the fields of the motor object describing the current status of the motor, including its final angular position. We could type

```
moveA.ReadFromNXT();
```

and we would get a list of the different properties and their values, but it would not be very useful for programming. Instead we use

```
motor_data = moveA.ReadFromNXT();
```

and it creates a structure named '`motor_data`' that contains fields describing the motor's current state. If we want to know the motor's current position, we type

```
disp(motor_data.Position);
```

to find the current tachometer reading. This reading is relative to whatever the motor position was at the time it was initialized. To reset the tachometer, use

```
moveA.ResetPosition();
```

Always good to end with

```
moveA.Stop('off');
```

so you do not leave the motor consuming any power once your script ends. Do not forget the '`COM_CloseNXT all`' to close the channel.

H) Reading from sensors

To activate a sensor port, use a command of format:

```
status = NXT_SetInputMode(port, SensorTypeDesc, SensorModeDesc,
ReplyMode);
```

which tells the NXT brick to activate the sensor port indicated by *`port`*, which will probably be '`SENSOR_1`' or '`SENSOR_2`'. The term *`SensorTypeDesc`* must be one of the many sensor types listed on the *NXT_SetInputMode* help page and tells the brick what type of sensor you have plugged into the port. For this activity, we are using a pressure sensor supplied by Omega (PX40-50BHG5V) which is powered by a 5 VDC supply voltage and returns an analog signal between 0 and 5 V that is proportional to the pressure. There is no pre-defined *`SensorTypeDesc`* for the pressure sensor, so we have to improvise. Lego sells a reflected light sensor intended to be used for robot vision that turns on a LED to illuminate the robot's path and uses a photoreceptor to measure the light reflecting off any objects in front of the robot. It is similar to

our pressure sensor in that it returns an analog voltage between 0 and 5 V, so we will use *SensorTypeDesc* of 'REFLECTION'. Because we are measuring pressure, not reflected light, we will specify *SensorModeDesc* as 'RAWMODE', which means the reported value will simply be a digital number between 0 and 1023 (the number of bytes we get at 1 kbyte resolution). You can read about the other mode choices, but they will not be useful to us as they all apply some sort of mathematical analysis to the results that would obscure the relationship between pressure and the returned value. The only two choices for *ReplyMode* are 'reply' or 'dontreply', depending on whether you want to receive an error message or not.

In order to supply the 5 VDC needed to power the sensor, we will also activate sensor port 2. Activating the sensor port causes it to power up with 7 VDC; in order to get the 5 VDC needed for the sensor, the customized cable includes an integrated circuit voltage regulator spliced in-line that drops the 7 V down to 5 V. So, the commands to activate the sensor ports are: [is the variable 'status' necessary?]

```
status = NXT_SetInputMode(SENSOR_1, 'REFLECTION', 'RAWMODE', 'reply');
status = NXT_SetInputMode(SENSOR_2, 'REFLECTION');
```

Once the sensor is activated, to obtain signal from the sensor you simply query it with

```
raw_value = GetLight(SENSOR_1);
```

where 'raw_value' is just a variable of your choosing.

At the completion of the script, you will want to power off the sensor ports to conserve the batteries in the NXT brick with:

```
CloseSensor(SENSOR_1);
CloseSensor(SENSOR_2);
```