1 Basic Programming Tutorial

CHAPTER REV. 552, DATED 05/21/2010, 14:49. USER GUIDE REV. 552, DATED 05/21/2010, 14:49.

1.1 Introduction

The purpose of this tutorial is to give you a start on learning how to control the MFP-3D at a low level using Igor. You might want to do this to make the AFM do things not available in the higher level software. Before running through this tutorial you should be very comfortable using the AFM and, ideally, have practiced a little bit of Igor programming. At the very least, it is important to understand the difference between "X scaling" and "P scaling" in Igor and doing waveform assignments from the command line. The "Getting Started" tutorial that is part of the Igor manual will teach you this much. This tutorial will focus on commands specific to the Asylum AFMs. There are more detailed descriptions of some of this elsewhere in the Asylum help. For example, under the *Help menu*, take a look at *Help* ▷ *AR Help Files* ▷ *Xop Help Files* ▷ *MFP3D Xop Help* to see some of those help files.

1.2 Conventions

Anytime in this tutorial where you see something preceded by a bullet point, such as

```
print td_ReadValue("Head.Temperature")
```

it is meant to be executed from the Igor command line. You can just copy and paste them, but you will learn more quickly if you type them out yourself. The value that the function returns to the Igor history will often be shown below the command, e.g.

```
print td_ReadValue("Head.Temperature")
31.625
```

This is just to show you what you might expect to see. You shouldn't enter the `31.625` into the command line.

1.3 Help

The majority of commands you will be learning about are commands that start with `td_` . The td stands for 3D and/or Todd Day, the programmer that created the commands. Help for these commands can be found in the Xop Help file. The easiest way to get to this is to go to the menu

Help ▷ *Command Help*. This brings up the Help Browser and you can just scroll down to get help on the command you are interested in learning about.

Another very useful help item is that if you type the command onto the command line and right click on it you can go to the help for that command or insert a template. Inserting a template is very useful if a command takes several parameters and you don't remember the order.

1.4 Low Level Parameter Names

Everything on the instrument that has a value you might want to read or control has a parameter name. These parameter names are constructed a bit like email addresses to make it easier to organize them. For example, to read the value of a thermometer that is built into the MFP-3D head, you would execute

```
print td_ReadValue("Head.Temperature")
31.625
```

The "head" refers to a specific device that is part of the instrument, while the the "temperature" is a specific parameter that the head owns. Some examples of devices are the head, the XY scanner, and the controller. Additional devices such as heaters can also be added to the instrument. You can see all the devices hooked up to the instrument by clicking on the picture of the controller at the bottom left-hand side of the screen.

Since many devices have a lot of parameter names, there is an additional level of organization called groups. For example the controller has a lot of analog-to-digital converters (ADC's) and digital-to-analog converters (DAC's) in it. There is a group named "Input" that contains the ADC's and a group named "Output" that owns the DAC's. So, for example, to read the value of the X sensor, you would execute

```
print td_ReadValue("ARC.Input.X")
-6.84063
```

You should read the "." as "is a member of" so that the above parameter name would be read "X is a member of Input at the ARC Controller". Depending on which version of the software, the name of the controller may be "Controller" rather than "ARC". The name of the controller appears in the lower left toolbar.

Similarly,

```
print td_WriteValue("ARC.Output.X", 60)
0
```

would apply 60 volts to the X piezo. Note that when you use commands like `td_WriteValue` where you don't expect an output, you should still precede them by a print command. They will return 0 if everything worked and a non-zero error code if it didn't. So, for example, if you mispell something you will know.

Note that some outputs can also be read as inputs. You might be interested in knowing what the current voltage to the X piezo is before you change it. So

```
print td_ReadValue("ARC.Output.X")
60
```

Another way to access these parameters is to use the menu that pops up when you click on the controller picture at the bottom left-hand side of the screen. A popup menu with varioius pictures of the devices currently hooked up will appear. For example, to see the changes you just made, float the cursor over the picture of the controller in the menu and an additional popop menu will appear. Float the cursor over parameters menu and yet another popup menu will appear with a list of all the devices owned by the controller. Select *Output* and a panel will popup that contains all the parameters in the output group. Click on the read button and you should see that the X output has a value of 60 V.

Before you start, the HighVolage Relays must be checked to see if they are open or closed. Click on the controller icon in the toolbar and navigate to *ARC* ▷ *Parameters* ▷ *Default*. Then use the read and write buttons to look at and set the state of the *HighVoltageXYRelay* and *HighVoltageZRelay*. You can also change the state of the relays at the command link

```
print td_WriteValue("Arc.HighVoltageXYRelay",1)
0
```

At the top of the MFP3D Help file there is a complete list of parameter names as well as the error codes.

1.5 High Level Parameter Names

There are also parameters that exist in Igor rather than down at the device level. If you click on the Programming menu and then select "Master" from the "Global Variables" popup menu, you will see a table full of parameters and descriptions of what they control. For example, the top item is the ScanSize and shows the size current image in meters.

While you can read these values from the table, while programming it is useful to access them with the GV (Get Value) routine. Executing

```
print GV("ZPiezoSens")
```

will print the sensitivity of the Z piezo (in meters per volt) into the history.

1.6 The Crosspoint Switch

To make the instrument a lot more flexible, the controller has something called a crosspoint switch inside of it. The crosspoint switch is a lot like an old telephone patch board. It has 16 inputs and 16 outpus and you can connect virtual wires between any of the inputs and any of the outputs. You can have multiple wires running from one input to many outputs.

The crosspoint switch can make programming a bit more complicated because a given ADC isn't always connected to the same things. From the main menu bar select *Programming* ▷ *Crosspoint Panel* to bring up a panel (Figure 1.1 on page 4)which allows you to see and control the wiring of the crosspoint switch. Switch between contact mode and ac mode and see how some of the signals are rerouted.

When programming you often find yourself using the BNCs on the front of the controller. These appear on the crosspoing switch inputs and outputs. To help keep you sane, we named all the extra

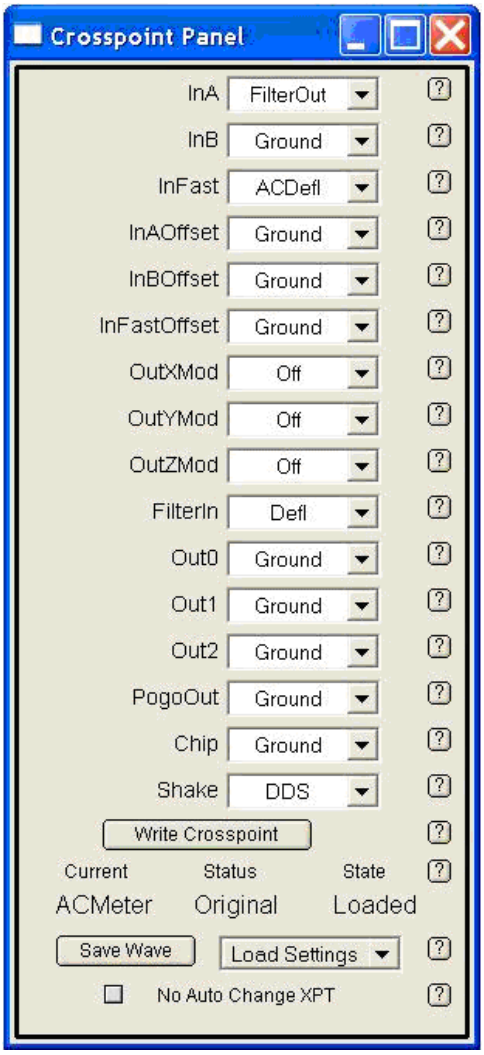


Figure 1.1: Crosspoint Panel

ADCs and DACs with letters (e.g. InA) and the physical BNCs with numbers. So, for example, you can connect the BNC In0 to the ADC InA using the top drop down menu in the Crosspoint Panel.

The crosspoint switch can also be set with `td_WriteString` commands. For example,

```
print td_WriteString("Controller.Crosspoint.InA","BNCIn0")
0
```

would accomplish the same thing.

Probably the most confusing out to the crosspoint is the *FilterIn*. By routing a signal to *FilterIn*, you are routing it into a low pass filter of about 36 kHz. The filter signal then reappears as an input to the crosspoint switch named FilterOut. This filter is useful for filtering high bandwidth signals (e.g. Defl) before passing them to a 100 kHz ADC like *InA*.

A Simple Example with `td_ReadValue` and `td_WriteValue`

You can use the `td` commands to move the X stage around and measure its motion with the X

sensor. The stage is approximately centered at 70 volts. If you execute the following commands in a row you should see output similar to those listed here.

```
print td_WriteValue("ARC.Output.X",60)
0
print td_ReadValue("ARC.Input.X")
-1.20091
print td_WriteValue("ARC.Output.X",70)
0
print td_ReadValue("ARC.Input.X")
-0.487317
print td_WriteValue("ARC.Output.X",80)
0
print td_ReadValue("ARC.Input.X")
0.297895
```

You just drove the X stage to 60, then 70, then 80 volts. As you can see, the sensor voltage measured this movement and returned three different values. To see what happened in meters instead of volts we can use the sensitivities to convert the voltages.

```
print GV("XPiezoSens")*20
9.6164e-06
```

This means moving the stage from 60 to 80 volts (20 volt change) should have produced about 9.6 microns of movement. In reality, the sensors measured

```
print (0.297895 -1.20091)*GV("XLVDTSens")
1.18161e-05
```

This is about 11.8 microns. The fact that the piezo didn't quite move as much as we told it to is the reason we have the sensors. How far the piezo is off is dependent on scan size. Try the above experiment for a 2 volt change (drive X to 69 volts and then 71 volts) and you should get better agreement.

While you could make the sample move around by stringing together a lot of `td_WriteValue` commands, that is tedious. A better way is to use an Igor wave to control the stage and collect data. You'll do this next.

1.7 Parameter Control and Collection

Before doing anything, it is best to stop all activity in case the Asylum Research software is doing something. You can use the `td_stop` command. It is a bit brutal in that it halts all inputs, outputs, and feedback loops. There are slightly less brutal commands that stop things individually that you can learn about as you get better.

```
print td_stop()
```

Now make two Igor waves, one to store voltages to drive the X stage and and one to store the sensor values we read.

```
Make/N=1024 PiezoVoltage SensorVoltage
Next make a graph of each
Display PiezoVoltage
Display SensorVoltage
```

and move them so you can see both of them. You'll notice that right now, the wave scaling of both is just p scaling. The X axis on each wave is from 0 to 1023. To use this wave to drive the piezo voltage, it will be much more convenient to have the x scaling in time. While you could set it by hand the easiest way is to use a `td` command. To set up an output wave you will use the `td_xSetOutWave` command. You can find the detailed help for the command as described in Section 1.3 on page 1.

This command will take the data in an Igor wave and download it to the DSP where it will sit ready to be output to the X piezo. Of course we have to tell the DSP how quickly we want the wave output, when to output it, what DAC we want to drive and a few other things.

The syntax of the command is

```
td_xSetOutWave(whichBank, eventString, channelString, wave,
  interpolation)
```

The parameters are:

whichBank determines which memory bank in the DSP the wave will be stored in. There are three banks of memory, labeled 0,1, and 2, and each can hold up to two pairs of waves. These waves can be up to 87,380 points long and must be single-precision (32 bit) floating point.

eventString determines when the data will be output. You can think of them as a trigger. You usually don't want the wave to be output as soon as it reaches the DSP because you might want to synchronize the output with some data collection. The events are group of parameters owned by the controller, so a typical eventString would be "ARC.Event.0". When we later change the value of this event parameter, the wave will be output to the DAC that drives the X voltage.

channelString is simply a low level parameter name like we learned about. In this case it will be "ARC.Outtput.X".

wave is simply the name of the Igor wave with the data to be sent. Here it is PiezoVoltage.

interpolation determines how quickly the wave will be output. The DAC's and ADC's all run at 100 kHz, so an interpolation of 1 means that each point in the Igor wave will correspond to a 100 kHz sample. For the 1024 point waves you made, this means the entire wave would be output in about 10 milliseconds. This is very fast, so we will choose an interpolation of 100 to give us about a 1 second output.

Putting this all together gives

```
print td_xSetOutWave(0, "ARC.Event.0", "ARC.Output.X", PiezoVoltage, 100)
0
```

If you did everything right you should have gotten a zero in the history.

If you look at the graph of PiezoVoltage that you made, you will notice that the x scale has been updated to reflect the time that the wave will take to be output.

Now make a ramp over the full range of voltage (-10 to +150) by altering the first half and the second half of PiezoVoltage (Figure 1.2 on page 7).

```
PiezoVoltage(0,0.511) = -10 + 160*x/0.511
PiezoVoltage(0.512,1.023) = 150 - 160*(x - 0.511)/0.511
```

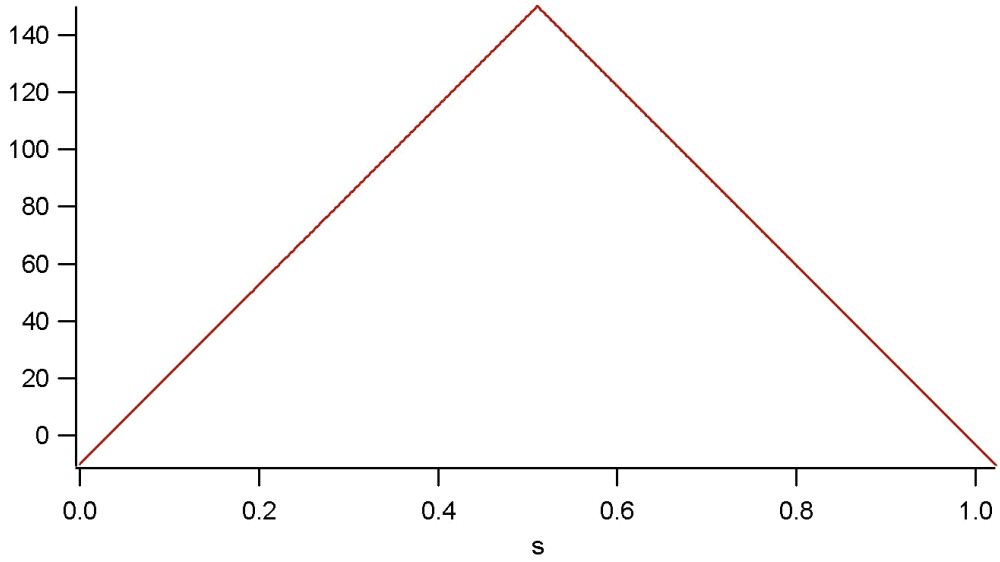


Figure 1.2: Piezo Voltage vs Time

Since we have changed the data, we need to download it to the DSP again:

```
print td_xSetOutWave(0, "ARC.Event.0", "Output.X", PiezoVoltage, 100)
0
```

Now you need to set up the SensorVoltage wave to record data. The syntax of the command is fairly similar to that of `td_xSetOutWave`:

```
td_xSetInWave(whichBank, eventString, channelString, wave, callback,
  decimation)
```

The first four parameters are conceptually identical except `channelString` will now be an input instead of an output.

We also have a callback parameter and decimation instead of interpolation. Decimation is the opposite of interpolation. Rather than creating points we want the DSP to reduce the number of points coming back. Sometimes getting data at 100 kHz is a bit like drinking from a firehose and you want less information.

The callback function is an Igor function that will be called when the input data has been collected. This is useful so that control returns to Igor while the data is being collected (otherwise Igor would freeze during data collection). This function might be something that does some analysis on the data. We won't use a callback in this example.

One major conceptual difference here is that rather than the data being stored on the DSP like the output data, it is continually streamed back to the PC. This means the banks correspond to "pipes" on the USB connection rather than memory. Due to bandwidth limitations, the banks have slightly different personalities.

- Bank 0 can bring back two 16-bit waves at 100 kHz (decimation 1).
- Bank 1 can bring back two 16-bit waves at 50 kHz (decimation 2).
- Bank 2 can bring back one 32-bit wave at 50 kHz.

Putting all this together gives

```
print td_xSetInWave(0, "Event.0", "Input.X", SensorVoltage, "", 100)
0
```

You will see that the x scaling on the SensorVoltage graph has now changed to seconds. You have now set up everthing and all you have to do is trigger ARC.Event.0. For this it is easier to use `td_WriteString` instead of `td_WriteValue`. They are identical except the first is used for passing strings instead of numbers.

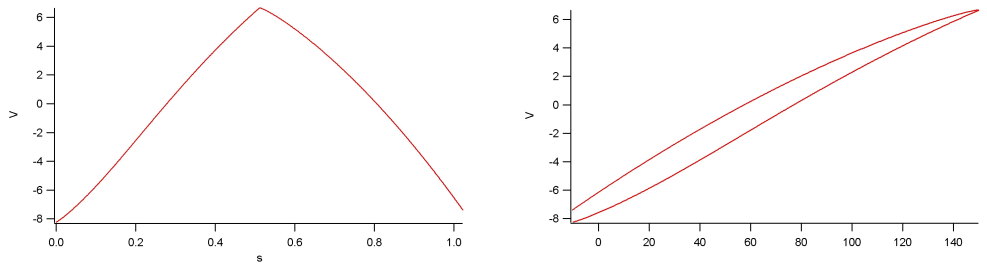
```
print td_WriteString("ARC.Event.0", "once")
0
```

If all went well after about a second you should see the SensorVoltage wave get update with data. The curvature shows that the piezo isn't moving linearly. If the graph shows noise then the high voltage relays are in the open state so the voltage is not getting to the piezos. you can check this by clicking on the Controller Icon botton of the Igor window and navigate to *ARC* ▷ *Parameters* ▷ *Default*. Then use the read and write buttons to look at and set the state of the *HighVoltageXYRelay* and *HighVoltageZRelay*. You can also change this at the command line:

```
print td_WriteValue("Arc.HighVoltageXYRelay",1)
0
print td_WriteValue("Arc.HighVoltageZRelay",1)
0
```

If you graph the SensorVoltage versus the PiezoVoltage you will see the hysteresis loop the stage traced out (Figure 1.3 on page 8).

```
display SensorVoltage vs PiezoVoltage
```



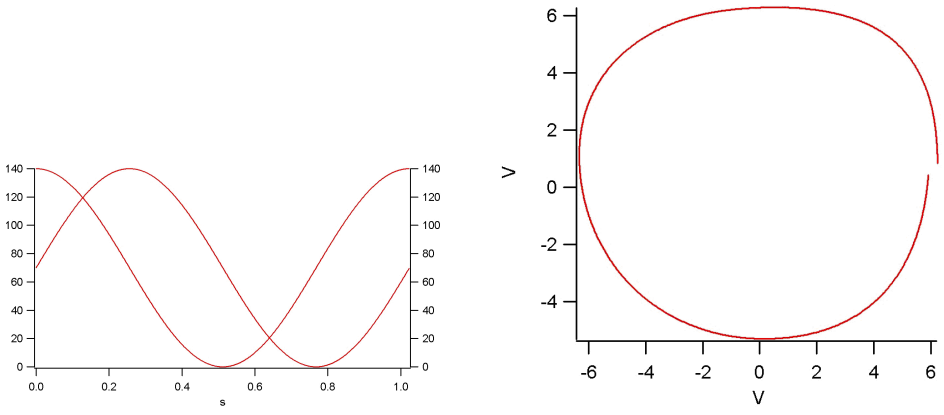
(a) Sensor Voltage vs Time **(b)** Sensor vs Piezo Voltage

Figure 1.3

1.8 Drive the XY Stage in a Circle

The set of commands below will drive the XY stage with voltages corresponding to a circle almost full scale for the stage. The sensors will measure what really happened. The example uses `td_xSetInWavePair` and `td_xSetOutWavePair` which are very similar to the commands you learned above. You can watch the sample movement using the optics of the MFP-3D stand-alone or with the optics of your inverted optical microscope if you have a MFP-3D-BIO.

```
make/N=(1024)/O XVoltage YVoltage XSensor YSensor
display XVoltage; appendtograph/R YVoltage
display XSensor; appendtograph/R YSensor
print td_xSetOutWavePair(0, "0,0", "ARC.Output.X", XVoltage,
  "ARC.Output.Y",YVoltage,100)
0
XVoltage = 70 + 70*cos(2*pi*x/1.024)
YVoltage = 70 + 70*sin(2*pi*x/1.024)
print td_xSetOutWavePair(0, "0,0", "ARC.Output.X", XVoltage,
  "ARC.Output.Y", YVoltage,100)
0
print td_xSetInWavePair(0, "0,0", "ARC.Input.X", XSensor,
  "ARC.Input.Y", YSensor, "", 100)
0
print td_WriteString("Event.0", "once")
0
print td_WriteString("Event.0", "once")
0
display YSensor vs XSensor
ModifyGraph width={Plan,1,bottom,left}
```



(a) X,Y Voltage vs Time **(b)** Y Sensor vs. X Sensor

Figure 1.4

As you can see by Figure 1.4 on page 9, because of piezo non-linearities and hysteresis, the circle isn't very round.

1.9 Feedback Loops

For a lot of stuff that you might want to do (for example, to make a round circle), you will want to use a feedback loop of some sort. The controller is able to run six PIDS loops. These loops have

- proportional gain P
- Diferrential Gain D
- integral gain I
- double-integral gain S

Each parameter belonging to the loop can be access, for example, *PIDSLoop.0.IGain* is the name for the Integral gain of loop 0.

One problem with feedback loops is that if you pick the wrong gains, the feedback loop can oscillate. Besides being incredibly loud, this can also push the sensors out of alignment and require a short trip back to the factory (nothing serious though so don't be too paranoid).

The easiest thing to do is to look at the values that the software uses. Be careful not to look at the gains in the panels themselves (like integral gain on the Main panel or the X and Y gains on the XY gains panel). These are scaled to make it more convenient for users to think about.

To peek at the value that the software uses for Z in contact mode, hit Engage on the meter panel. It doesn't matter if you have a cantilever in or not. This turns on the Z feedback loop. The following commands will then read the three gain values. This result is typical for contact mode.

```
print td_ReadValue("PIDSLoop.2.PGain")
0
print td_ReadValue("PIDSLoop.2.Igain")
1000
print td_ReadValue("PIDSLoop.2.SGain")
0
```

It is easiest to look at thie values in the PIDS Loop Panel, go to *Programming ▷ XOP Tables ▷ PIDS Loop Panel*.

For AC mode, the gain will be the opposite sign and smaller.

Similarly, to see the X and Y gains, click "Do Scan" and look at the PIDS Lopp Panel, or you could execute the following:

```
print td_ReadValue("PIDSLoop.0.PGain")
0
print td_ReadValue("PIDSLoop.0.IGain")
6887.8
print td_ReadValue("PIDSLoop.0.SGain")
1075803
print td_ReadValue("PIDSLoop.1.PGain")
0
print td_ReadValue("PIDSLoop.1.IGain")
6357.8
print td_ReadValue("PIDSLoop.1.SGain")
1007645
```

In almost all software versions, the X loop is on the first PISLoop (PISLoop.0), the Y is on the second (PISLoop.1) and Z is on the third (PISLoop.2). There is a small chance that the X and Y are on PISLoop.3 and PISLoop.4. You would see this went you went to read the gains.

To set up a feedback loop you will use the `ir_SetPISLoop` command, this is because the PIDSLoop panel is automatically updated by the AFM software and does not allow manual overwrite. The syntax of the `ir_SetPISLoop` command is

```
ir_SetPISLoop(whichLoop, eventString, inChannelString, setpoint, pgain,
  igain, sgain, outChannelString,OutputMin,OutputMax)
```

The parameters are

whichLoop determines which of the six feedback loops you will use. They all run at 100 kHz.

eventString is just like the events described above. Unlike with input and output waves, with feedback loops you often want them to start running immediately. Passing a string of "Always" for the eventString will make this happen for the start string, there also needs to be a comma and the a string for the stop event, in our examples "Never".

inChannelString is the channel string for the input to the feedback loop. For example, in contact mode, this is one of the ADCs (InFast usually), which is wired up to Defl.

setpoint, pgain, igain, sgain are the setpoint and gains for the loop. After the loop is set up, the setpoint can be driven by a wave. This would be used, for example, to move the XY stage around under feedback control. Note the gains are scaled to be mathematically right. A pgain of 1 will give you a 1 volt output for a 1 volt input. Similarly, the same 1 volt input with an igain of 1 will give you a ramp from 0 to 1 in one second.

outChannelString is the output of the feedback loop. For contact mode, this would be the Z DAC.

OutputMin and OutputMax is the range of values used for the OutputChannel.

1.10 Setting Up Your Own Z Feedback Loop

So, to set up your own feedback loop in Z, first turn on the meter and make sure you have a deflection close to zero. Also make sure the cantilever is close to the surface by doing an engage the normal way and then withdrawing. Above you saw that for contact mode, typical gains are P=0, I=1000, and S=0. First you will hook up the Defl channel to the InA ADC by executing

```
print td_WriteString("ARC.Crosspoint.InA","Defl")
0
```

Next you will set up the feedback loop with

```
print ir_SetPISLoop(2,"Always,Never","ARC.Input.A",1,0,1000,0,
  "ARC.Output.Z",-10,150)
0
```

If all went well, you should see on the meter that the Zvoltage went to a stable value and that the deflection is at 1 volt.

Fig Closed Loop Circle We need to figure out what the gains are for X,Y and Z. This is done with the same read value commands as above. The microscope should be engaged and scanning.

```
variable/G X_PGain, X_IGain, X_SGain, Y_PGain, Y_IGain, Y_SGain,
  Z_PGain, Z_IGain, Z_SGain
X_PGain = td_RV("PIDSLoop.0.PGain")
X_IGain = td_RV("PIDSLoop.0.IGain")
X_SGain = td_RV("PIDSLoop.0.SGain")
Y_PGain = td_RV("PIDSLoop.1.PGain")
Y_IGain = td_RV("PIDSLoop.1.IGain")
Y_SGain = td_RV("PIDSLoop.1.SGain")
Z_PGain = td_RV("PIDSLoop.2.PGain")
Z_IGain = td_RV("PIDSLoop.2.IGain")
Z_SGain = td_RV("PIDSLoop.2.SGain")
printf "X Gains: P:%.4g I:%.4g S:%.4g\r", X_PGain, X_IGain, X_SGain
printf "Y Gains: P:%.4g I:%.4g S:%.4g\r", Y_PGain, Y_IGain, Y_SGain
printf "Z Gains: P:%.4g I:%.4g S:%.4g\r", Z_PGain, Z_IGain, Z_SGain
```

Now let's make a variable Radius (the size od the Radius circle in volts). The range of the sensor is between -10V and +10V but they don't use the full range. Try using 3V for a radius.

```
variable Radius
Radius=3
```

We then need make all the waves.

```
Make/N=(1024)/O XVoltage YVoltage XSensor YSensor XCommand YCommand
Display/K=1 /W=(5.25,41.75,399.75,250.25) XVoltage
Appendtograph/R YVoltage
Display/K=1 /W=(7.5,275.75,402,484.25) XSensor
Appendtograph/R YSensor
Display/K=1 /W=(409.5,41.75,662.25,250.25) YSensor vs XSensor
ModifyGraph width={Plan,1,bottom,left}
```

Since we haven't set up the waves yet, we will still have to do p (point) scaling so we will do the sign and cosine based on p (point) scaling rather than x scaling (time) like we did for Open Loop Circles.

```
XCommand = Radius*cos(2*pi*p/1024)
YCommand =Radius*sin(2*pi*p/1024)
```

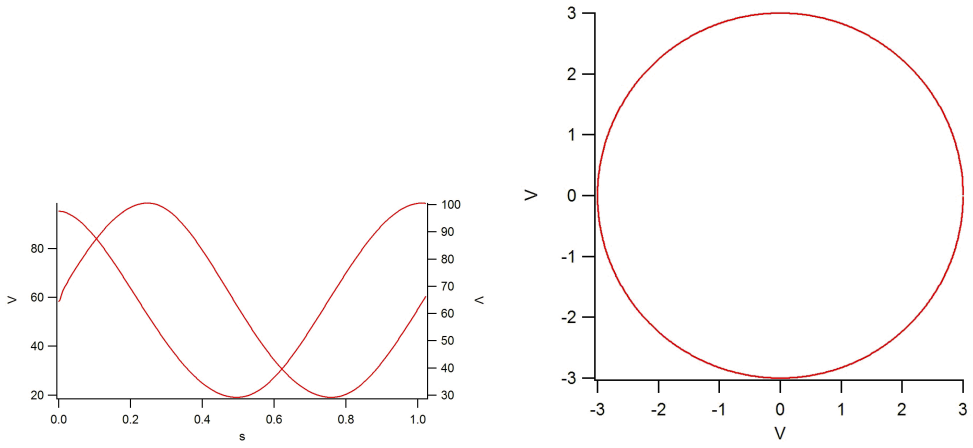
We should do a stop to make sure nothing else is running.

```
print td_stop()
0
```

We now setup the feedback loops for X and Y. Notice how the initial setpoint is set to the Radius for X and 0 for Y. This is because the XCommand starts at a Voltage equal to the Radius and the YCommand starts at 0V.

```
print ir_SetPISLoop(0,"Always,Never","ARC.Input.X",Radius,X_PGain, X_IGain,
  X_SGain,"ARC.Output.X",-10,150)
0
print ir_SetPISLoop(1,"Always,Never","ARC.Input.Y",0,Y_PGain, Y_IGain,
  Y_SGain,"ARC.Output.Y",-10,150)
0
```

We then pass the feedback loop voltages that we would like using a command signal and it will vary the XVoltage and YVoltage to try to achieve this. We will use the trick to look at outputs (Output.X) as inputs to see what the voltages are and similarly we display the input voltages.



(a) X,Y Voltage vs Time **(b)** Y Sensor vs. X Sensor

Figure 1.5

```
print td_xSetOutWavePair(0, "0,0", "PIDSLoop.0.Setpoint", XCommand,
 "PIDSLoop.1.Setpoint",YCommand,100)
0
print td_xSetInWavePair(0, "0,0", "ARC.Output.X", XVoltage, "ARC.Output.Y",
  YVoltage, "", 100)
0
print td_xSetInWavePair(1, "0,0", "ARC.Input.X", XSensor, "ARC.Input.Y",
  YSensor, "", 100)
0
```

As you can see in Figure 1.5 on page 13, the circle should be perfectly round.

1.11 Creating Your Own Programs

While all the examples here have been done from the command line, it is pretty easy to start copying and pasting commands to make your own functions. The functions above for open loop circle, closed loop circle and finding the gain parameters are included in the Programming procedure file

you should have received with this file . You now know enough to be dangerous. Have fun and feel free to contact us at

Support@AsylumResearch.com

or by calling us if you need some more help.