21M.370 Digital Instrument Design

Lab 4b - Mar 4

This week we will be exploring working with sensor signals and mapping them to synthesis parameters.

Our goal for today is to take some of the approaches from monday and use them with analog input from optical sensors.

By the end of today you should:

1. be receiving at least two analog values into PD
2. have those signals mapped to a synthesizer
3. understand how to scale sensor values to appropriate ranges for PD
4. be able to convert control signals to audio signals to control automatonism parameters
5. be able to use the shmitt and ischmitt objects to generate triggers
6. have a minimal musical instrument you can play!

The file we will be using is NIME/Puredata/MappingExamples/excitation_analog

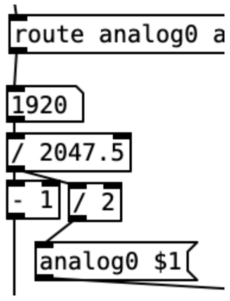
Part 1: Getting our system running

Our system consists of 3 parts:

1. an ESP32 microcontroller, which converts analog signals to digital signals, and sends them over USB/serial to python.
2. a python script, which receives the serial data and converts it to a format PD can read, then sends the data to PD.

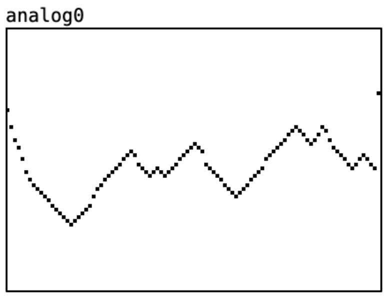
3. a PD patch which contains the dataMonitoring object. dataMonitoring receives the signals from python, scales them appropriately, and sends them out labelled as analog0, analog1, etc. dataMonitoring also has a small graphical display of incoming data. This can be useful to make sure data is coming in correctly and give you a visual indication of the data range and behaviour.

The scaling in dataMonitoring for analog0 ->



```
route analog0 a
1920
/ 2047.5
- 1   / 2
analog0 $1
```

The graphical display for analog0 ->



Here is what you need to do to get sensor data into PD:

1. Program the M370_ALL_ANALOG firmware onto the ESP32. You only need to do this once - after that the ESP32 will start sending data to python as soon as it is plugged in.
2. Open the 370_HELLO_PD.py python script. Run it and you will likely get an error saying the serial port is not available. At the top of the python console the script will print out a list of available serial ports - the port associated with the ESP32 will hopefully be called something like 'usbserial'. If not, you will have to figure out which port the ESP32 is using. If you look at Arduino->Tools->Port you will also see a list of ports, and the port in python (should) use the same name.
3. Edit the python script to use the correct serial port (it should be line 45: ser = serial.Serial("your serial port goes here")

4. Run the script again and it should list the serial ports, say it is sending data, and then may have a superfluous 'none' message. After that you won't see anything in the python console
5. Open a PD patch with dataMonitoring in it.

After that you should see values inside dataMonitoring in the graphical display.

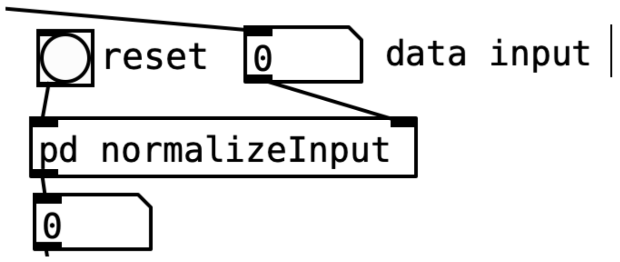
Part 2: Working with analog values

To get an analog signal into PD, we first have to convert it to a digital value. Most microcontrollers convert analog signals into integers with a certain bit depth. In our case, the ESP32 has 12-bit convertors, so output a value from 0 to 2^12, or from 0-4095.

It is often helpful to scale this to a standard range, often from 0.-1, just by dividing the incoming range (0-4095) by 4095. dataMonitoring does this for you.

The resulting signal, though, will not actually range from 0-4095 - instead it will be some subsection of that range. What we need to do is find the *actual* low and high values for our analog signal, and then scale that range to 0-1.

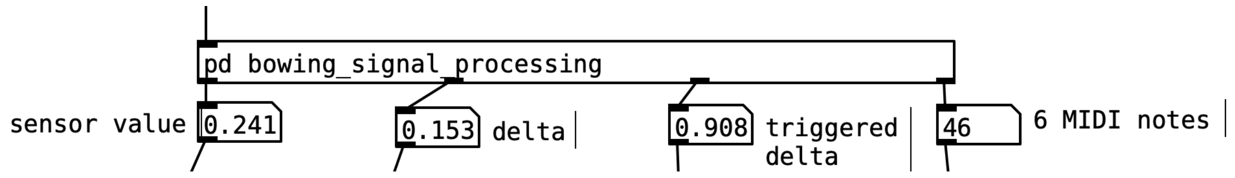
We do this in the 'pd normalizeInput' subpatch. If you click it you can see how it works. You can also just see the results in the number box below the subpatch.



One thing you should be aware of is that we are monitoring the incoming analog values and just looking for the lowest and highest range. To get this to work correctly, we need to calibrate 'pd normalizeInput' by sending in the highest and lowest values we get. However, if for some reason the environmental conditions change or there is a spurious value our calibration may get off. If that happens we can press reset and recalibrate.

part 3: sensor signal processing

In the 'pd bowing_signal_processing' subpatch we have the same algorithms we worked with last week tidied up into subpatches and configured to work with our sensor data. This time, though, instead of choosing a particular algorithm the subpatch uses all of them and sends them out individually:



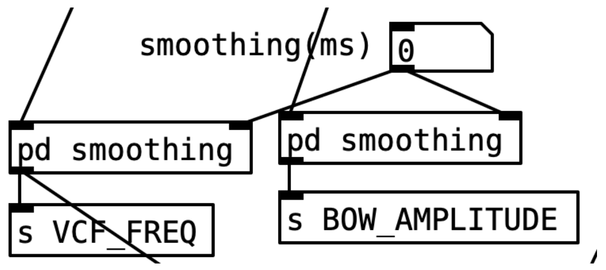
Sensor value: just follows incoming sensor values

Delta: changes in sensor value

Triggered delta: the current delta value when the sensor value exceeds a threshold

6 MIDI notes: multiple thresholds triggering a series of MIDI notes

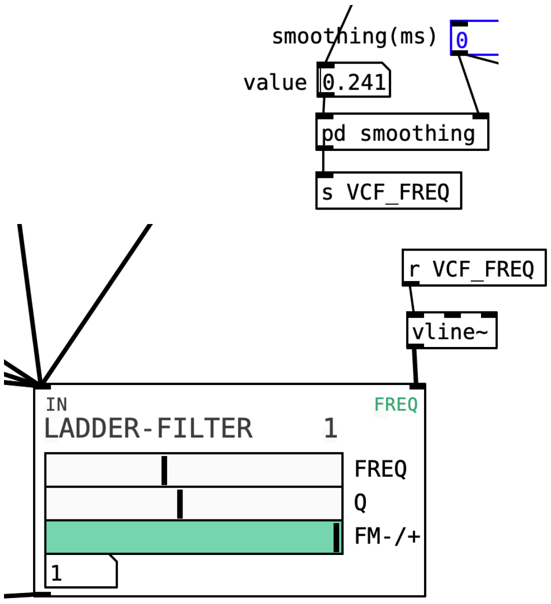
We also have a smoothing subpatch which we use to smooth the sensor signals and prepare to send them to vline~ objects, which convert the values to a smooth audio signal. Different values for smoothing (from 10 - 500, maybe) will change the feel of the instrument.



part 4: mapping

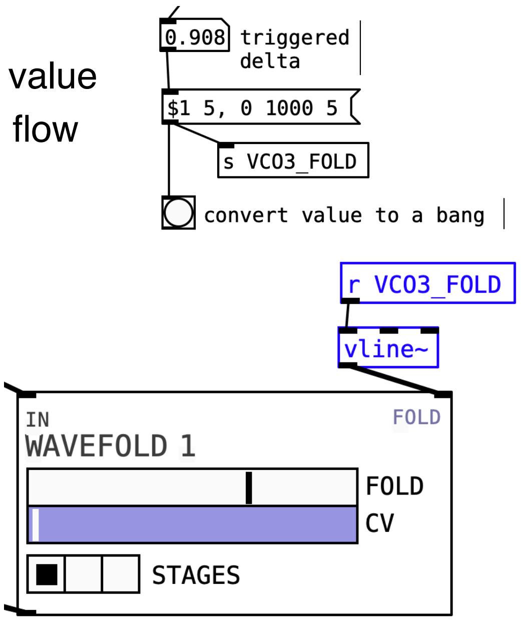
The synth used in this patch is relatively simple, but still has lots of parameters we can map sensor signals to. We have two kinds of signals in this patch:

1. continuous signals (sensor value and delta): continuously put out a value. Suitable for sending to any CV input.
2. discrete values (triggered delta and MIDI notes): only send out data when a threshold is exceeded. Can be used to trigger envelopes, change pitch, trigger specific events, etc.



For continuous signals, the signal flow is:

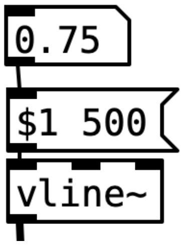
1. continuous value
2. smoothing subpatch
3. send object with parameter name
4. receive object with parameter name
5. vline~
6. input to automatonism module



For discrete signals which are sending out a value (the triggered delta, for example) , the signal flow is:

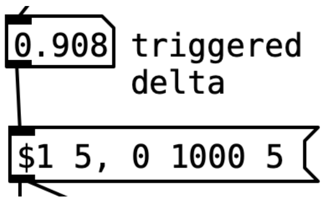
1. discrete value
2. envelope message with appropriate timing values
3. send object with parameter name
4. receive object with parameter name
5. vline~
6. input to automatonism module

Digression: vline~



As a reminder, vline~ expects messages in one of two formats:

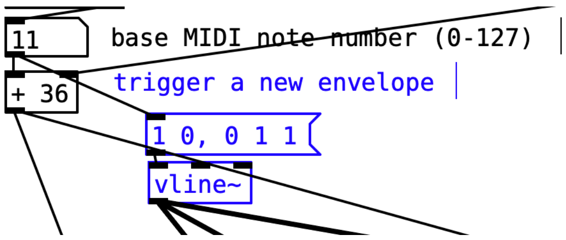
1. <destination value> <time to reach the destination>
    1. for example, a message box with ‘0.75 100’ will take 100ms to reach the value of 0.75
    2. the line~ object is a simpler object and also accepts value-time messages.
2. a compound message which starts:
    1. destination value - time (just like above)
    2. followed by a comma. The comma splits the message into two - the first one described above and a 2nd message described below.
    3. then a 2nd message with value-time-delay
    4. the value-time part of this is the same, but the delay is how long to wait after the message is received to start moving towards the destination.



Using our triggered delta as an example, the image to the left shows a delta value of 0.908. When this value is triggered, it goes to the message box and then two messages are sent to a vline~ which interprets them message as:

<0.908 5> ’immediately start going to and get there in 5ms’

<0 1000 5> ‘wait 5ms and then start going to 0 and take 1000ms to get there’



We can also have discrete signals which don’t have a value but are just a trigger. We can convert a discrete value to a trigger in two ways:

1. message boxes which don’t have a $1 will send out the message as written anytime they receive input. The ’11’ in the image above causes the messages <1 0> and <0 1 1 > to be sent to vline~
2. by using the ‘bang’ object. You can find this under the put menu. The bang object literally sends a bang message when it receives any input. This bang message can be used to for lots of purposes - including causing objects to send out their current value without changing it.

/digression

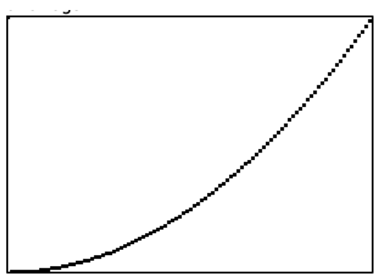
In the patch we are using, all of the available parameters are made available by using send and receive objects (’s’ and ‘r’). When send and receive objects have the same name anything sent into the send appears at the output of the receive.

Remember, all of the parameters we are using are expecting values from 0-1 (except MIDI pitch!). So feel free to map any control value to any synth parameter. Experimentation is key - just try some things out.

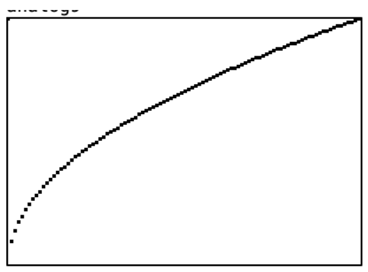
A few things to consider:

- sometimes you don't want to use the whole range from 0-1. you can use the scale object to map a control range to any desired parameter range. the arguments are (input value low, input value high, out low, out high)
- the 5th argument of scale is an exponent which can cause the output data to have an exponential or logarithmic curve. values lower than 1 will cause the output to spend more time below 0.5, while values greater than 1 will cause the output to spend more time above 0.5

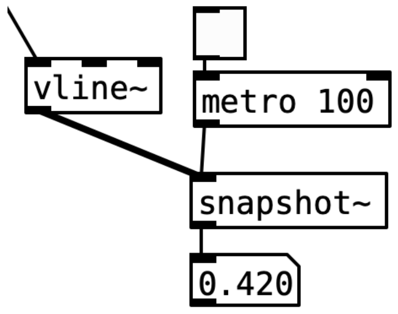
.exponent of 2:



exponent of 0.5:



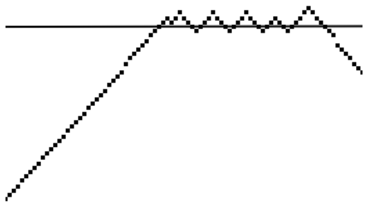
- the actual parameter value will also be scaled by the cv slider in most automanism objects. Follow the signal flow and pay attention to every math operation - monitor the signal at different points - and listen to the results.
- You can monitor the actual signal being sent to automatonism by using these objects:
  You will see these sprinkled around the patch - feel free to copy and paste and use them. Click the toggle to turn monitoring on and off.



Schmitt triggers

A final point about triggers - when you want to create a trigger you often set a threshold which causes the trigger to be generated when the threshold is exceeded. It is good practice to prevent accidentally retriggering when noise in the signal causes it to go above and below the threshold repeatedly.

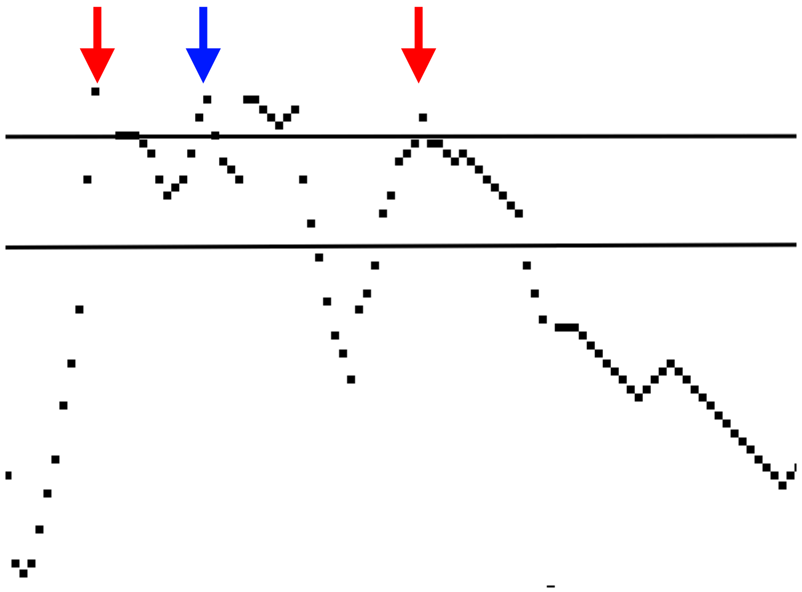
Here we can see such an object - using a simple threshold will cause several triggers when we really only want one.



To prevent this, we will set two thresholds - a trigger will be generated when the signal goes above the first threshold, but the signal won't generate another trigger until it goes below the 2nd threshold. This is called a shmitt trigger.

Main trigger->

retrigger threshold->



In this example the red arrows point toward when triggers occur. The blue arrow does not cause a trigger because the signal hasn't dropped below the 2nd threshold yet.

The ‘schmitt’ and ‘ischmitt’ objects implement a schmitt trigger. ‘ischmitt’ is an inverted schmitt, which cause a trigger when a signal goes below a threshold.

The first inlet is the data input. The 2nd input sets the main threshold. The third input sets the 2nd threshold as a ratio of the first. e.g. a main threshold of 0.5 and a ratio of 0.8 will set the second threshold to be (0.5*0.8) = 0.4.