AS MEX H5 processing in GeoPIXE

Contents

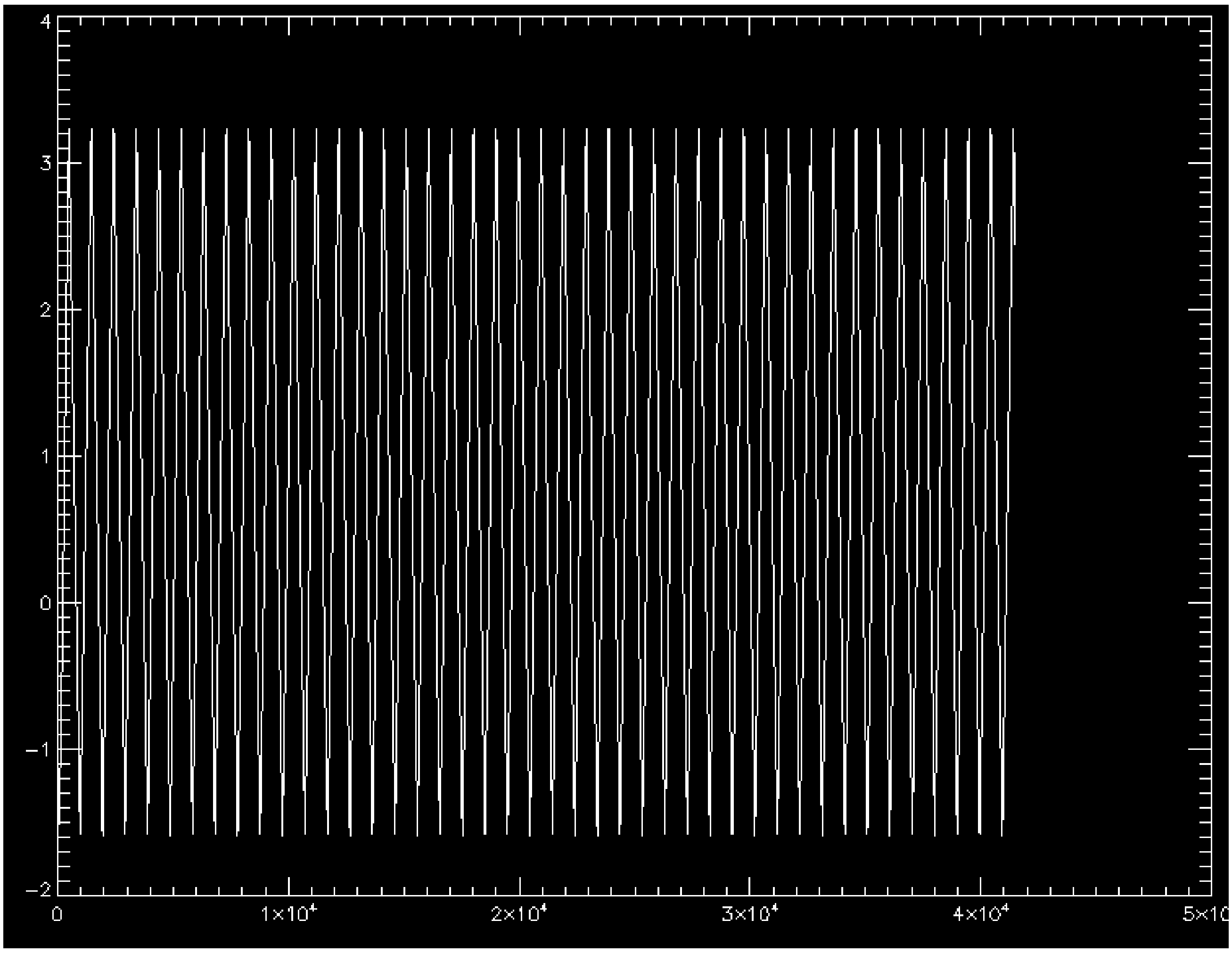
Debug contents of MEX hdf5 file

Notes about what we find …

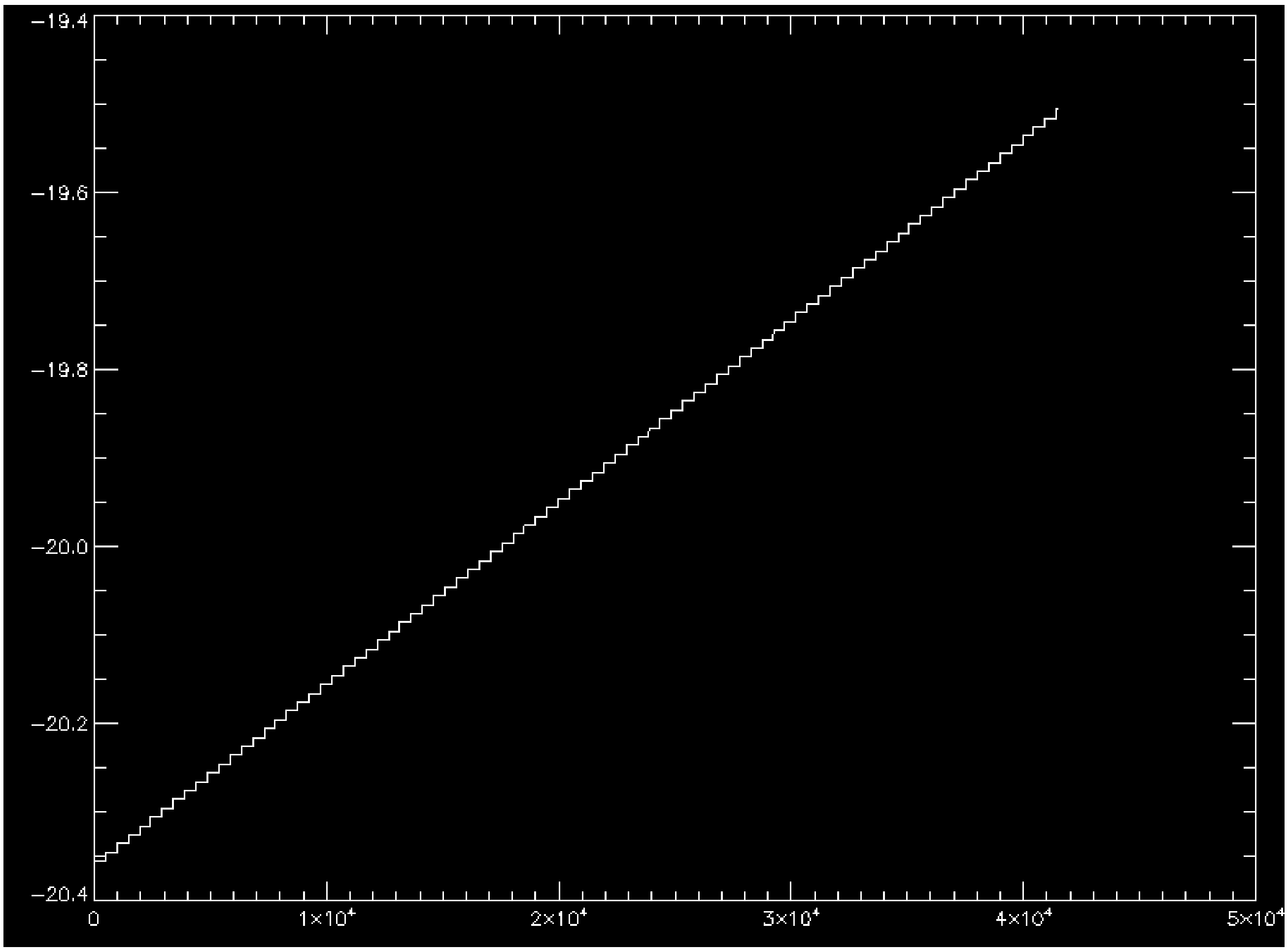
1. There is no metadata for 'nx', 'ny', nominal dwell, beam energy and detector energy cal.
2. There is an 'abs_x', which is absolute 'x' position (float) and an 'x', which is relative 'x' (but first item seems corrupted). Need to determine pixel x,y from these and an 'effective nx,ny?
3. Records 'x_ts' and 'y_ts', 'i0_ts', etc. are time stamps (seconds since 1970 epoch)
4. 'position' is a sequence number index (starts at 0). Useful to use, as this permits treatment of multiple files with extended sequence.
5. 'spectrum' is the spectra from 4 detectors over pixel count/sequence index (1,41496,4,4096). The first redundant index is energy (for XANES maps). In Fortran/IDL index order this data has dimensions [4096,4,41496,1], i.e. channels in spectra fastest.
6. There does not seem to be a time-stamp for spectra, so will assume that these use the 'x_ts' time-stamps.
7. 'i0' is a flux, 'i1' is a flux, 'i0_ts', 'i1_ts' are time stamps (s).
8. 'dcm_energy_ev', energy (eV) by sequence. 'mex1_dcm_bragg' by Bragg angle.
9. 'Accelerator_ring_current', ring mA versus sequence.

Test data

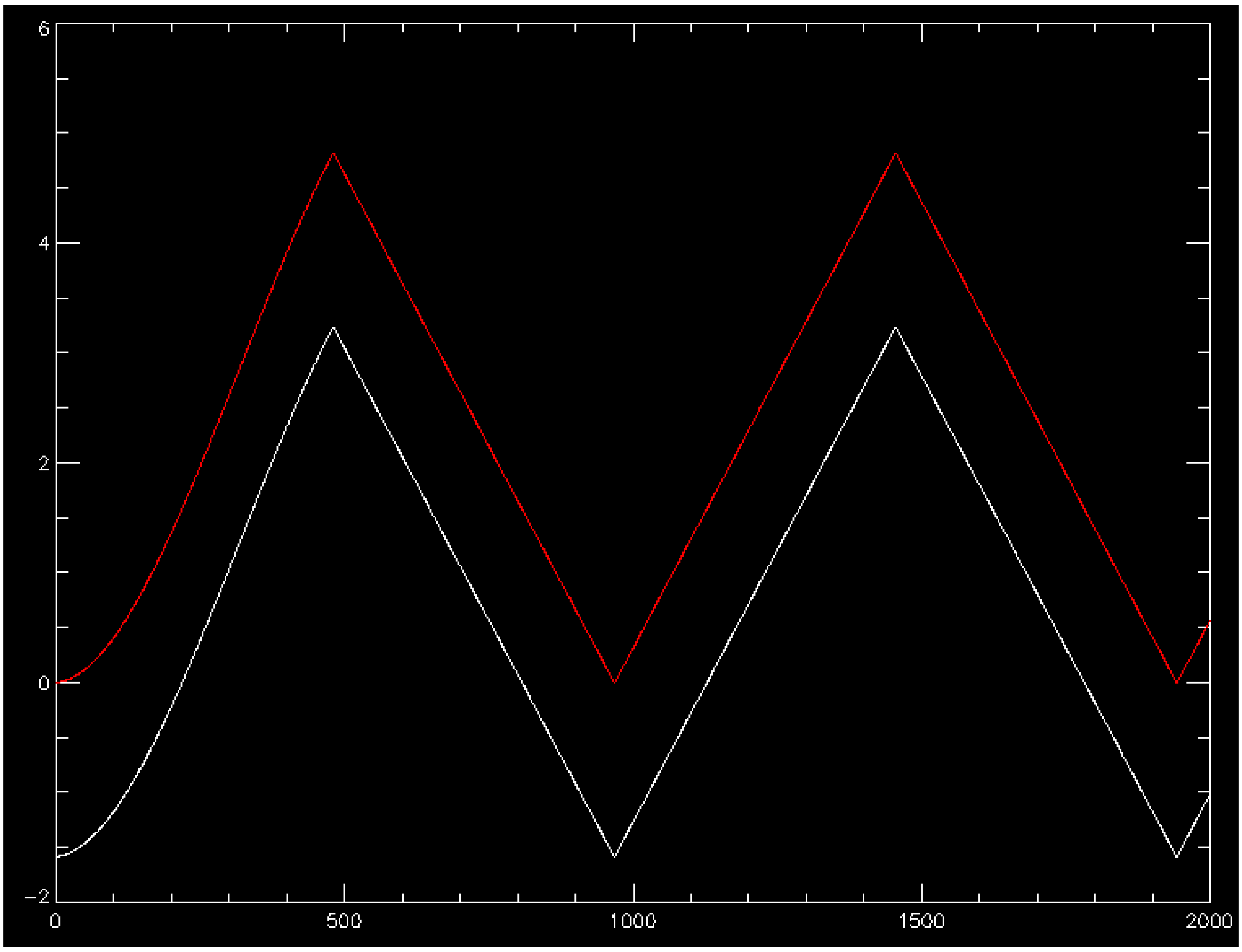
Data file: microprobe_20250411_011259+1000_merged.hdf5 (note "+" in filename causes problems). Energy 11 keV, major peaks due to IL, Fe, Ni, Cu, Zn, minor Cl, Ca, Ti. Fitted using a garnet skarn setup for ME3 at XFM, for 11 keV, which is close enough for now.



Abs_x



Abs_y



Abs_x[0:2000] (white) versus x[0:2000] (red) – red is just starting at (near) zero

Seems to have about 43 swings in 'abs_x'. Does not step evenly at first. Used this code to get an average step size (assumed equal in x and y) … Total number of sequence steps (41496) is less than the inferred from product nx (488) * ny (85).

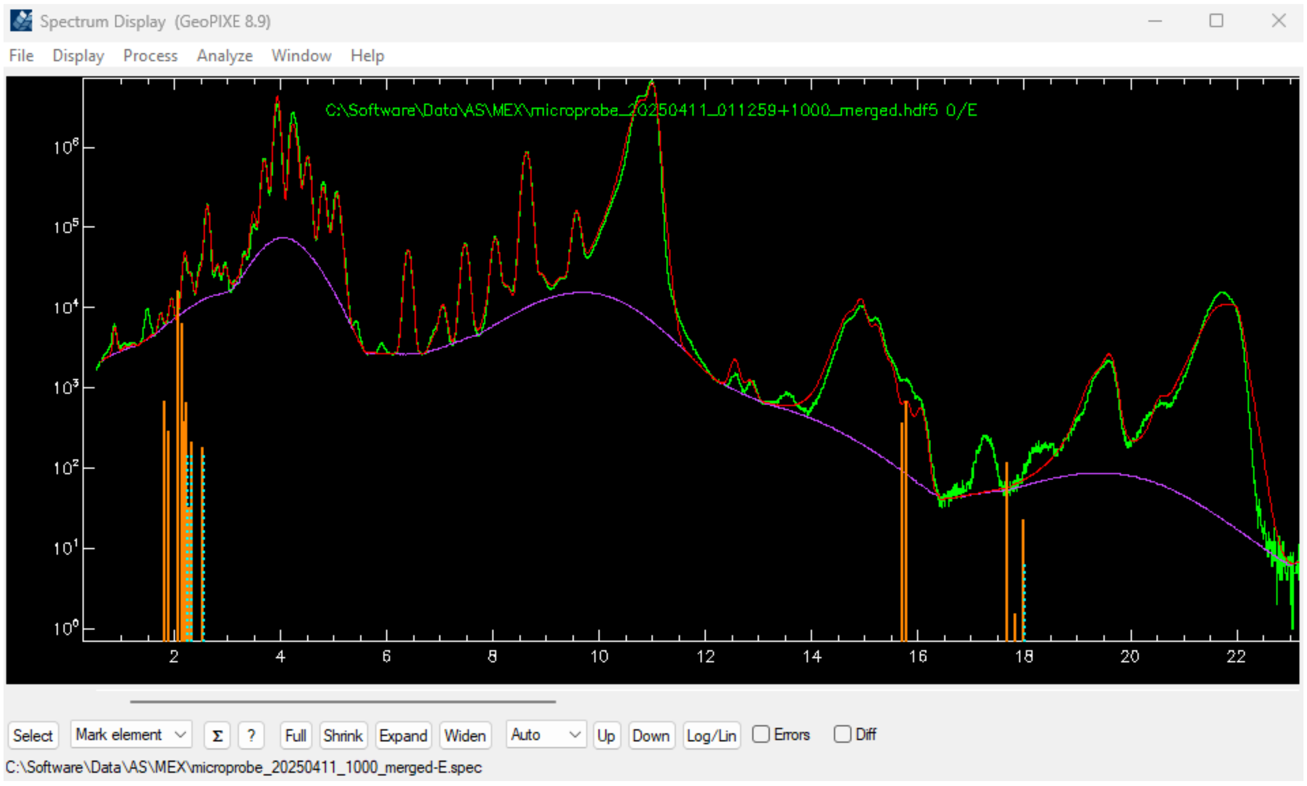
```
step_x = x - shift(x,1)
step_x[0] = step_x[1]                                   ; fix wrap in first value
step_x = mean( median( abs(step_x[nseq/10:*]),5))       ; best shot at ave. step in X
step_y = step_x                                         ; assume step in Y is the same as X?

pixel_x = round( (x - min(x)) / step_x )                ; pixel addresses
pixel_y = round( (y - min(y)) / step_y )
min_x = min( pixel_x)
min_y = min( pixel_y)
nx = max(pixel_x) + 1                                   ; effective nx, ny
ny = max(pixel_y) + 1
```

Since we work out the pixel x,y from the relative x,y for each sequence step, easier was to read all sequence steps for one detector channel and assign x,y this way … Then step through detector channels (4). Progress was then simply by detector channel. Later with energy, could step through detector and energy per pass.

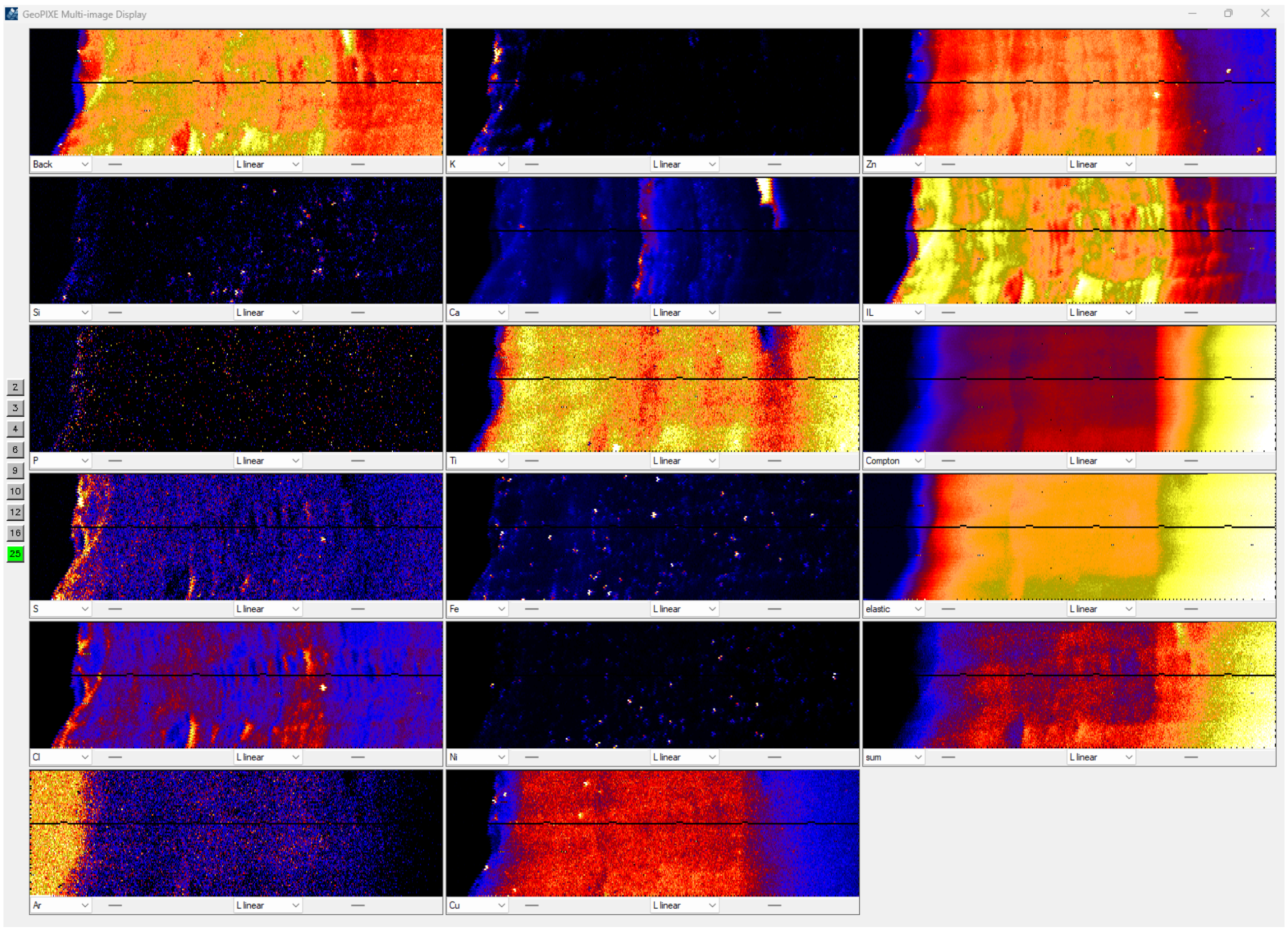
```
for j=0L,nseq-1 do begin
        pnc_hdf_x1[*,j] = x[j]
        pnc_hdf_y1[*,j] = y[j]
        pnc_hdf_e[*,j] = ramp
endfor
```

Ignore X time-stamp variation

Ignoring the fine-grained time-stamp variation (see below), we get this …



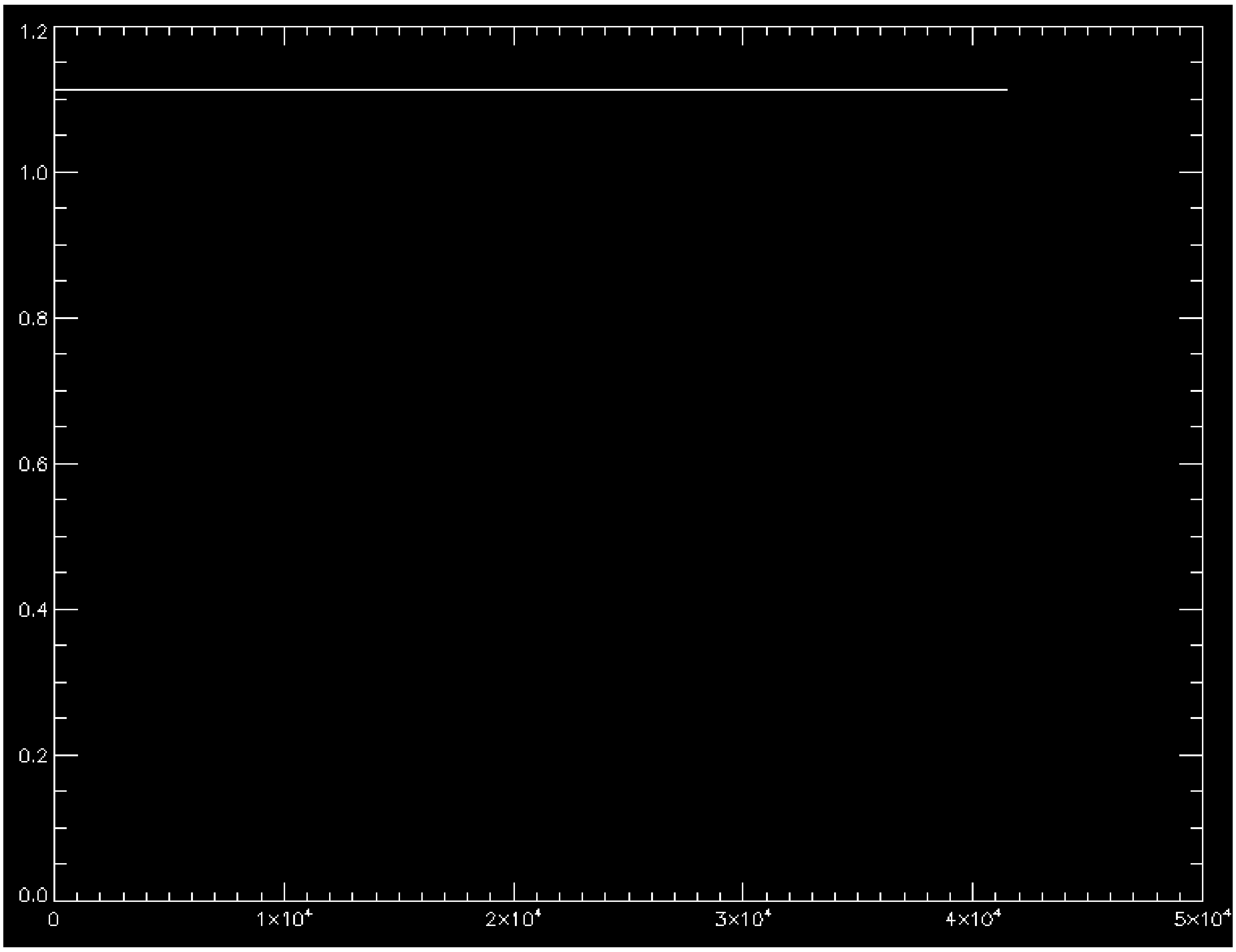
C:\Software\Data\AS\MEX\microprobe_20250411_1000_merged-E.spec



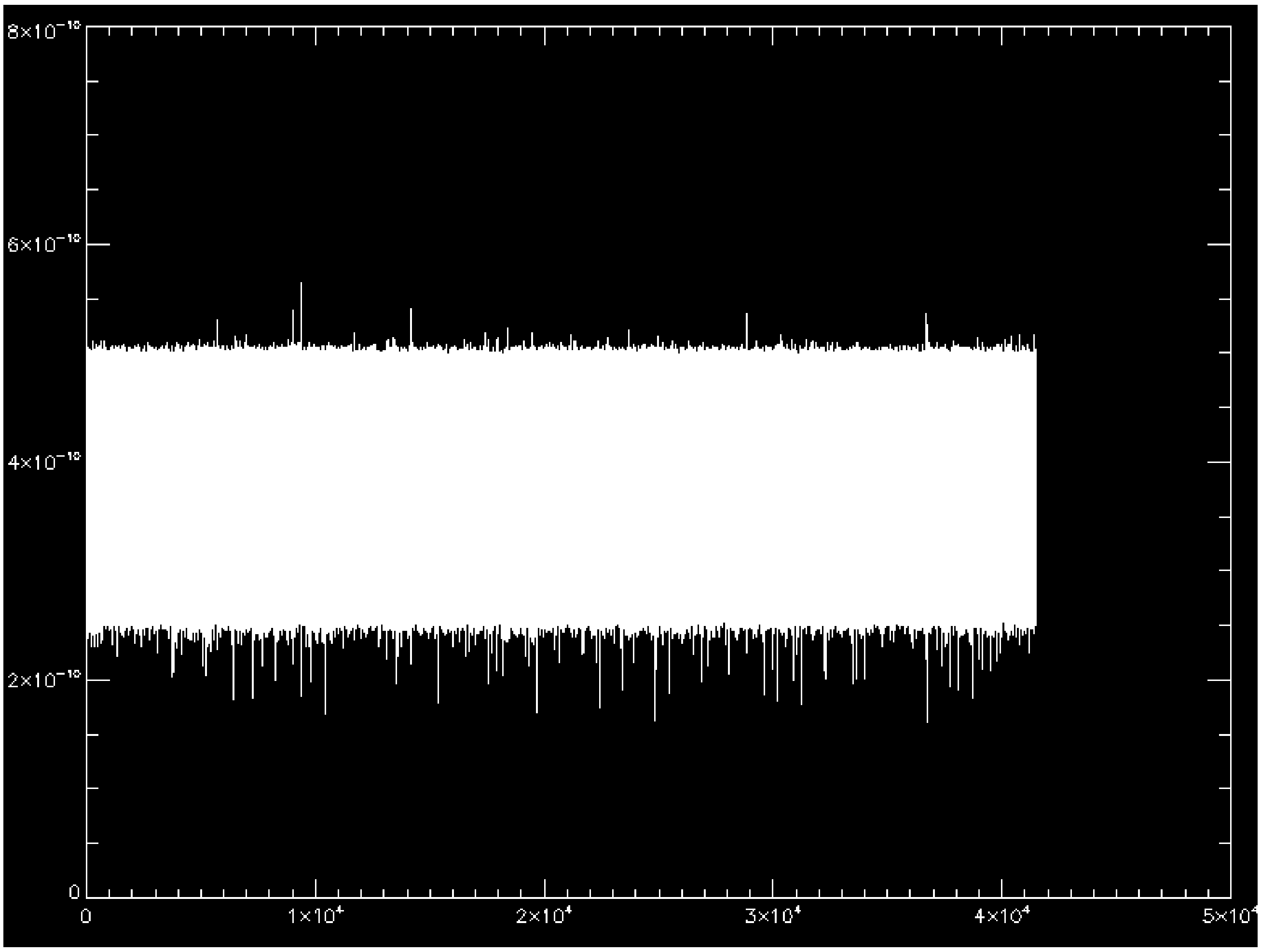
Black (missing) pixels across images may be an artefact of determining pixel address from supplied x,y positions or assumption of equal step_x, step_y. Seems to be some “jaggies” shearing, evidence for back-lash or logging delays.

More on time stamps

Looking at the time-stamp ‘x_ts’ values, we see this … and the differences should be dwell …

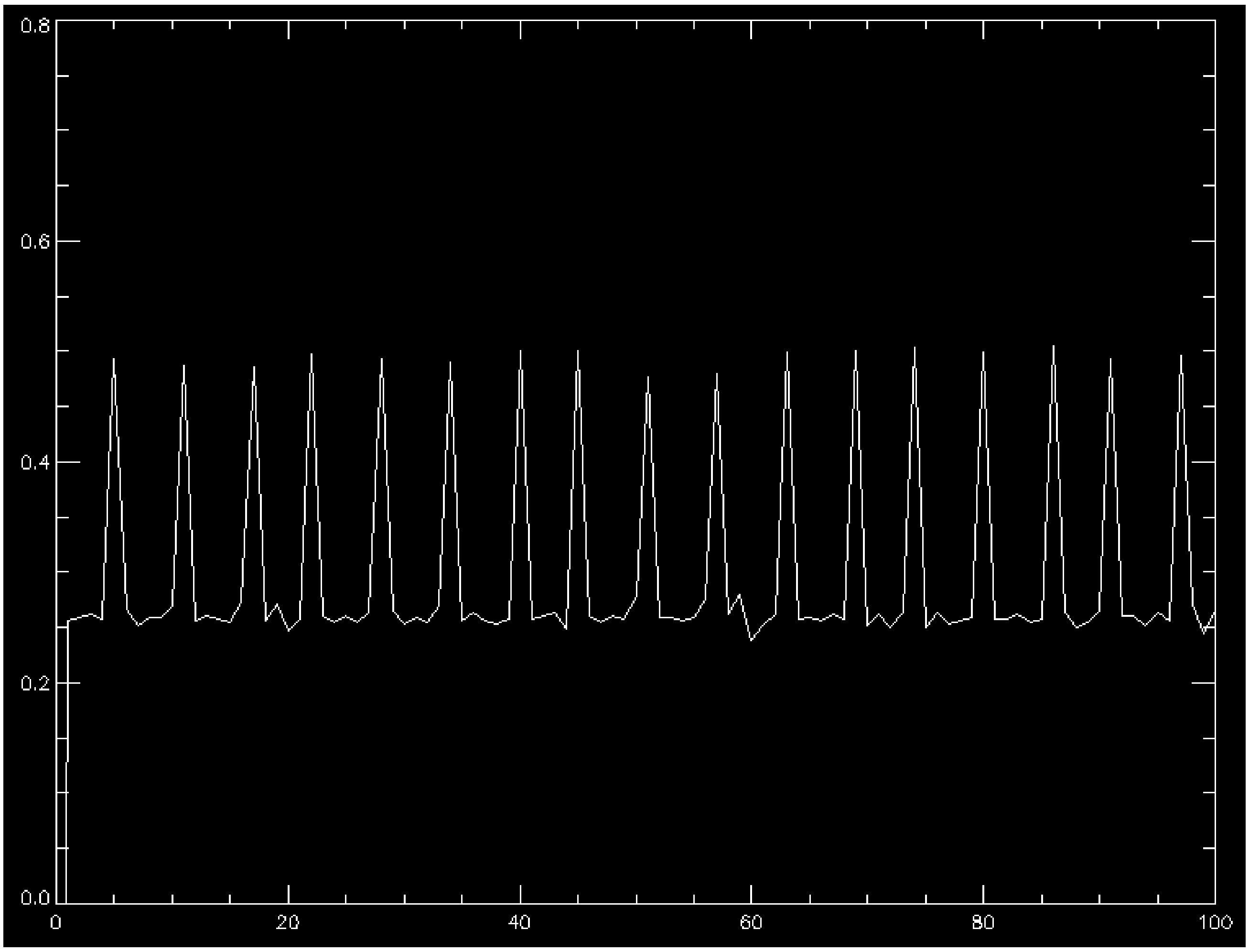


X_ts



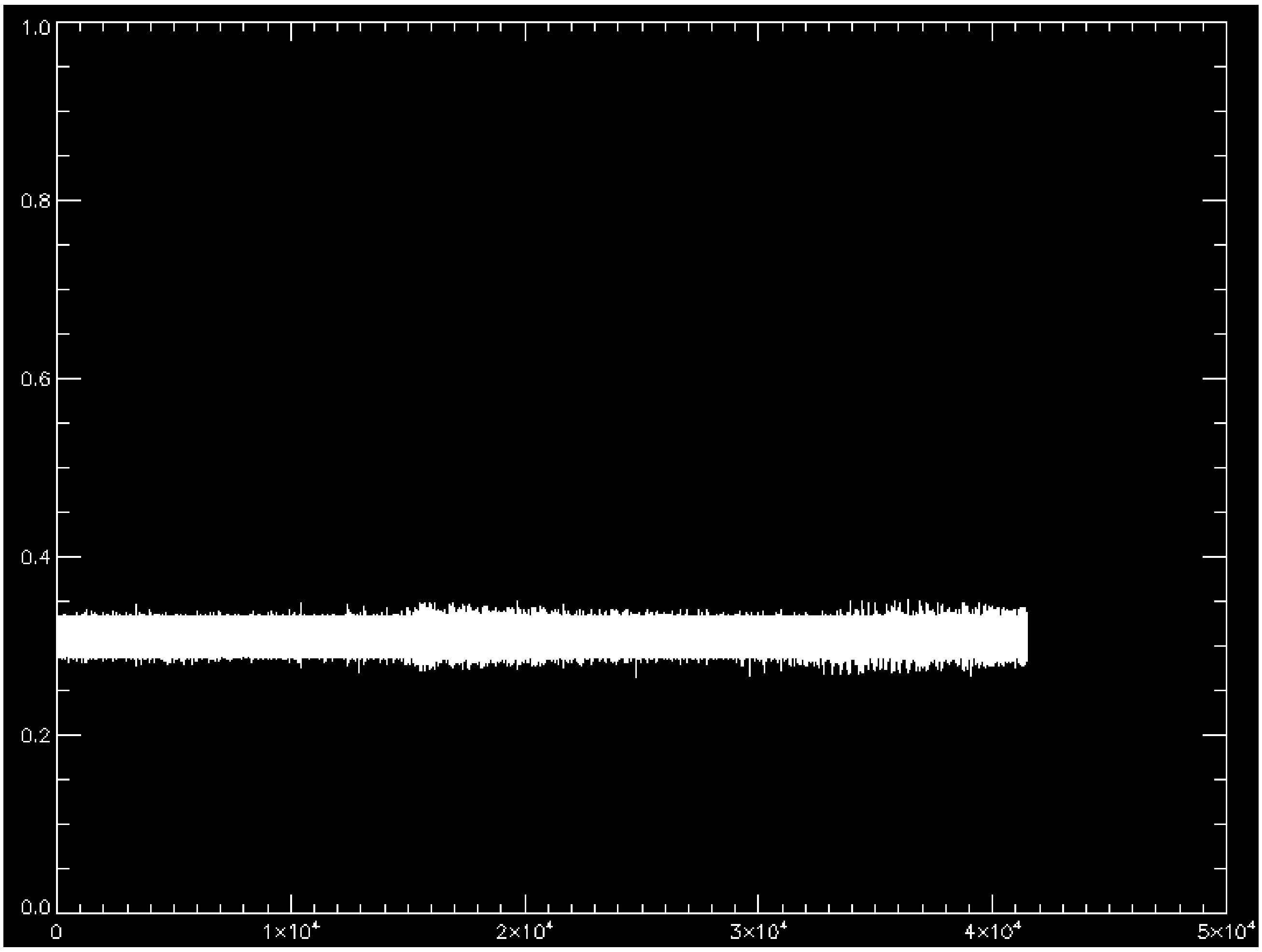
*x_ts - **shift**(x_ts,**1**), yrange=[**0**,0.**7**] – mean around 0.4 seconds? Why such large scatter?*

Looking in detail, we see this …

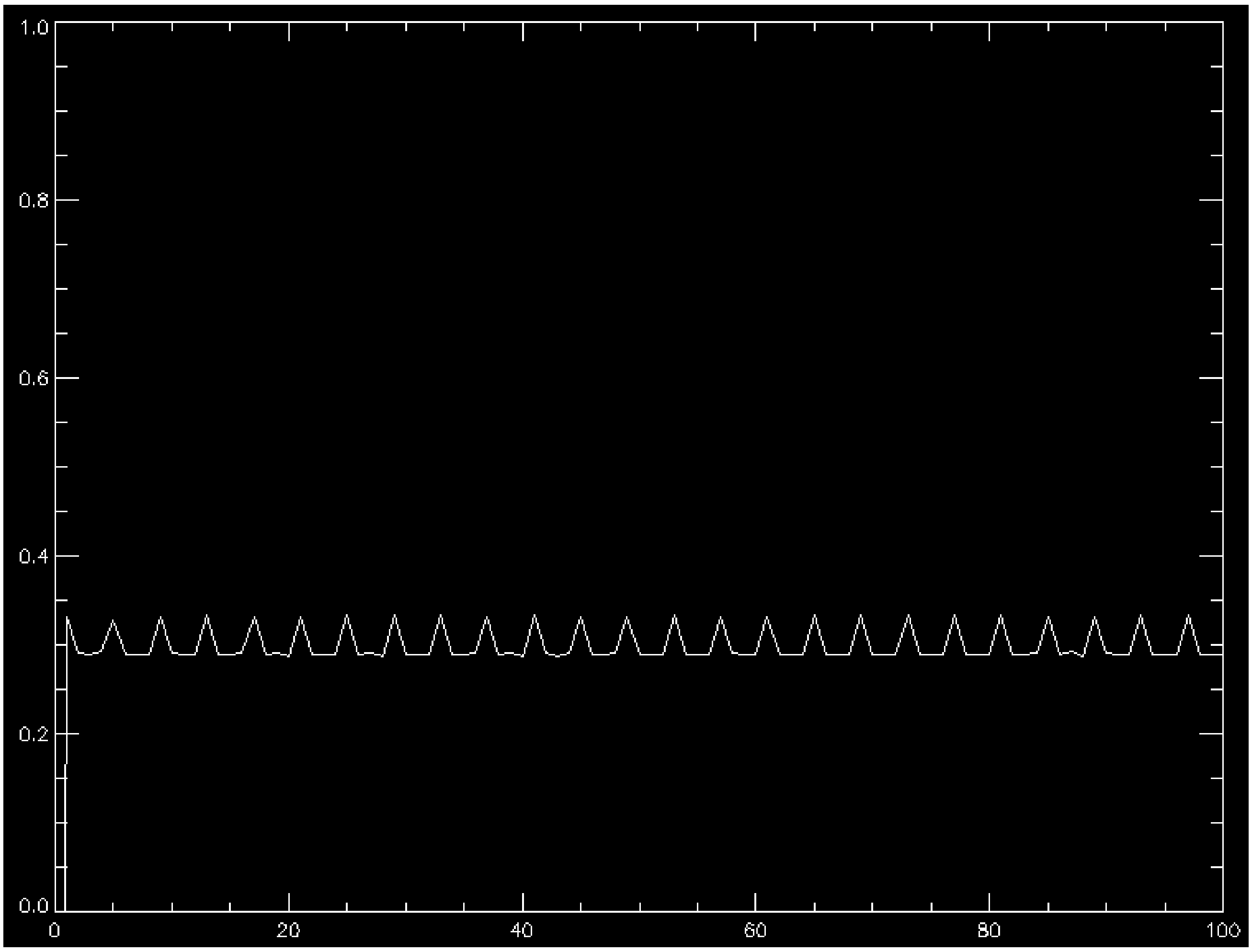


First 100 values

The dwell (as differences between time-stamps) averages about 0.3 seconds. But jumps over about 0.2 seconds (every ~7th sample is high). This is probably not real, some artefact of the logging software? (Later below it is clear that this is an artefact, as the spectra counts do not change like this).



*I0_ts - **shift**(i0_ts,**1**)*



*I0_ts - **shift**(i0_ts,**1**), first 100 values*

These time-stamps vary less. But they average around 0.3 seconds with jumps of 0.05 s. The average is similar to the x_ts, but the jumps are different. However, the most common dwell from x_ts is 0.26 s, while the most common from i0_ts is 0.29 s. These differences are offset by the different jumps. Clearly, the time-stamps are seriously error probe, systematically.

Revisit time-stamped data

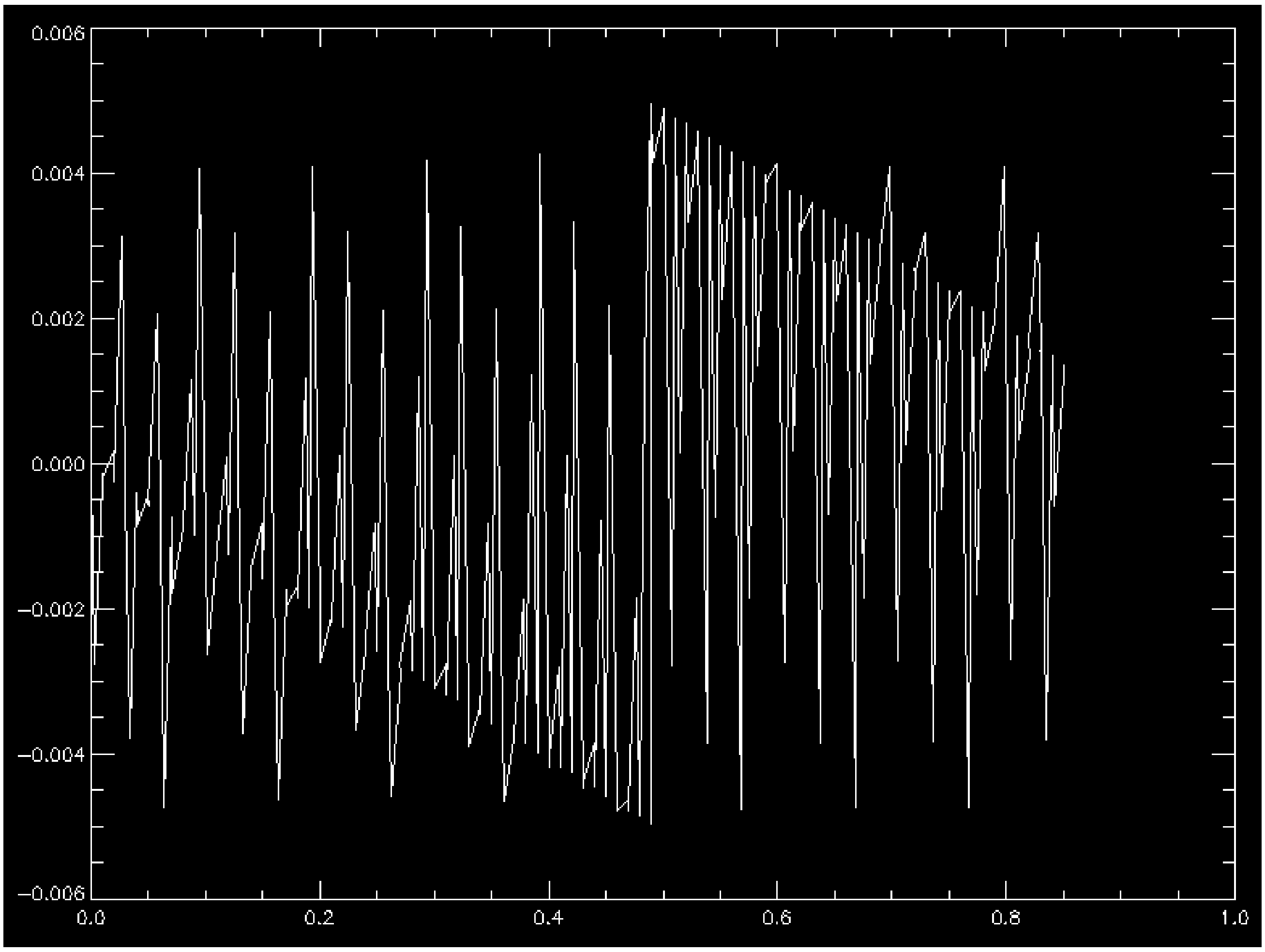
Really should be doing this, to allow for a different time-step for the x,y versus i0 time-stamping:

1. After pixelating x,y into even spaced pixels, interpolate to get the effective time-stamp for each pixel. Start with the rounded pixel_x, scaled by step_x to effective (offset) position.
2. Determine step_y independently, and find that it differs from step_x.
3. Then interpolate into i0 time-stamps to get the correct i0 for each pixel. This is interpolated.
4. Do this <u>separately</u> for any other time-stamped streams, such as i1, dcm-energy, etc.

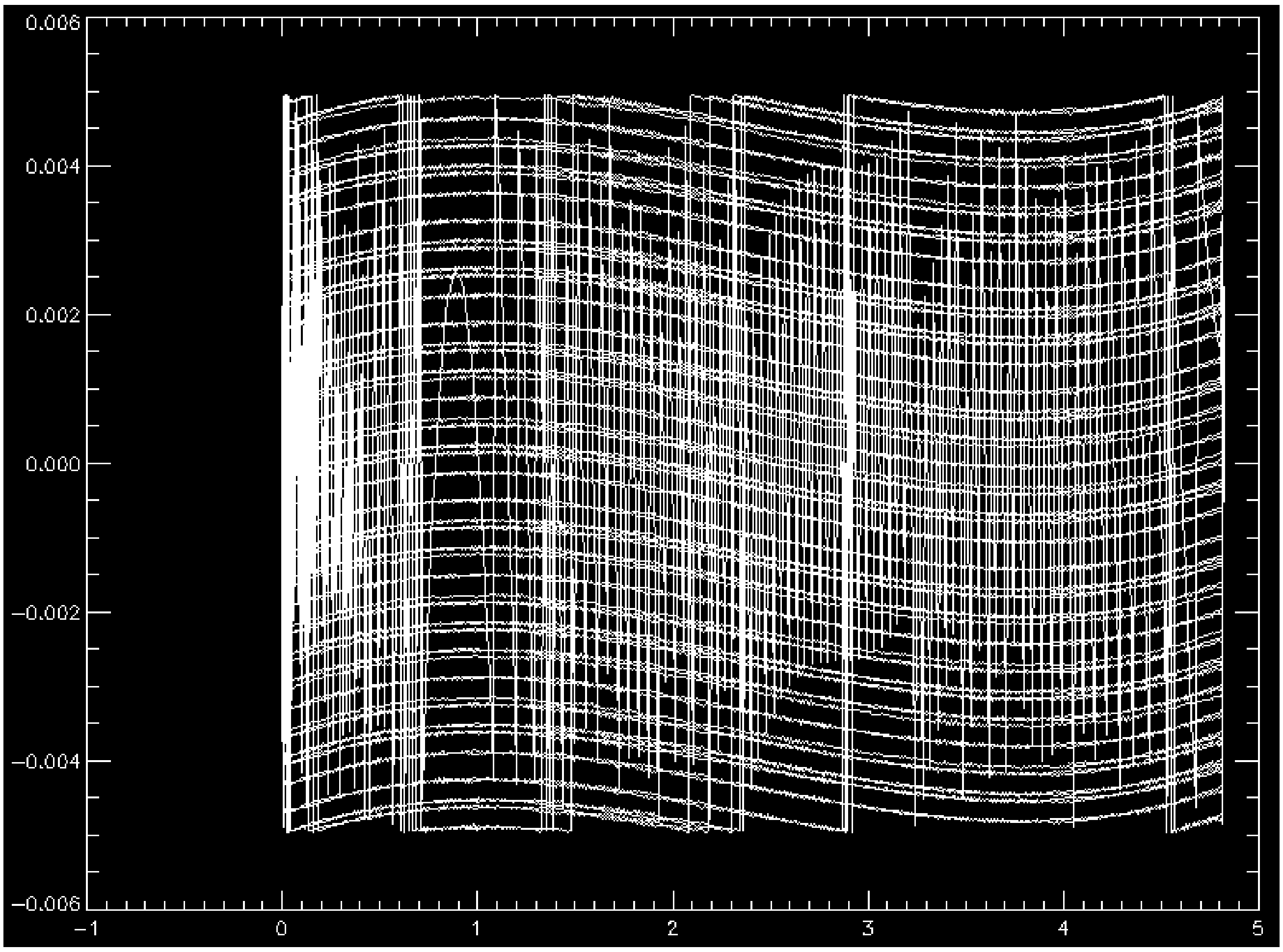
The first step needs to use the pixellated effective ‘x’ position, e.g. pixel_x * step_x + min(x), where pixel_x was rounded (pixel_x = round((x - min(x)) / step_x)). Could simply use an interpolation function, which is easy as both TS tables are strictly monotonic. Do we assume that ‘x’ and ‘y’ time-stamps are equivalent? Yes for now. Later we’ll interpolate into ‘y_ts as’ well.

The problem with this scheme is that it assumes that ‘x’ is monotonic and the scan only visits each x once. This is not the case in general. How to offset ‘x’ to make it ‘visit’ each x only once, or at least seem monotonic? If ‘step_x’ is less than 1/nx, could simply add lindgen() to pixel_x to offset it, and subtract this back after interpolation. But this does not work because the interpolation source vector ‘x’ does not extend that far. But can also extend source ‘x’ in the same way as well … Note may need to scale ‘offset’ by some integer factor if not large enough.

```
pixel_x = round( (x - min(x)) / step_x )                 ; find middle of equi-spaced X pixels
x_eff = float(pixel_x) * step_x + min(x)
offset = lindgen(nxy)                                    ; offset (integer) to make all visit once
x_eff_ts = interpol( x_ts, x+offset, x_eff+offset)       ; effective time-stamp for each pixel, based on 'x'
```

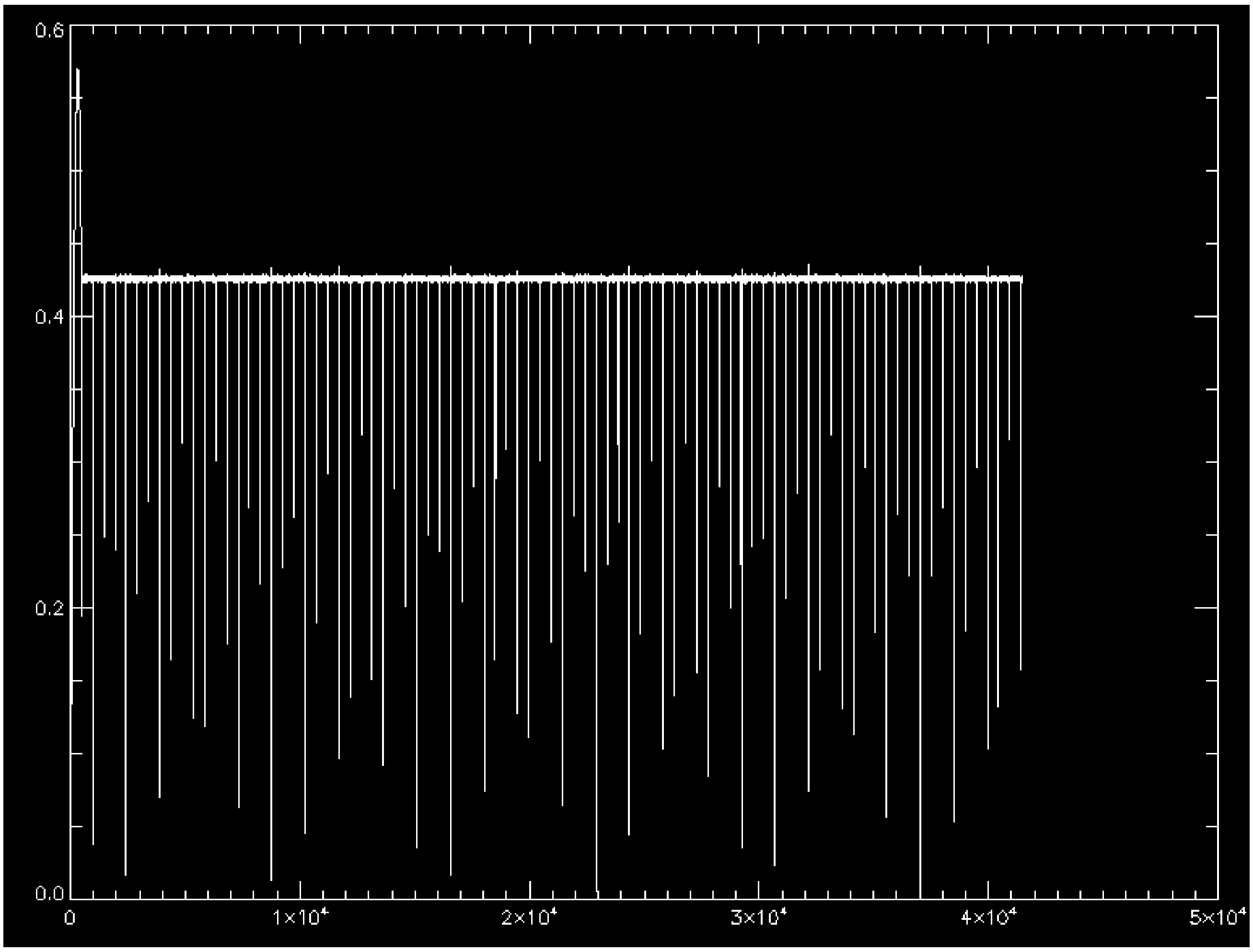


y_eff-y versus y. Note y visits only once

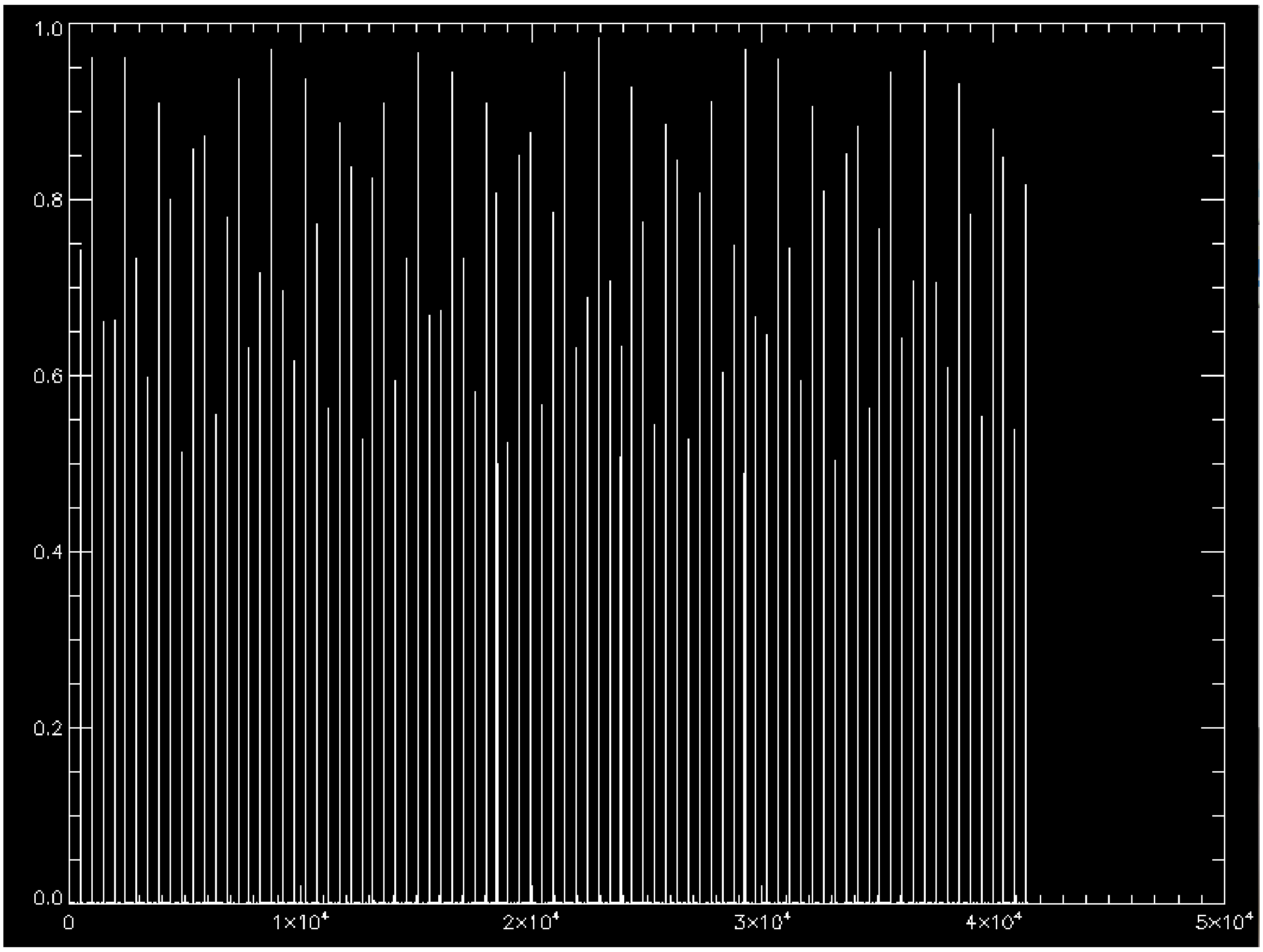


x_eff-x versus x. Note x visits many times without the offset

While ‘y’ visits only once in this single scan, in a XANES map, it too will visit many times versus energy. Need a way to determine the fast axis. Look at the deltas sx = x - shift(x,1) and fix sx[0].



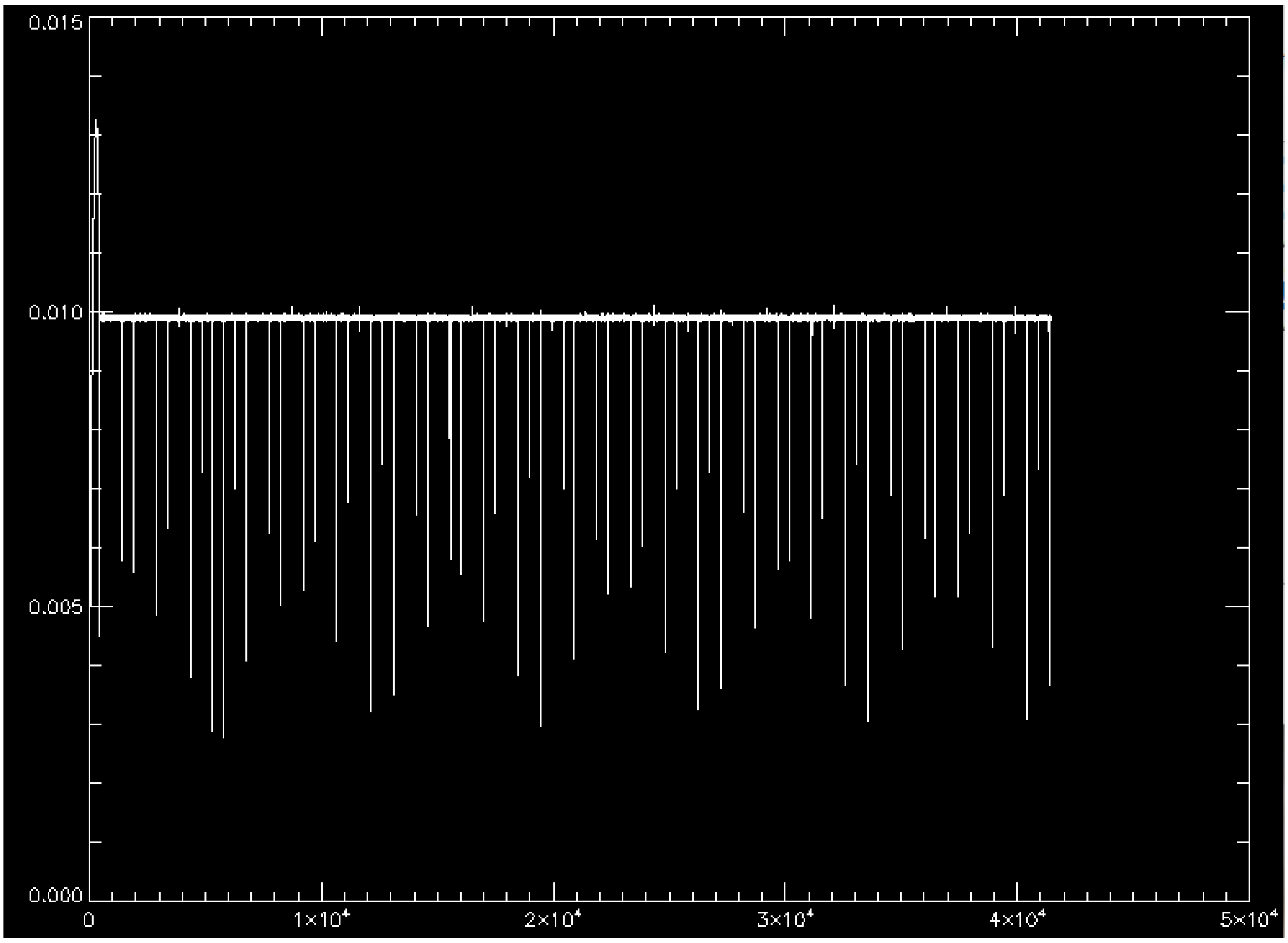
***rx = abs**(sx)/(**max**(sx)-**min**(sx))*



***ry* = *abs*(*sy*)/(*max*(*sy*)-*min*(*sy*))**

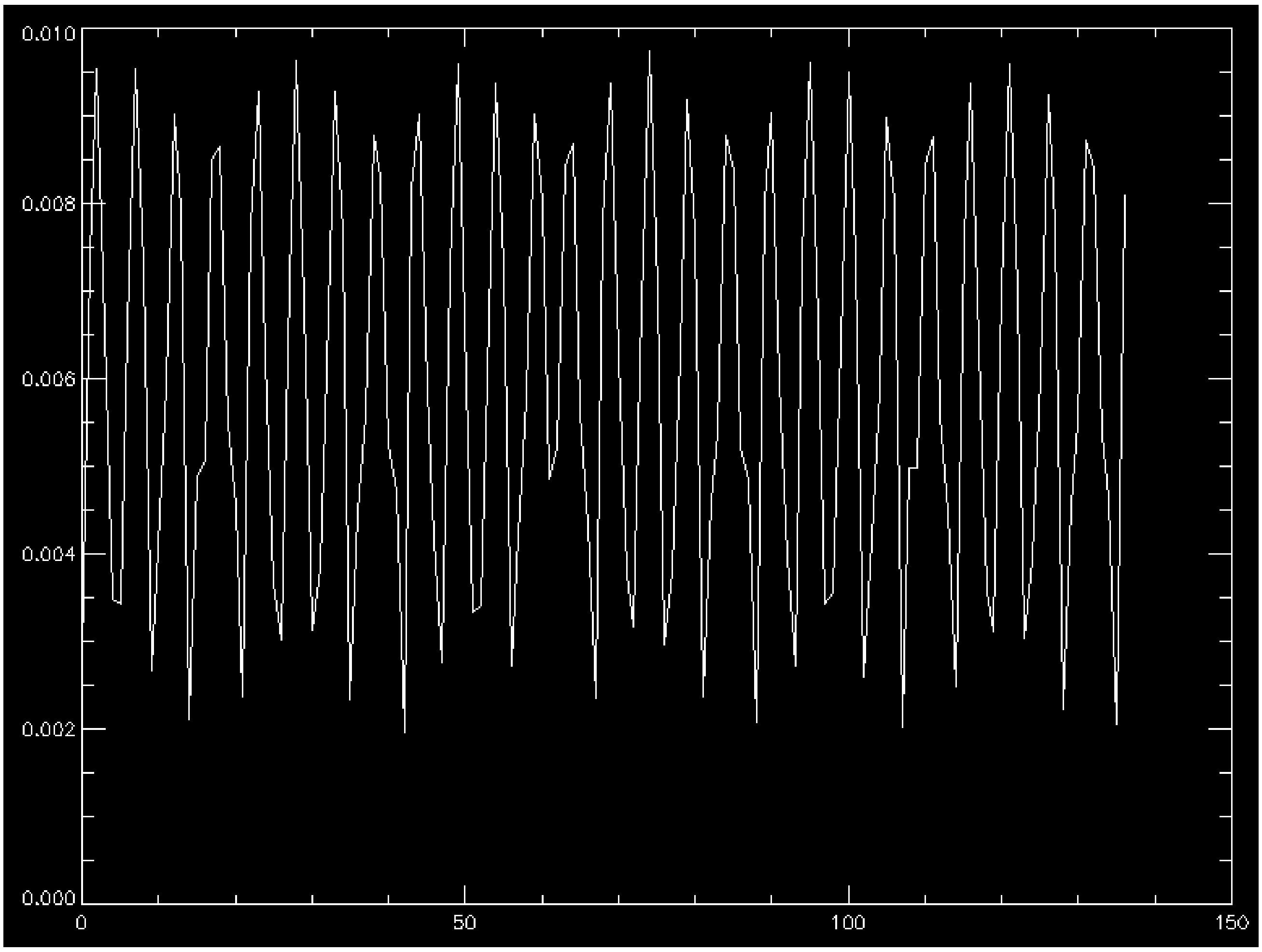
```
qx = where(rx gt 0.2*max(rx))                    ; where significant step changes occur
qy = where(ry gt 0.2*max(ry))
```

The changes in x (sx) are quite consistent, after start-up strangeness, with just a few showing steps less than half the average.



Abs(sx[qx])

Changes in y (sx) are more variable, showing values ranging between 50% above and below the average …



Abs(sy[qy])

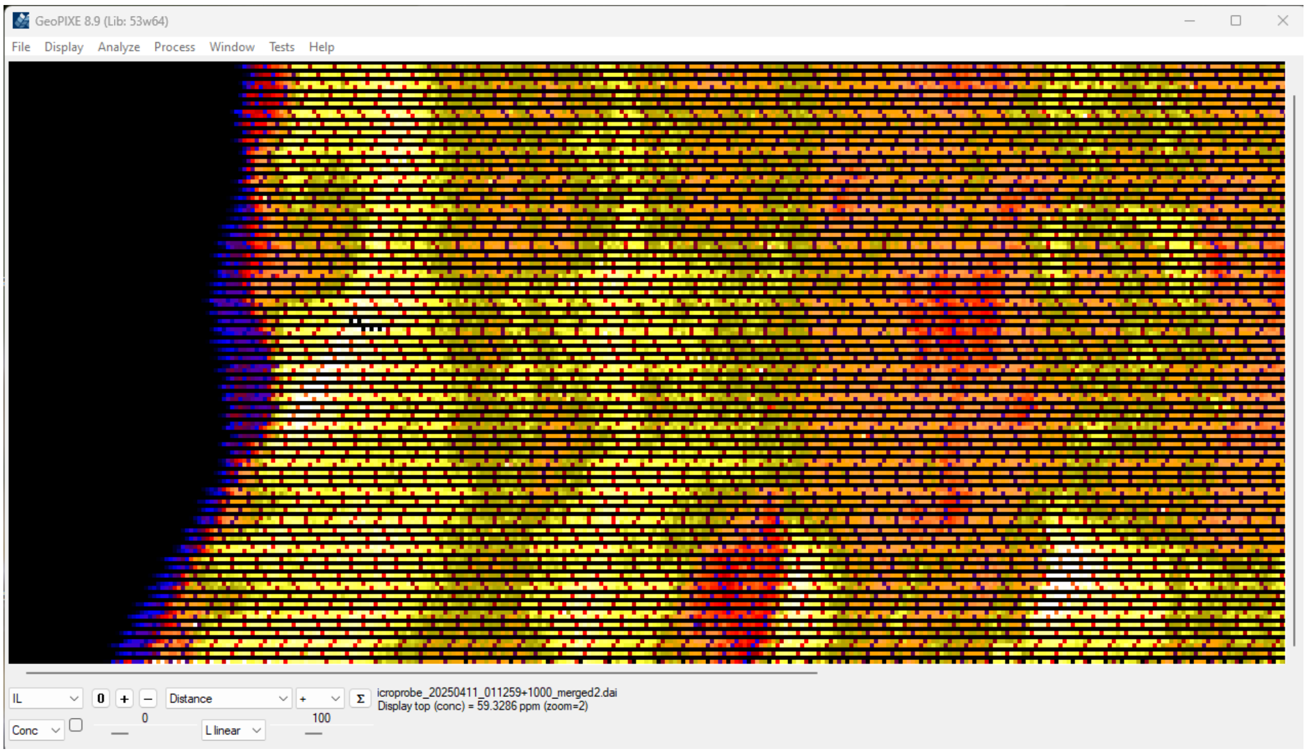
Then determine the average step sizes using …

```
step_x = mean( median( abs(sx[qx]),5))          ; average step sizes
step_y = mean( median( abs(sy[qy]),5))
```

This gives from the header read …

```
AS MEX H5 device: read_as_mex_h5_header ...
Effective X,Y step size = 0.0099037008 0.0056603795
Fast axis = X
Pixel counts X,Y = 488 151
```

These smaller Y steps cause Y gaps in the pixels …

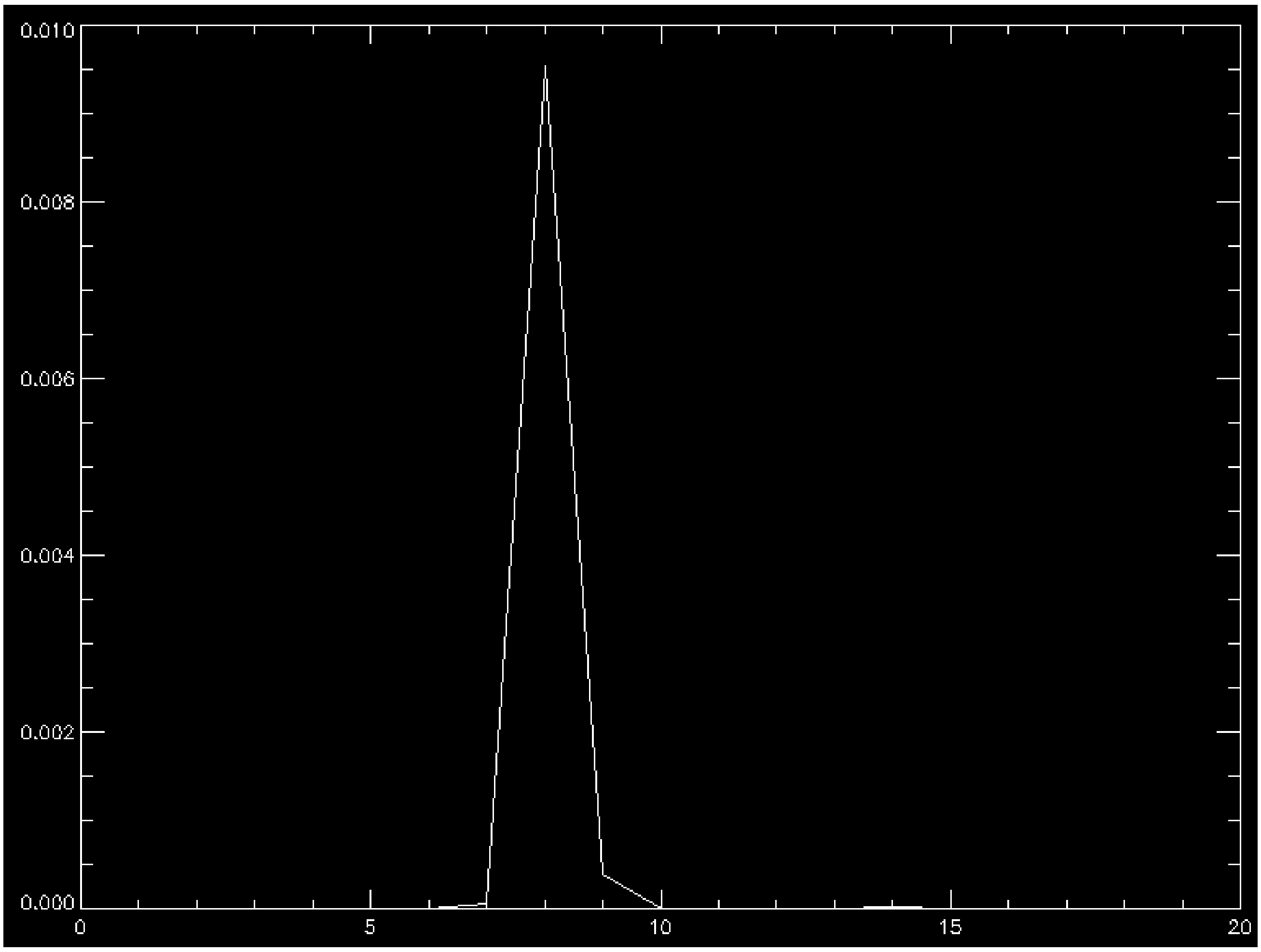


icroprobe_20250411_011259+1000_merged2.dai
Display top (conc) = 59.3286 ppm (zoom=2)

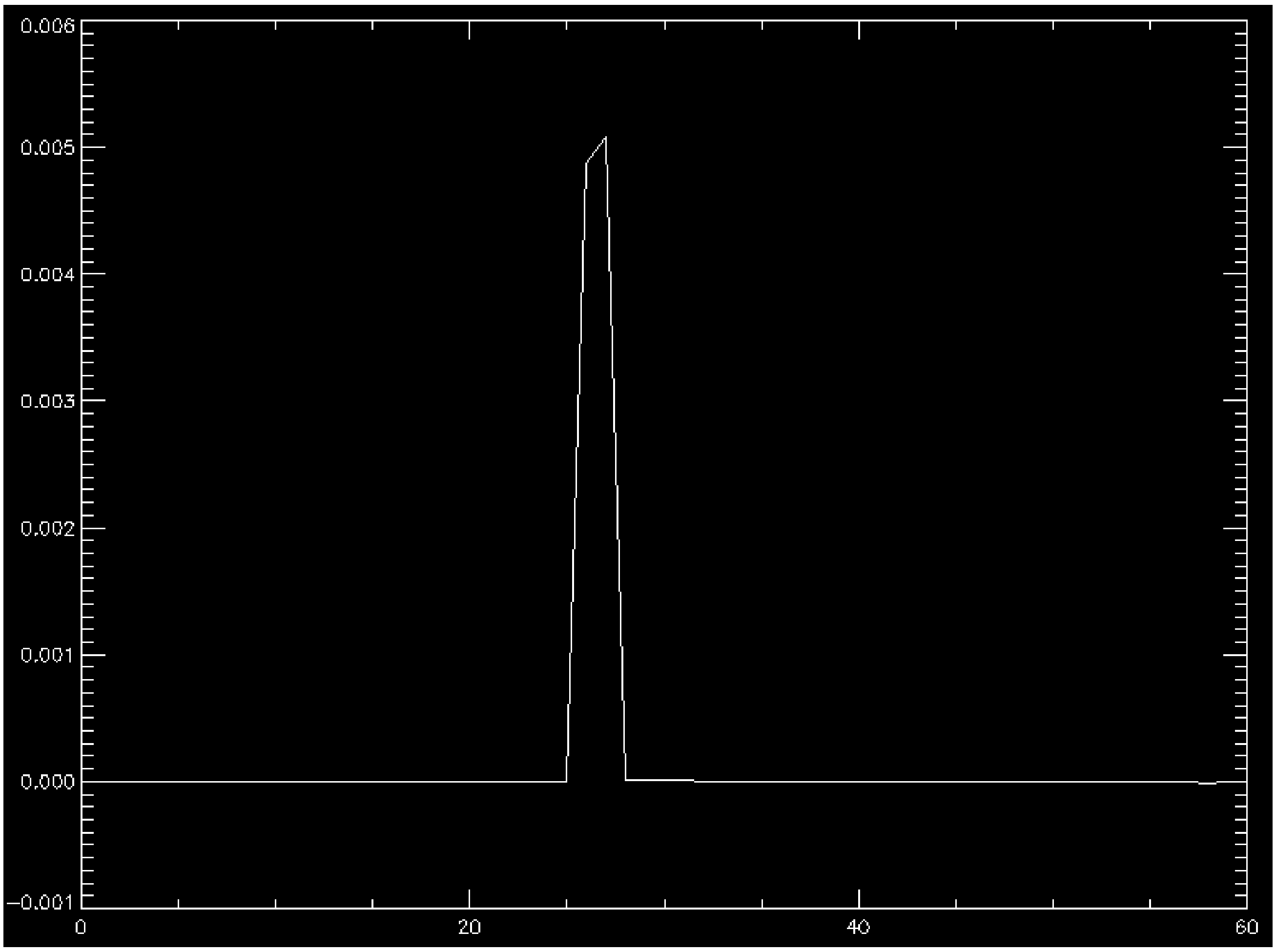
The added pattern in X is impressed by the changes in dwell time (and hence flux), as noted above, which normalize down every ~7th pixel. As noted above, this seemed to be unlikely. Scaling the flux rate (from i0) with the dwell time, to get flux count, has produced this issue. It seems that the real dwell times do not vary this much. The time-stamps are seriously in error.

Y steps in clusters

Looking more closely at the sy steps, we see they can come in clusters, which add up to more like the step_x values … These clusters should be combined to give the total Y steps, as they just represent Y steps spread over multiple time-stamps.



*sy[**960**:**980**], a single Y step*



*sy[**4840**:**4900**], a cluster of consecutive Y steps*

```
print, sy[4860:4875]
-4.0663047e-07 -2.3347720e-07 1.2978889e-08 1.0410485e-06 1.4118259e-06 1.0311048e-06
0.0048707200 0.0050818961 5.5291151e-06 6.1223851e-06 6.8996372e-06 8.4755383e-06
2.9698219e-06 -7.1044904e-07 -1.6863398e-06 -3.5630147e-06
```

This shows some clusters in qy = 4379 + 4380 and 4866 + 4867 (highlighted) …

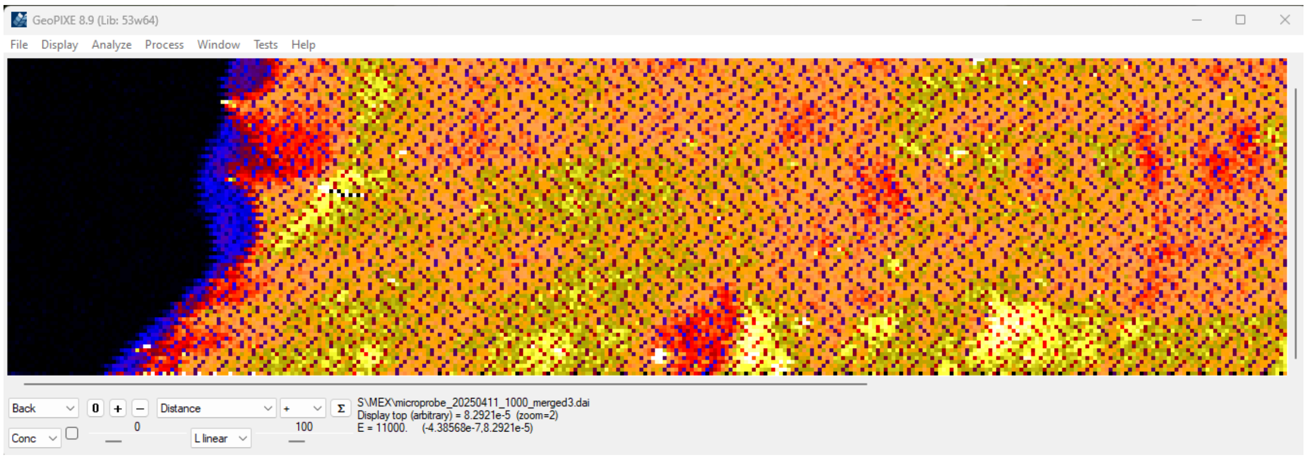
```
print, sy[qy[13:20]]
0.0090191848 0.0079282004 0.0021163984 0.0048707200 0.0050818961 0.0016100056
0.0084999787 0.0086402633
print, qy[13:20]
3892 4379 4380 4866 4867 5353 5354 5841
```

Need to combine pair-wise (or more) these clusters. This code will merge these clusters …

```
        q = where( qy eq (shift( qy,-1) -1), nq)                    ; pairs of sig moves together
        while nq gt 0 do begin
                sy[qy[q]] = sy[qy[q]] + sy[qy[q+1]]                 ; combine pairs
                sy[qy[q+1]] = 0.0                                   ; remove from list
                qy[q+1] = -1
                q = where( (qy eq (shift( qy,-1) -1)) and (qy ge 0) and ((shift( qy,-1) -1) ge 0), nq)
        endwhile
        q = where(qy ge 0, nq)
```

step_y = **mean**(**median**(**abs**(sy[qy[q]]),**5**))

This has fixed the missing Y lines issue. We still have the "7th pixel low blues" caused by the errors in 'x_ts' time-stamps.



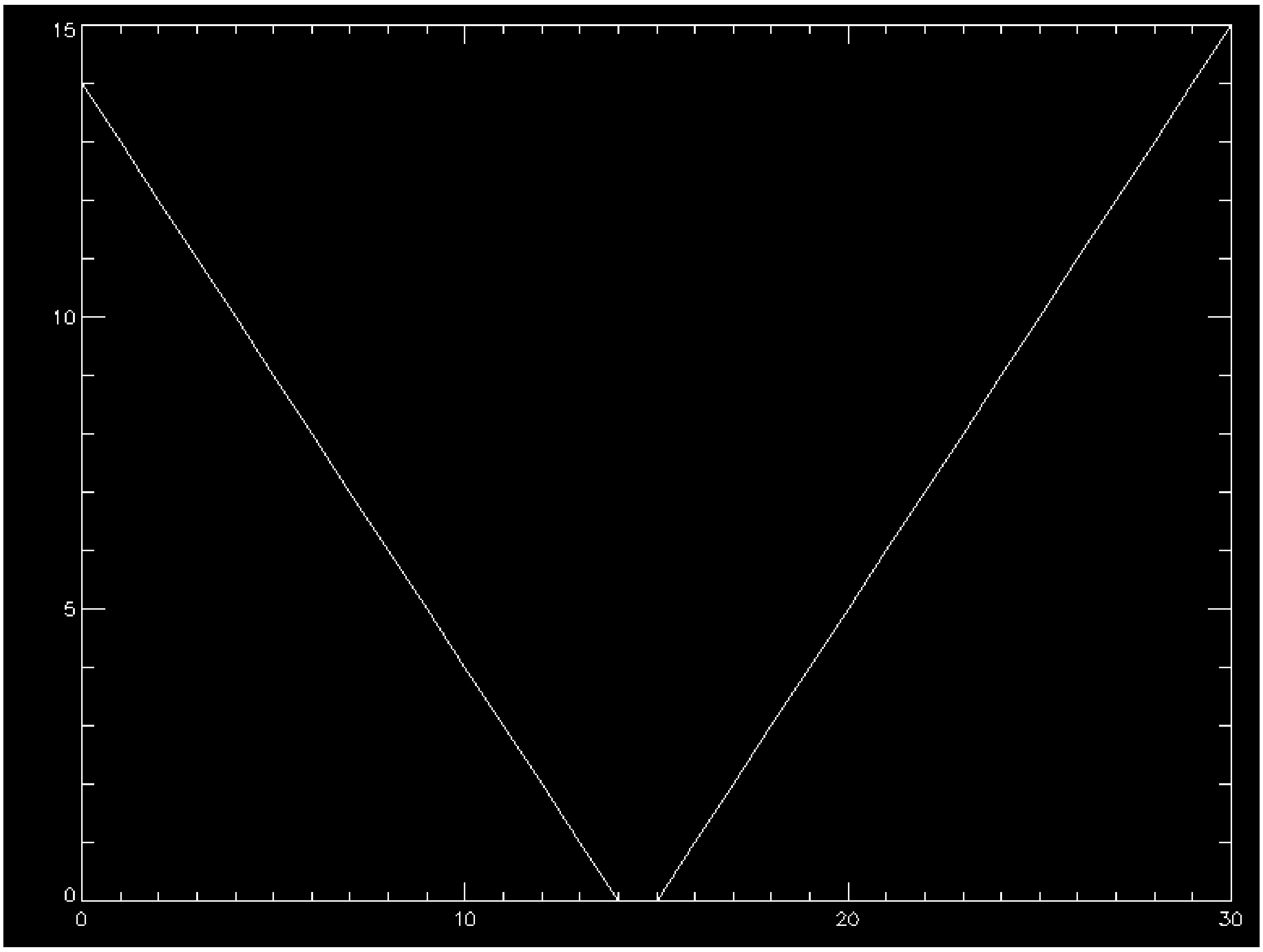
S\MEX\microprobe_20250411_1000_merged3.dai
Display top (arbitrary) = 8.2921e-5 (zoom=2)
E = 11000. (-4.38568e-7,8.2921e-5)

Spectra versus X steps

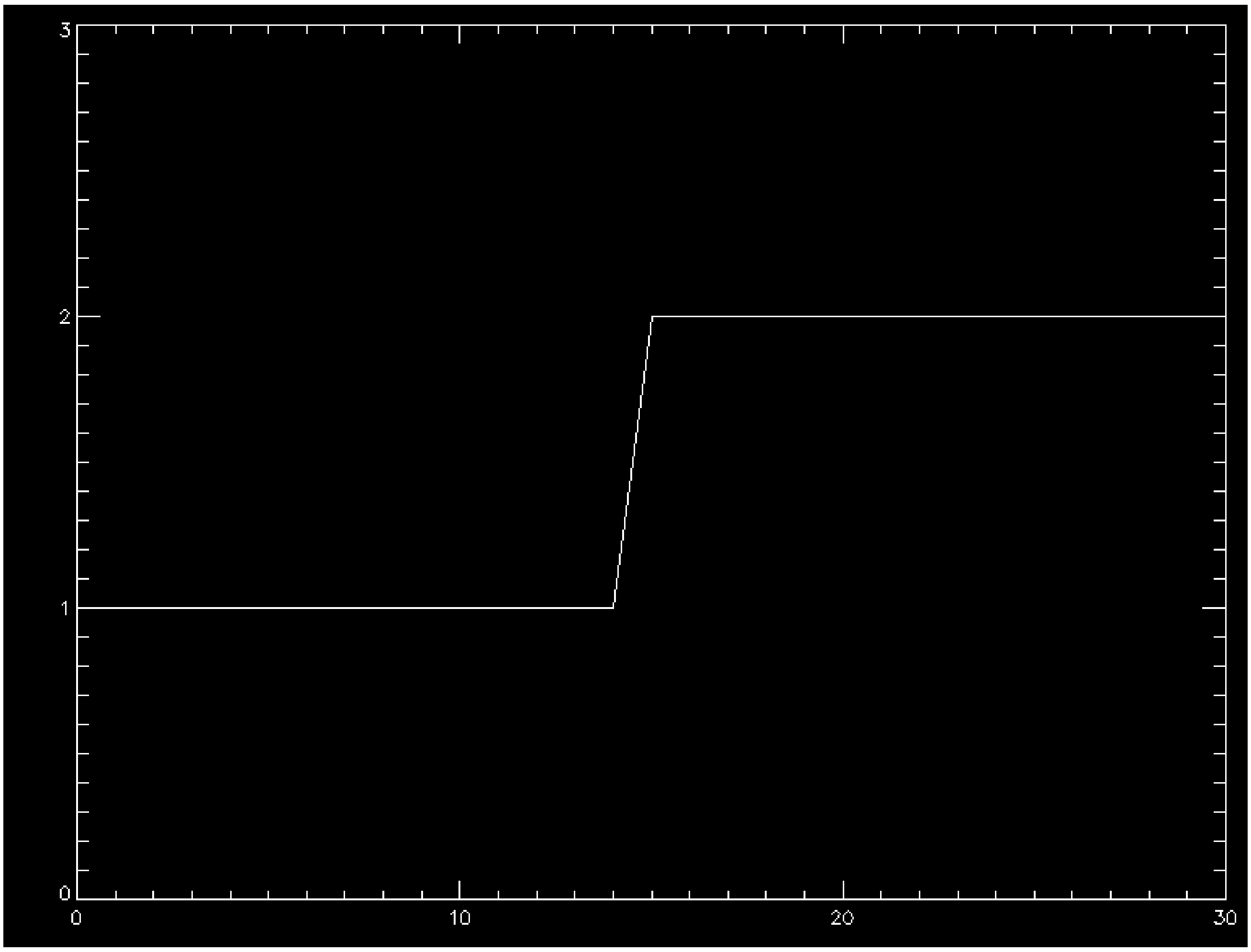
Have assumed that 'x_ts' monitors the steps integrated into each spectrum. But is this the case? The above result would suggest otherwise. How do the individual spectra (total) compare with the 'x_ts' steps (dwell)?

Spectra

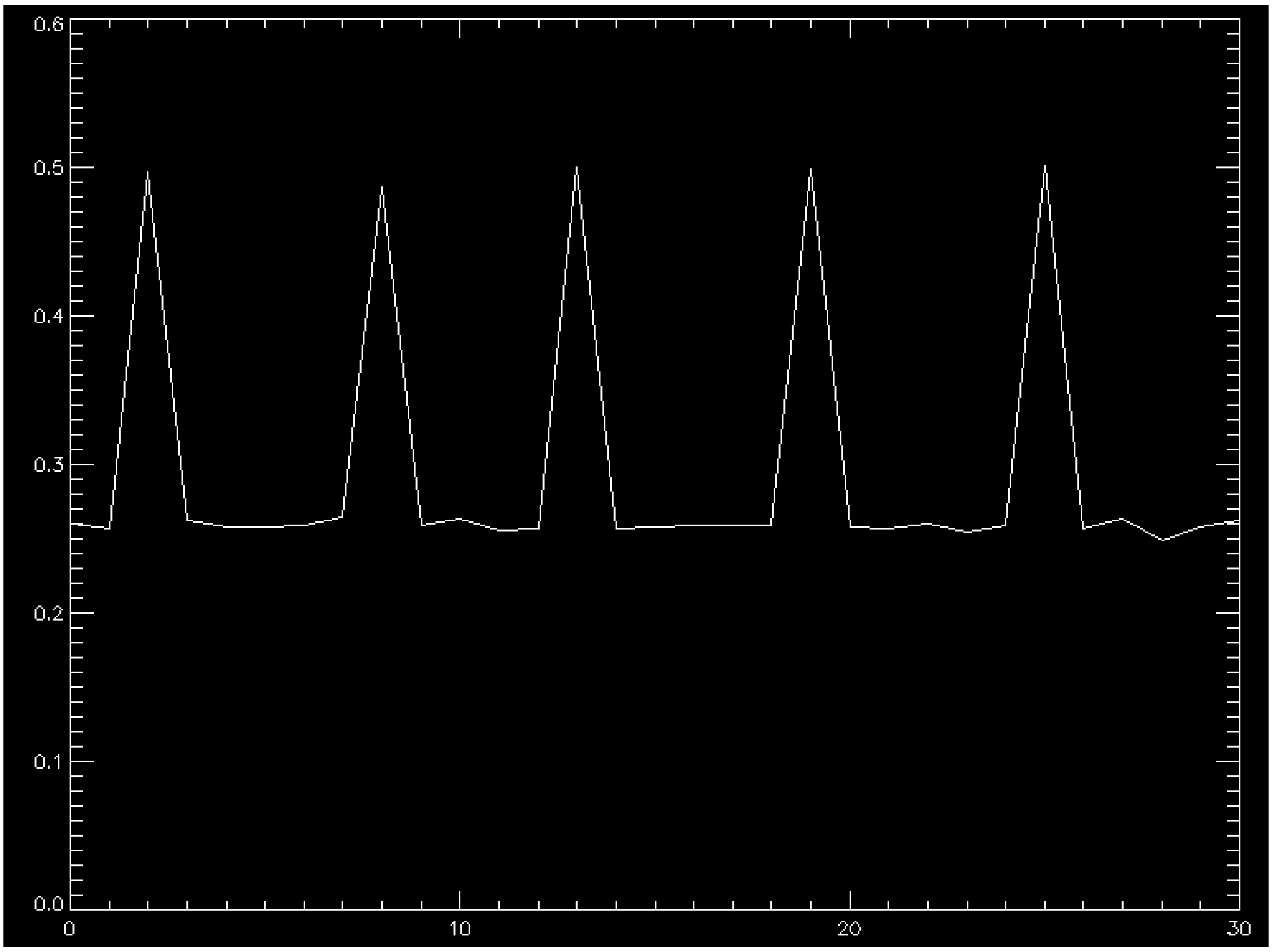
Looking at the totals of spectra (detector channel 0) for 31 steps around the reversal in X and step in Y (sequence [953:983]) (and away from the weirdness near the start of the scan) …



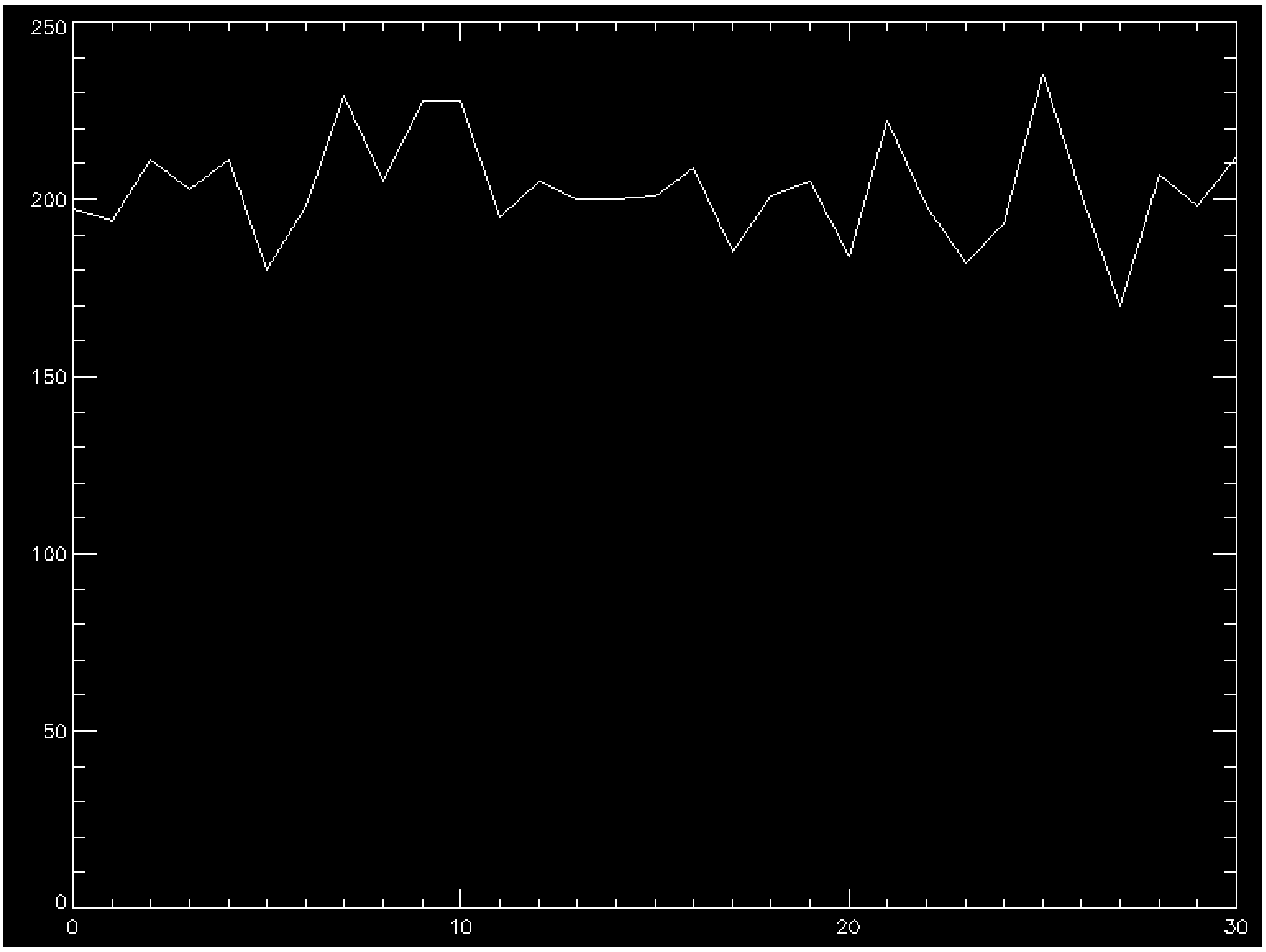
Pixel_x[953:983]



Pixel_y[953:983]

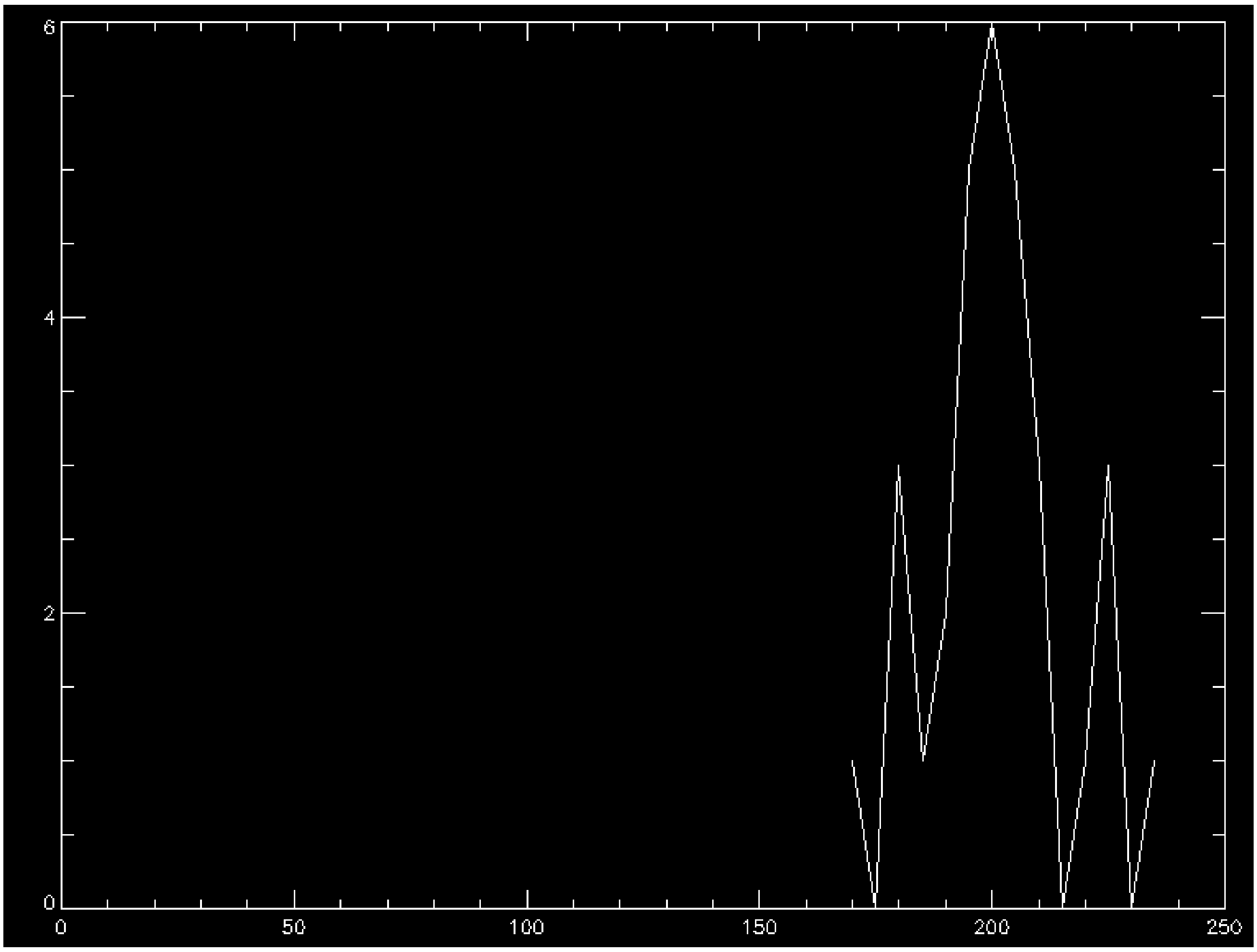


*Dwell = reference_ts[**953**:**983**] - reference_ts[**952**:**982**]   (relative s.d. 30%)*



Totals for spectra, for index 953:983   (relative s.d. of 7.4%)

Clearly, there is no evidence for a factor of 2 (~0.25 to 0.5 dwell) change in spectra totals. These spectra seem to be on a fairly even time step (s.d. 7.4%, inconsistent with the dwells inferred from 'x_ts' (which becomes 'reference_ts'). The standard deviation of these totals is 15 (relative s.d. of 7.4%), which matches root-N (14.2) given by Poisson statistics and suggests even sampling to within a couple % variation at most. In contrast, the relative s.d. of the time stamps is 30%.

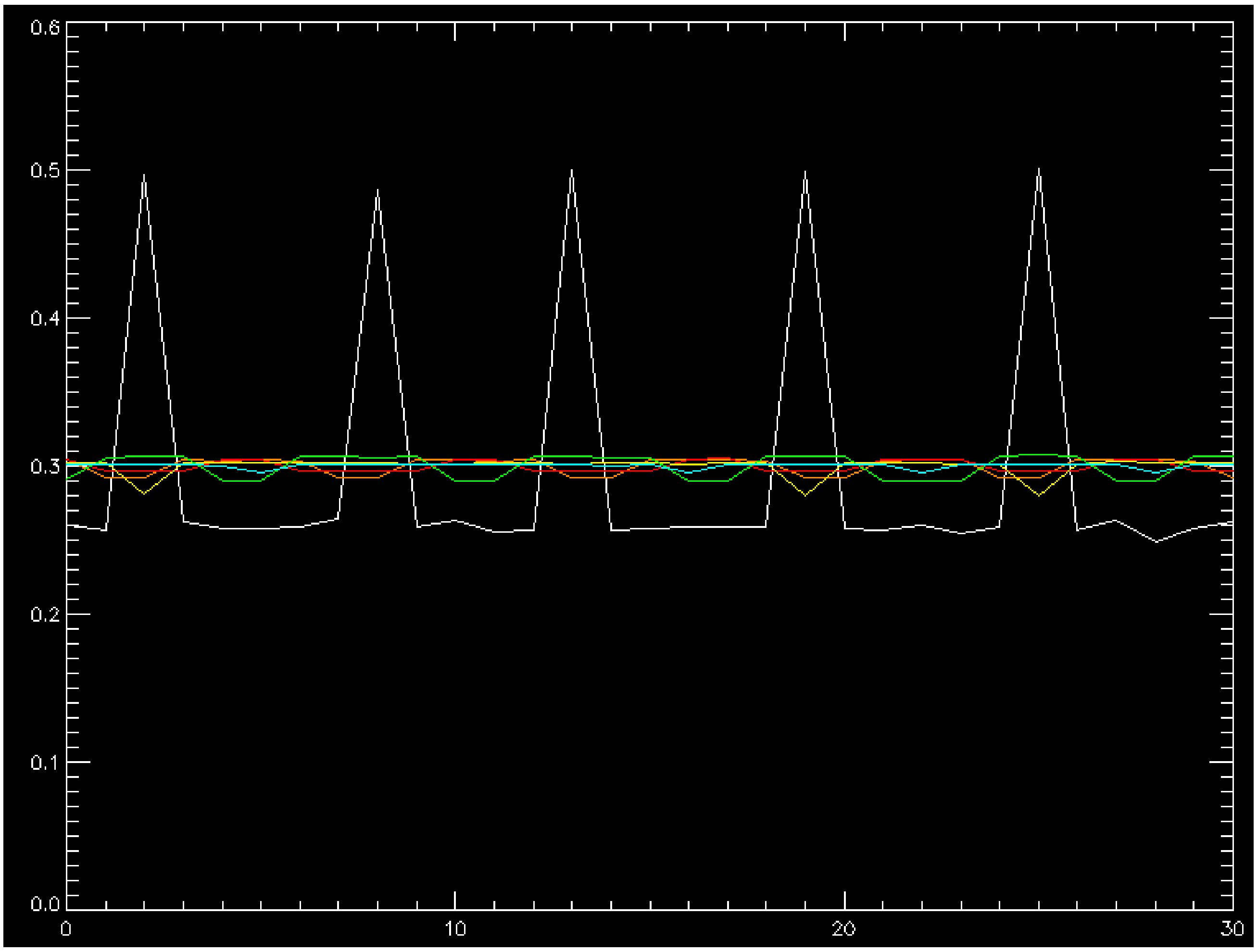


Histogram of totals across 953:983

Can only conclude that the spectra are NOT sampled on a time base given by the ‘x_ts’ time stamp. Their time base is much more even, probably to better then a couple %.

Options to deal with errors in time-stamps

As we do not appear to have time-stamps for the spectra sequence, we need to use the ‘x_ts’ (or possibly ‘i0_ts’). Hence, perhaps the approach is to provide the ability to smooth out errors in the time-stamps to better reflect the real spectra time steps. Need a smoothing window of about 20-50 perhaps. A width of 30 gives a relative s.d. of 12.8%, which is still not small enough. A width of 50 gives a relative s.d. of 5.3%, which seems better. Note that we use an average (smooth) rather than a median to preserve the total time passed.

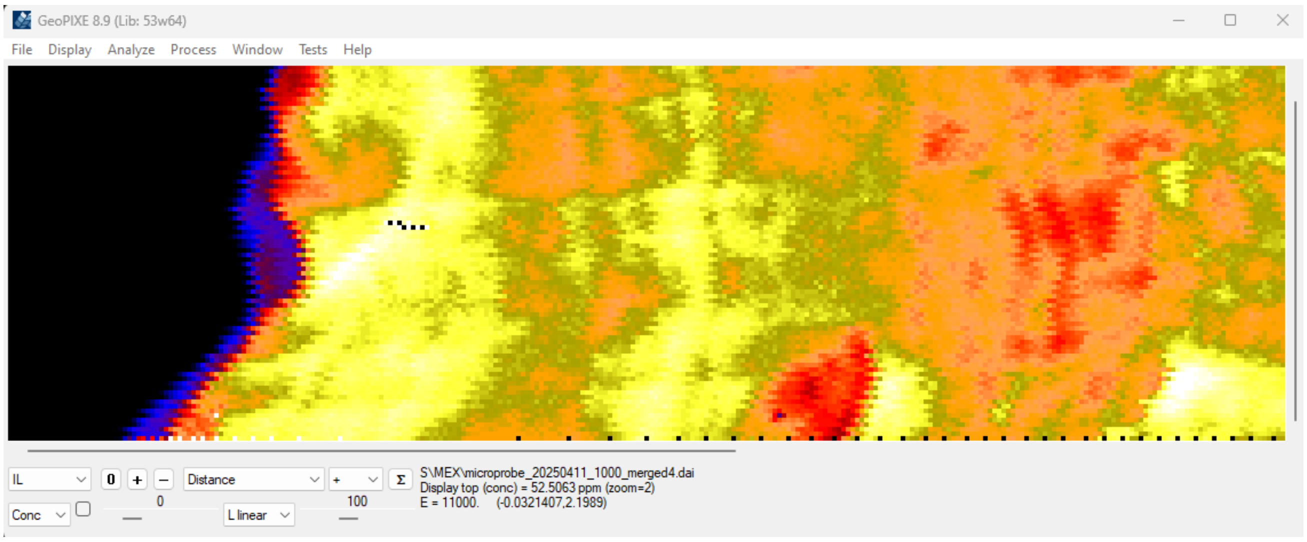


Smooth time-steps in 'i0_ts' (boxcar) of width 10 (yellow), 15 (green), 20 (orange), 30 (red), 50 (l.blue)

Add a device object control on the "Device" tab (e.g. *Sort EVT* window or *Import Spectra* pop-up) to enable the smoothing of 'x_ts'.

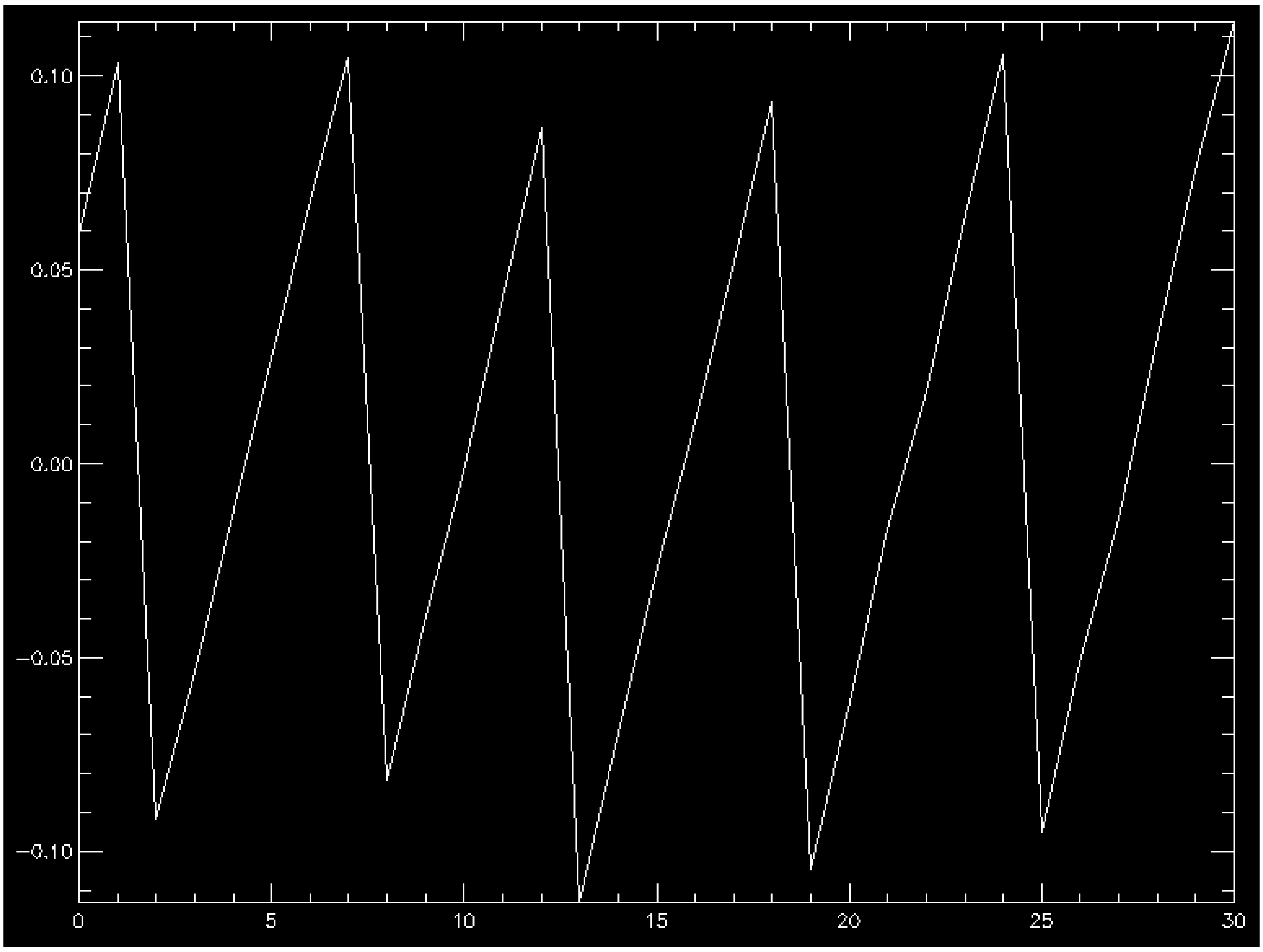
Smoothed time-stamps

This seems to work better, at least for this dataset … However, it still shows a delay, which shows as row "jaggies" as seen before.

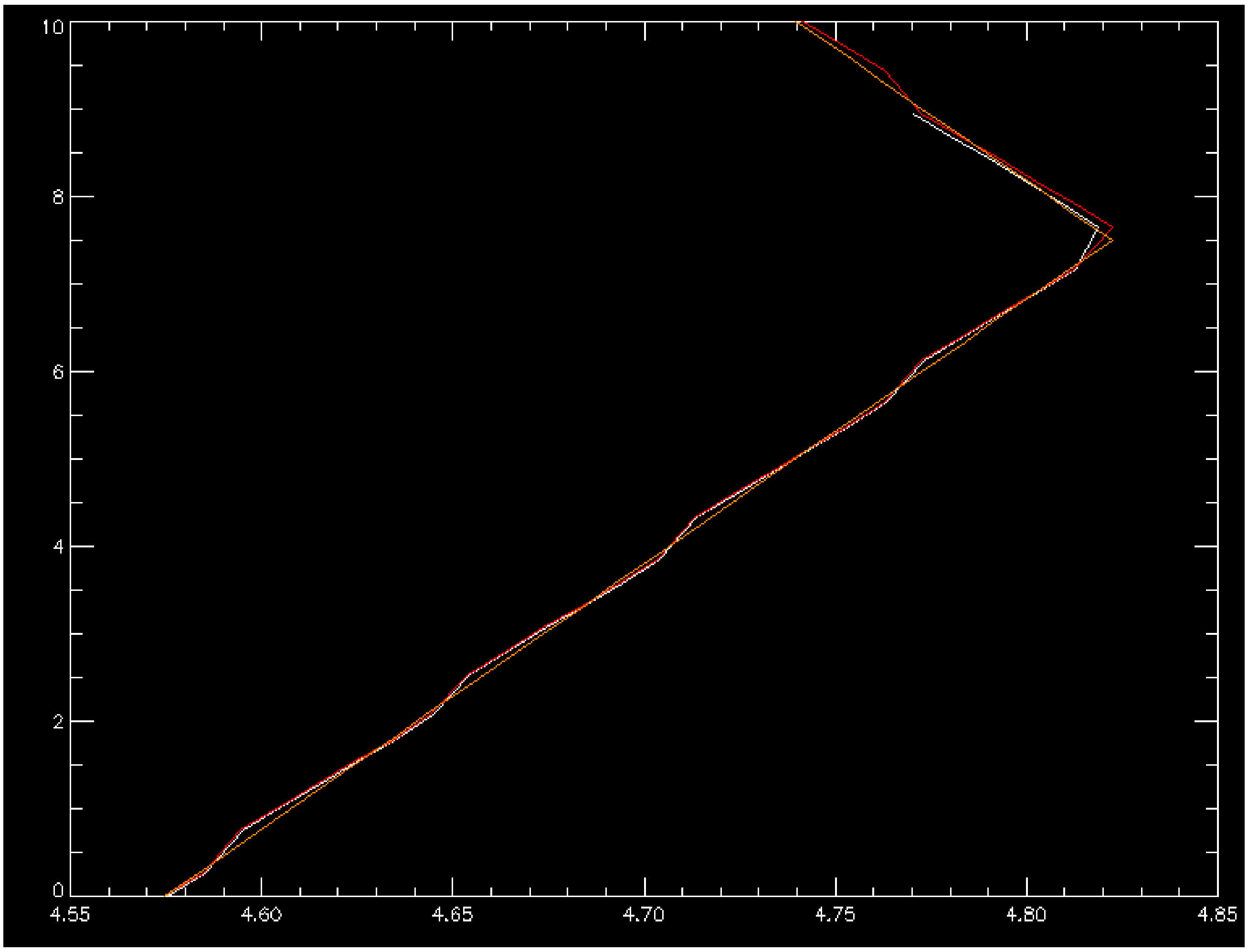


S\MEX\microprobe_20250411_1000_merged4.dai
Display top (conc) = 52.5063 ppm (zoom=2)
E = 11000.   (-0.0321407,2.1989)

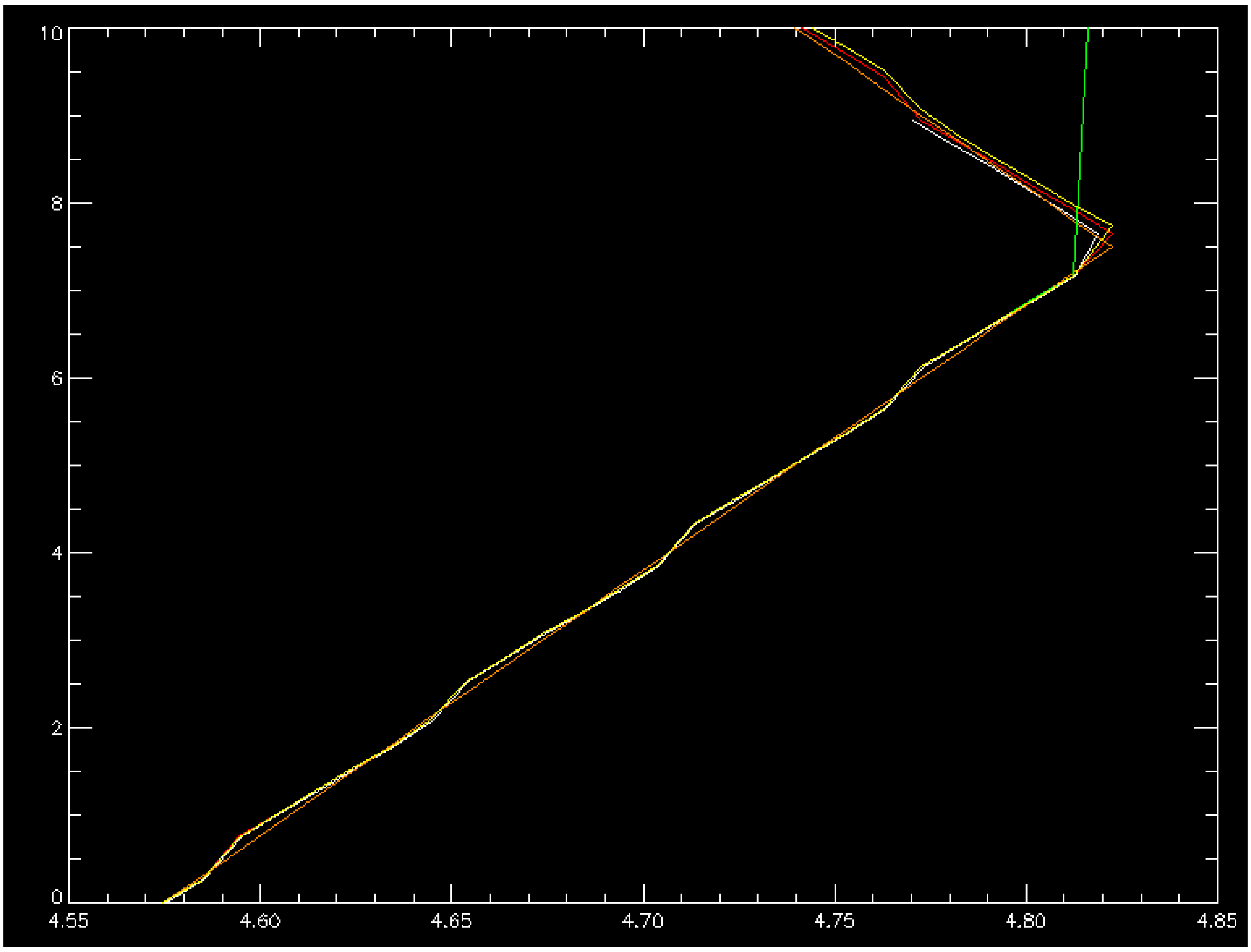
No, analysis of the smoothing suggests it does not introduce the delay. Seems the jaggies are inherent in the data.



Difference between smoothed and reference_ts [953:983] just shows the 7 step repeat errors

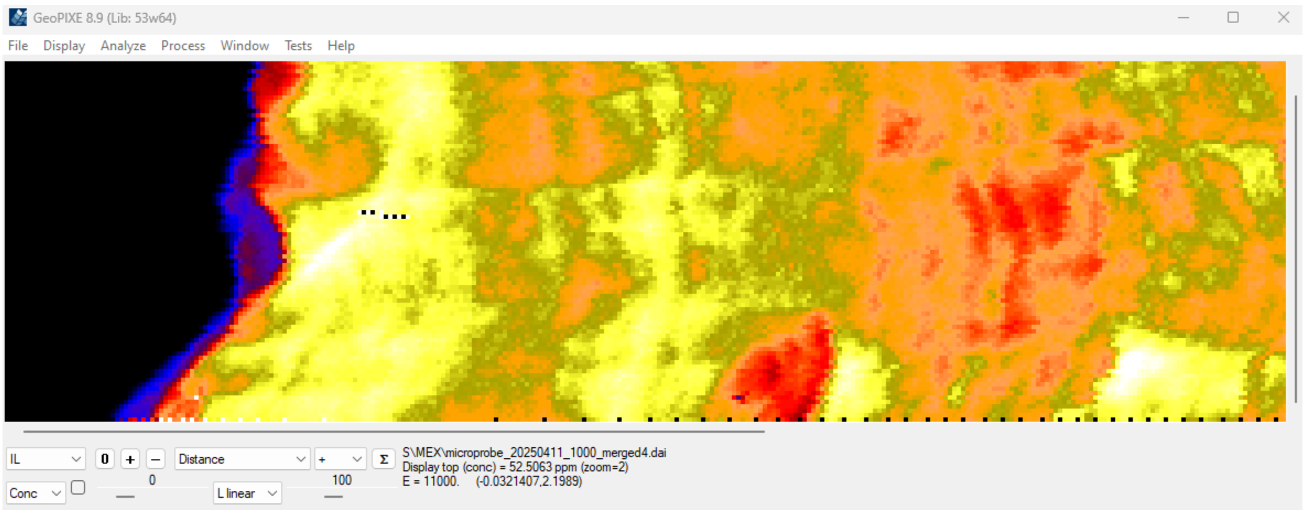


Orange: smoothed reference_ts – reference_ts[1430] versus x_eff; Red: reference_ts – reference_ts[1430] versus x_eff; White: x_ts – x_ts[1430] versus x, over sequence range [1430:1480]



*Green: offset near critical scale (nx/(max(x)-min(x)) about 1/100 times Red; Yellow: offset * 2. (1/50 times Red). Settle on 4x critical.*

Shifting odd rows -2 does better …



Distance

+

Conc

L linear

0

100

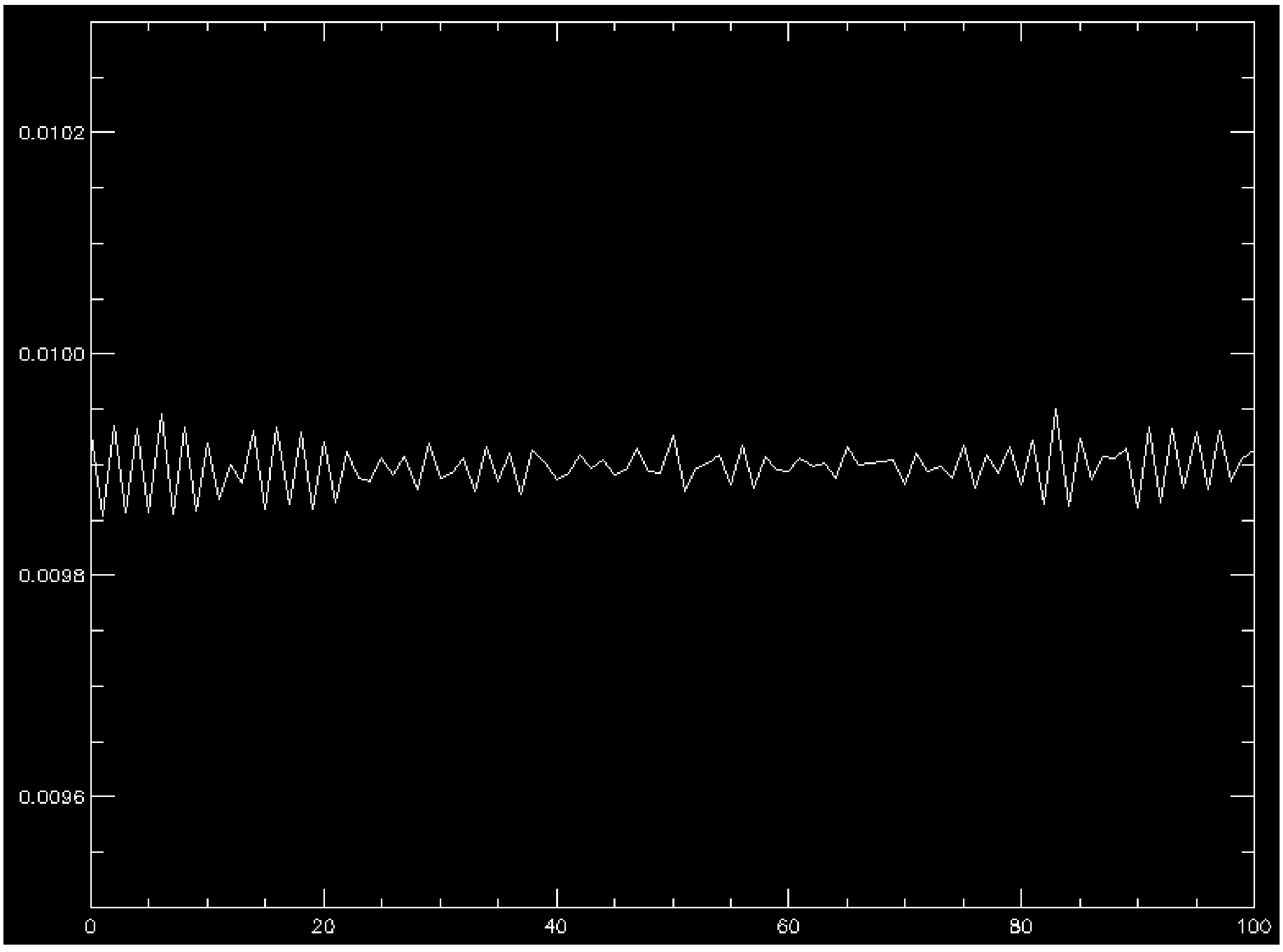
S\MEX\microprobe_20250411_1000_merged4.dai
Display top (conc) = 52.5063 ppm (zoom=2)
E = 11000.   (-0.0321407,2.1989)

Indeed, the jaggies represent a delay, which suggests that the x_ts are not properly synchronized to the writing of the spectral data. To make a 2-pixel jaggie suggests delays of the order of 2x the dwell (~0.6 s), which is comparable to the observed jitter in the x_ts at ~0.3 s (jumps from ~0.2 to ~0.5 s every 7th step). This suggests that the time-stamps are not properly synchronized to the signals they "stamp" as well as having huge jitter, suggestive of software driven logging, where processes are interrupted by other operations for some significant (and often random) times. These errors are very regular, so it is possible that it has been done in hardware (e.g. FPGA) with poor accounting for lost clock cycles.

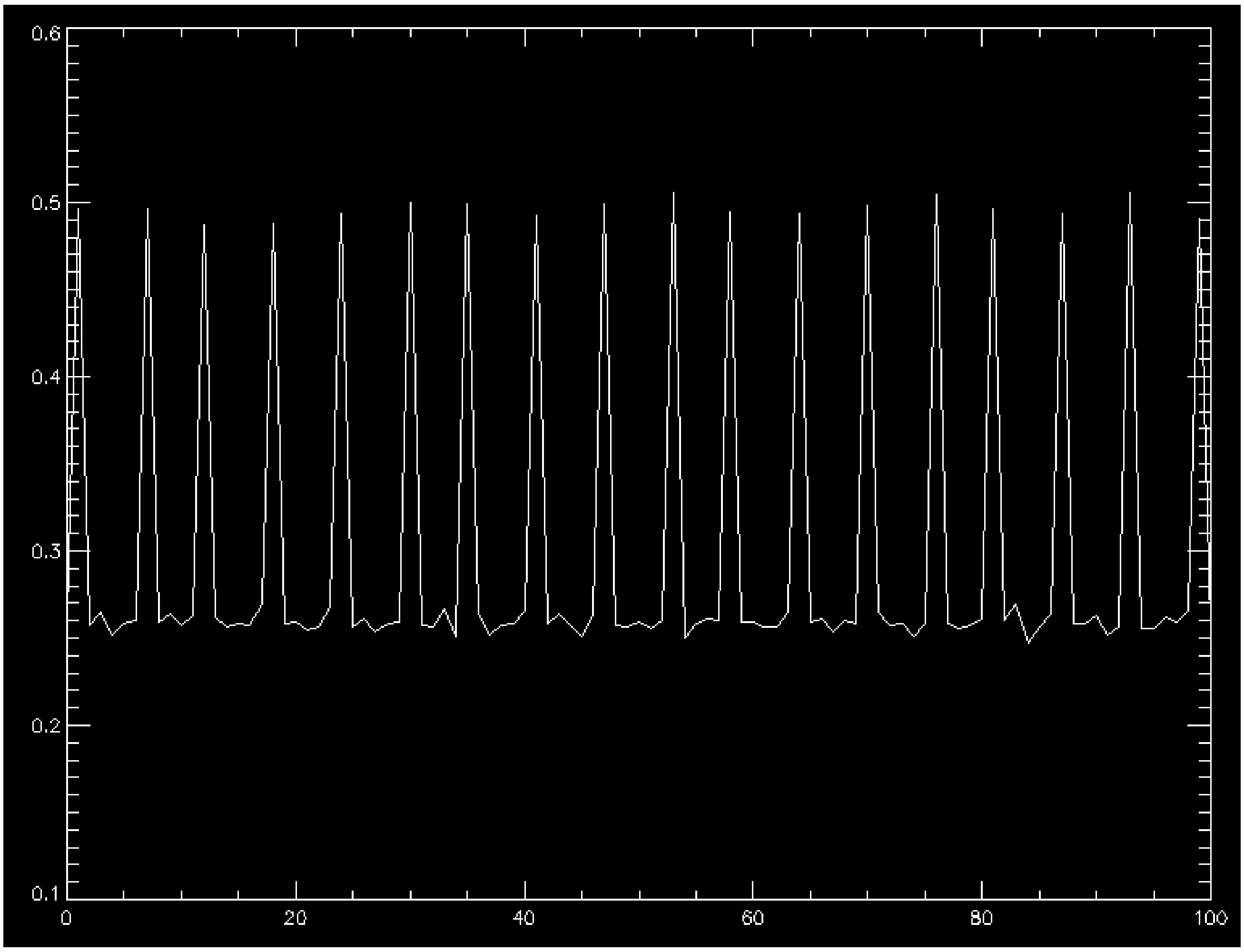
What is needed

There are a number of issues in the data that these results highlight:

1. The time-stamps are not properly synchronized to the data they are supposed to track. There are two aspects to this:
    a. There is a consistent delay between the time-stamps and the signals/data they track – a systematic error. This shows as “jaggies” as the X axis reciprocates back and forth.
    b. There are regular jumps, or larger errors that repeat on some strange period – (pseudo-)random errors. This makes the dwells extracted from ‘x_ts’ have values that jump from the common ~0.25s to ~0.5 every 7th sample. This totally corrupts the dwell measure.
    c. These ‘x_ts’ time stamp differences are totally inconsistent with the very regular steps in ‘x’ (see below).
2. There are no time-stamps for the spectra:
    a. Clearly, spectra are sampled more uniformly in time that ‘x_ts’ suggests. It should have its own time stamp series (which accounts for any delays between acquisition and writing to disk).
    b. The spectra are more consistent with the very uniform changes in ‘x’ (after the strange start-up period of varying acceleration has passed).



Steps in ‘x’ [1000:1100] are very uniform, qualitatively in agreement with the spectra



Steps in 'x_ts' [1000:1100] are crazy, suggesting that 'x_ts' are rubbish (not tied to 'x' at all)

What is needed is tighter logging of time-stamps for each parameter that:

1. Is tightly aligned with the events they log (i.e. generated in hardware).
2. Corrects for any delays calculated between the time as sampled and the events they log (time stamps need to be "pre-dated" appropriately).

For fast axes, such as 'x' in this case, but also 'i0' at least, the time-stamps need to be done in hardware. Software delays in most computers will incur systematic and random delays, depending on the OS. Realtime OS do better, but are still not perfect. FPGA "software" works much better, and detailed accounting for lost "ticks" can be done to account for delays and rise-time, etc. These can be tuned to avoid "jaggies" even at 10's µs dwell per pixel (e.g. as in Maia).

Slow axes, such as 'i0' and energy/DCM angle, can be sampled in software as they are values that change slowly, so errors in sampling are not too serious. Stage 'y' would still be a problem in software even for fast X scanning.

Overview of pipeline

The two main methods for access to the data are (i) reading "header" metadata to extract parameters like the pixel size of an image (nx,ny), energy, sample name, etc. and also pull out data for the stage encoders, dwell (exposure time) as a function of sequence number, and (ii) reading the HDF file to get spectral data and associating that with pixel address x,y to form vectors of photon events (E,x,y, …).

The methods in the device objects generally include (i) a few that handle "options", (ii) 'get_header_info', which calls the 'read_as_mex_h5_header' routine, (iii) 'update_header_info', which copies the read header info into the 'self' structure of the object (can avoid unnecessary re-reading of long header files), (iv) 'flux_scan', which is usually used to look for the PV names and settings for flux (it also calls 'read_as_mex_h5_header'), (v) 'read_setup', setup photon event vectors for each spectral data read from the HDF file, (vi) 'read_buffer' read next buffer of data from the HDF file, (vii) a few that handle "import" approaches and (viii) 'init', which is the object initialize method, which is only called when a new instance of the object is created.

"Option" GUI elements

"Options" are GUI elements that can appear across GeoPIXE (e.g. "device" parameters in the *Import spectra* and *Sort EVT* windows) for control of internal "device" parameters. These are not really used at this time for this AS MEX H5 device object. They are place holders for later features, and currently just save a default 'version' value.

Import spectra

"Import" methods provide parameters to use for (i) the *Import Spectra* call (Menu: "Import→Spectra" in the *spectrum display* window), and (ii) custom local spectral data reads. In this device, we are only using the first, which is handled via the *Import Spectra* approach (see routines "import_select" and "spectrum_load" called from *spectrum display* window), which scans all raw spectra/event data in Xspress3 NXS files to accumulate spectra for 'E' and the projection of all events onto X,Y axes. The definition for this is …

```
opt_39 = define(/import)                                ; MEX new HDF5 file read as a list-mode
        opt_39.name =        'as_mex_h5_evt'            ; unique name of import
        opt_39.title =       'Extract E,X,Y from MEX map HDF5 file as list-mode'
        opt_39.in_ext =      '.hdf5'                    ; input file extension
        opt_39.request =     'Select MEX HDF file to scan for all spectra, X,Y'
        opt_39.preview =     0                          ; allow spectrum preview
        opt_39.raw =         1                          ; flags use of separate Raw data path
        opt_39.multifile =   0                          ; denotes multi-file data series
        opt_39.separate =    ''                         ; char between file and run #
        opt_39.spec_evt =    1                          ; uses call to 'spec_evt' to extract events
        opt_39.use_IC =      1                          ; pop-up the 'flux_select' PV selection panel
        opt_39.IC_mode =     1                          ; default to using PV for IC
```

This tells GeoPIXE that HDF data is found in one file (multifile=0) and a file extension of "hdf5" (in_ext=".hdf5"). 'Title' is for the *Import Spectra* popup title, 'request' is a title for the *File-Requester* popup, 'raw' flags a separate path for raw data and analyzed data, 'use_IC' flags using a popup 'flux_select' to choose the PV to use for flux values, 'IC_mode'=1 means a default mode of using a PV selection for flux with gain settings.

Note: Later it may be necessary to spread data over multiple files (e.g. for XCANES maps). Then "multifile" will be set to 1. The use of the sequence number will be important the.

Header read ("read_as_mex_h5_header.pro")

Reading "header" metadata is needed to extract parameters like the effective pixel size of an image (nx,ny), energy, sample name, etc. and also pull out data for dwell and flux as a function of sequence number.

It reads 'x' and 'y' data per sequence number. This gives pixel position as a function of sequence number. From these we calculate pixel address "pixel_x, pixel_y" versus sequence number using these lines.

```
    sx = x - shift(x,1)                                    ; step change in x
    sx[0] = sx[1]                                          ; fix wrap
    sy = y - shift(y,1)                                    ; step change in y
    sy[0] = sy[1]                                          ; fix wrap
    rx = abs(sx)/(max(sx)-min(sx))                         ; relative step changes
    ry = abs(sy)/(max(sy)-min(sy))
    qx = where(rx gt 0.1*max(rx), nqx)                     ; where significant step changes occur
    qy = where(ry gt 0.1*max(ry), nqy)

    step_x = mean( median( abs(sx[qx]),5))                 ; average step sizes
    step_y = mean( median( abs(sy[qy]),5))
```

and then if X is the fast axis …

```
    print,'   Fast axis = X'
    pixel_x = round( (x - min(x)) / step_x )               ; find middle of equi-spaced X pixels
    nx = max(pixel_x) + 1

;   Think about effective time-stamp for pixellated x, as a reference TS for
;   interpolating corresponding values in the other TS tables (e.g. y_ts, i0_ts).

    x_eff = float(pixel_x) * step_x + min(x)               ; effective 'x' positions in pixels
    nabs_x = x_eff + abs_x[0] - x[0]                       ; effective absolute 'x'

    scale = 4*(max(x) - min(x))/nx
    offset = scale*lindgen(nxy)                            ; offset (integer) to make all visit once

    if nx le (max(x) - min(x)) then begin
        offset = offset * (round((max(x) - min(x))/nx) > 1)
    endif

    reference_ts = interpol( x_ts, x+offset, x_eff+offset) ; effective time-stamp for each pixel

    if smooth_ref_time then begin                          ; smooth out errors/jitter
        reference_ts = smooth( reference_ts, 50)
    endif

    y_at_x = interpol( y, y_ts, reference_ts)              ; effective 'y' at time of 'x' time-stamps
    nabs_y = y_at_x + abs_y[0] - y[0]                      ; effective absolute 'y'

    pixel_y = round( (y_at_x - min(y_at_x)) / step_y )     ; effective middle of equi-spaced Y pixels
    ny = max(pixel_y) + 1
```

We also pull out dwell time per pixel from “reference_ts” time-stamp differences as a function of sequence number. We then populate a 2D array of dwell time (“maia_dwell”) making use of the pixel address and sequence number. The array “maia_dwell” is stored simply in a common block, so it can be accessed when scanning data later in ‘read_setup’ and ‘read_buffer’.

```
    t = reference_ts - shift(reference_ts,1)
    t[0] = t[1]                                            ; fix wrap
```

```
dwell_array = t                                         ; dwell time (s)

maia_dwell = fltarr(nx,ny)
maia_dwell[pixel_x,pixel_y] = dwell_array * 1000.       ; ms dwell
```

A common/representative dwell "common_dwell" is also estimated from the most common dwell in a histogram.

Processing spectra/image data

Reading spectral data is done in the methods "read_setup" and "read_buffer". 'read_setup' is called for each new HDF5 file and then 'read_buffer' is called, normally in a loop until all data is read. In this device object, this looping is used access all detector channel data found in the HDF file (indexed using 'i_petra_channel').

On entry into 'read_setup', the HDF5 data file is already open and 'fstat' is used to get its filename so we can open that using HDF5 routine 'H5F_OPEN'. We first read sequence number vector for this file and determine the pixel address vectors 'x,y' for this sequence number vector ('sequence') using the saved 'pixel_x, pixel_y'. Note that HDF5 sequence number start at 0.

'read_setup' then determines how many detectors are present from 'dims'and sets 'n_petra_channel'. It then creates some arrays that will be used in 'read_buffer' to assemble vectors for the events stream for pixel address (x,y), "station" or channel number (ste), and photon energy (e).

'read_setup' then looks for the *IC PV* to find the vector of flux values as a function of sequence number. The chosen PV is given by 'flux_ic.pv'. Its gain units are set in 'flux_ic.val' and 'flux_ic.unit', which for now are set to unity (1 nA/V, i.e. nsls_flux_scale=1.0). It then sets the corresponding pixel flux values in an array 'flux' across the image. Here we distinguish between imaging mode, where we need to maintain a 2D flux array and spectrum mode, where we just accumulate a total flux.

```
nsls_flux_scale = flux_ic.val * flux_ic.unit
i0 = H5D_read(rec_id)
flux[x,y] = nsls_flux_scale * i0
```

'read_buffer' reads the spectra data from the HDF5 "spectra" for each channel 'i_petra_channel'. It then builds vectors of photon events 'e,ste,x1,y1,multiple for return to GeoPIXE (e.g. to 'da_evt' or 'spec_evt' routines). The vector 'multiple' makes it simple to convert a spectrum into a pseudo photon event stream by setting 'multiple' to the counts in each bin of the spectrum histogram.