HS - Madison Bootcamp 2016

June 11, 2016

1 Objective

Explain IceCube low level hardware through examination of the low level data product *hitspooling*. This will be presented as a **live** notebook so please ask questions as I am going - we can explore the data interactively!

2 Intended Audience

This notebook is given as a short 45-min course for the IceCube Bootcamp series to incoming students to IceCube or those otherwise needing some introduction to IceCube and IceCube software. Some basic familiarity with Python is assumed to understand the code presented here.

3 Hitspooling

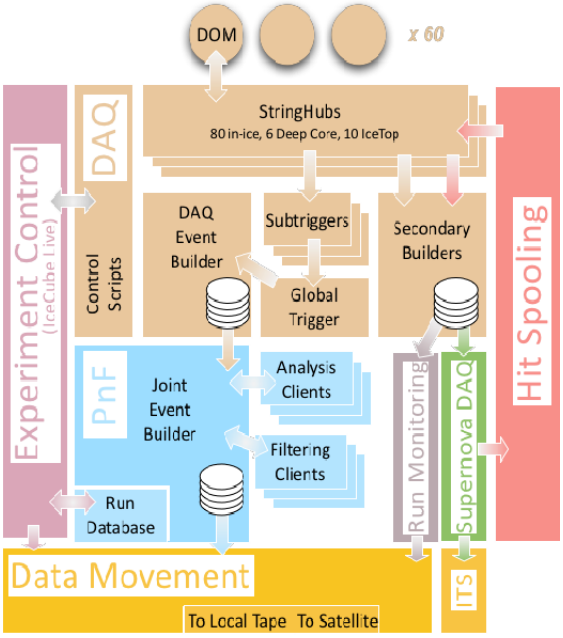
3.1 Why are we looking at hitspool data?

It's raw and it's simple. The idea is that you get very close to the metal and can appreciate how the detector works at a low level without the layers of (eventually very useful) decoding, calibration, and analysis software. Plus it uses some Python utilities that one doesn't often use in high-level code (bit stuffing routines, for example).

3.2 What is hitspooling?

- Each IceCube DOMHub has a 2 TB hard disk spinning away ... what a waste to not use that;
- The data rate of hits from a Hub is about 2 MB/s $\rightarrow$ in principle 1 million seconds of storage;
- This is RAW as RAW gets - only trigger is the DOM MB SPE/MPE trigger (and then there are beacon hits)
- A good way to save very low threshold data for use by external triggers
- Supernova DAQ (latency about 5-10 minutes)
- HESE triggers
- GRB
- FERMI triggers

3.3 How does it work? A picture ...



Detector Dataflow

The StringHub process in the DAQ is continuously reading out hit buffers from the DOMs (along with the supernova scalers, DOM slow control monitoring, and performing periodic TCALs). The hit data is *lightly* processed and then sent to a sort tree where all DOM channels readout by that Hub are merged into a single time-ordered list. This is conventionally called the *StringHub Frontend*. The hits are then passed to the *StringHub Backend* or *Sender* as Dave G calls it. The Sender puts the full hits into a large list but then extracts some channel and time information and sends that to the trigger. If the trigger spots an interesting pattern of hits (it merges these streams from *all* DOMHubs and looks over the entire in-ice or IceTop array) it generates a *ReadoutRequest* for the EventBuilder which then will go back to the Hubs and ask for the full hit information in a time window (-4 μs to +6 μs) around the trigger.

That is the classical DAQ. The Hitspool extension simply puts a T in the link between frontend and backend and the other output path of the T is written to local disk. The bigger effort of the Hitspool system is the architecture which pulls the data from the hub-resident disk files upon external request. I'm not going into details here - if you are interested you can read the PhD thesis of David Heereman.

3.4 Hitspool in the Data Warehouse

We will start here with hitspool files already extracted and transferred to the Madison Data Warehouse. The files are contained in `/data/exp/IceCube/YYYY/internal-system/hitspool` where `YYYY` is the 4-digit year. There are now several possible origins:

- Supernova alerts - these have a filename like `SNALERT`
- HESE triggers - `HESE`
- Anonymous - I guess test triggers `ANON`
- Solar Flare FERMI trigger

4 Python Code

So let's get to business! First the standard import sequence - everything is standard Python here except for the `slchit` and `hspack` modules which I wrote. It's good practice to put code in library modules for later easy re-use. I find the notebook format to be very seductive so I struggle to stick to this rule all the time.

Please note that I am using Python3 viz. Python2 - due to minor but nonetheless annoying incompat between 2 and 3 this code will not work on Python2.

```
In [37]: import numpy as np
         import pylab as plt
         import math, os, re, struct
         import datetime
         import hspack, slchit
         %matplotlib inline
```

I am going to define some file locations but these are local to my laptop's filesystem so will need to be changed for you. We will be looking at hits from String 22 only. The hitspool files come as a tarball of binary files divided up by Hub. There may be one or more files per hub. In the case of this hitspool event there should be 7 15-second files per hub.

```
In [2]: dirtop = "/Users/Kael/Documents/IceCube/hitspool"
        a_dir  = "SN20140815_062550"
        a_hub  = "ichub22"
        a_file = "HitSpool-19.dat"
```

4.1 A Look at Basic HitSpooling Structure

4.1.1 The TestDAQ Header

Let's take some baby steps to show you how easy this is. Hitspooling data is essentially a long list of binary formatted DOM delta compressed hits concatenated together in a file and separated by some verbose headers. Let's start by explaining the bitstream structure of the so-called *TestDAQ* header:

Offset	Type	Length	Description
0	i	4	Record length inclusive of this header
4	i	4	Record type identifier
8	q	8	48-bit mainboard ID
16	x	8	8 bytes of padding zeros
24	q	8	64-bit timestamp

Now let's verify this on real data:

```
In [3]: fhs = open(os.path.join(dirtop, a_dir, a_hub, a_file), 'rb') # Must use 'rb
        buf = fhs.read(54)
        struct.unpack(">iiq8xq", buf[:32]) # Get to know pack/unpack facility - ess

Out[3]: (54, 3, 175849839247268, 1954953742735004O4)
```

Some comments:

- Record length is 54 which means that there are 22 additional bytes in this record - get to this in a nanosecond
- Record type is 3 - always 3 for hitspool
- Mainboard ID is some long integer number here and one of 4 ways DOMs are referenced. There is a long and somewhat amusing history on DOM names. Most everyone uses the *OMKey* namespace for DOMs but there is also
- The production ID - an 8-character code which contains information on when and where the DOM was made and what kind of DOM it was (long penetrator, short penetrator, or IceTop qualified which means it passed additional cold testing). Consult IceCube DWG 9000-0038 for decoding.
- The mainboard ID - this is actually derived from hashing a serial number in the nonvolatile flash on the DOM. There are two so actually the DOM has the possibility to masquerade as someone else but normally the 2nd flash is not used for boot unless there is a hardware failure on the first;
- The OMKey name - this is string and position on string, 1 being at the top and 60 being at the bottom. Note that IceTop DOMs are numbered 61-64, 61 and 63 are the high gain DOMs, 62 and 64 the low gain DOMs.
- Nickname - each and every DOM has a nickname. Not just for amusement, it *is* easier to remember exceptional channels by their name as opposed to one of the other identifiers.

- Timestamp is linked to UTC time - it tells you the number of 0.1 ns ticks since the beginning of the year.

There is a nicknames class in the `hspack` module which reads in a file database called `nicknames.txt` (available in the pDAQ `config` project) and can do various lookups for you.

```
In [4]: recl, fmtid, mbid, utc = struct.unpack(">iiq8xq", buf[:32])
        hexid = "%12.12x" % mbid
        names = hspack.lookup(hexid)
        hittime = datetime.datetime(2014,1,1) + datetime.timedelta(seconds=1E-10*ut
        print (names, ":", hittime)

('9fef3b33bfa4', 'TP9P3737', 'Mad_Monty', '22-28') : 2014-08-15 06:25:37.427350
```

4.1.2 The Compressed Hit / Soft Local Coincidence (SLC) Structure

We still have 22 bytes to go in this record. These bytes are almost exactly what comes out of the DOM, there is a bit of reordering done in the StringHub process. The raw data format is:

Offset	Type	Length	Description
32	h	2	Version ID - should be 1
34	h	2	Pedestal flag - if 1 then no ped subtract; 2 ped subtraction in DOM
36	h	2	Some flag which I now forget the meaning
38	q	8	48-bit DOM clock counter 40 MHz frequency
46	I	4	Compressed word 1
50	I	4	Compressed word 3

The last two *compressed words* give information on which triggers fired in the DOM and chargestamp information. The authoritative document on the format of these words, as well as the secrets behind the Delta 1-2-6-3-11 lossless waveform encoding is Delta Compressor Data Format and Processes. If you want to understand the trigger bitmask you will need to read the DOMAPP CPU Firmware API Document. I do remember that bit 0 is SPE trigger (the most common), bit 1 is MPE, and bit 2 is the forced trigger.

We dump out first the raw bits but then use the utility decoders found in the `slchit` module to gain more insight into this hit - we know when and where it occurred but what caused it and how big was the PMT pulse? The trigger below is 1 which corresponds to bit 0 and only bit 0 being set so this was an SPE trigger. The chargestamp for in-ice DOMs (different for IceTop) is a sequence of 4 values which inform the location of the peak sample in the FADC: the 1st number gives the sample index of the peak sample relative to the trigger (but then you also have to account for FADC pipeline depth), and quantities 2, 3, and 4 give the samples around the peak and including the peak. Note that the FADC is not zero subtracted so there is going to be a baseline of approximately 127 which these samples ride on.

```
In [5]: print ("Raw dump: %x %x %x %6.6x %8.8x %8.8x" % struct.unpack(">3hq2I", buf
        hit = slchit.SLCHit(buf)
        print ("Trigger:", hit.trigger, "Chargestamp:", hit.chargestamp)

Raw dump: 1 2 1 c51d6123f9b 8004000c 3a151c8c
Trigger: 1 Chargestamp: (7, 133, 142, 140)
```

That's about all that we can do for SLC hits. The other bits in the compressed header will simply tell you that there is no waveform information and no LC bits set. In order to look at non-trivial information here we will have to hunt down a waveform bearing hit which had a neighbor to setup the HLC condition. I will just keep on reading in records from the file until I find a record which when decoded says it has waveform information. Note that, were I speed-optimizing I would probably opt to *not* decode the record and use the knowledge that SLC records are *always* 54 bytes long - but that is a little cryptic so let's do it this way:

```
In [7]: while True:
            buf = fhs.read(54)
            recl, fmtid, mbid, utc = struct.unpack(">iiq8xq", buf[:32])
            buf += fhs.read(recl-54)
            hit = slchit.DeltaCompressedHit(buf)
            if hit.fadc_avail: break
```

4.2 DOM Raw Waveforms

Gotcha! Now we have a hit that claims to have waveform information. There is another set of slides which should give some details on the waveform capture hardware of the IceCube DOM but let me again summarize here in case those slides are not available.

4.2.1 ATWD - The Analog Transient Waveform Digitizer

First there are two ATWDs per DOM. These are custom-designed chips from the late 90's early 2000's which have 4 channels of 128 analog sampling capacitors. When the ASIC is launched,

input voltages are sampled and held analogically on the caps at a settable rate of 100's of MHz - we use a setting which corresponds to about 290 MHz. When the analog sampling is complete, a relatively slow ADC comes along and converts the capacitor voltages to a 10-bit digital quantity. In the case of IceCube there are 4 channels because we want to extend the dynamic range beyond the 10-bits to span a factor of about 10,000 which is near the dynamic range of the PMT:

- Channel 0 is the high-gain channel: 16× amplification before the ATWD
- Channel 1 is the mid-gain channel: 2×
- Channel 2 is the low-gain channel: 0.25×
- Channel 3 is the swiss army knife channel - used for other things

Why are there two ATWDs you ask? (A) if one fails there is always another one; (B) deadtime - the ATWD samples relatively fast for hardware of its time and is lower power. But, you pay the piper in the ability to only capture about 450 ns of PMT pulse and there is a significant deadtime in the digitital conversion - about 30 μs per channel. If another pulse comes along 10 μs later you would miss it. The dual-chip solution helps here. The ATWDs operate in *ping-pong* mode whereby chip A and chip B always alternate in triggering. There is kind of a funny edge case where readouts can happen in inverted order - you might actually get hits that appear slightly out of time order - due to the variable conversion time. If for example chip A starts a conversion in which all 3 gain channels are readout, but chip B starts 50 μs later and only reads out the high gain channel, chip B will actually get inserted into the data stream first. Fortunately, the StringHub works this out for you so you don't have to worry about it.

4.3 FADC - The 'Fast' ADC

Because of the limited capture of the ATWDs, the DOM has a third digitizer - a commercial pipelined AD9215 10-bit, 40 MSPS A/D converter from Analog Devices. In principle it could keep on capturing until memory got filled but it is fixed in hardware to always deliver 256 samples - 6.4 μs depth. It has limited dynamic range relative to the multiple gain ATWD and so is relatively susceptible to digital overflow.

4.4 A/D Pedestals

A quick note on pedestals. Both the ATWD and FADC are readout as unsigned quantities. To avoid overflows they are riding on a non-zero pedestal. The pedestal in the ATWD is rather complex and varies sample to sample. For this reason, at the beginning of each run, the DOMApp CPU application acquires several hundred waveform captures with no input (any pulses present by accident are actually rejected by the algorithm) to build up the average pedestal pattern. It then stores the pattern in an FPGA register which is then used by the firmware to hardware subtract out the pattern for each waveform acquired. A bias is subtracted from the pattern so that the ATWD ideally sits around 128 samples with shorted input.

The FADC also has a pedestal but it is a fixed flat pattern also about 128 counts high.

4.5 Delta Compression

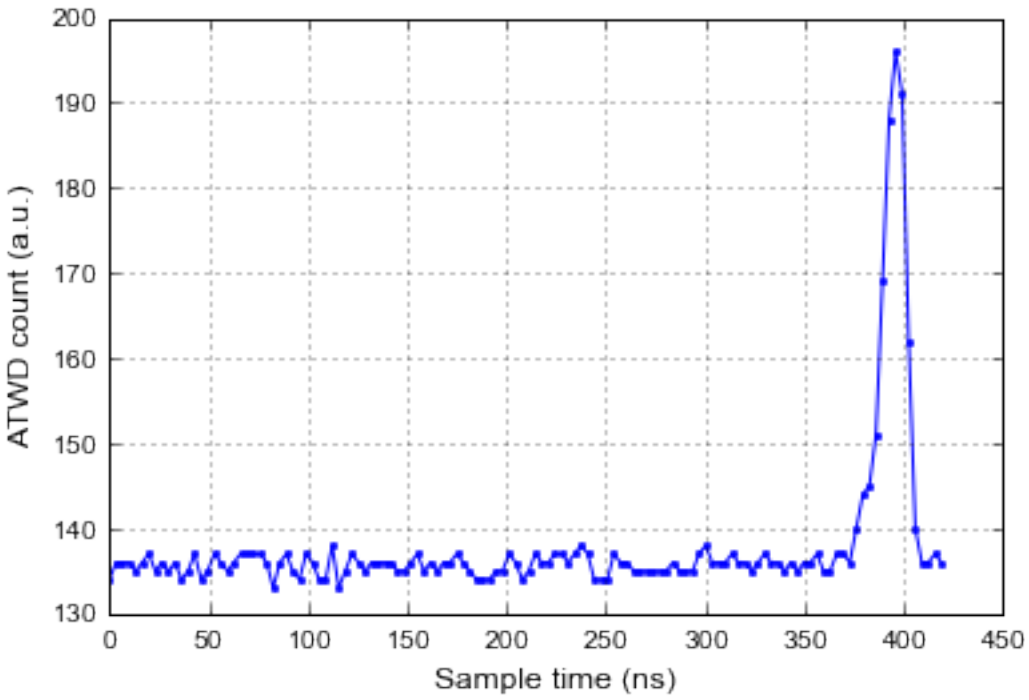
The ATWD and FADC waveforms are losslessly compressed using sample difference and bit compression. Not enough time to cover here. The `slchit` module contains a decoder for this encoding scheme. Use it as a black box - it works, even though I wrote it I would have a hard time to

explain it. Anyway, decoding takes some time so I force the user to explicitly request a decoding. There are more elegant ways of doing this in Python - TODO.

```
In [8]: print ("Trigger mask ......", hit.trigger)
        print ("FADC available ....", hit.fadc_avail)
        print ("ATWD available ....", hit.atwd_avail)
        print ("LC bits ...........", hit.lc)
        print ("ATWD Chip .........", hit.atwd_chip)
        hit.decode_waveforms()
        t = np.arange(128)*3.3
        plt.plot(t, hit.atwd[0], '.-', ms=5)
        plt.grid()
        plt.xlabel('Sample time (ns)')
        plt.ylabel('ATWD count (a.u.)')
```

```
Trigger mask ... 1
FADC available ... True
ATWD available ... True
LC bits ... 1
ATWD Chip ... 1
```

```
Out[8]: <matplotlib.text.Text at 0x1d253e3f320>
```



OK, cool so we just got our first waveform but you may notice that it's a bit, uh, rotated relative to how normal PMT pulses look (the PMT multiplies electrons, not positrons, so the pulse should

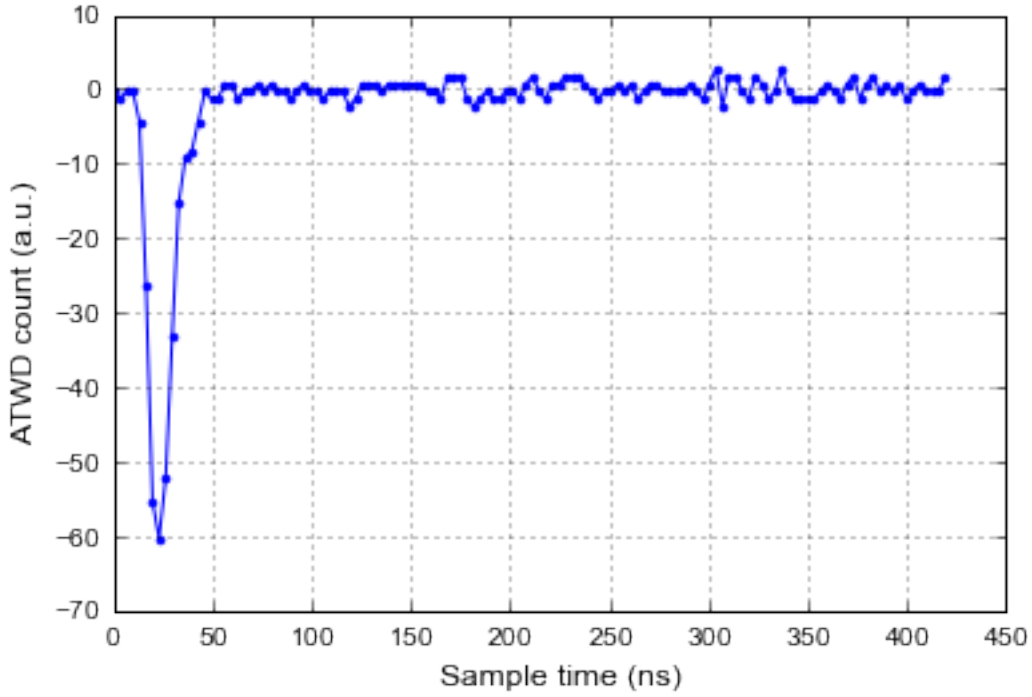
be negative going). It also happens to be backwards in time. OK we can fix that with NumPy's array functions - good opportunity to introduce you to some array manipulation in Python. Note that what comes from the hit is a plain-old Python `list` type but it is easy enough to construct a full-fledged NumPy array:

```
In [9]: atwd0_raw = np.array(hit.atwd[0], 'd') # The 'd' means convert it to 64-bit
```

Now we can do some syntatically compact ops on the array - let's invert it and time reverse it but also try to set the baseline to zero. This is not safe in general but I will assume the following samples are just noise rattling around zero.

```
In [10]: ped = np.average(atwd0_raw[0:80])
         atwd0 = ped - atwd0_raw[::-1]
         plt.plot(t, atwd0, '.-')
         plt.grid()
         plt.xlabel('Sample time (ns)')
         plt.ylabel('ATWD count (a.u.)')

Out[10]: <matplotlib.text.Text at 0x1d2547f65c0>
```



Ah, that looks better. Notice how close the pulse is to the extreme left edge. The ATWD style of SCA has been replaced by samplers which continually sample over the analog capacitors and get *stopped* on a trigger as opposed to *started*. This gives ample pre-trigger window and obviates the need for things like the IceCube DOM's delay board which delays the signal going into the ATWD for 72 ns to allow the ATWD to start before the pulse arrives. Even then we are just at the edge.

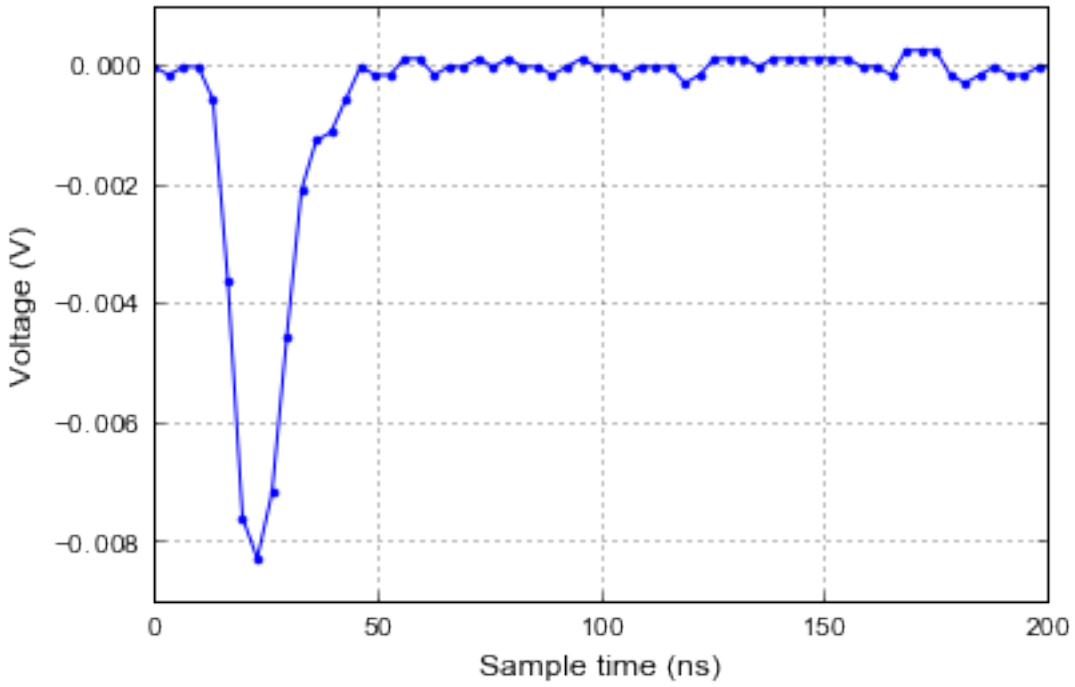
You know what? Let's try to put the samples into physical voltages - we will make the ATWD an oscilloscope. Disclaimer: please don't try this at home: getting calibrations good to the few percent level requires much more work, but this illustrates the basic concepts. Needed for this are the following pieces of general information:

- ATWD digital gain is about 2.2 mV per count;
- 16 × amplifier between the ATWD and the PMT signal input connector.

That's it!

```
In [11]: v0 = atwd0 * 0.0022 / 16
         plt.plot(t, v0, '.-')
         plt.grid()
         plt.xlim((0, 200))  # Tedious to look at whole waveform - zoom into pulse
         plt.xlabel('Sample time (ns)')
         plt.ylabel('Voltage (V)')

Out[11]: <matplotlib.text.Text at 0x1d255b89358>
```



OK, but wait a second. How do we figure out how many electrons came out of the PMT? We have a voltage and we need a charge. The PMT is a current source which gets turned into a voltage by the front-end impedance of the DOM MB which is about 47 Ω. You remember Ohm's Law, right? I = V/R and charge, Q, is the integral of the current:

$$Q = \sum_i \frac{V_i}{R_{\text{in}}} \Delta t$$

```
In [18]: q = -sum(v0 / 47) * 3.3E-09
         q_spe = 1.602E-19 * 1E7
         print ("Charge is {:.1f} p.e.".format(q / q_spe)) # This is the Python3 wa

Charge is 1.7 p.e.
```

4.5.1 FADC

Nominally 1 mV per FADC count and 15 × amp gain in front. However, the amplifiers are strongly shaping at signal frequencies so the pulses are broadened.

```
In [36]: fadc = np.array(hit.fADC, 'd')
         ped_fadc = np.average(fadc[50:])
         t_fadc = np.arange(256)*25
         v_fadc = 0.001*(ped_fadc - fadc) / 15
         plt.plot(t_fadc, v_fadc, '.-')
         plt.grid()
         plt.xlim((0, 1000))
         q_fadc = -sum(v_fadc / 47) * 25E-09
         print (q_fadc / q_spe)

1.24044489936
```