Giving *pandas* ROOT to chew on: experiences with the XENON1T Dark Matter experiment

D Remenska[1], C Tunnell[2], J Aalbers[2], S Verhoeven[1], J Maassen[1], J Templon[2]

[1]Netherlands eScience Center, Science Park 140, Amsterdam, The Netherlands
[2]National Institute for Subatomic Physics (NIKHEF), Science Park 105, Amsterdam, The Netherlands

E-mail: d.remenska@esciencecenter.nl

Abstract. In preparation for the XENON1T Dark Matter data acquisition, we have prototyped and implemented a new computing model. The XENON signal and data processing software is developed fully in Python 3, and makes extensive use of generic scientific data analysis libraries, such as the SciPy stack. A certain tension between modern "Big Data" solutions and existing HEP frameworks is typically experienced in smaller particle physics experiments. ROOT is still the "standard" data format in our field, defined by large experiments (ATLAS, CMS). To ease the transition, our computing model caters to both analysis paradigms, leaving the choice of using ROOT-specific `C++` libraries, or alternatively, Python and its data analytics tools, for developing physics algorithms. We present our path on harmonizing these two ecosystems, which allowed us to use off-the-shelf software libraries (e.g., NumPy, SciPy, scikit-learn, matplotlib) and lower the cost of development and maintenance. To analyse the data, our software allows researchers to easily create "mini-trees"; small, tabular ROOT structures for Python analysis, which can be read directly into *pandas* DataFrame structures. One of our goals was making ROOT available as a cross-platform binary for an easy installation from the Anaconda Cloud (without going through the "dependency hell"). In addition to helping us discover dark matter interactions, lowering this barrier helps shift particle physics toward non-domain-specific code.

1. Introduction

With the recent inauguration of the new XENON1T instrument at the Gran Sasso National Laboratory, the most sensitive dark matter experiment in the world will boost-up the hunt for this rare-signature particle. In preparation for the data acquisition, a new computing model has been designed and implemented, making a paradigm shift from the typical HEP-specific frameworks and tools. Python has received a widespread adoption among the scientific community, due to its ease of use for fast prototyping, and the rich ecosystem of scientific data analysis packages (Numpy, SciPy, pandas, and matplotlib, to name but a few). Our quick experiment progress gives us more in common with a tech startup than most other experiments, which is why we explored community-supported Big Data solutions. Large HEP experiments have an investment in software spanning 20 years, and mostly use the ROOT framework and data format.

To ease the tension between these "novel" Big Data solutions and the ones typically used in HEP communities, we interfaced Python *pandas* to ROOT, effectively harmonizing these two

software jungles, which allowed us to use existing software libraries and leverage our Python experience, while leaving the choice of using ROOT-specific `C++` libraries as an alternative. PhD students and researchers should not have to spend too much time getting software tools and libraries to work in their own local environment; they should rather focus on the physics research. For this, we built cross-platform binary ROOT packages that can be easily installed in user space, taking care of all library dependencies. In this paper we report the technical choices we made in order to meet these challenges, and the resulting impact on the XENON (and wider) community.

2. The XENON Dark Matter experiment

Most matter in our Universe is "dark matter", yet it remains invisible to our instruments. What constitutes dark matter is one of the most intriguing questions in modern physics. The XENON dark matter experiment [1], installed underground at the Laboratori Nazionali del Gran Sasso (LNGS), aims to detect dark matter in the form of interactions of Weakly Interacting Massive Particles (WIMPs) with xenon nuclei. The XENON100 [2] detector uses a total of 161 kg of pure liquid xenon (LXe) divided in cylindrical two-phase (gas/liquid) time projection chambers. Its successor, the XENON1T detector [3], deployed in late 2015 at LNGS, is realized by scaling up the existing XENON100 detector by a factor of 10 and reducing the background by a factor of 100. With the detector being able to distinguish between the WIMP and possible background radiation, the researchers hope to be able to determine the existence of a new subatomic particle, and determine its mass and likelihood of interaction with ordinary matter.

2.1. Computing model and data flow

In preparation for the XENON1T data acquisition, a new computing model is envisaged. The technology chain behind it, is shown on Figure 1. The experiment data flow contains three parts: acquisition, processing, and analysis. The data acquisition system (DAQ), in proximity to the detector, acquires data at a rate of up to 2.4 Gbps, digitizes the PMT signals and transfers the digitized waveforms from the FADCs (Fast/Flash Analog-to-Digital Converters) to a local MongoDB storage. The acquisition stage produces cumulatively 1PB of raw BSON-based data files (binary serialization format) that must be saved to tape. The raw data is subsequently partially processed and reduced using highly optimized trigger routines. The Celery distributed task queue [4] is used for parallelization. At the end of acquisition, all of the raw data remains within a database at the LNGS site. The utilization of a cloud or grid infrastructure is foreseen

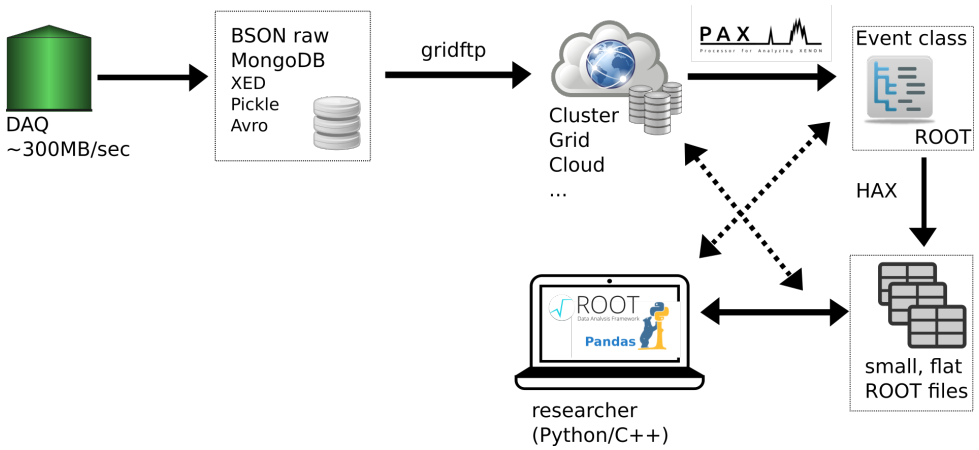

Figure 1: XENON1T computing technology chain and data flow

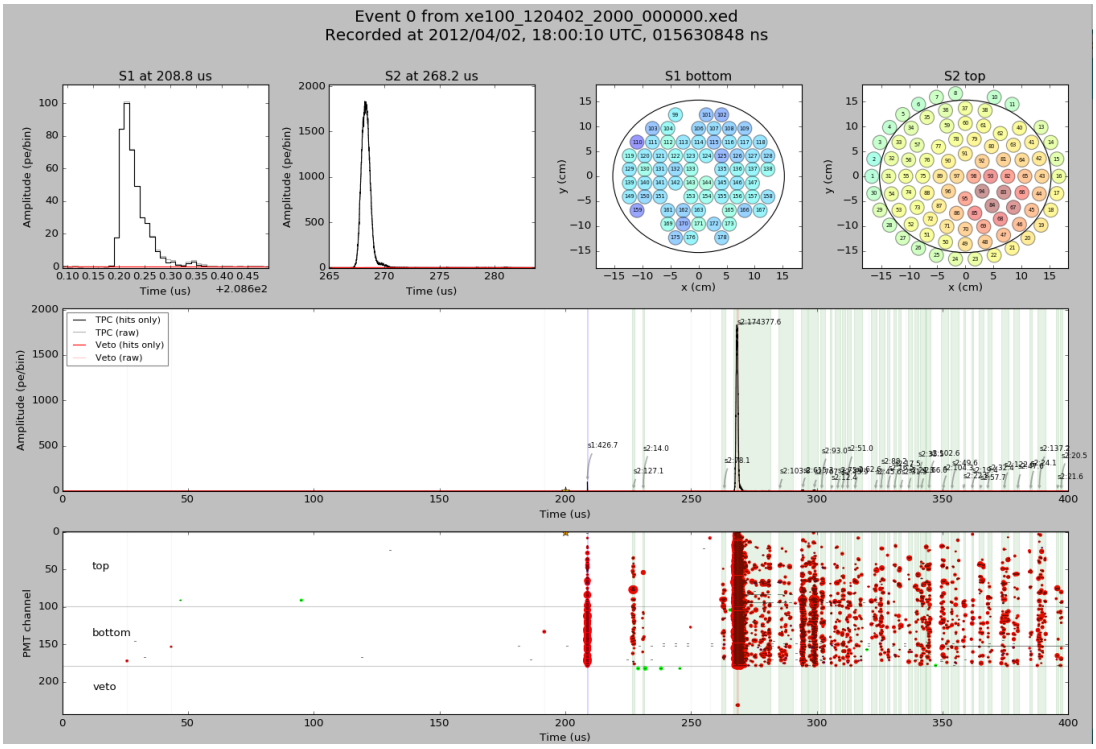

Figure 2: Screenshot from the Processor for Analyzing XENON (PAX) software for raw data processing

in the future, for more efficient larger-scale processing. The Processor for Analyzing XENON (PAX, Fig. 2), developed fully in Python 3, is used for digital signal processing and other data processing on the XENON100/XENON1T raw data. With a series of I/O and transformation plugins developed for intermediate processing steps, PAX reduces a single event from roughly 1MB of raw data to 50 KB of processed data.

The default PAX output format is a ROOT file containing the Event class. The processing stage will produce about 50 TB of ROOT files per year, which the analysts must be able to process in an efficient manner. For this purpose, the Handy Analysis for XENON (HAX) software was prototyped in Python 3, which allows easier and flexible "pythonic" access to the ROOT data, without the inconvenience of manually re-looping over the events in one or more ROOT files, each time a new plot needs to be created or adjusted. To extract and further reduce the data (make cuts), HAX lets researchers create "mini-trees": small, tabular ROOT structures for Python analysis that can be read directly into *pandas* DataFrame structures. The main advantage of using them is the easiness of computing over values which are not directly in the ROOT tree, compared to the usual `TTree.Draw` method of analysis where access to the actual values of the data is not directly possible. In the following section we explain in more details how we implemented the tools for analysis.

3. Beyond HEP-specific tools

The XENON scientific community consists of about 150 researchers, with no dedicated software-expertise manpower. The new computing model was prototyped to address some of the issues that small-to-medium-sized particle physics experiments face: limited resources and manpower to develop, maintain, and support the necessary software libraries. The bigger LHC experiments,

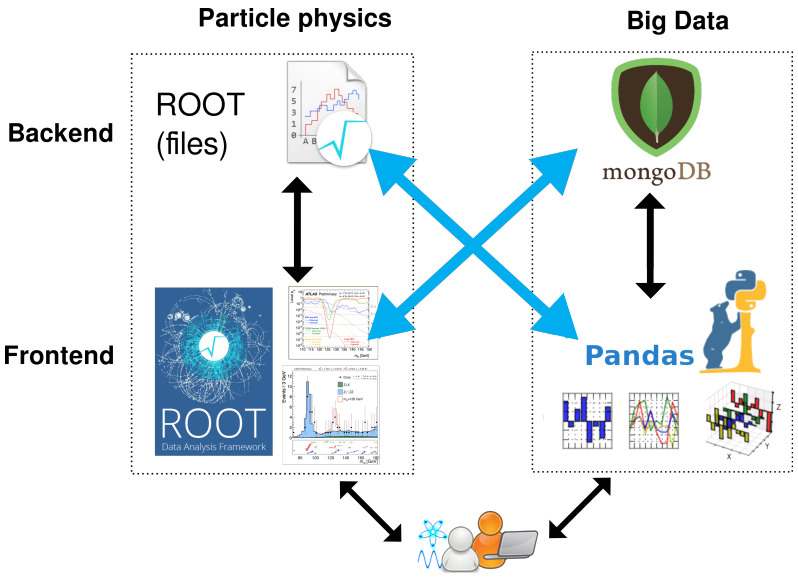

Figure 3: Data flows in Particle Physics and Big Data. Interfacing ROOT I/O in Python *pandas* will transitively allow interoperability between existing tools in the two ecosystems

such as ATLAS and CMS, define the standards of particle physics computing for more than two decades, relying heavily on CERN's ROOT framework and file format for physics analysis. The observation that the HEP computing community is becoming less isolated, is already reported [5]. However, the process is rather slow, as the HEP community seems to be neither very interested nor ready to accept more mainstream generic solutions, or provide such solutions. There are exceptions to this, of course, but the list is rather short. Examples of projects with beyond-HEP applicability and impact are DIRAC [6], GEANT4 [7], and PanDA [8].

On the other hand, generic Big Data analytics tools are gaining a momentum in scientific data analysis, due to the (much) larger and dedicated community support behind them. Python has become the language of choice for high-level applications where fast prototyping and efficient development are important, while glueing together low-level libraries for performance-critical tasks. Our data processing software (including the Event data model) is developed fully in Python 3. We have found that ensuring correctness via unit testing, integration testing, documentation, and code sharing in Python and IPython notebooks has many advantages compared to our existing particle physics codes. We put significant efforts into following Open Source standards and procedures necessary to maintain coherent, sustainable, and easy-to-use software. In addition, relying on existing analytics code, instead of particle-physics specific code, greatly simplifies our work and allows us to focus on the physics.

However, within the XENON collaboration, there is a tension between these "novel" Big Data solutions, and the existing ones used in the wider HEP community. To ease the transition, our computing model should cater to both analysis paradigms, leaving the choice of using ROOT-specific C++ libraries, or alternatively, Python and its data analytics tools, as front-end framework for developing physics algorithms. Hence, our goal is to harmonize these two ecosystems, shown on Fig 3, by organizing software and data such that we can work with the existing particle physics infrastructure, yet still use community Big Data tools. Specifically, we interfaced Python *pandas* DataFrames to the ROOT format, which allows us to use existing software libraries (e.g., NumPy, SciPy, scikit-learn, matplotlib) and lower the cost of developing and maintaining the XENON software stack. With pandas I/O to ROOT, we can also take

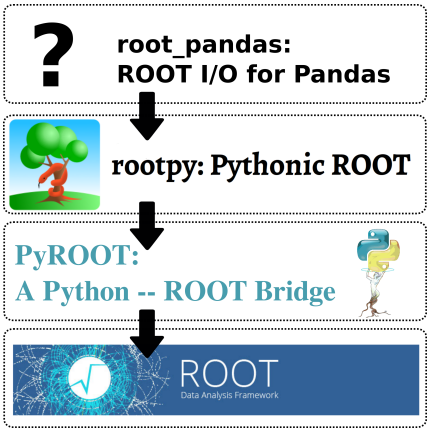

Figure 4: Interfacing ROOT to Python pandas DataFrames. Arrows between layers represent usage dependency.

advantage of tools such as the *odo* project [9] for data migration between storage systems (part of the PyData stack), and allow for conversions between pandas, HDF5, and MongoDB formats. In addition to helping us discover dark matter interactions, lowering this barrier helps shift particle physics toward non-domain-specific codes.

3.1. Interfacing ROOT to Python pandas DataFrames

Besides the "social" reasons (hesitance to make a big radical leap) for keeping ROOT as the backend file format, there are valid technical ones as well: column stores (efficient column scans), continuously improved techniques for pre-fetching, caching, and flushing of data buffers. Over the past years, the ROOT I/O team has significantly enhanced the I/O performance of ROOT [10]. However, in our experience, nowadays many "physics-specific" analysis algorithms are not that "physics-specific" in fact. Python has tools and libraries for efficient data structures and array manipulations, with a steeper learning curve than the ROOT analysis framework. We further elaborate on the existing layers (Fig. 4) we used and adapted to fulfill the goal of interfacing ROOT to the Python pandas DataFrames.

The PyROOT [11] module is a bridge between Python and C. Despite its ROOT bindings, it is merely a mechanical translation of C++ function calls, and as such, programming is not really "pythonic", i.e., one can not fully leverage existing experience with Python; the native Python data structures are not used as expected, and these function calls do not adhere to the common Python idioms. Furthermore, Python 3 is not yet fully supported, and at the moment of writing, there is no official maintainer of PyROOT. Often both Python 2 and 3 versions are available on users' systems. However, the official binary releases of ROOT have the PyROOT module built with only Python 2 support, so PAX users must resort to compiling ROOT from source. Steering the compilation process to look for the correct Python executable and libraries is cumbersome, and requires setting up environment variables and symlinks. One of our goals was to significantly simplify this process for the collaboration. In Section 4 we elaborate our approach to achieving this goal. In addition, there were blocking Python 3 issues with memory leaks [12] when using array fields, which we had to resolve by patching PyROOT.

The rootpy [13] project is a community-driven effort aiming to bring PyROOT closer to the Python arena. In addition to its main capability of creating ROOT tree data models purely in Python (Listing 1) and easier navigation through ROOT files, rootpy provides an interface with matplotlib, a popular Python publication-quality plotting library. It can further redirect

```
from rootpy.tree import Tree, TreeModel
from rootpy.tree import FloatCol, IntCol
from rootpy.tree import FloatArrayCol, CharCol
from rootpy.io import root_open
from random import gauss, choice

f = root_open("test.root", "recreate")
# define the model
class Event(TreeModel):
    s = CharCol()
    x = FloatCol(default = 'nan')
    y = IntCol(default = 0)
    f = FloatArrayCol(5)
tree = Tree("test", model=Event)
# fill the tree
for i in range(5):
    tree.s = ord(choice(ascii_letters))
    tree.x = gauss(.5, 1.)
    tree.y = i
    for j in range(5):
        tree.f[j] = gauss(-2, 5)
    tree.fill()
tree.write()
f.close()
```

Listing 1: Creating tree models with rootpy

```
class ReconstructedPosition(object):
    x = float('nan')
    y = float('nan')
    ....

class Peak(object):
    area = 0.0
    detector = 'ptc'
    ...
    rec_positions = list(ReconstructedPosition)

class Hit(object):
    channel = 0
    center = 0.0
    ...
class Event(object):
    event_number = 0
    dataset_name = 'Unknown'
    ...
    peaks = list(Peak)
    hits = np.array([], dtype=Hit.get_dtype())
```

Listing 2: The pax Event model

ROOT error messages through Python's logging system, turning them into Python exceptions. The related root_numpy [14] library is used for efficient conversions between ROOT TTrees and structured NumPy arrays. One can then take advantage of the statistical and numerical packages that Python offers. The root_pandas [15] package is a convenience wrapper built around the root_numpy library. It allows to easily load and store pandas DataFrames in the ROOT format.

One essential problem that we stumbled upon, with both rootpy and root_pandas, is that creating trees with branches of user-defined types (and arrays/lists thereof) still requires the necessary "glue" C++ code. We currently autogenerate the C++ classes inheriting from TObject, based on introspection of our existing Python Event data model (sketched in Listing 2). Without it, these tools cannot handle nested (json-like) structures, such as the ones PAX uses. Storing non-rectangular data into Python DataFrames is possible, but it results in object structures that are not easily queried in a manner that flat data can be. This is why we created another layer, the HAX analysis tool, which essentially uses root_numpy under the hood, to load extracted ROOT "mini-trees" data into pandas DataFrames. We were not able to find a suitable mature library or a file format that deals with this type of "ragged" data in a more elegant and flexible manner for users.

4. ROOT in the Anaconda Cloud

The XENON data processing software makes extensive use of scientific data analysis libraries (such as the SciPy stack). To make it straightforward for the collaboration users to install the software in user space, we strongly recommend the Open Source Anaconda Python distribution [16], which includes many Python scientific modules bundled in a single installation. This distribution installs all the dependencies cleanly in a single directory (known as *environment*), and does not interfere with other Python installations present on a system. Anaconda comes with a binary packaging, dependency, and environment management system named Conda [17]. It is the main interface for managing multiple installations of various binary packages in separate environments. Conda environments are not Python-specific, and offer additional advantages compared to virtualenv or pip. We aimed to make it easy to re-create

consistent environments and be able to reproduce the work of collaboration members, so we adopted Anaconda as a long-term solution in our workflow.

As already mentioned, one of the bigger barriers users typically faced was getting ROOT installed in user space. Rather than relying on users compiling it with the specific Python 3 bindings and resolving all dependencies themselves, our goal was to provide a ROOT binary distribution that would be as easy to install as `conda install root={version}`. Building a cross-platform binary conda package involves creating a *recipe*, containing metadata on the source code, the build and run dependencies, as well as the scripts needed to build the package. In many cases creating a recipe is straightforward (in particular for pure Python software); however, making a "as portable as possible" ROOT installation is somewhat involved, due to its dynamic dependencies on `GCC` and libstdc++.

Linux distributions typically ship a particular version of `GCC` and `glibc`. Our XENON users need to be able to install and run ROOT on a variety of OS versions, ranging from reasonably old to new. Given that ROOT 6 requires `GCC>=4.8` to use some of the latest `C++` features, while we aimed to create a single binary distribution that would work out-of-the-box on older systems, we decided to ship `GCC 4.8` as a ROOT runtime dependency package in conda. To maintain the ABI (binary) compatibility, and go "as low as possible" with `glibc`, the solution we found is a hybrid one: provide a `GCC 4.8` conda binary that is compiled against an older Linux kernel. To meet this challenge, Scientific Linux 6 offers a good compromise: the Developer Toolset (v2) from CERN provides a newer complier and binutils, while the kernel remains untouched (`glibc 2.12`). This way we can provide a `GCC/libstdc++` that supports the latest `C++` features, but is still compatible with the oldest `libc` and kernel we can support. These technical choices allowed us to produce ROOT (version 5 and 6) conda binaries which have been successfully tested on Ubuntu (11.10, 12.04, 14.04, 15.04) and SLC (6.5 and 7). The ROOT binaries (with Python bindings for 2.7 and 3.4) and dependencies are available from the Anaconda Cloud, a service for hosting (free public) conda packages. This also allowed us to do automated integration testing of the XENON data processing software within the Travis CI. All recipes are also publicly available [18], along with a documentation on their usage.

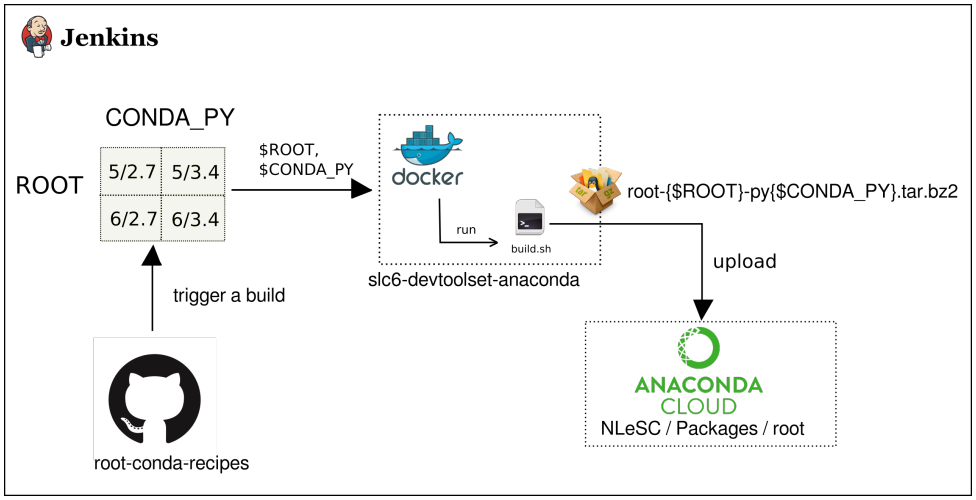

Figure 5: Jenkins CI setup for building ROOT binaries and uploading them to Anaconda Cloud

4.1. Continuous integration

Our goal was to have the ROOT (and dependencies) conda packages be automatically rebuilt, tested and uploaded to the Anaconda Cloud, triggered by pushing changes to the GitHub recipes repository (for instance, when a new version of the ROOT source code is released). There are two mature options for this: Travis and Jenkins CI. Travis CI is the most convenient way to automate the builds for both Linux and OSX platforms, and it is well integrated as a free service with GitHub. Unfortunately, the current build time restrictions do not allow us to build ROOT. Depending on the number of features which are enabled, building and uploading ROOT 6 binaries can take up to (more than) 120 minutes, even with the most bare-bones configuration.

A public Travis CI setup is still used for testing the sanity of existing ROOT conda binaries. Once the build-time restriction is lifted or relaxed, switching the building process on, is rather trivial. For the moment, we rely on an externally-hosted Jenkins machine where we have more resources. The setup is shown in Fig. 5. This Jenkins server runs an Ubuntu distribution (12.04, `glibc 2.15`), where `GCC 4.8` is not available. Docker is a container technology that allows for packaging software with all of its dependencies into one image ("lightweight VM"), with a guarantee that it will always run exactly the same, regardless of the host environment it is deployed in. We use it to ship the full build environment for ROOT[1]. Each packaged binary of the build-matrix is created using a publicly available Docker image of SLC 6.5, equipped with CERN's Developer Toolset (v2), cmake, and conda/Anaconda installed. `CONDA_PY` and `ROOT` are environment variables (configurable in Jenkins), and are passed to the *build.sh* script, which contains the `conda build` and `upload` commands. Each commit to the `root-conda-recipes` repository triggers a new build, and the responsible committer will receive an email in case of a build failure.

5. Evaluation

...some XENON1T workshop trainings? Jeff's anaconda Nikhef setup at Stoomboot? centralized installation? (which experiments? what kind of analysis they did?) xecluster? Pandas examples from XENON1T/hax?

Figure 6

[1] Running a Docker engine requires admin priviledges, which is why we did not use it to provide ROOT binaries directly

In [9]:
```
# I also load the S2 width, for use in the plot in the appendix
class NewDriftTime(hax.minitrees.TreeMaker):
    __version__ = '0.0.1'
    extra_branches = ['peaks.index_of_maximum', 'peaks.hit_time_std']

    def extract_data(self, event):
        if not len(event.interactions):
            return dict()
        s1 = event.peaks[event.interactions[0].s1]
        s2 = event.peaks[event.interactions[0].s2]
        return dict(drift_time_2=(s2.index_of_maximum - s1.index_of_maximum) * 10,
                    s2_width=s2.hit_time_std)

data = hax.minitrees.load('xe100_111110_1127', ['Basics', NewDriftTime])
```

```
WARNING:hax.paxroot:Root file /mnt/xecluster/archive_lngs/common/PaxReprocessed_9/good/xe100_111110_1127.root does not include pax event class. Normal for pax < 4.5.Falling back to event class for pax 430
WARNING:hax.paxroot:Root file /mnt/xecluster/archive_lngs/common/PaxReprocessed_9/good/xe100_111110_1127.root does not include pax event class. Normal for pax < 4.5.Falling back to event class for pax 430

Created minitree Basics for dataset xe100_111110_1127
Created minitree NewDriftTime for dataset xe100_111110_1127
```

`data` is now a pandas DataFrame containing basic info (see below) for all the events. For more details, and instructions on how to select your own variables, see the `hax` tutorial. Here is a look inside the data we just loaded:

In [10]:
```
data.head()
```

Out[10]:

	index	cs1	cs2	dataset_number	drift_time	event_number	event_time	largest_coi
0	0	533.132248	117434.248258	1111101127	102700.195312	0	1320920877023216128	0
1	1	NaN	NaN	1111101127	NaN	1	1320920877024649984	0
2	2	916.212603	89846.426385	1111101127	159293.125000	2	1320920877028420096	0
3	3	NaN	NaN	1111101127	NaN	3	1320920877029669120	0
4	4	337.785598	32902.233486	1111101127	160682.093750	4	1320920877032971008	0

5 rows × 22 columns

We'd like to have the drift time in us, and the S2 area on the bottom. Let's compute these from the basic variables:

In [12]:
```
drift_time = data['drift_time_2'].values / units.us
s2_bottom = data['s2'].values * (1 - data['s2_area_fraction_top'].values)
```

Exploratory analysis

Before fitting, let's have a first look at the data we have:

In [13]:
```
plt.scatter(drift_time, s2_bottom,
            c=(data['s1'].values), vmax=2 * data['s1'].mean(),
            marker='.', edgecolors='none', s=10)
plt.yscale('log')
cb = plt.colorbar(label='S1 (pe)')
plt.grid(which='minor', linestyle='-', alpha=0.08)
plt.grid(which='major', linestyle='-', alpha=0.15)
plt.xlim(0, 200)
plt.ylim(1e4, 5e5)
plt.xlabel('Drift time (us)')
plt.ylabel('S2 (pe in bottom array)')
plt.title('Figure 1: A first look at the data')
plt.show()
```

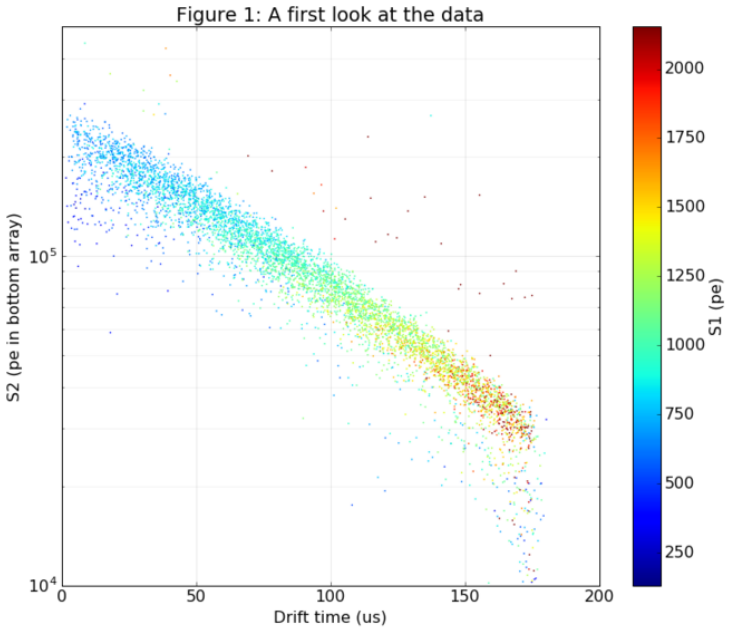

Figure 6: HAX in action: IPython notebook demonstrating how to: (1) create a mini-tree containing selection of variables in the dataset; (2) load the data in a pandas DataFrame; (3) perform exploratory visual analysis with matplotlib

6. Acknowledgments

This work was supported by a pathfinder project in collaboration with the Netherlands eScience Center. The authors would like to thank the XENON, PyROOT, and rootpy community for their valuable contribution and impact on this work.

References

[1] Aprile E *et al.* 2012 *Astroparticle Physics* **35** 573–590 ISSN 0927-6505 URL http://www.sciencedirect.com/science/article/pii/S0927650512000059

[2] Aprile E, Alfonsi M, Arisaka K, Arneodo F, Balan C, Baudis L, Behrens A, Beltrame P, Bokeloh K, Brown E *et al.* 2014 *Astroparticle Physics* **54** 11–24

[3] Aprile E 2013 The XENON1T dark matter search experiment *Sources and Detection of Dark Matter and Dark Energy in the Universe* (Springer) pp 93–96

[4] Solem A 2014 Celery: Distributed Task Queue

[5] Smirnova O, Brun R, Cailliau R, Carminati F and Elmer P 2014 *Journal of Physics: Conference Series* **513** 052033 URL http://stacks.iop.org/1742-6596/513/i=5/a=052033

[6] Tsaregorodtsev A, Bargiotti M, Brook N, Ramo A C, Castellani G, Charpentier P, Cioffi C, Closier J, Diaz R G, Kuznetsov G, Li Y Y, Nandakumar R, Paterson S, Santinelli R, Smith A C, Miguelez M S and Jimenez S G 2008 *Journal of Physics: Conference Series* **119** 062048 URL http://stacks.iop.org/1742-6596/119/i=6/a=062048

[7] Agostinelli S, Allison J, Amako K a, Apostolakis J, Araujo H, Arce P, Asai M, Axen D, Banerjee S, Barrand G *et al.* 2003 *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **506** 250–303

[8] Borodin M, De K, Jha S, Golubkov D, Klimentov A, Maeno T, Nilsson P, Oleynik D, Panitkin S, Petrosyan A, Schovancova J, Vaniachine A and Wenaus T 2014 PanDA Beyond ATLAS : A Scalable Workload Management System For Data Intensive Science Tech. rep. ATLAS Collaboration, CERN URL https://cds.cern.ch/record/1670021

[9] Continuum Analytics Odo: Shapeshifting for your data http://odo.pydata.org

[10] Canal P, Bockelman B and Brun R 2011 *Journal of Physics: Conference Series* **331** 042005 URL http://stacks.iop.org/1742-6596/331/i=4/a=042005

[11] Lavrijsen W 2015 *Journal of Physics: Conference Series* **664** 062029 URL http://stacks.iop.org/1742-6596/664/i=6/a=062029

[12] CERN Central Issue Tracking Service https://sft.its.cern.ch/jira/browse/ROOT-7854

[13] Dawe N, Waller P, Friis E K, Deil C, schmitts, Turra R, Klukas J, Stevenson S, Buat Q, chrisboo, Alessandro, Kreczko L, Hegeman J, Verzetti M and Hollingsworth M 2015 rootpy: 0.7.0 URL http://dx.doi.org/10.5281/zenodo.18815

[14] Dawe N, Ongmongkolkul P, Deil C, Stark G, Waller P, Howard J and Babuschkin I 2015 root_numpy: 4.4.0 URL http://dx.doi.org/10.5281/zenodo.31192

[15] Babuschkin I, Remenska D, Nöthe M and Pearce A 2016 root_pandas: Initial release URL http://dx.doi.org/10.5281/zenodo.45464

[16] Continuum Analytics Anaconda: a free Python distribution for scientific Big Data analysis https://www.continuum.io/why-anaconda

[17] Continuum Analytics Conda: open source package and environment management system http://conda.pydata.org/

[18] Remenska D 2016 root-conda-recipes: Initial release URL http://dx.doi.org/10.5281/zenodo.47512