PROCEEDINGS OF SCIENCE

ctapipe: a low-level data processing framework for the Cherenkov Telescope Array

K. Kosack, M. Peresano*for the CTA Consortium†
CEA Paris-Saclay Astrophysics Department
E-mail: karl.kosack@cea.fr

The low-level data processing of the Cherenkov Telescope Array (CTA) and indeed all other existing Cherenkov Telescopes can be broken into four general steps: 1) the processing of air-shower event image time-series, 2) the stereo reconstruction of the incident air showers, and 3) the discrimination of gamma-ray induced showers from those from cosmic rays 4) the determination of the overall system response. The final output for science users is a list of reconstructed gamma-ray-like events and their associated parameters, along with a set of instrumental response functions needed for doing astrophysics. We present a python-based framework, ctapipe, for writing the algorithms required for these processing steps along with a reference prototype pipeline. The code is written with a focus on simplicity and usability by developers with a diverse range of skill sets, and leverages existing code from the science community (AstroPy, SciPy/NumPy, SciKit-Learn, etc). This concept is intended to be a prototype for the final CTA low-level data processing pipeline, allowing physicists to quickly explore low-level Cherenkov telescope data and develop new algorithms. Thanks to the framework modularity, computer engineers and data scientists will be able to simultaneously optimize the algorithms and parallelize them using modern computing and big-data architectures to support the high data volumes of CTA.

36th International Cosmic Ray Conference -ICRC2019-
July 24th - August 1st, 2019
Madison, WI, U.S.A.

*Speaker.
†for collaboration list see PoS(ICRC2019)1177

PoS(ICRC2019)717

1. Introduction

The Cherenkov Telescope Array (CTA) Observatory will be a next-generation gamma ray *Imaging Atmospheric Cherenkov Telescope* (IACT) consisting of telescope arrays in two sites. The baseline design consists of 99 telescopes of 3 sizes (small, medium, and large) in the Southern Hemisphere, located in Paranal, Chile, and 29 telescopes of medium and large size in the Northern Hemisphere, located on La Palma in the Canary Islands. Each array operates as a single gamma-ray detector, or may be split into smaller sub-arrays depending on the science case.

In IACTs, individual gamma-ray photons are detected by imaging the UV/blue Cherenkov light emitted by secondary particles in extensive air showers that are created when a high-energy gamma-ray hits Earth's atmosphere. The sequence of images of each shower and its time-development are used to reconstruct the likely point-of-origin of the incident particle via stereoscopy, and also to distinguish morphologically between showers originating from gamma rays vs cosmic rays, which also produce similar air showers and have a rate typically 10^4 to 10^5 times larger than gamma rays. The arrays use a multi-level *trigger* system that is able to allow each telescope's Cherenkov camera to be read out only when a shower-like event has been detected (an "event"), with a resulting event trigger rate on the order of 10,000 events per second. The raw data from such an instrument for each trigger event consists therefore of what are essentially short, low-resolution movies of the Cherenkov light time-development of each air shower, taken at up to GHz frame rates from multiple view points.

The first level of *science data* that will be provided to users of the observatory will include two main products: *event-lists* and *instrumental response functions* (IRFs). Event lists are tables of reconstructed information about each candidate gamma-ray event detected by CTA including such parameters as the estimated energy, point-of-origin on the sky in ICRS coordinates, and a measure of the reconstruction quality, and likelihood of being a gamma-ray vs a background cosmic ray. It should be noted that even after the gamma-hadron discrimination step describe above, IACT event data still include an irreducible residual background from gamma-like cosmic rays and electrons, and thus gamma rays may only be described as statistical quantities (hence why these are called "event lists" and not "photon lists"). The IRFs consist of a set of matrices that allow one to transform from physical quantities to detector quantities, and include a measure of the effective gamma-ray collection area, the estimated background rate, the point-spread function, and the reconstructed energy dispersion.

In this proceeding, we describe the design for a prototype software framework intended for implementing a pipeline that processes CTA data from raw to this first-level science data.

1.1 Data Processing Algorithms

The data processing algorithms needed to reconstruct a single air shower into gamma-ray parameters include:

- **Image processing**: calibration (for example flat-fielding and noise-level subtraction), waveform peak detection and integration, image de-noising, inpainting, detection of image features (e.g. Hillas, muon, or time evolution parameters), all on both hexagonal and square pixel grids.

- **Reconstruction**: 3D plane intersection, transformation of coordinates from camera to horizon frames (Alt/Az), multi-dimensional interpolation, astrometric corrections, machine learning regression.
- **Event Discrimination and Classification**: machine learning classification, possibly including deep, recurrent, or convolutional networks.

1.2 Challenges and Constraints

CTA will face challenges that were not as critical in previous IACTs.

- CTA will produce vastly more data, resulting in the order of 10 PB/year of raw data after a combination of lossy and lossless compression. It will additionally generate a similar volume of simulated data, needed to characterize the instrumental response for science results. This implies that the code must run in parallel on, in principle, multiple data centers and possible be able to use modern computing technologies.
- CTA will be an open observatory catering to the wide scientific community over a lifetime of 30 years. The code used for analysis must therefore be robust, easy to understand, and maintainable for at least 10 years without replacement.
- CTA will be an observatory that serves users, with strict requirements to provide high-quality, verified science data to the public that are suitable for all science cases without requiring the users to process low-level raw data themselves.
- CTA will have official data releases, with previous data re-released after being re-processed with new algorithms, updated simulations, and better calibration.
- CTA software will be maintained by the observatory staff, and not necessarily by scientists (who may not be available for consultation for the full lifetime of the instrument). However, due to the complexity of the analysis, scientists must be heavily involved in the software development. This implies the pipeline code must be accessible to people who are not computer scientists.
- the CTA instrument itself is more complex than past IACTs, with at least three telescope optics and camera designs (for example see Figure 1.2), the choice of multiple sub-arrays, and varying atmospheric conditions at two sites with different array layouts. This means that the software must treat complex differences in telescope technologies in a generic way.

1.3 Lessons Learned

A study was done early on to extract both good and bad experience from users and developers of the frameworks from previous IACTs and similar telescopes. These included MARS[21] (from MAGIC), EventDisplay[17] and VEGAS[10] (from VERITAS), SASH [14, e.g.] (from HESS), as well as frameworks from non-IACTS like IceTray[11] (Ice Cube, ANTARES), and the Fermi-LAT Data Processing Pipeline[22]. In all cases, the previous reconstruction pipeline frameworks were written in C++ and based on the CERN ROOT framework. The main lessons learned were:

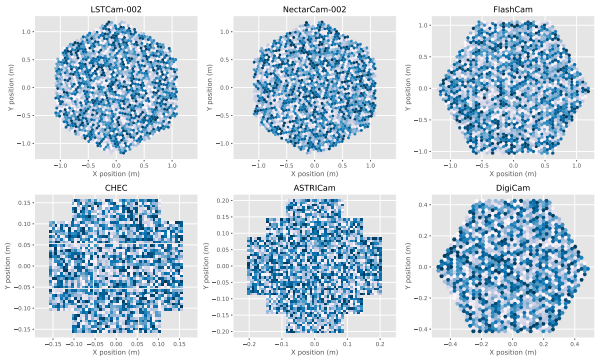

Figure 1: Examples of different camera geometries used in current CTA prototype telescopes. Each camera has differing physical size, pixel geometry (both square and hexagonal), and readout characteristics. The images shown are random noise.

- Physicists want to be able to contribute to the analysis and demand a small learning curve for inexperienced developers (who may be post-docs or PhD students that participate for only a short time). It should be possible to involve expert computer scientists as well for e.g. optimization without making the code unusable by non-experts.
- It is important that it is easy to play with data and explore algorithms interactively, even if this is not the final mode of operation
- Reproducablity of results needs to be considered in the design.
- Software should be written in a way that makes it possible to leverage new technologies like high-performance computing or parallelization, even those that arise after the software has been initially designed.
- Too heavy a framework (even if very clever) is confusing to users and complicates maintenance.
- Leverage the wider astronomy community (and not as much particle physics), particularly existing open source astronomical software rather than "rolling your own".
- Make it friendly : Rich visualizations , tutorials, notebooks, easy to discover and explore
- Use standards and open tools (minimize custom code)
- Don't be too clever with how algorithms are chained together: can be confusing to users, difficult to debug, and you can achieve the same thing later by wrapping in a big-data framework (spark, celery, etc)
- ROOT, while powerful, suffered from poor API design and presented a high barrier-of-entry for new users. Since it also lacks a well-maintained standard astronomy library, it was not recommended as a basis for a new astronomical observatory framework.

- C++ is a great language, but is sometimes too flexible, complex, and low-level, making it too easy for unskilled developers to make mistakes or write unreadable or un-maintainable code. Furthermore the vast majority of code in astronomical data processing frameworks does not benefit strongly from speed or hardware-level optimizations provided by languages like C/C++. Nevertheless, allowing some fraction of code to be written in C++ is helpful for high-performance optimization.

2. Framework Design

The ctapipe[3] package was created as a basis for prototyping a full pipeline to process CTA raw data, starting with simulations and also supporting data from prototype telescopes. It contains both core data structures and algorithms needed for such a pipeline. The framework and its design are still rapidly evolving, and this represents only the current status. Based on the lessons learned and the basic requirements of CTA and its developers and users, the following design choices were made:

Framework Language Python (v3.6 and above) was chosen as the core language of ctapipe due to its adoption by the wider scientific community, ease of use, and vast library of existing code for everything from data access to astronomical and statistical calculations. The core dependencies (other than the python standard library) are essentially the standard science and astronomy stack: *NumPy/SciPy*[19, 13] (for numerics, histogramming, fitting, interpolation, integration, signal processing and linear algebra), *AstroPy* (particularly for coordinates, units, table access), *Matplotlib*[12] (for visualization and plotting), *Scikit-Learn*[20] (for classical machine learning), and Pandas[18] (for high-level data analysis). We also use an associated package *pyeventio*[7, 8] for access to simulation data.

High Performance Code and Optimization Clear and simple code is valued higher than high speed code in ctapipe, nevertheless since algorithms implemented in python (and in particular the CPython implementation used) can be very slow compared with lower-level compiled languages like C++, it was identified early on that there is a need to identify and optimize certain parts of the pipeline that are speed bottlenecks. Python provides many friendly tools for code performance monitoring (like *cProfile*, *snakeviz*, *line-profiler*) that can be used to identify the minimal amount of code that needs optimization. The following technologies were then used to write faster code (in order of importance):

- *NumPy/SciPy*: the use of `numpy.ndarray` data structures and associated operations rather than explicit for-loops provides in general a reasonable level of speed, particularly when optimized versions of *NumPy* are used. This also has the effect of making code smaller and easier to read.

- *Numba*[16] for code where for-loops are needed or where complex NumPy operations result in a slow-down due to many temporary arrays being created, Numba's `@jit` decorator allows for code that has similar speed to hand-optimized C to be written without sacrificing simplicity or the need to write external code in another language. It supports automatic loop parallelization and even GPU output (though the latter is so far not used by ctapipe).

- *Cython*. Cython is a meta language that provides a easy way to hand-optimize c-like code with a python syntax. It allowed the *pyeventio* project (used to read current-generation simulation data) to achieve speeds similar to a pure C implementation while maintaining python-like code.
- *wrapped C/C++ libraries* In the case where none of the above are sufficient or where a common library must be shared with other non-Python code, providing well-written wrappers for C/C++ code allows a clean separation between low-level optimizations and the needs of high-level users.

Pipelineing of algorithms and parallelization The frameworks allows algorithms to be pipelined at several levels: algorithms can be called for each event (or on data within an event), grouped into higher-order algorithms, or combined into a command-line application (`Tool` for batch processing). The I/O layer allows for data at any stage to be serialized to disk and read back later for continued processing.

The framework intentionally does not enforce a specific parallelization framework, but is designed with parallelization down to the event-level in mind. For example the internal data structures and algorithm steps are designed to be easily serializable and thus may be parallelized using the standard library `concurrent.futures`, or with frameworks like *dask*, *Apache Spark*, as well as simple batch processing.

I/O layer Since current CTA prototype telescopes and simulations each output data in different formats, the need to develop a standardized IO system with back-end plugins for each format was recognized early in the development. ctapipe provides plug-in `EventSources` supporting all currently used raw and simulated data formats, plus a set of hierarchical classes that describe the internal data model, based on custom `Container` classes that store data items (`Field` names) along with their metadata such as the unit, description, default value, and uniform content descriptor (UCD). These classes are used to exchange information in memory within an application, and also can be used like an object-relational mapper with a `TableWriter` class that uses the metadata to serialize them to and from output formats (so far only HDF5 tables in pytables format are supported for output, but others can be easily implemented). This allows for writing each `Container` to a row of an output table. Similarly, a `TableReader` class is provided to read back rows of a table into `Containers`. `Containers` can also be converted to and from Python *dict*s.

Instrumental Description The instrumental data model is described a set of `InstrumentDescription` classes that allow the details of the hardware to be captured. It supports all of the differences in hardware of current CTA prototype hardware as well as for some other Cherenkov telescopes. This includes descriptions of a (sub)array, a telescope, and the associated Cherenkov camera and optics structure. This model is automatically built when reading simulation data, or can be read or written from/to `astropy.table.Table` objects or any data file format they support (e.g. FITS, text, HDF5).

Configuration and Provenance Tracking ctapipe provides a `Tool` base class to create command-line applications and a `Component` class to create algorithms or functions with user-configurable parameters. These class provides both a top-level configuration system supporting command-line

and text configuration files (based on the `traitlets.config` library), as well as logging setup and provenance tracking. Components that inherit from the same base class can additionally be configured by name using a *factory* design pattern. Provenance, the metadata that describes the inputs and outputs to a particular "activity" is tracked automatically, storing for each activity (application) a unique identifier along with the filename of all files opened or written, and details of the computing environment on which the application ran. This information is output as in JSON format, and can be later read into a database for tracking a full chain of data processing.

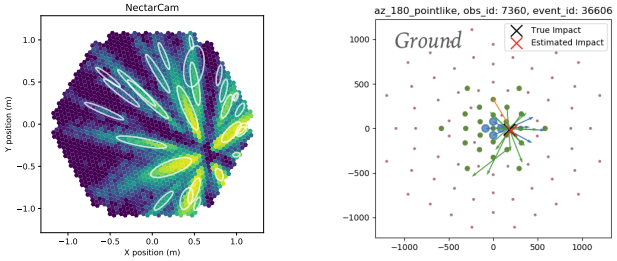

Figure 2: Example of visualizing the shower point-of-origin reconstruction viewed from multiple cameras (left) and the shower ground-impact reconstruction (right)

Visualization Rich and interactive visualization of things like Cherenkov camera images, image parameterizations, 3-D reconstructions, histograms and profiles, etc, are critical for writing and debugging good scientific code. ctapipe provides functionality based on matplotlib to generate some common visualizations, like interactive camera displays and reconstruction plots (an example is shown in Figure 2). The *Bokeh*[9] library is also used for some experimental web displays.

Development Methodology ctapipe is developed using modern development practices using GitHub as a repository, TravisCI[6] for continuous integration, *Coverage.io*[2] for unit-test coverage, *Codacy*[1] for code quality checking, and *Slack* for team chat. We require at least 2 independent reviewers as well as all automated checks to pass in order to accept any pull request added to the main code base. Unit tests are required for all code added, and are updated to avoid regression of bugs. This methodology has greatly improved the quality of code, contributions, and interaction between developers. Documentation is built automatically along with the unit tests, using *Sphinx*[5], and we use *nbsphinx*[4] to be able to run and include *Jupyter*[15] notebooks directly in the documentation[3].

3. Conclusions from Prototype

Overall, ctapipe has been a successful prototype and is a strong candidate to be the basis of the final CTA low-level pipeline. The use of Python and standard scientific libraries greatly reduced the complexity of algorithms, increased code readability, and lowered the barrier to entry for users. The negative aspects of Python such as slow speed and lack of a true compiler are mitigated with several technologies as well as by the heavy use of unit and regression tests. It contains code contributions from over 35 developers to date, and the total code base including all algorithms is only 15,000 lines of code (one third of which are unit tests), rather small compared with previous implementations, due to use of code from the scientific community. The Analysis and Simulations Working group is

currently using it to prototype a full reconstruction pipeline to help with CTA design studies and it will soon replace the modified MAGIC and VERITAS pipelines currently being used for this purpose.

Acknowledgements This work has been conducted in the context of the CTA Analysis and Simulations Working Group. We gratefully acknowledge financial support from the agencies and organizations listed here: http://www.cta-observatory.org/consortium_acknowledgments

This paper has gone through internal review by the CTA Consortium.

References

[1] Codacy. URL: https://codacy.com.

[2] coverage.io. URL: https://codecov.io/.

[3] ctapipe: Low-level data processing prototype for cta. Accessed: 2019-06-27. URL: http://github.com/cta-observatory/ctapipe.

[4] nbsphinx: Jupyter notebook tools for sphinx. URL: https://nbsphinx.readthedocs.iob/.

[5] Sphinx python documentation generator. URL: http://www.sphinx-doc.org/.

[6] Travisci continuous integration. URL: https://travis-ci.org/dashboard.

[7] Pyeventio: Python read-only implementation for the eventio data format, 2019–. URL: https://github.com/cta-observatory/pyeventio.

[8] K. Bernlöhr. eventio: a machine-independent hierarchical data format and its programming interface, 2014. URL: https://github.com/cta-observatory/pyeventio.

[9] Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2014. URL: http://www.bokeh.pydata.org.

[10] P. Cogan. VEGAS, the VERITAS Gamma-ray Analysis Suite. *International Cosmic Ray Conference*, 3:1385–1388, Jan 2008. arXiv:0709.4233.

[11] T. DeYoung. IceTray: A software framework for IceCube. In *Computing in high energy physics and nuclear physics. Proceedings, Conference, CHEP'04, Interlaken, Switzerland, September 27-October 1, 2004*, pages 463–466, 2005. URL: http://doc.cern.ch/yellowrep/2005/2005-002/p463.pdf.

[12] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. URL: http://matplotlib.org/.

[13] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL: http://www.scipy.org/.

[14] Bruno Khelifi, Arache Djannati-Ataï, Lea Jouvin, Julien Lefaucheur, Anne Lemiere, Santiago Pita, Thomas Tavernier, and Regis Terrier. HAP-Fr, a pipeline of data analysis for the HESS-II experiment. *PoS*, ICRC2015:837, 2016. doi:10.22323/1.236.0837.

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[16] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[17] G. Maier and J. Holder. Eventdisplay: An Analysis and Reconstruction Package for Ground-based Gamma-ray Astronomy. *International Cosmic Ray Conference*, 301:747, Jan 2017. arXiv:1708.04048.

[18] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[19] Travis E. Oliphant. NumPy: Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. URL: http://www.numpy.org/.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Roberta Zanin. MARS, the MAGIC analysis and reconstruction software. In *Proceedings, 33rd International Cosmic Ray Conference (ICRC2013): Rio de Janeiro, Brazil, July 2-9, 2013*, page 0773. URL: http://www.cbpf.br/%7Eicrc2013/papers/icrc2013-0773.pdf.

[22] S. Zimmer, T. Johnsson, T. Glanzmann, C. Lavalley, L. Arrabito, and A. Tsaregorodtsev. The Fermi-LAT Dataprocessing Pipeline. Computing in High Energy and Nuclear Physics (CHEP2012), May 2012. Poster. URL: http://hal.in2p3.fr/in2p3-00703727.