Dissertation
submitted to the
Combined Faculties for the Natural Sciences and for Mathematics
of the
Ruperto-Carola University of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

Put forward by
Ramin Marx
Born in: Flensburg

Algorithms for Imaging Atmospheric Cherenkov Telescopes

Referees: **Prof. Dr. Peter Fischer**
Prof. Dr. Werner Hofmann

Kurzfassung

Abbildende atmospärische Cherenkov-Teleskope (IACTs) sind komplexe Instrumente für die bodengebundene γ-Strahlenastronomie und erfordern eine anspruchsvolle Software für die Verarbeitung der Messdaten.

Im ersten Teil dieser Arbeit wird ein modulares und effizientes Software-Framework vorgestellt, das es erlaubt, die komplette Kette vom Einlesen der Rohdaten der Teleskopen, über Kalibration, Hintergrundreduktion und Rekonstruktion, bis hin zu den Skymaps. Mehrere neue Methoden und schnelle Algorithmen wurden entwickelt und werden vorgestellt.

Darüber hinaus wurde festgestellt, dass die derzeit in den IACT-Experimenten verwendeten Dateiformate in Bezug auf Flexibilität und I/O-Geschwindigkeit nicht optimal sind. Daher wurde im zweiten Teil ein neues Dateiformat entwickelt, das es erlaubt, die Kamera- und Subsystemdaten in ihrer ganzen Komplexität zu speichern. Es bietet eine schnelle verlustbehaftete und verlustfreie Kompression, die für die hohen Datenraten der IACT-Experimente optimiert ist.

Da auch viele andere wissenschaftliche Experimente mit enormen Datenraten zu kämpfen haben, wurde der Kompressionsalgorithmus weiter optimiert und verallgemeinert, und ist nun in der Lage, auch die Daten anderer Experimente effizient zu komprimieren.

Schließlich wird für diejenigen, die es vorziehen, ihre Daten als ASCII-Text zu speichern, ein schnelles I/O-Schema vorgestellt, einschließlich der notwendigen Kompressions- und Konvertierungsroutinen.

Obwohl der zweite Teil dieser Arbeit sehr technisch ist, könnte er dennoch für Wissenschaftler interessant sein, die ein Experiment mit hohen Datenraten planen.

Abstract

Imaging Atmospheric Cherenkov Telescopes (IACTs) are complex instruments for ground-based γ-ray astronomy and require sophisticated software for the handling of the measured data.

In part one of this work, a modular and efficient software framework is presented that allows to run the complete chain from reading the raw data from the telescopes, over calibration, background reduction and reconstruction, to the sky maps. Several new methods and fast algorithms have been developed and are presented.

Furthermore, it was found that the currently used file formats in IACT experiments are not optimal in terms of flexibility and I/O speed. Therefore, in part two a new file format was developed, which allows to store the camera and subsystem data in all its complexity. It offers fast lossy and lossless compression optimized for the high data rates of IACT experiments.

Since many other scientific experiments also struggle with enormous data rates, the compression algorithm was further optimized and generalized, and is now able to efficiently compress the data of other experiments as well.

Finally, for those who prefer to store their data as ASCII text, a fast I/O scheme is presented, including the necessary compression and conversion routines.

Although the second part of this thesis is very technical, it might still be interesting for scientists designing an experiment with high data rates.

Contents

I. Algorithms for the Calibration and Analysis of γ-ray Data taken with IACTs

1. Overview of Ground-Based γ-ray Astronomy **11**
- 1.1. History . . . 11
- 1.2. Origins of VHE γ-rays . . . 12
- 1.3. Air Showers and Cherenkov Radiation . . . 15
- 1.4. Water Cherenkov Detectors . . . 16
- 1.5. Imaging Atmospheric Cherenkov Telescopes (IACTs) . . . 17

2. HESS and CTA **23**
- 2.1. HESS . . . 23
  - 2.1.1. HESS camera data . . . 24
- 2.2. CTA . . . 25
  - 2.2.1. CTA camera data . . . 26

3. The MESS Software Framework for Data Processing in γ-ray Astronomy **29**
- 3.1. Developer Tools . . . 29
  - 3.1.1. Automatic header file generation . . . 29
  - 3.1.2. Documentation generation . . . 29
  - 3.1.3. Parameter parsing . . . 29
  - 3.1.4. Logging . . . 30
  - 3.1.5. Enum-to-string conversion . . . 30
  - 3.1.6. Code statistics . . . 30
  - 3.1.7. Automatic versioning . . . 30
  - 3.1.8. Build system . . . 30
- 3.2. Module system for the on-the-fly creation of analysis pipelines . . . 31
  - 3.2.1. Description of modules and pipelines . . . 31
  - 3.2.2. Command line interface . . . 32
  - 3.2.3. No init scripts or environment variables . . . 32
- 3.3. Message passing . . . 33
- 3.4. Examples of MESS pipelines . . . 34
- 3.5. Data Structures . . . 36
  - 3.5.1. Messages . . . 36
  - 3.5.2. Telarray . . . 36
  - 3.5.3. Telescope . . . 37
  - 3.5.4. Camera . . . 37
  - 3.5.5. Events . . . 37
  - 3.5.6. Parameter sets . . . 38
  - 3.5.7. Histograms . . . 38
  - 3.5.8. Time and IDs . . . 42
- 3.6. Non-pipelined programming with MESS libraries . . . 43
- 3.7. Miscellaneous . . . 44
  - 3.7.1. Creating a module . . . 44
  - 3.7.2. Dependencies . . . 44
  - 3.7.3. Memory usage . . . 44
  - 3.7.4. Python . . . 44

3.7.5. Parameter parsing . . . 45
3.7.6. Advanced Vector Extensions 2 . . . 45
3.8. Components of MESS . . . 47
3.8.1. List of 50 selected modules . . . 47
3.8.2. List of selected MESS programs . . . 49
3.9. Summary . . . 49

4. The MES File Format for IACT Data **51**
4.1. Description of the MES File Format . . . 51
4.1.1. Header . . . 51
4.1.2. Camera data . . . 51
4.1.3. Parameter sets . . . 52
4.1.4. Histogram sets . . . 53
4.1.5. Other data . . . 53
4.1.6. Synchronization . . . 53
4.1.7. Versioning . . . 53
4.1.8. Message records . . . 53
4.2. Built-in Compression Algorithms for Camera Raw Data . . . 55
4.3. Built-in Quantization and Compression for Derived Pixel Values . . . 55
4.3.1. Compression ratio for calibrated HESS data . . . 57
4.3.2. Reduced resolution vs. reconstruction error . . . 57
4.3.3. Summary . . . 57
4.4. Regions of Interest in IACT Camera Images . . . 59
4.4.1. Benchmark . . . 60
4.4.2. Summary . . . 60
4.5. Reading and Synchronization . . . 61

5. Data Quality **63**
5.1. Run Selection . . . 63
5.2. Data Quality Explorer . . . 63
5.3. Broken Pixels . . . 65
5.3.1. Broken pixel cut on mean reconstruction error . . . 69
5.3.2. Weighting broken pixel patterns during reconstruction . . . 70
5.4. Summary . . . 71

6. Calibration **73**
6.1. Statistical Methods . . . 73
6.1.1. Robust statistics . . . 73
6.2. ADC-to-PE Ratio . . . 74
6.3. Online pedestal estimation . . . 74
6.3.1. Known algorithms for online pedestal estimation . . . 75
6.3.2. Robust adaptive baseline estimation (RABE) . . . 79
6.3.3. Overshooting and the final RABE algorithm . . . 80
6.3.4. Conclusion . . . 83
6.4. NSB level estimation . . . 84
6.5. Broken pixel (per gain channel) identification . . . 85
6.5.1. Too often no signal . . . 86
6.5.2. Unusual differences in time . . . 86
6.5.3. Signal not often enough . . . 87
6.5.4. Unusual standard deviations . . . 87
6.5.5. Pixels with no signal that have pixels with signal as neighbors . . . 87
6.6. High Gain/Low Gain Ratio and Conversion to Intensities . . . 88
6.7. Comparison to *sim_telarray* calibration . . . 90

6.8. Comparison to *HESS* calibration . . . 92
6.8.1. Number of events . . . 92
6.9. Interpolation . . . 92
6.10. Normalization with Broken Pixel Parameters of Monte Carlo Simulations . . . 93
6.11. Applying Calibration Parameters to Monte Carlo Data . . . 94
6.12. HESS Calibration . . . 94
6.13. Summary . . . 96

7. Analysis **97**
7.1. Waveform Analysis Algorithm . . . 97
7.2. Image Cleaning . . . 100
7.3. Hillas Analysis . . . 101
7.3.1. Square grid . . . 101
7.4. γ/Hadron Separation . . . 102
7.5. Direction Reconstruction . . . 102
7.6. Monte Carlo Simulations . . . 102
7.7. Complete Analysis Pipeline . . . 102
7.8. Results . . . 103
7.9. Summary . . . 104

8. Online Analysis and Data Reduction **107**
8.1. Online Analysis . . . 107
8.1.1. Example pipeline . . . 107
8.1.2. Results . . . 108
8.2. Data Reduction . . . 110
8.2.1. Results . . . 110
8.3. Summary . . . 111

9. γ/Hadron Separation **113**
9.1. Introduction . . . 113
9.2. Input data . . . 113
9.3. Hillas parameters . . . 118
9.4. Performance curves . . . 119
9.5. Benchmark . . . 119
9.6. Hillas analysis with different image cleanings . . . 121
9.7. Projection along the Hillas axes . . . 123
9.8. Projection along the Hillas axes plus Hillas parameters . . . 126
9.9. Lorentzian fit . . . 128
9.10. Geometrical and histogram features . . . 131
9.10.1. Distances to Hillas axes . . . 131
9.10.2. Other distance related parameters . . . 133
9.10.3. Neighborhood related parameters . . . 134
9.10.4. Convex hull . . . 136
9.10.5. Segmentation related parameters . . . 136
9.10.6. Histogram related methods . . . 138
9.11. Correlation matrix . . . 142
9.12. All methods combined . . . 147
9.13. Calculating moments in each bin . . . 148
9.14. Deep Neural Networks . . . 151
9.14.1. Training procedure . . . 151
9.14.2. Network architecture . . . 151
9.14.3. Results . . . 152
9.15. Summary . . . 153

9.16. Appendix . . . 154
9.16.1. Hillas parameters . . . 154
9.16.2. Lorentzian fit errors . . . 155
9.16.3. Distances to Hillas axes . . . 156
9.16.4. Distances to Hillas axes . . . 157
9.16.5. Other distances . . . 158
9.16.6. Neighborhood . . . 159
9.16.7. Convex hull . . . 160
9.16.8. Segmentation related parameters . . . 161
9.16.9. Histogram related parameters . . . 162
9.16.10. Density and entropy . . . 163

II. Algorithms for the Compression of Scientific Data 165

10. Raw Data of Scientific Experiments 167
10.0.1. GERDA . . . 167
10.0.2. FlashCam . . . 167
10.0.3. CHEC . . . 168
10.0.4. HAWC . . . 168
10.0.5. Monte Carlo (MC) Simulations . . . 168
10.1. Advantages of compression . . . 169

11. Fast Lossless Compression of 16 bit Time-Sampled Waveforms 171
11.1. Storing Blocks with Reduced Range of 4, 8 or 16 bits . . . 171
11.1.1. Benchmark . . . 171
11.2. Integer Wavelet Transform with Fixed Huffman Residual Coding . . . 174
11.2.1. Differential coding . . . 174
11.2.2. Integer Wavelet Transform . . . 174
11.2.3. Summary . . . 177
11.3. Morphological Wavelet Transform and Residual Range Encoding . . . 179
11.3.1. Algorithm . . . 179
11.3.2. Mathematical morphology . . . 179
11.3.3. Morphological Wavelet Transform . . . 180
11.3.4. 4-6-8-16 Bitmask Encoding . . . 181
11.3.5. Summary . . . 182
11.4. `fc16` - Morphological Wavelet Transform with Dynamic Bit Range Residual Encoding . . . 183
11.4.1. Dynamic encoding of residuals of the Wavelet transform . . . 183
11.4.2. Benchmark . . . 185
11.4.3. Results . . . 186
11.4.4. Usage . . . 190
11.4.5. Summary . . . 190
11.4.6. Outlook . . . 190
11.4.7. Update . . . 190

12. Fast Reading and Writing of Text Files Containing Numbers 191
12.1. Introduction . . . 191
12.2. Reduced Character Set . . . 192
12.3. Fast Compression of Numbers Stored in ASCII Format . . . 193
12.3.1. `fc4` . . . 193
12.4. Converting Integer Numbers from Text to Binary . . . 194
12.4.1. `fatoi` . . . 194

12.5. Converting Floating Point Numbers from Text to Binary . . . 195
12.5.1. `fatof` . . . 195
12.6. Converting Integer Numbers from Binary to Text . . . 197
12.6.1. `printint` . . . 197
12.7. Summary . . . 197

13. Conclusion **199**
13.1. Impact on Data Acquisition and Storage . . . 199
13.1.1. Relevance for CTA . . . 199

A. Appendix **201**
A.1. Abbreviations and special terms . . . 201
A.2. Acknowlegdements . . . 205

Part I.

Algorithms for the Calibration and Analysis of γ-ray Data taken with IACTs

1. Overview of Ground-Based γ-ray Astronomy

This chapter gives a very short overview over the discovery and the origins of γ-rays and explains briefly how modern detectors work. The experiments HESS (High Energy Stereoscopic System)[11] and CTA (Cherenkov Telescope Array)[18] are explained in detail, because their data was analyzed and used to test the algorithms developed in this work.

1.1. History

Hundred years ago, very-high energy (VHE, $E > 100$ GeV) γ-rays were still unknown. Only γ-rays from radioactive decay with energies of some MeV had been detected, and x-rays with energies of several keV could be produced with x-ray tubes. After Victor Hess discovered cosmic rays in 1912[1], the scientific community became more and more interested in the non-thermal processes in- and outside the galaxy.
Because γ-rays cannot penetrate Earth's atmosphere, the scientists had to continue their research with balloon experiments and from the 1960s on with satellite experiments. Around 1990, Whipple and HEGRA detected several γ-ray sources and the successful era of ground-based γ-ray astronomy began. Thanks to the orders of magnitude larger collection areas as compared to satellite instruments, ground-based Cherenkov telescope arrays are able to detect TeV photons, which are so rare that they practically never hit the small detector of a satellite experiment.
Today, a large number of galactic and extragalactic sources have been detected that emit γ-rays with energies of several TeV [3].
A short timeline of relevant milestones is given below:

1912 Victor Hess discovers radiation coming from space (later called *cosmic rays*)
1938 Pierre Auger discovers extensive air showers[5]
1961 the γ-ray telescope on the Explorer 11 satellite detects cosmic γ-rays
1967 OSO-3 detects extragalactic γ-rays
1972 SAS-2 correlates γ-ray background with structural features of our Galaxy
1975 COS-B provides a more detailed γ-ray map of our Galaxy
1989 Whipple Observatory detects the Galactic source Crab Nebula[2]
1991 Compton Gamma Ray Observatory is launched
1992 Whipple Observatory detects the extragalactic source Markarian 421[4]
1996 HEGRA stereoscopic system first light
2002 HESS telescope system first light
2004 MAGIC telescope first light
2007 VERITAS telescope system, first light
2008 Fermi satellite is launched
2013 HAWC detector first light
2018 First CTA LST telescope deployed on La Palma

1.2. Origins of VHE γ-rays

Very high energy $\gamma-$ rays are secondary products created in interactions of very high energy particles – cosmic rays – with interstellar gas matter or with radiation fields. The large amounts of energy that are needed to create cosmic rays are believed to be provided by supernovae, rotating magnetic fields or accretion onto black holes.
In cosmic-ray interactions, γ-rays can be produced by different mechanisms:

Bremsstrahlung and Synchrotron Radiation When a moving charged particle is deflected by another charged particle or a magnetic field and loses kinetic energy, the energy is converted into a photon.
Inverse Compton Scattering When a low-energy photon is scattered on a light charged particle, some or most of the kinetic energy of the particle can be transferred to the photon.
Pion Production and Decay When two hadrons traveling with a high velocity collide, π^0 and other unstable particles can be produced; π^0's almost instantly decay into two γ-rays.
Annihilation or Decay of Dark Matter Particles Another mechanism for γ-ray emission is the decay or annihilation of heavy particles. WIMPs (Weakly Interacting Massive Particles, in the GeV to TeV mass range) have been proposed to explain the problem of dark matter, and they are believed to produce γ-rays during annihilation.

The following source classes are the prime candidates for observations:

Supernova Remnants (SNRs) After a star explodes in a supernova, some of its material is ejected light years into the interstellar environment. The resulting shock fronts move with several thousand kilometers per second and can accelerate electrons and nuclei close to the speed of light. According to leptonic models, γ-rays can then be produced by Inverse Compton scattering of infrared and microwave photons on the accelerated electrons. Hadronic models assume that γ-rays are created by pion production and decay. Which γ-ray emission mechanism dominates depends on the ratio of accelerated protons and electrons, on the density of target material, i.e. the ambient gas density, and the intensity of radiation fields.

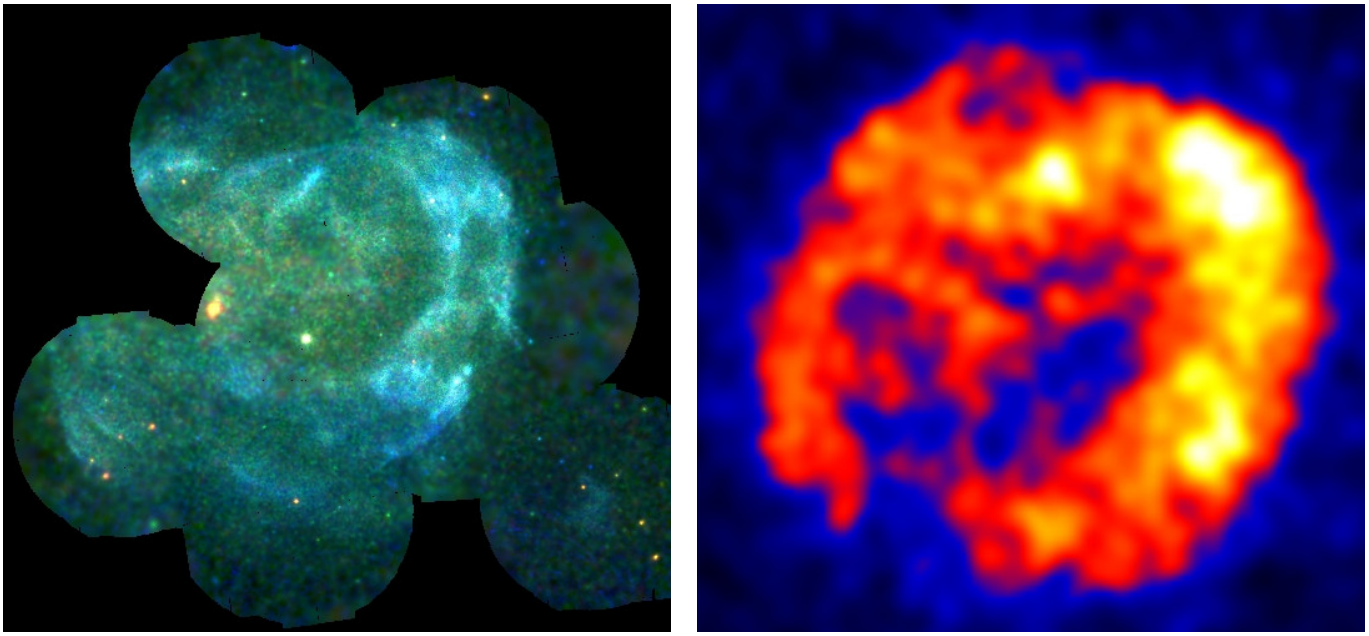


Fig. 1.1.: Left: SNR RXJ 1713-3941 seen with the XMM Newton x-ray satellite (credit: NASA/XMM). Right: RXJ 1713 seen with the HESS telescopes (credit: HESS collaboration).

Pulsars and Pulsar Wind Nebulae After a supernova, the remaining matter can collapse and form a neutron star with a diameter of few kilometers. Since the angular momentum must be conserved, the rotational velocity of the neutron star and its magnetic fields increases. The electric fields induced by the rotating magnetic field then accelerate charged particles from the surface of the neutron star to close to the speed of light. This pulsar wind is stopped in a termination shock, beyond with particles flow outward more slowly. These particles can then produce γ-rays when they are deflected in magnetic fields, or interact with matter or radiation fields. Interactions within the light cylinder, where fields co-rotate with the pulsar, result in pulsed emission; beyond the termination shock, the emission traces the pulsar wind nebula.

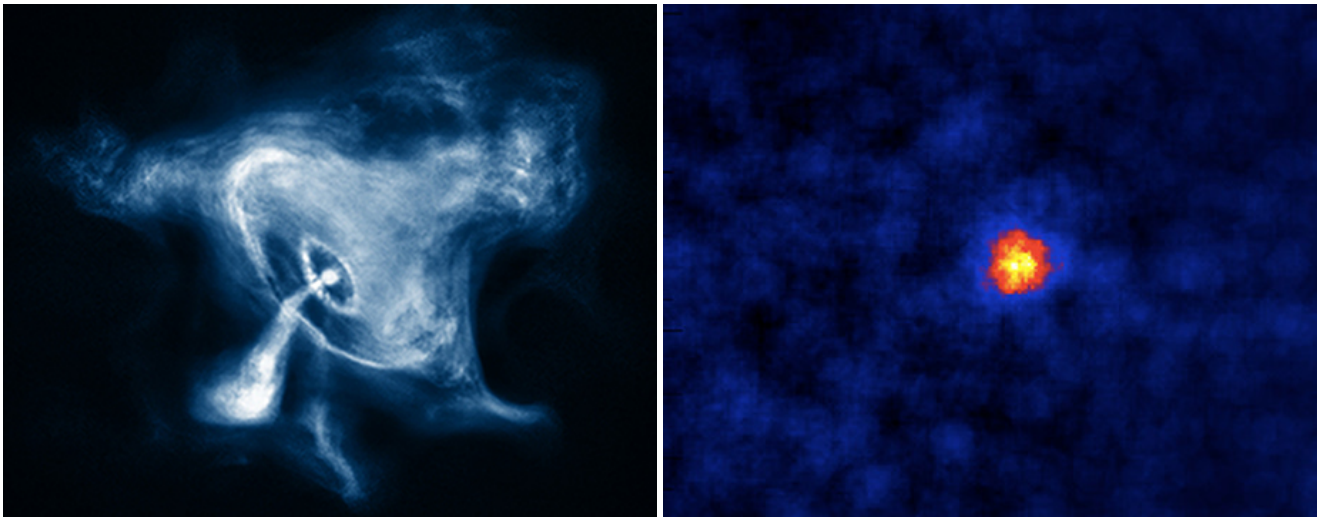


Fig. 1.2.: Left: The Crab Nebula observed with the Chandra x-ray satellite (credit: NASA/Chandra). Right: The Crab Nebula seen with the HESS telescopes (credit: HESS collaboration).

Binary Systems When two massive objects (e.g. two stars or a star and a neutron star) orbit each other, VHE radiation can be created when winds of two massive stars collide, forming a shock zone, or when accretion of stellar material on a compact companion object – a neutron star or black hole – results in the formation of jets which can induce shocks both along the jet and when terminating against the ambient pressure.

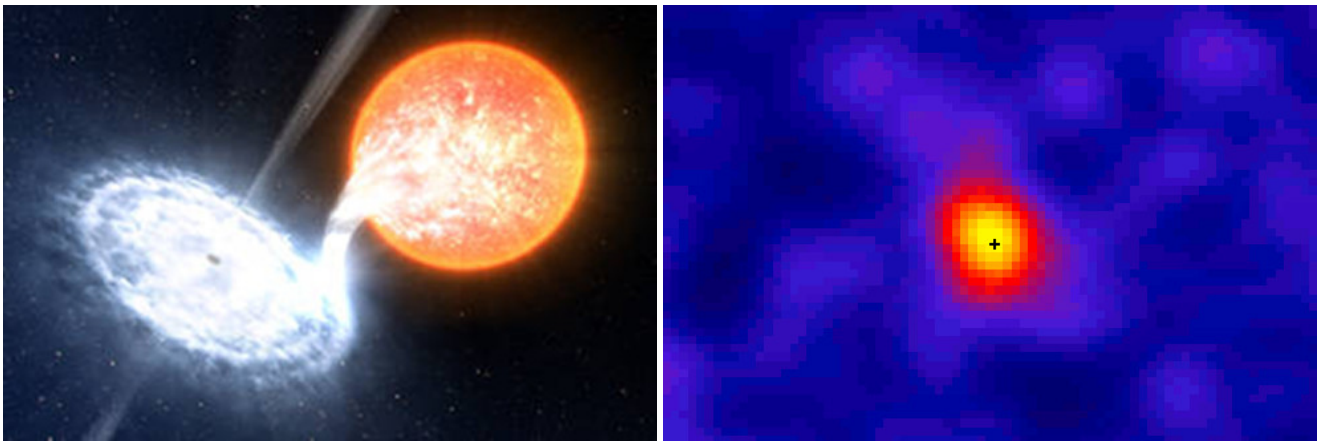


Fig. 1.3.: Left: Illustration of a star orbiting a light black hole with ten solar masses (credit: ESO/L.Calçada). Right: Observation of the binary system LMC P3 with HESS (credit HESS).

Active Galactic Nuclei (AGN) If the center of a galaxy contains a supermassive black hole ($m > 10^6$ solar masses), often an accretion disk around the black hole is formed. By mechanisms that are not fully understood – magnetic collimation is likely to play a role – part of the infalling matter is ejected in form of a jet and accelerated close to the speed of light. Beyond the accretion disks, also the rotational energy of the black hole can potentially serve as energy source. Several mechanisms of γ-ray production are possible, along the jet and it its termination shock.

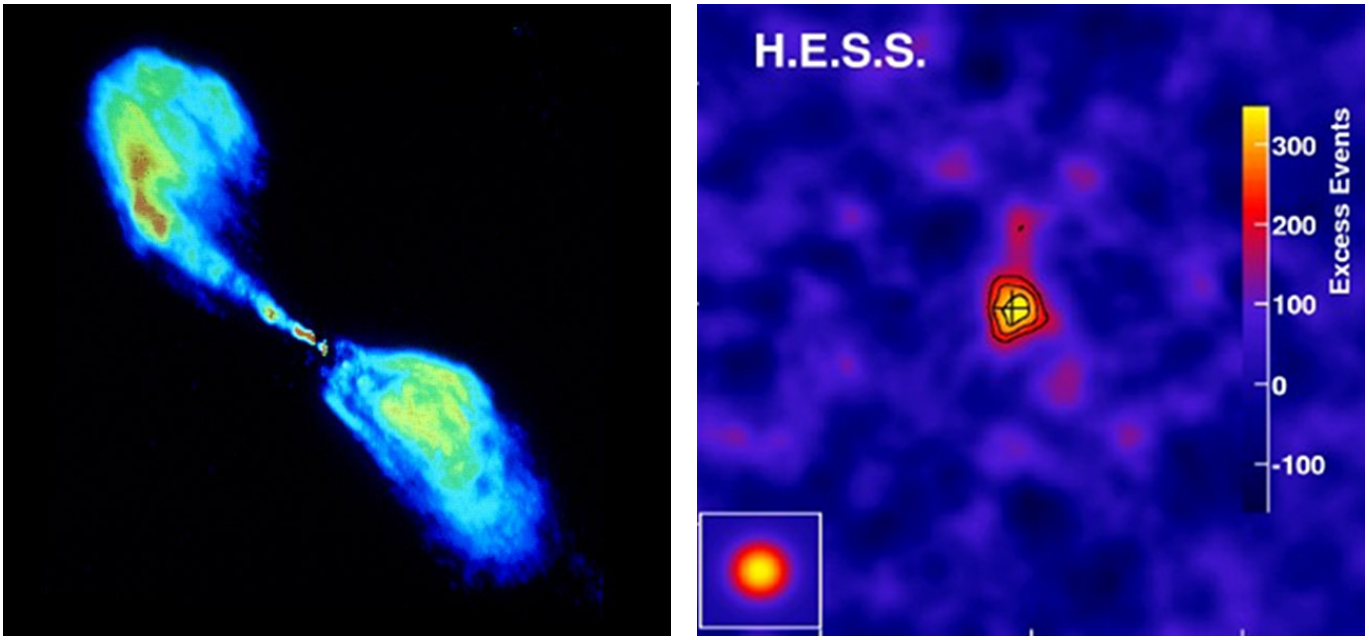


Fig. 1.4.: Left: Radio Galaxy Centaurus A seen with the Very Large Array (VLA) (credit: VLA). Right: Centaurus A seen with the HESS telescopes (credit: HESS collaboration).

Gamma-Ray Bursts (GRBs) It is believed that when a heavy star dies as a hypernova or when two neutron stars merge, the central region collapses into a fast rotating black hole, again resulting in the formation of – relatively short-lived – jets that accelerate particles and produce γ-rays.

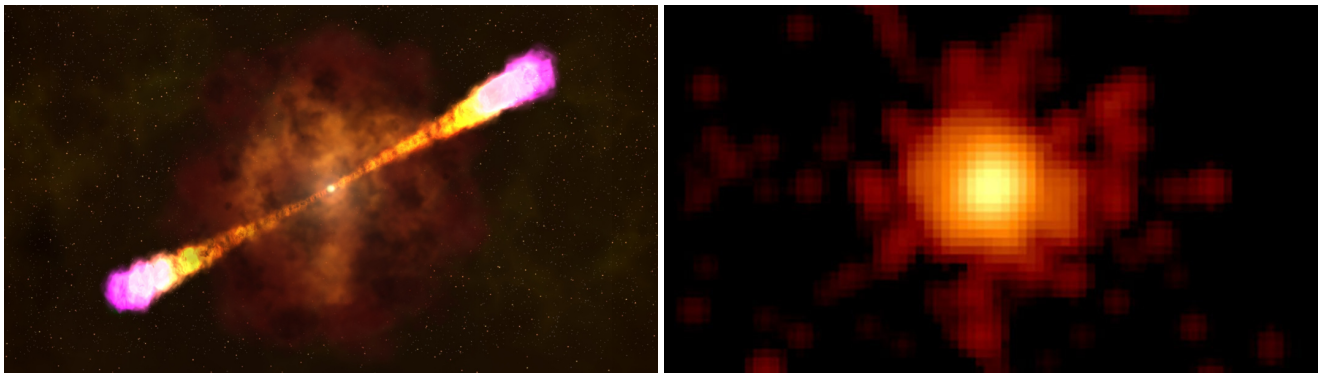


Fig. 1.5.: Left: Illustration of a γ-ray burst. Right: Image of GRB 130427A, taken with Swift's X-Ray Telescope. Credits: NASA/Swift/Stefan Immler.

Importance of γ-ray astronomy for Physics The distribution of particles, their energies and their light curves can be analyzed to learn more about the high-energetic processes in those sources. However, in contrast to electrons and nuclei, light can travel through space without being deflected by magnetic fields. Many experiments exist that observe the sky in different wavelengths, but for understanding the universe at the highest energies, it is essential to see the sky in γ-rays.

1.3. Air Showers and Cherenkov Radiation

If a γ-photon does not react with interstellar matter or the cosmic microwave background, it reaches Earth and can here be detected with a γ-ray detector on a satellite. This is done for γ-rays up to a few 100 GeV. Higher-energetic γ-rays are too rare for the small collector areas on a satellite, but can be detected indirectly.
When a VHE γ-ray or a cosmic ray enters the atmosphere, it interacts with atoms in the air. Through repeated collisions, decays, pair production and Bremsstrahlung on nuclei a cascade of particles (called *air shower*) is generated (see figure 1.6). In the early stages of cascade evolution, the number of particles increases exponentially, while their energies decrease. With decreasing energy of cascade particles, energy loss by ionization dominates over creation of new particles. Beyond the shower maximum, more particles are stopped than new particles are created, and the number of particles decays exponentially. At TeV energies, few if any particles reach sea level. γ-ray air showers are much more symmetric and compact than hadronic showers, which are more irregular because of the subshowers that are generated in hadronic interactions.

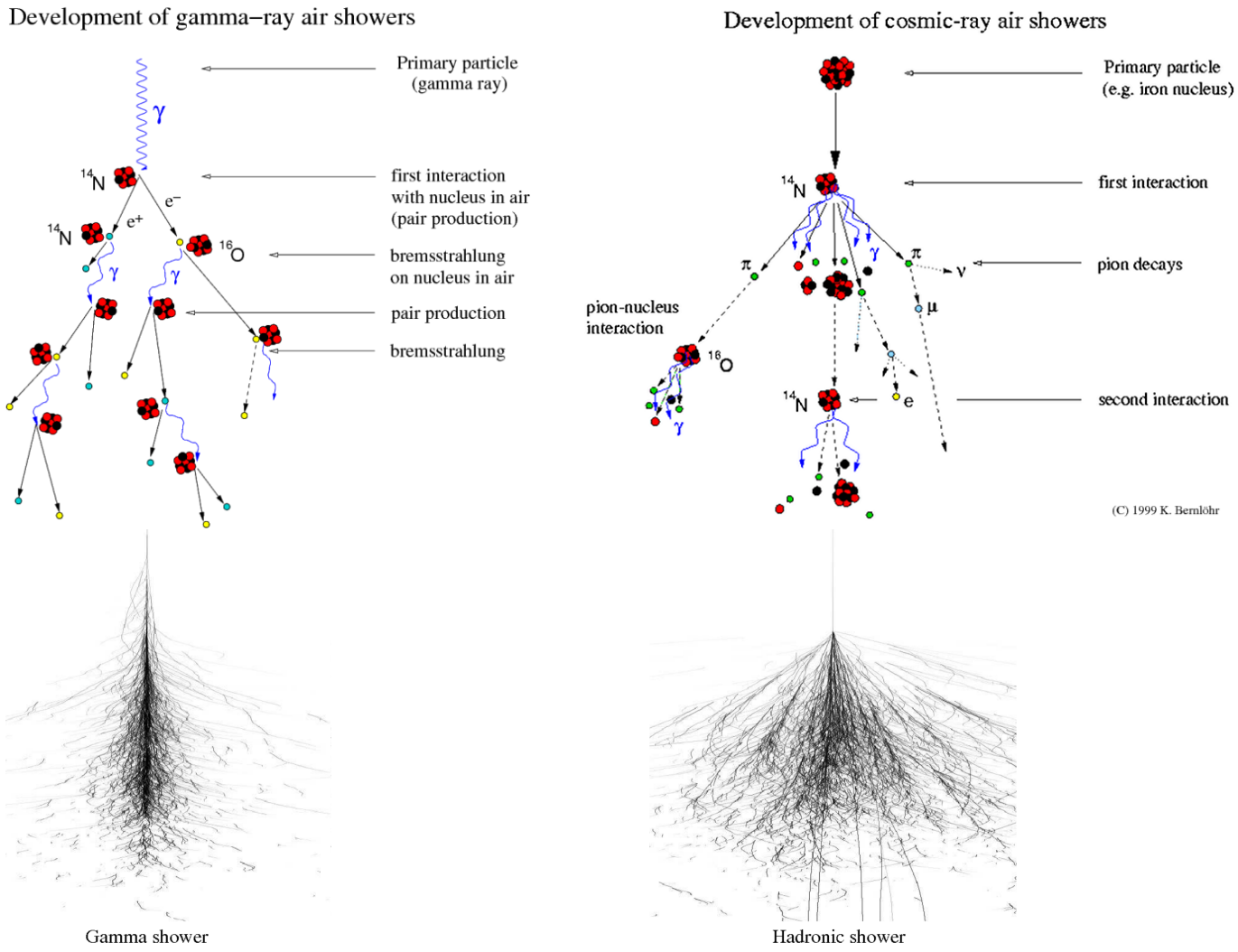


Fig. 1.6.: Left: γ-ray air shower. Right: cosmic-ray air shower. Top: sketch. Bottom: simulation. Credit: Konrad Bernlöhr.

When a charged particle travels through water, air or any other transparent medium whose molecules can be polarized, its electromagnetic field polarizes the molecules of the medium for a short time. If the particles move faster than light in the medium, the short electromagnetic pulses that are radiated from the molecules of the medium when they return to their original position interfere constructively (like a Mach-cone) and an only few nanometer thick wave front of blue Cherenkov light emerges. More details can be found in [7]. For wavelengths larger than 300 nm, the atmosphere is largely transparent for Cherenkov light and it can be observed from the ground.

1.4. Water Cherenkov Detectors

At sufficiently high altitude ($\approx$ 4000 m above sea level), even at TeV energies a sufficient number of cascade particles reaches the ground to allow particle-based detection of the cascade and reconstruction of its direction. Techniques for particle detection include scintillator arrays and arrays of particle tracking devices such as Resistive Plate Chambers (RPCs), but as the most powerful technique the measurement of cascade energy flow using water Cherenkov detectors emerged. In densely spaced, large-volume water tanks, the cascade electrons and muons directly produce Cherenkov light, and cascade photons interact in the dense medium and produce secondary particles that in turn emit Cherenkov light. The Cherenkov light is detected with PMTs (photomultiplier tubes). Water Cherenkov detectors are relatively inexpensive, have a large (1-2 sr) field of view and can be operated day and night.
For the **HAWC** (High Altitude Water Cherenkov) experiment [8], 300 water tanks were placed on a plateau in the Sierra Negra, Mexico. Each tank contains 188.000 liters of water and four PMTs (see figure 1.7). HAWC has an energy range of 100 GeV to 50 TeV and an angular resulution of 0.1° for energies above 10 TeV; below 1 TeV the angular resolution is above 1°. Its energy resolution is below 50% for energies above 10 TeV.
HAWC will not be further discussed, but its data will be used for the compression benchmarks in part two of this work.

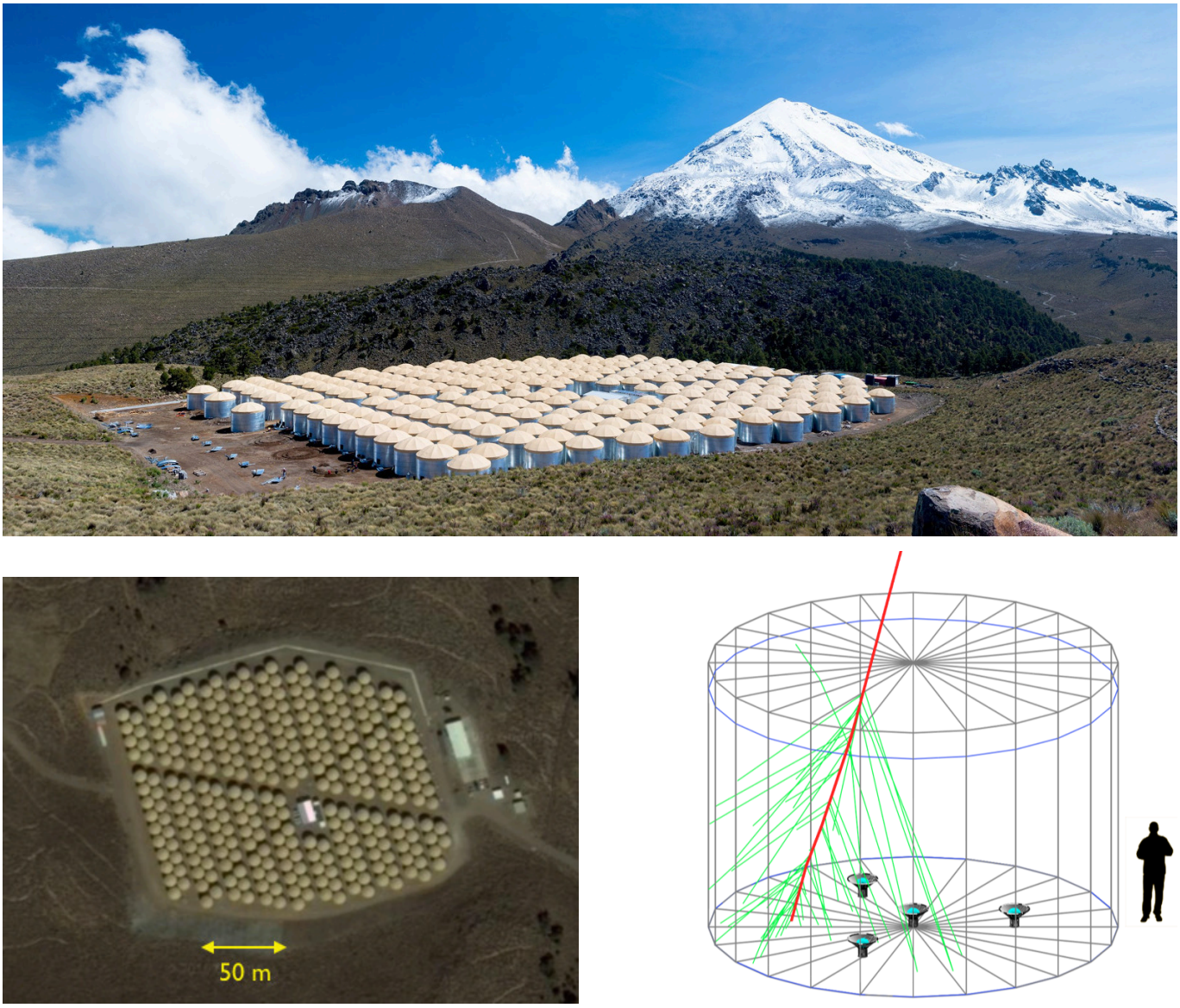


Fig. 1.7.: Left: HAWC array in Mexico during construction. Credit: Wikipedia. Bottom: HAWC array seen with Google Earth (left) and sketch of a HAWC water tank (right). Credit: HAWC Collaboration.

1.5. Imaging Atmospheric Cherenkov Telescopes (IACTs)

In order to detect the Cherenkov light in the air, telescopes with special cameras and large mirrors are needed. Cherenkov telescopes need to be placed at dark and cloud-free sites, just like astronomical optical telescopes. Telescope structures and optics are adapted to the special needs, with relaxed optics requirements compared to optical telescopes – typical point spread functions are in the range of several arc-minutes as opposed to arc-seconds – but high-tech cameras allow triggering and capture of the faint nano-second Cherenkov flashes. Extensive calibration schemes and software image analysis are applied to reconstruct the direction and energy of the primary particle from the Cherenkov images provided by multiple telescopes that stereoscopically view each cascade.

Location The telescopes are usually installed at a height of ≈ 2000m above sea level, because there, shower evolution (at TeV energies) has terminated and little Cherenkov light is emitted at larger depth, yet the other absorption of Cherenkov light is modest. It is important to have clear skies and high atmospheric transparency, to minimize absorption of Cherenkov light and to allow the reliable determination of the energy of primary particles. Any light source would disturb the Cherenkov telescopes, so the site must be far away from cities, streets (because of the headlights of passing cars) and even airline routes. If the central Milky Way is going to be observed, the site should be located on the southern hemisphere.
On sites that meet these criteria, it often proves difficult to provide the telescopes with power and to install a computing center for data acquisition, online analysis and data reduction. Constructing a high-speed data line to a large town is an additional challenge.
Since such experiments usually run for decades, the political stability in the country is another important aspect. Historically, Namibia and Chile have proven to be suitable locations

Mirror A modern Cherenkov telescope has a mirror area of ten to several hundred square meters. The mirror area consists of several smaller mirrors, which usually have circular or hexagonal shape (see figure 1.8). Their reflectivity is optimized for Cherenkov light (above 80% for 300 to 600 nm). The imaging systems are focused for the typical distance of an air shower (10 km).

Mechanical Structure and Drive System Because of the focal length of 15-35 m, the heavy camera and the large mirror area, the mount and dish of an IACT must be very stable. Medium-sized Cherenkov telescopes (like the HESS I telescopes) have a mass of 50-100 tons, while larger Cherenkov telescopes (like the HESS II telescope) can have a mass of over 500 tons. The structure of the large HESS II telescope can be seen in figure 1.8.
Since an IACT has a limited field-of-view of a few degrees, it must also have a drive system, so it can be pointed at any source in the sky. In order to save time during transitions between different targets and to be able to point to a GRB quickly, the drive system must be very fast, which is not trivial, both because of the power requirements and the oscillation of the structure that are induced during rapid acceleration and deceleration. A modern drive system can rotate the telescope with speeds up to several 100°/minute.
For smaller telescopes, like the SSTs (Small Size Telescopes) in CTA, which have a mass of only 8-20 tons, the mechanical structure and the drive system are much simpler.

Camera The reflected light from the mirrors is captured by a special camera with typically 1000 to 2000 pixels (see figure 1.9). Each pixel consists of a PMT or a SiPM (Silicon Photomultiplier) and has elaborated electronics installed, so that it can record even very faint and short Cherenkov flashes.
In analog cameras such as installed in the HESS telescopes, the PMT signals are recorded for a duration of 1 μs with a resolution of 1 ns by Analog Ring Samplers (ARS). When a number of pixels (typically 3-5) in a neighborhood have signals above a threshold (typically 5 photoelectrons), the camera is triggered and the PMT signals of the whole camera are read out. The pixel coincidence prevents the camera from triggering on photons of the night sky background (NSB). In HESS, not the full 1 μs signal trace is read out for each pixel, but instead only the ARS samples within a fixed smaller window around the trigger time are digitized by an analog-to-digital converter (ADC) with 12 bit resolution. In the standard readout mode, samples within 16 ns around the trigger time are summed up, and only the sum value – so-called *ADC sum* – is stored, to reduce the data rate [13]. Newer analog cameras such as the HESS I

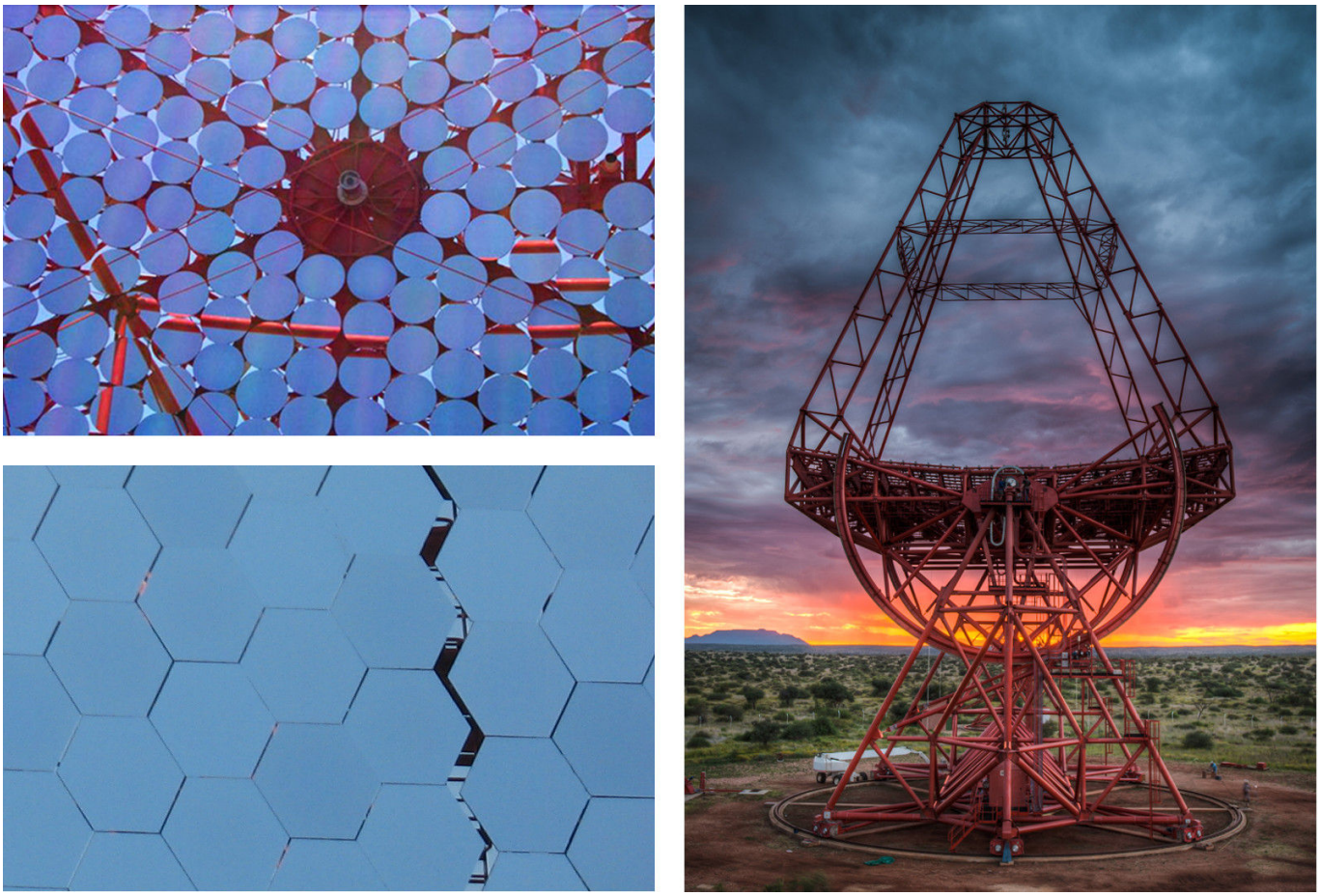


Fig. 1.8.: Top left: Circular mirrors of a HESS I telescope. Bottom left: Hexagonal mirrors of the HESS II telescope CT5. Credit: HESS collaboration.
Right: Structure of the HESS telescope CT5. Credit: Christian Foehr.

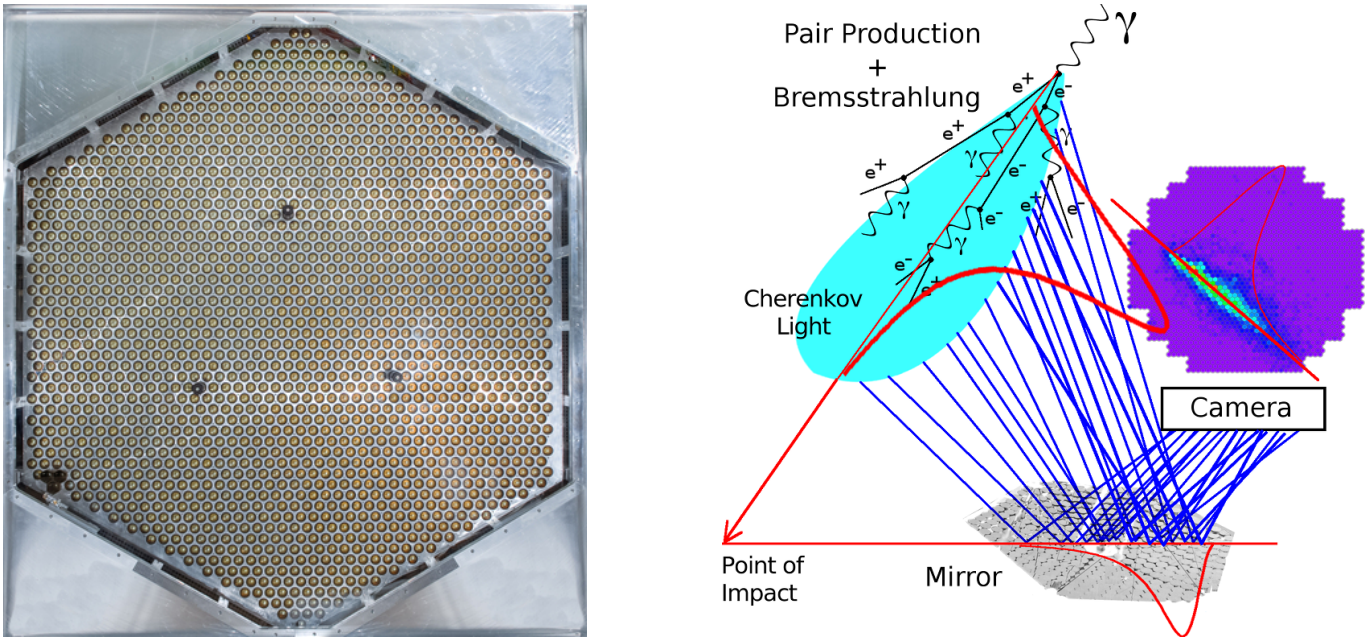


Fig. 1.9.: Left: Prototype of a FlashCam camera for the upcoming CTA experiment (credit: MPIK).
Right: Sketch of Cherenkov light hitting the mirror and then being detected by a HESS camera.

Upgrade Cameras [14] have larger readout bandwidth and allow storing sampled data (also called *trace*). There are also fully digital cameras with Flash ADCs [15], which also provide signal traces of around 100 ns for each event.
An ADC sum image can be understood as a photo of the incoming Cherenkov light taken with long exposure (and thus blurry along the time axis), and traces like video consisting of images with short exposure. The size of the window over which samples are summed represents a compromise; too long a window will collect more NSB photons resulting in increased noise, too short a window will truncate images in particular for large shower impact distances, where the time gradient across the image can reach tens of nanoseconds. Traces contain more information about the shower development and allow proper recording of images with large time gradient, at the cost of more than an order of magnitude more data, and a waveform analysis is required to extract the signal.
For a good resolution over a large dynamic range, many cameras have two gain channels for each pixel. The high gain channel amplifies the signal by a large factor and is sensitive to faint signals. The low gain channel amplifies by a small factor and is sensitive to strong signals. Figure 1.10 shows the same shower in the high gain and the low gain channel. It can be seen that for large amounts of light the high gain channel saturates and the low gain channel must be used.
Whenever a shower is recorded and the camera is read out, the data of both gain channels are provided. Which gain channel is used or if they are merged is decided later in software.

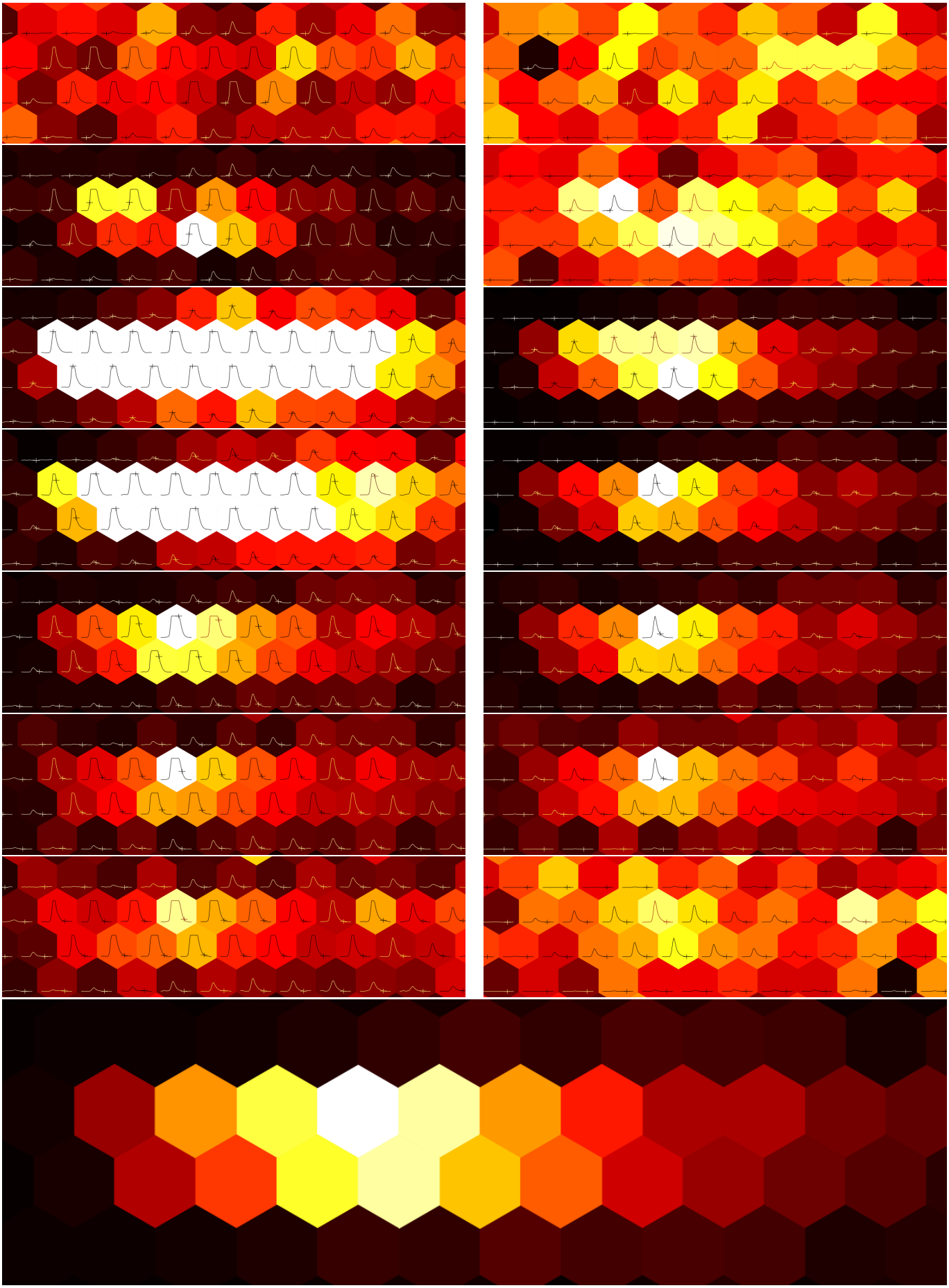


Fig. 1.10.: Sampled ADC values of a shower in a NectarCam of the CTA experiment. For each image, the maximum value is drawn in white and the minimum value in black. Left: High gain channel. Right: Low gain channel. From top to bottom: different positions in the trace (marked by a small cross inside each pixel), so the development of the shower in time can be seen. The bottom row shows the calibrated shower after integration of the signal and merging both gain channels.

Monte Carlo (MC) Simulations In all IACT experiments, MC simulations are essential for the design and optimization of the telescopes, for the development of algorithms for the reconstruction of air shower parameters and the creation of the required lookups and the supervised training of machine learning algorithms, as well as for the calibration of the telescope response.
An air shower simulation package like CORSIKA [28] is used to simulate an air shower induced by a primary particle of certain type, direction and energy, and calculate its development in the atmosphere, considering the magnetic field of the earth and the atmospheric profile. The response of the telescope(s) to the photon bunches is simulated with simulation codes such as sim_telarray [28]. The simulations take into account the particle type, its energy and direction, different parameters of the atmosphere, the magnetic field of the earth at the location of the telescope, as well as the mirrors on the telescope, the optical elements of the camera, the response of the photo sensors and the signal-processing electronics of the camera. Comparisons to laboratory measurements and increasingly detailed simulations have made the simulations more and more realistic over the years. Simulations output data in formats identical to those for real data, for processing by the identical reconstruction chains.
Several Petabytes of MC simulations already exist for CTA, and they are used on a daily basis by many researchers. The CORSIKA/simtelarray simulations, which are used for HESS and CTA, are very similar to real data, both regarding physical parameters and technical parameters, like data structures and size of the data. So the results of the analysis and compression algorithms obtained with MC simulations give a good estimate of what to expect with real data.

Telescope Arrays When at least two telescopes are used, it is possible to geometrically reconstruct the direction of the primary particle (see *Image Analysis*). With assumptions regarding the shape of shower images, a limited reconstruction is also possible with a single image, but stereo imaging significantly reduces errors on the estimated type, the direction and the energy of the primary. Thus, most γ-ray experiments use an array of telescopes and require at least two triggered telescopes (in a short time window), before an event is recorded.
Some experiments combine different types of telescopes, each optimized for a different energy range. MC simulations can be used to support the site selection and to find the best arrangement of the telescopes for different science cases. This was done for CTA, and the resulting layouts for the two array sites are shown in figure 1.11 [16, 17].

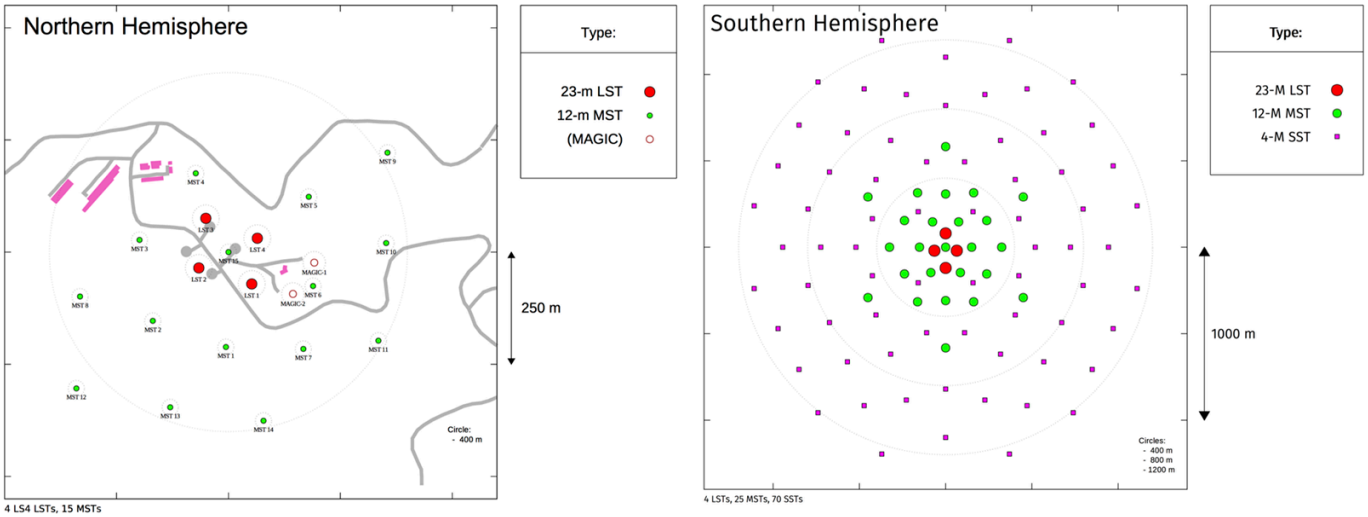


Fig. 1.11.: Layouts for the upcoming CTA experiment in the Northern (left) and the Southern (right) Hemisphere.

Image Analysis The primary particle (i.e. the particle that induced the air shower) can be identified by analyzing the geometry of the air shower image in the camera. Cosmic rays generate more wide-spread and irregular images, because the hadronic collisions in the atmosphere spread the resulting particles farther away from the shower axis, where they produce sub-showers. γ-ray induced air showers are more collimated and produce elliptical images.
This is exploited to reject protons and other charged hadrons, which arrive orders of magnitude more often than γ-rays, but do not contain useful directional information, because they are deflected by magnetic fields.
If two telescopes record the shower, their Hillas parameters [19] are calculated and the direction of the γ can be estimated geometrically (see figure 1.12 and [9]).

If more than two telescopes record the shower, the shower geometry is over-determined and one can average over the pairwise estimated directions of the primary particle or apply a suitable optimization scheme to determine the shower parameters that match the images best. Consistency between multiple images can also be used for γ/hadron separation: in case of proton-induced background showers, telescopes often respond to different subshowers resulting in poor consistency between different views of the shower.
However, if at the very beginning of a cosmic-ray-induced air shower a π° is created that carries a large fraction of the primary energy, and decays to two γ-rays, the analysis of the resulting air shower image will assume that the primary particle was a γ-ray.
In addition to the Hillas parameters, other shower parameterizations can be calculated and fed into a machine learning algorithms that are trained with MC events, in order to classify the images as γ-ray or hadron showers.

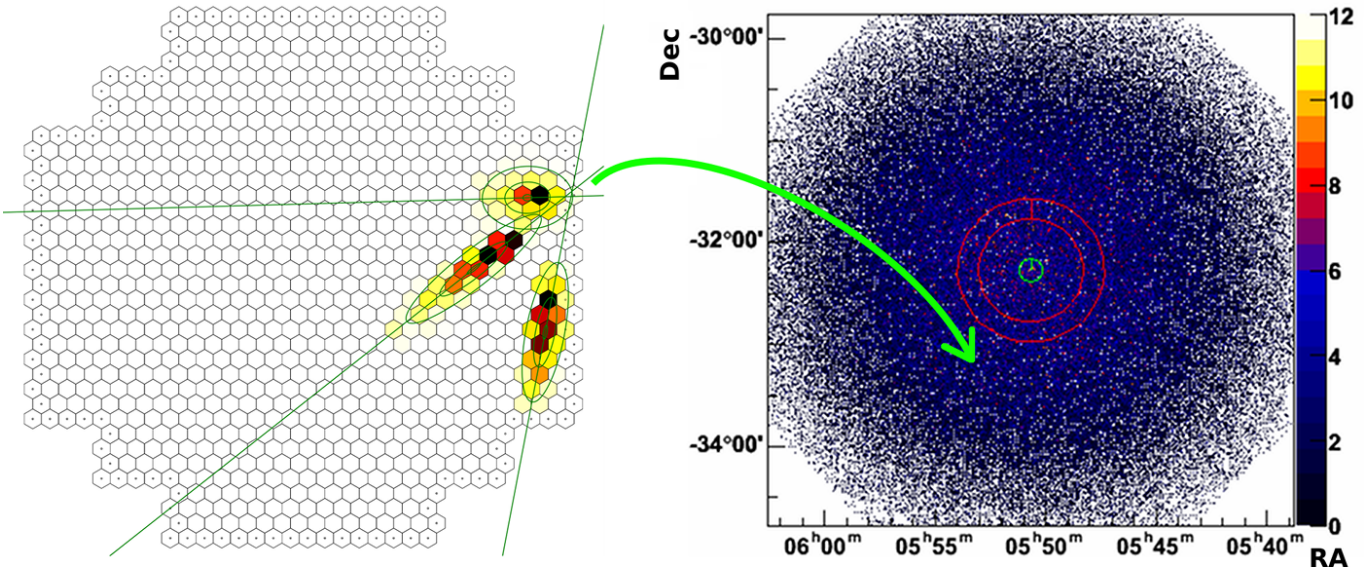


Fig. 1.12.: Each reconstructed direction in the nominal camera system (left) gives an additional count on the sky map (right). Geometric reconstruction is based on the feature that images of the shower axis in different superimposed views intersect at the image of particle source.

Sky Maps Unlike optical telescopes, an IACT cannot instantly provide an image of the observed source. Instead, each reconstructed shower becomes a point on a sky map (1.12). Despite γ/hadron separation, the sky map is still dominated by hadronic events and thus contains significant background that fills the field of view more or less uniformly. Diffuse γ-rays from large-scale γ-ray sources may also contributed to the background.
From the density in the field of view of γ-ray candidates outside a source region, an estimate of the background is calculated and subtracted. The probability of detection of a source can be estimated by counting the number of events in the source region (N_{on}) and the predicted number of background events in the source region (N_{off}) and then calculating the Li-Ma-significance[6]. Depending on the flux of the source and its neighborhood, it can take several hours or even months for data taking until it sets itself apart from the background.
In complex regions, for example in the Galactic plane, exclusion maps need to be applied to prevent wrong background estimates. Sources with complex morphology (as opposed to point sources) require more complicated analyses.
Modern IACTs have an angular resolution of $< 0.1^\circ$, which allows to image some extended sources in our galaxy. Extragalactic sources mostly appear as point sources.

2. HESS and CTA

The High Energy Stereoscopic System (HESS) is one of the leading IACT experiments, and the Cherenkov Telescope Array (CTA) is a next-generation facility under construction. HESS is central in this work, because most of the analysis and calibration algorithms described in this document have been developed in the context of the HESS experiment and were tested with HESS data. CTA also motivated a large part of this work, because its complexity and the expected data rates of 70 GB/s require modular, efficient software. CTA Monte Carlo simulations and real data from a prototype camera for the medium-sized CTA telescope – FlashCam – are used in the second part of this work.

2.1. HESS

HESS (named after Victor Hess) is an array of five IACTs, located near the Gamsberg mountain in Namibia. The four smaller (12 m) telescopes CT1-4 (CT: Cherenkov Telescope) were built around the year 2000. They form a square of 120 m side length and have a trigger rate of $\approx$ 300 - 400 Hz, requiring coincident detection of an air shower by two telescopes. This original array is called HESS I.
Ten years later, a larger 28 m telescope (CT5), suited for lower energies, was added. To maximize coincide triggers with the other telescopes, CT5 was placed at the center of the original array. This upgraded array is called HESS II and is shown in figure 2.1.

Fig. 2.1.: The HESS array in Namibia. The large CT5 is in the center of the four smaller CT1-4. Credit: HESS Collaboration.

HESS has an energy range of 50 GeV to several tens of TeV, an angular resolution of less than 0.1° and a high sensitivity. It can detect a source with a flux of 1% of the Crab in 25 hours observation time with 5σ. One of the highlights of HESS is the Galactic Plane Survey [10]. A cut-out of the obtained sky map can be seen in figure 2.2.

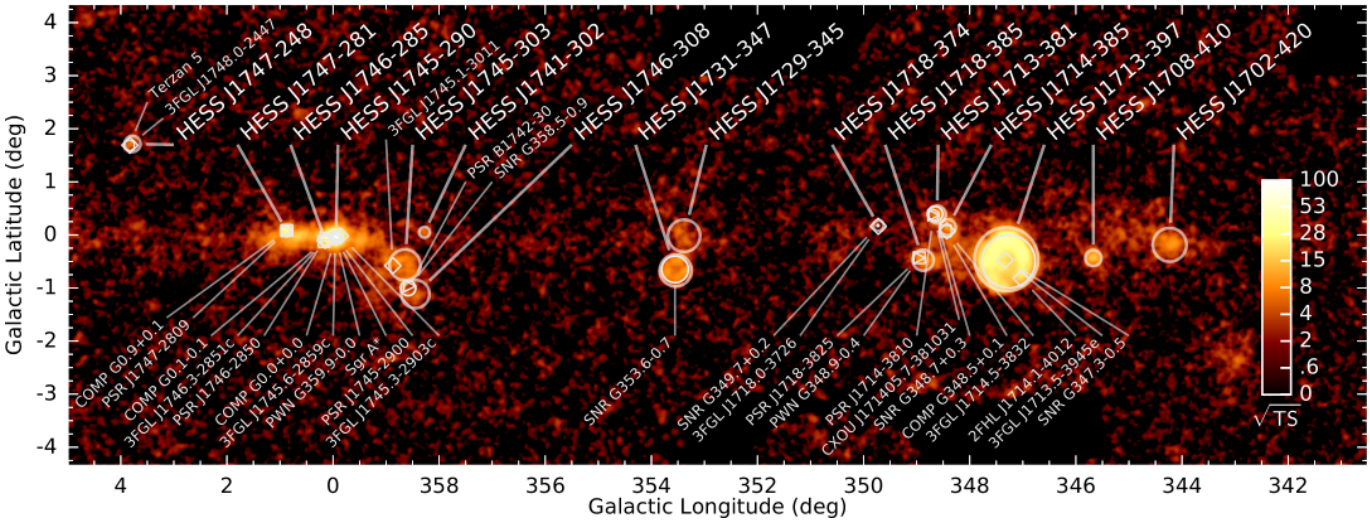


Fig. 2.2.: HESS Galactic Plane Survey (cut-out taken from [10]).

2.1.1. HESS camera data

The four HESS-I telescopes CT1-4 have 960 pixels each. Each pixel has a high gain and a low gain channel, and when an event is recorded, the signal from both gain channels is recorded. The 12 bit ADC values are stored as 16 bit unsigned short integers. Thus, a HESS-I event has a size of about 10 kB (not all four telescope participate in a given event), and with a trigger rate of $\approx 300 - 400$ Hz, the data rate is $\approx 3 - 4$ MB/s.
The HESS-II telescope CT5 has 2048 pixels with high gain and low gain channel, and triggers with ≈ 1.5 kHz. Furthermore, the time of maximum of each trace is stored, so the data rate of CT5 alone is ≈ 20 MB/s. Even with all five telescopes taking data, the data rate is be below 25 MB/s. Since the beginning of the experiment, several hundred TeraBytes of data have been accumulated. Camera images from a HESS-I telescope and from CT5 are shown in figure 2.3.

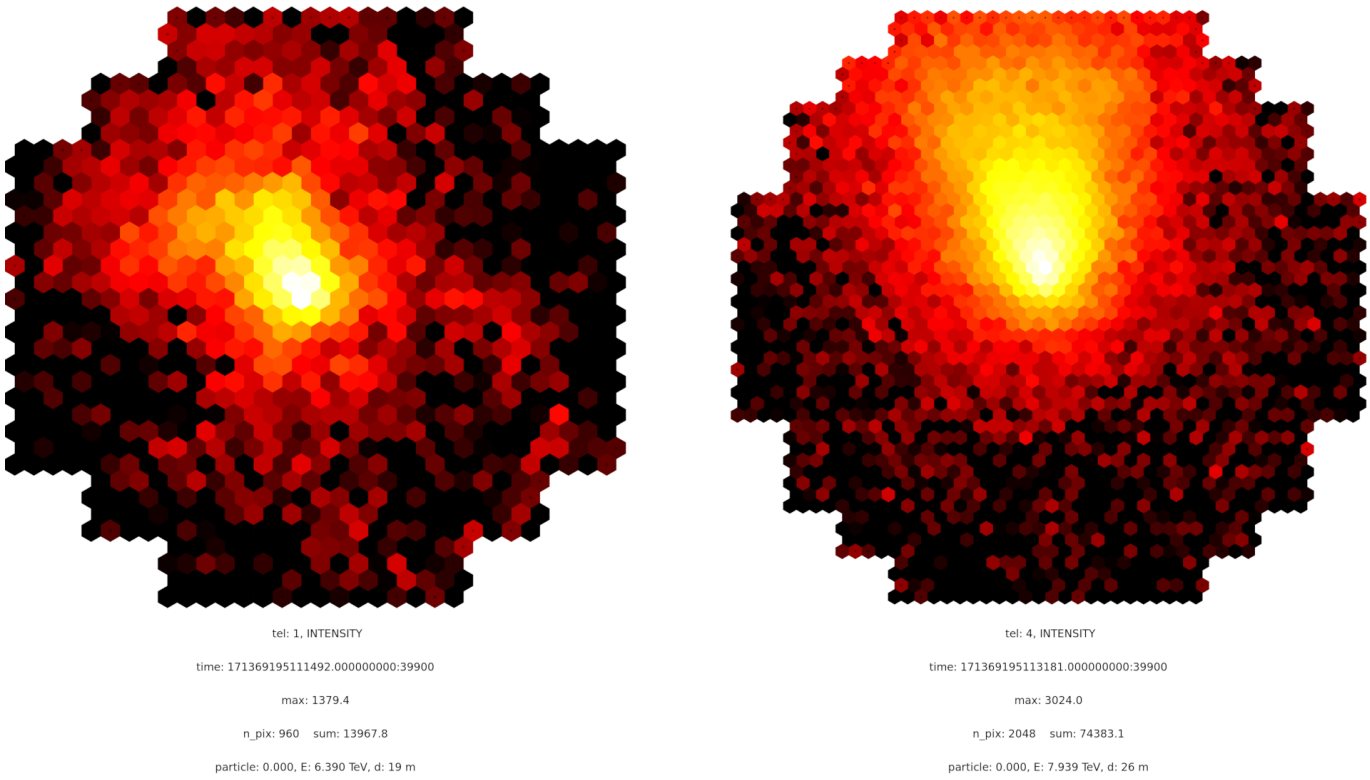


Fig. 2.3.: A simulated γ-ray event in a HESS-I telescope (left) and in CT5 (right). Camera sizes are not shown to scale.

2.2. CTA

CTA will be the next large ground-based gamma-ray observatory and will consist of over 100 Cherenkov telescopes in both hemispheres. A smaller array with 19 telescopes will be installed on La Palma, Spain, and a larger array with 99 telescopes will be installed in Paranal, Chile (see figure 2.4).

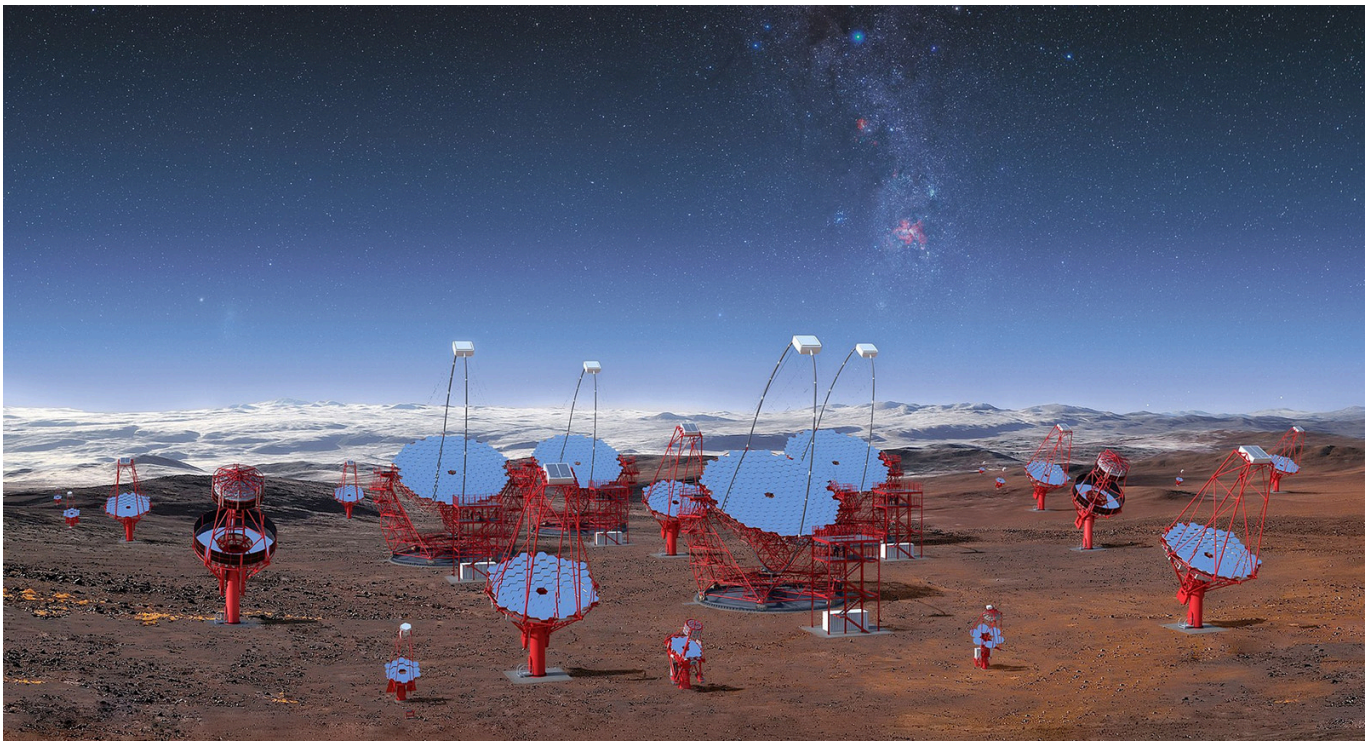


Fig. 2.4.: Illustration of a part of the CTA South array in Paranal, Chile, showing all three types of telescopes. Credit: CTA/M-A. Besel/IAC (G.P. Diaz)/ESO.

There are three types of telescopes (see 2.5) that will record photons in the energy range from 20 GeV to 300 TeV:

Small-Sized Telescope (SST) They have an energy range of 3 TeV to 300 TeV and will be used primarily to study sources in our own galaxy. Consequently, they will be deployed in the southern hemisphere. Their field of view is $\approx 9^\circ$ and they are inexpensive compared to the other telescopes; a large number of 70 SSTs will provide a ≈ 10 km^2 detection area for the highest gamma ray energies. There are three different prototypes of SSTs under development: the dual-mirror ASTRI and GCT telescopes and the single-mirror SST-1M. The three SST variants have different cameras, each with different number of pixels (2368, 2048 and 1296) and field of view ($10.5^\circ, 8.3^\circ$ and $8, 8^\circ$).

Medium-Sized Telescope (MST) With a mirror size comparable to the HESS-I telescopes, a total of 40 such telescopes is planned to cover the core energy range of CTA of 100 GeV to 30 TeV. In this range the highest spectral sensitivity is achieved, appropriate to observe both galactic and extragalactic sources. There are two different MST cameras under prototyping for the MST structure: FlashCam and NectarCam, differing primarily in their signal recording electronics. FlashCam has a field of view of 7.5° and 1764 pixels, and NectarCam has a field of view of 7.7° and 1855 pixels.

Large-Sized Telescope (LST) With an energy range of 20 GeV to 3 TeV, four LSTs on each of the two sites will be used to observe galactic and extragalactic sources. The LST cameras provide a field of view of 4.3° and have 1855 pixels. The high positioning speed of the LST of only 20 s to any point in the sky allows timely follow-up of flaring sources such as gamma-ray bursts.

2.2.1. CTA camera data

Both CTA array sites are far away from the data centers, where all data will be archived and where users will be able to access it. Since the raw data rates are expected to exceed 200 PB/year and only 4 PB/year can be transferred with the assumed Gigabit line connecting array sites with the data center(s), a data compression factor of at least 50 is be needed. Tests with different algorithms show that for lossless compression, a compression factor of maximal 3 should be expected. Thus, significant data reduction on site is required, e.g. by identifying the pixels that contain an air shower signal and only transmitting those, and/or by lossy compression. A prerequisite for efficient data reduction is the reliable on-site and on-line calibration of pixel signals. However, any data reduction scheme that works well with current analysis algorithms might still interfere with future more sophisticated algorithms, hence as much pixel information as possible should be kept. An efficient data format that allows flexible data reduction schemes and that compresses the recorded events well is therefore crucial.
Another use case for an efficient data format is the data acquisition and lowest-level calibration, where the data is dumped to disk and then reread several times in order to obtain the calibration parameters. The IO is usually limited by the disk and the network, and not by the CPU, so a fast compression/decompression algorithm could improve the IO significantly.
Most of the data that need to be stored are shower images recorded with Cherenkov cameras. Some of the cameras have two gain channel and provide 12-bit sampled data (also called *traces*) for each camera pixel, therefore an image is actually stored as a $2 \times n_{\text{pix}} \times n_{\text{samples}}$ array of 16 bit unsigned integers. Six different types of cameras are currently being considered in CTA, each with a different number of pixels and samples per pixel:

Camera Type	#Pixels	#Samples	Event Size [kB]	Trigger Rate [kHz]	Data Rate [GB/s]
ASTRI	2368	1	9	0.3	0.1
CHEC	2048	128	512	0.4	0.1
DigiCam	1296	22	114	0.6	0.4
FlashCam	1764	22	152	6-35	3.0
NectarCam	1855	60	435	7	2.0
LST Cam	1855	60	435	20	3.0

Table 2.1.: Overview of CTA cameras and their data rates. The number of samples and the trigger rates are variable, so some numbers might look inconsistent.

After calibration, the samples are integrated around the position of the signal, the information of the two gain channels is merged, and an additional array of n_{pix} 32 bit floating point values needs to be stored. Other derived values, such as the time of the signal maximum in each pixel or the integral of the complete trace might also need to be stored.
From these derived values, usually only the subset that contains all the signal pixels is needed. It can either be stored as list or as region of interest (see chapter 4, section 4.4). These reduced data sets are two orders of magnitudes smaller than the raw data, and they hopefully contain all the necessary information for further analysis. However, not all calibration parameters are available during the data acquisition, so the signal pixels cannot be reliably detected and any kind of on-the-fly data reduction would be dangerous.
So either each telescope:

- stores the raw data temporarily on its directly attached camera server, determines the calibration parameters, reduces the data, and sends it to the central on-site data center, where the per-telescope data is merged into array-event data structures. Although in this scheme, the critical parts of the I/O are parallelized, each camera server still has to manage up to 3 GB/s of I/O.
- sends the raw data directly to the central on-site data center, where the merged array events (still $\approx$ 20 GB/s) are written to disk. This requires a sophisticated RAID system inside the data center.

Any reduction of the data rate saves money, because less hard disks, CPUs, network equipment and electricity are needed.

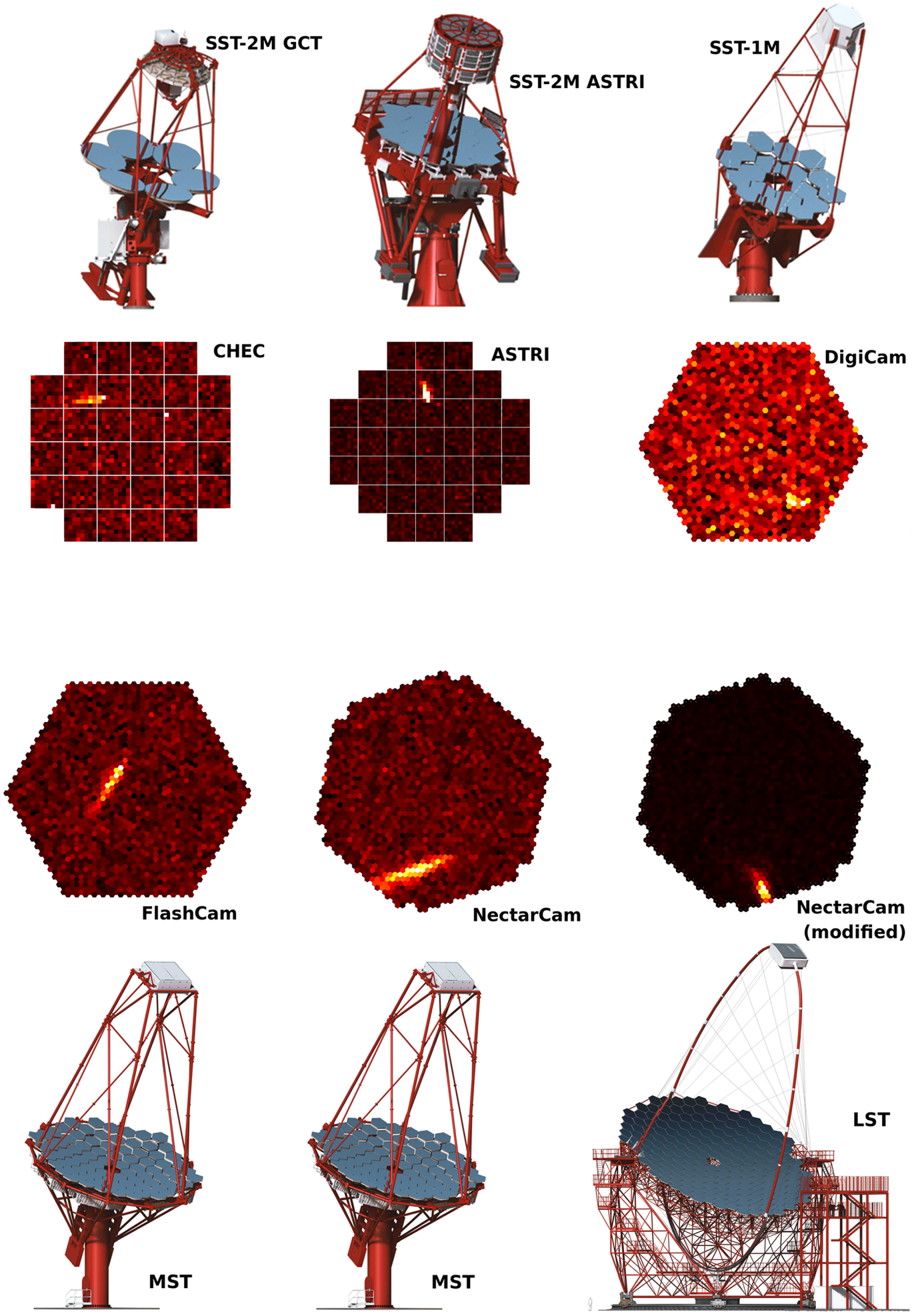


Fig. 2.5.: CTA telescopes and cameras. Credit for the telescope sketches: CTA Collaboration.

3. The MESS Software Framework for Data Processing in γ-ray Astronomy

MESS (Modular Efficient Simple System) is a software framework for γ-ray astronomy and consists of developer tools, a library and several modules that can be used to create complex analysis pipelines on the command line. It has been written from scratch in C.
The motivation for this software is to support the users with frequently needed data structures and algorithms in γ-ray astronomy, without forcing any programming philosophy onto them and without confining them to the framework. Instead, flat data structures and simple interfaces allow the users to easily interact with other algorithms, libraries and programs.
This chapter describes the main aspects of the MESS software framework. It assumes knowledge about the calibration and analysis of IACT data, which are explained in chapters 6 and 7.

3.1. Developer Tools

During developing, redundancy should be minimized and maintaining the code and documentation should be as simple as possible.

3.1.1. Automatic header file generation

Writing header files, is a redundant process, because the declarations of public functions is done twice: First in the source file and then in the header file. Whenever the interface changes, for example if a function is changed to accept different arguments or if its name changes, both the source and the header file need to be updated. MESS can generate header files from C/C++ files, so less code and only one file need to be maintained.

3.1.2. Documentation generation

Browsing documentation in a format like HTML is more convenient than reading the comments in the source code, so having a tool that creates HTML pages from the comments in the source files is helpful. Such tools exist (e.g.: `Doxygen`), but they are not as modular and simple to use, and they are heavy-weight and produce heavy-weight output. MESS can convert markup-like text to HTML (web page or presentation) and Latex, with support for tables, images, videos, links, Latex math, on-the-fly input from external programs etc. Directory hierarchies in the documentation are mapped to menus and subpages in HTML and parts, chapters, sections etc. in Latex.

3.1.3. Parameter parsing

Managing the set of possible parameters for each module or program is also a problem that every developer encounters. First, they have to appear in the source code, because they have to be parsed from the command line. Second, they have to be mentioned in the documentation, and if a parameter changes or a new one is added, the documentation needs to be updated as well. Putting the documentation in the code helps, but does not solve the problem, because still, two places would have to be kept up-to-date. MESS provides routines for parameter handling: parameters of modules or programs can be defined in a structure, then get automatically parsed from the command line and afterwards easily accessed and printed from within a program or the documentation generator. Flags are supported as well as parameters that receive one or more strings as arguments. If a parameter marked as mandatory is not found during parsing, the program terminates with an error message. If manual parsing of a parameter is desired, it can be marked. An example that demonstrates parameter parsing is done can be found in listings 3.4 and 3.5.

3.1.4. Logging

MESS provides infrastructure for logging. Each module can specify a file or a pipe where the log messages should be sent. By default, logging is sent to the global logging stream, which is by default *stderr*.

3.1.5. Enum-to-string conversion

Sometimes, an input string needs to be converted to an integer, for example if names shall be matched to global constants (e.g. `analyze -type EVENT`). Having an `enum` structure defined in a file, this tool generates conversion routines from the integer values of the enums to their names and back. This can also be used the other way round, for example if instead of an error code the enum name should be printed (like `ERROR_FILE_CORRUPT` instead of `N`).

3.1.6. Code statistics

In a large project, it can be helpful to gain an overview of the subprojects in terms of code complexity, coding style, code-to-comment ratio, high- or low-level code etc. MESS can calculate code statistics that can help classifying the code, like the number of code characters, comment characters, statements, loops etc. For projects that require certain code-to-comment ratios or that want to automatically find code with high complexity (high punctuation-to-alphanumeric characters ratio, many branches and loops per statement), this tool can be useful.

3.1.7. Automatic versioning

Keeping track of the version can be tedious, because it needs to be changed manually in the library every time a new major or minor version is created. MESS automatically obtains these numbers from `git` and inserts them into the library, so it always knows its version/tags, SHA checksum etc., and these can easily be accessed in order to simplify version checking and packaging.

3.1.8. Build system

MESS comes with a modular and easy-to-use build system, which does not have any dependencies except `make`. In order to build a module (or program), its name has to be added to the list of modules (or programs) to build. External libraries and readers/writers for external formats are cleanly integrated without imposing the dependencies on the core library or the rest of the modules. The locations of external libraries can be provided, otherwise they are found automatically, given they were installed at one of the standard locations (`/usr/lib`, `/usr/local`, `/home/$user`, ...). Thus, in many cases, other tools like *autoconf* or *cmake* are not necessary.

3.2. Module system for the on-the-fly creation of analysis pipelines

With MESS, it is possible to define a pipeline of modules on the command line. The main program `mess` is called with a list of modules, their parameters and the definition of the data flow as input:

```
mess -micropipe read.r: -in gamma.mes , plotevent.p:r
```

In this simple example, two modules are used: a `read` module named `r`, which is called with the parameter `-in` to define the input file, and a `plotevent` module named `p`, which receives its input from `r` and which does not have any parameters. The `read` module has no parents, so it is called first, and it reads the first event from the file. Then, the `plotevent` module is called, which simply receives the pointer to the event pointer and plots it. When the user hits `<return>`, the first iteration of the pipeline is done and the `read` module is called again. This continues until there are no more events in the file. In this example, the interactive module `plotevent` is involved, so user interaction was necessary to advance in the pipeline. If it had not been interactive, like most of the modules, the pipeline would have run without stopping at each iteration. MESS currently provides more than 70 modules, which cover:

- fast compressed I/O, event and parameter filtering, synchronization of several input streams
- trace analysis, image cleaning
- calibration, broken pixel identification
- event viewer with zoom, array view, support for showing traces, broken pixels and Hillas parameters, pdf/png export and a command line interface
- several shower image parameterizations that can be used for background reduction
- gamma/hadron separation using deep neural networks and support vector machines
- n-dimensional histogram generation with file I/O, classification performance plots (q-factor and ROC curve), and a 1D and 2D histogram plotting module

A list of modules can be found in the appendix. It is also possible to build a pipeline of executables that communicate via Unix pipes (`mess -macropipe`), but this is not covered here.

3.2.1. Description of modules and pipelines

Each module is a shared object file, written in C or C++, and must provide these three functions:

`<module name>_init` is called once, at the beginning of the pipeline. Here, parameters to the module are parsed, memory allocated and lookups filled. The input and output messages of the module are checked to be compatible with the input and output messages of its children and parents.

`<module name>_exec` is called at each iteration of the pipeline. Since the pointers to the input messages are fixed, this function call has no overhead and is as fast as a function call in a statically linked program.

`<module name>_exit` is called once, at the end of the pipeline. Here, summaries can be calculated and presented. This is useful for histograms and machine learning modules, where the processing can only start once all data is there. Closing files and cleaning up memory is also done here.

By sorting the modules topologically, their dependencies are resolved and the order of execution in the pipeline fixed. The syntax for defining a pipeline in MESS is as follows:

```
mess -micropipe \
  type.name:parent0,parent1,... -param value1 value2 -param value ... ,
  type.name:parent0,parent1,... -param value -flag -param value ... ,
  ...
```

The module type is the name of the shared object file (without the extension `.so`) and describes the purpose of the module. In the example above, the module types are `read` and `plotevent`. The module name is chosen by the user and needed to reference the module inside of the pipeline.

Input and output messages of a module Each module can receive several messages from its parent modules as inputs (either as requirements or as options), and it can provide several outputs to its children. Only the parent modules are given; the children modules can be inferred. Each module can have several messages as inputs (either as requirements or as options)

Module parameters Each module can have a list of flags, parameters and values, just like a normal program. They are are passed as `argc` and `argv` to the `init` function of the module and can there be automatically parsed by MESS functions or manually. A separate comma marks the end of the parameters of the module and the beginning of the next module. Note: If a command spreads across several lines, `bash` requires backslashes at the end of each line that breaks, but they are omitted here for better readability.

3.2.2. Command line interface

When a pipeline is executed, no files need to be compiled and no config files need to be written. It is run as any other program, but with much greater flexibility. This is convenient for scientists not familiar with programming.
However, as the other examples in this chapter show, more complex pipelines that involve branches and require a certain order of execution can also be created, for example to perform γ/hadron separation or calibration. In such cases, it is convenient to call the pipeline from inside a `bash` script, so the command line is saved and can be edited and shared. Also, `bash`'s full functionality can be used, for example to manage files and scan parameter spaces with loops.
Furthermore, by doing as much as possible on the command line, the reproducibility is high, because for a different pipeline, only the command line needs to be saved, whereas creating the pipelines in the source code means maintaining one file, one build command, and one command line for each pipeline. In scripting languages, pipelines are sometimes almost as easy to define as on the command line with MESS, however, if custom computationally expensive algorithms need to be created and executed, the performance of the scripting language will usually be poor. Writing the algorithm in a compiled language and create a library for the scripting language is more complicated than writing a module in MESS and compile it to a shared library.
For a better overview over a complex pipeline, it can optionally be written to a *graphviz* `.dot` file and then converted to an image that shows the data flow (see the following examples).

3.2.3. No init scripts or environment variables

Some software packages require certain environment variables to be set, which often leads to problems if the same command has to be executed on a different machine or if there are several init scripts on the same computer that override each other. MESS uses working directories, which contain the setup of the current analysis, such as the telescope positions, the camera configurations, and symlinks to the lookups used. Like this, the directory can be copied to a different location, and MESS will be able to run without any extra efforts. By default, the current directory is assumed to be the working directory, but it is also possible to specify a different working directory with the `-workdir` parameter. However, if any directories are explicitly specified on the command line (e.g. `read.r: -in /path/to/file.mes`), they are used.
When a pipeline is launched with the `mess` program, the location of the MESS software is found with the Linux `realpath` function, and the MESS modules can be loaded. This makes it easy to manage different versions of the MESS software, and no environment variables are needed and no paths need to be searched.
All MESS programs use this way of avoiding environment variables and init scripts, not only the pipeline program.

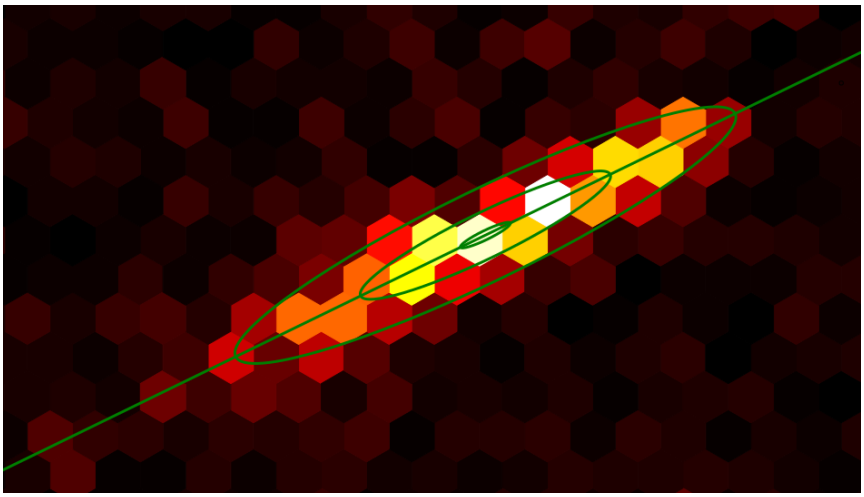
3.3. Message passing

Communication and data passing between modules is done by handing pointers to messages between the modules, so by default the memory is shared. This is the fastest way to pass data. Compatibility of interfaces between modules passing data is checked at the beginning, before the pipeline is executed, by comparing the message types of parent and child modules. The interface does not have to be fixed; depending on the parent module, the child module can change its input signature. Furthermore, it is possible for a module to have optional inputs. The `plotevent` module, for example, needs an event as mandatory input, and takes a broken pixel pattern or Hillas parameters as optional arguments; if they are given, they are drawn on top of the event.
If a module modifies the data and another one needs the original data, a flag can be set to control the order in which the modules are called. In cases where the order cannot be changed, for example because a module needs both the modified and the original data as input, the *dup* module can be used, which duplicates the data. Figure 3.1 shows an example where the duplication of a message is required: full camera images are read from a file and are to be plotted with the Hillas ellipses of the cleaned shower images on top of them. The only way for the `plotevent` module to receive a full camera image and the Hillas ellipse of the cleaned image is to make a copy of the camera image before the image cleaning:

1. the `read` module reads an event from a file
2. the `integratesamples` module does a trace analysis of the and fills the ADC sum container of the event with the integrated charges
3. the `dup` module duplicates the output of the `integratesamples` module
4. the star shows that this is done before the `cleanmn` module cleans it
5. the `hillas` module calculates the Hillas parameters of the cleaned shower
6. the `plotevent` modules plots the duplicated uncleaned event along with the Hillas parameters of the cleaned event overlaid

All modules have their inputs pointed to the outputs of their parent modules, so except for the `dup` module, no memory copy is done during the execution of the pipeline.

```
mess -micropipe read.r: -in gamma.mes ,
    integratesamples.i:r ,
    dup.dup:*i , cleanmn.c:i -m 4 -n 7 ,
    hillas.h:c , plotevent.pl:dup,h
```



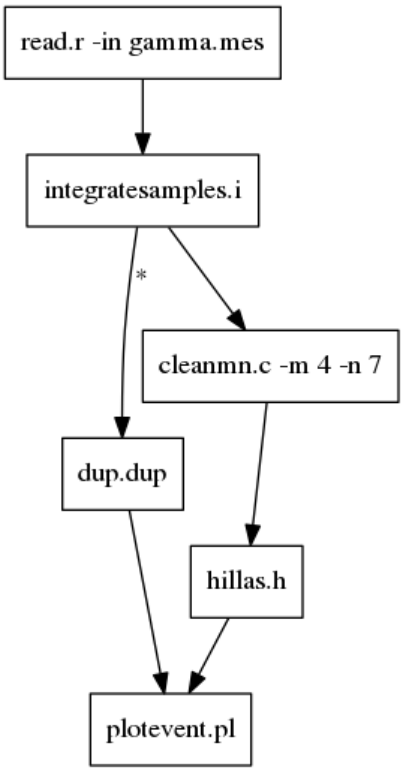


Fig. 3.1.: Example of a MESS pipeline that plots the Hillas parameters of a cleaned shower on top of the original shower. Left: the command line that defines the pipeline (top) and the result (bottom). Right: the corresponding graph.

3.4. Examples of MESS pipelines

Figures 3.2 and 3.3 show two pipelines that are used in the γ/hadron separation chapter. The first pipeline reads a file with γ-ray shower events and a file with proton shower events. Each event is cleaned with different image cleanings, and for each cleaned event, the Hillas parameters are calculated. The asterisk symbol before the name of the Hillas module means that the incoming cleaned event is processed before the cleaning module, which also receives it. From the Hillas parameters, only width, length and log(sum) are selected and then concatenated to a vector that is finally fed to a Support Vector Machine (SVM). The second pipeline is similar, but instead of being trained, the SVM now classifies the events on the basis of their Hillas parameters and the trained model. The responses are histogrammed and from the two histograms, a ROC curve is calculated and plotted at the end.

```
mess -graph hillas_multiclean_train.dot -micropipe
read.r: -in gamma_ct5_roi_0.3_0.4_train.mes , read.r2: -in proton_ct5_roi_0.3_0.4_train.mes ,
cleanmn.cA:r -m 2 -n 3 -nb_min 2 -nb_rows 0 -safe ,
cleanmn.c2A:r2 -m 2 -n 3 -nb_min 2 -nb_rows 0 -safe ,
cleanmn.cB:cA -m 3 -n 6 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2B:c2A -m 3 -n 6 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.cC:cB -m 6 -n 12 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2C:c2B -m 6 -n 12 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.cD:cC -m 10 -n 20 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2D:c2C -m 10 -n 20 -nb_min 1 -nb_rows 0 -safe ,
hillas.hA:*cA , hillas.h2A:*c2A , hillas.hB:*cB , hillas.h2B:*c2B ,
hillas.hC:*cC , hillas.h2C:*c2C , hillas.hD:*cD , hillas.h2D:*c2D ,
select.s:hA,hB,hC,hD -par hA.width hA.length hA.log_sum hB.width hB.length
    hB.log_sum hC.width hC.length hC.log_sum hD.width hD.length hD.log_sum ,
select.s2:h2A,h2B,h2C,h2D -par h2A.width h2A.length h2A.log_sum h2B.width h2B.length
    h2B.log_sum h2C.width h2C.length h2C.log_sum h2D.width h2D.length h2D.log_sum ,
vectorize.v:s , vectorize.v2:s2 , svmtrain.svm:v,r,v2,r2 -svmtype 0
    -model hillasmulticlean/ROC_hillasmulticlean_0.3_0.4_4_7_1_0_svmtype_0_C_100 -c 100
```

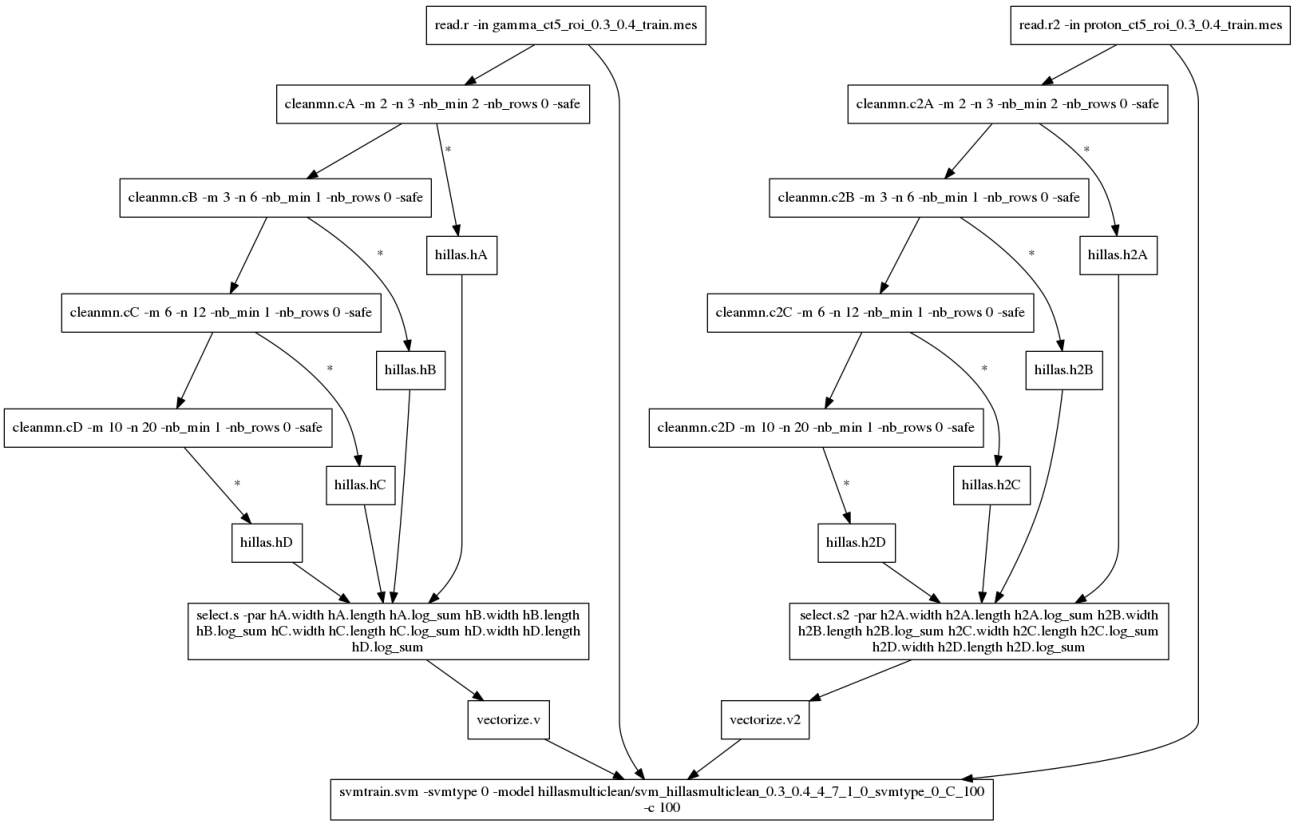


Fig. 3.2.: Pipeline that trains an SVM to distinguish shower images based on Hillas parameters of differently cleaned shower images.

```
mess -graph hillas_multiclean_test.dot -micropipe
read.r: -in gamma_ct5_roi_0.3_0.4_test.mes , read.r2: -in proton_ct5_roi_0.3_0.4_test.mes ,
cleanmn.cA:r -m 2 -n 3 -nb_min 2 -nb_rows 0 -safe ,
cleanmn.c2A:r2 -m 2 -n 3 -nb_min 2 -nb_rows 0 -safe ,
cleanmn.cB:cA -m 3 -n 6 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2B:c2A -m 3 -n 6 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.cC:cB -m 6 -n 12 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2C:c2B -m 6 -n 12 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.cD:cC -m 10 -n 20 -nb_min 1 -nb_rows 0 -safe ,
cleanmn.c2D:c2C -m 10 -n 20 -nb_min 1 -nb_rows 0 -safe ,
hillas.hA:*cA , hillas.h2A:*c2A , hillas.hB:*cB , hillas.h2B:*c2B ,
hillas.hC:*cC , hillas.h2C:*c2C , hillas.hD:*cD , hillas.h2D:*c2D ,
select.s:hA,hB,hC,hD -par hA.width hA.length hA.log_sum hB.width hB.length
    hB.log_sum hC.width hC.length hC.log_sum hD.width hD.length hD.log_sum ,
select.s2:h2A,h2B,h2C,h2D -par h2A.width h2A.length h2A.log_sum h2B.width h2B.length
    h2B.log_sum h2C.width h2C.length h2C.log_sum h2D.width h2D.length h2D.log_sum ,
vectorize.v:s , vectorize.v2:s2 ,
svmclassify.svm_g:v,r -model
    hillasmulticlean/ROC_hillasmulticlean_0.3_0.4_4_7_1_0_svmtype_0_C_100 ,
svmclassify.svm_p:v2,r2 -model
    hillasmulticlean/ROC_hillasmulticlean_0.3_0.4_4_7_1_0_svmtype_0_C_100 ,
hist.hg:svm_g -axis r1.particle_type:100,0,1 , hist.hp:svm_p -axis r1.particle_type:100,0,1 ,
quality.q:hg,hp -algorithm 0 -roc_bin_n 100 , interval.int:q -last_only -set_tag 2 ,
txt.plh:int -filter_tag 2 -idx 0 -idx2 0
    -out hillasmulticlean/ROC_hillasmulticlean_0.3_0.4_4_7_1_0_svmtype_0_C_100.txt
```

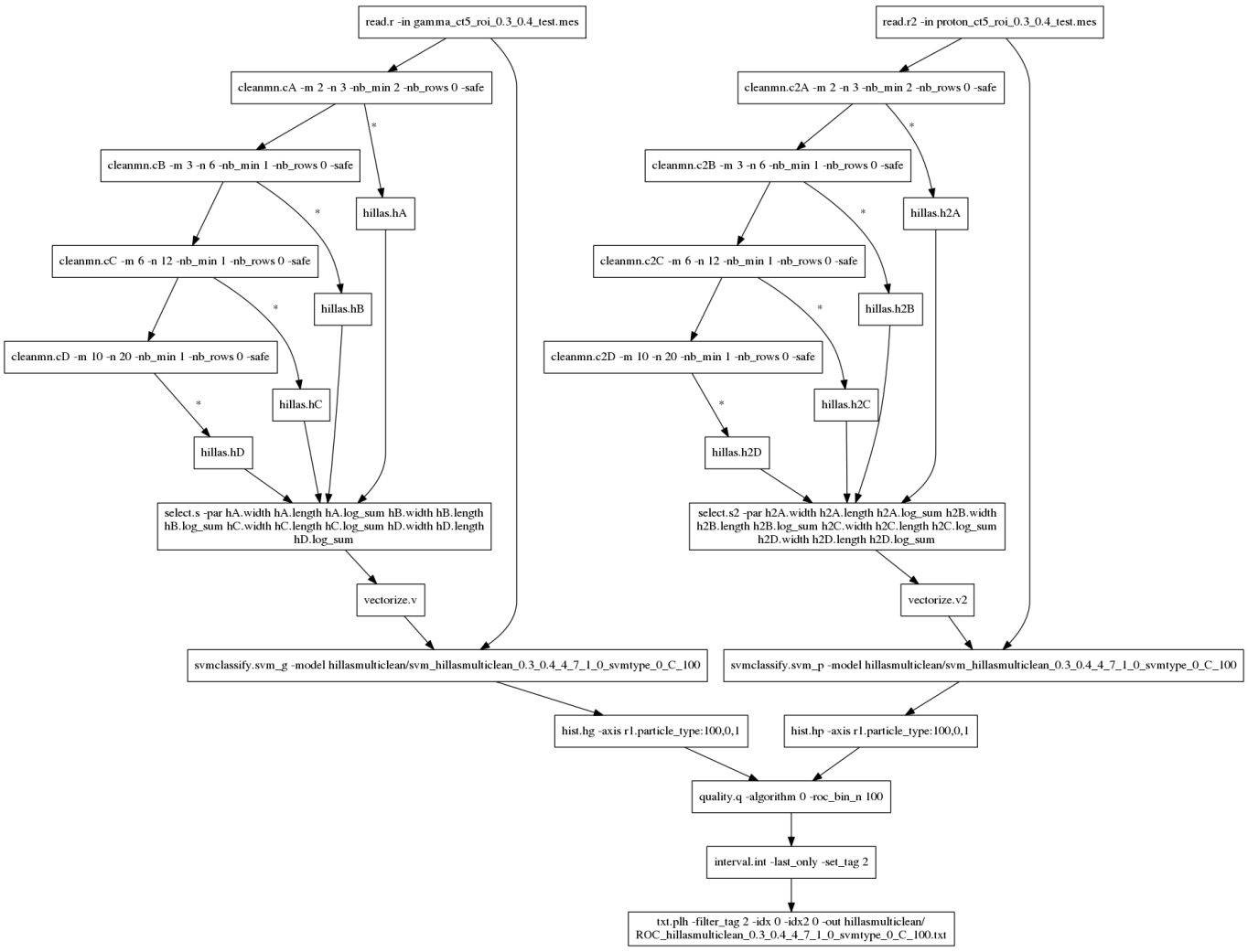


Fig. 3.3.: Pipeline that classifies shower images with a trained SVM model, based on Hillas parameters of differently cleaned shower images.

3.5. Data Structures

MESS has a simple design with very few orthogonal data structures. In any IACT experiment, file I/O, message passing, parameter handling (selecting, cutting, ...), histogramming, plotting, and almost all other operations can be done using the MESS data structures `Event`, `Parset`, `Histset` and few others that store the cofiguration of the experiment. Using such data structures ensures static type safety, so the correct module I/O can be checked at runtime. On the other hand, they are flexible enough to be used in any task from data acquisition over calibration to analysis. In case other data structures (e.g. the definition of the telescope array) are needed, they can be easily added by extending a `Message` structure.

3.5.1. Messages

All MESS data structures are derived from the `Message` structure:

```
typedef struct {
    Message_Header h;
} Message;
```

The `Message_Header` structure is defined as:

```
typedef struct Message_Header {
    unsigned int size; // size of the message, needed e.g. during IO, when a message is skipped
    int type;          // type, like EVENT or PARSET_HILLAS
    TimeID time_id;    // timestamp, used also for synchronization
    int version;       // major library version, so old software can skip messages of new version
    int status;        // if content is valid etc.
    int changed;       // during a step in the pipeline: has the message content been changed?
    long tag;          // can be set by modules to filter out a message
    int last;          // is this the last message in the stream?
    char name[256];    // a human readable `id`, if there are several messages of the same type
    Message_IO_Func encode_toc; // pointer to the message header encoder, e.g. `event_encode_toc`
    Message_IO_Func decode_toc; // pointer to the message header decoder, e.g. `event_decode_toc`
    Message_IO_Func encode;     // pointer to the message data encoder, e.g. `event_encode`
    Message_IO_Func decode;     // pointer to the message data decoder, e.g. `event_decode`
} Message_Header;
```

For any MESS data structure, the first entry must be the `Message_Header` structure. This ensures that each data structure can be casted to a `Message` and be treated the same way during the execution of the pipeline. When the flow in the pipeline is determined or a `Message_Record` for file I/O (see chapter 4) is created, the algorithms only consider the `Message_Header` and ignore the rest of the structure.
In the future, if a data structure is modified (e.g. new properties added), the `version` field in the `Message_Header` must be changed.

3.5.2. Telarray

The `Telarray` structure contains the position of the telescope array and pointers to the telescope structures.

```
typedef struct {
    Message_Header h;
    double lat, lon;            // position of the array on Earth
    double x0, x1, y0, y1;      // dimension of the array (needed e.g. for plotting)
    unsigned short telescope_n; // number of telescopes
    Telescope **telescope;      // the telecopes
} Telarray;
```

Before any data of an experiment can be analyzed, the telescope array structure with information about the telescopes and their cameras must be created, filled and stored in the file `TELARRAY.mes`. When a MESS program is started, the file is assumed to be in the current directory, but a different location can be passed as a parameter.

3.5.3. Telescope

The `Telescope` structure hold the position of a telescope relative to the centre of the array, plus some properties of the telescope are that are needed during the analysis.

```
typedef struct {
    Message_Header h;
    short id;                   // telescope id
    float pos_x, pos_y, pos_z; // position of telescope
    float focal_length;         // focal length
    float mirror_area;          // mirror area
    Camera *camera;             // the cameras
} Telescope;
```

3.5.4. Camera

The `Camera` structure holds all information of a Cherenkov camera that is relevant for analyzing shower images. This includes pixel geometry, pixel positions, number of gain channels, number of samples per gain channel, and the number of neighbours of a pixel and their ids:

```
typedef struct {
    Message_Header h;
    int pix_n;              // number of pixels in camera
    int pix_n_pad;          // allocate 32 if 0<=n<=32, allocate 64 if 32<=n<=64 etc.
    char geometry;          // hexagonal lying, upright, square, ...
    unsigned char nb_max;  // max. number of neighbours a pixel in this camera can have
    float dx, dy;           // distance between pixels in x and y
    float x_min, x_max;     // dimensions of the camera in x
    float y_min, y_max;     // dimensions of the camera in y

    float pix_vt_res[PIXEL_VT_MAX]; // resolution for each pixel value type, if 0: type not supported

    int gain_n;             // number of gain channels
    int sample_n[2];        // number of samples in each gain channel
    int sample_n_pad[2];    // allocate smallest n*32 that is >= pix_n+4 (see pix_n_pad)

    float *x;               // x coordinates of all pixels
    float *y;               // y coordinates of all pixels
    float *nom_x;           // x coordinates of all pixels in the nominal system
    float *nom_y;           // x coordinates of all pixels in the nominal system
    unsigned char *nb_n;    // number of neighbours of each pixel
    unsigned short *nb_flat; // the neighbour ids for each pixel
    unsigned short **nb;    // convenience pointers to `nb_flat`
} Camera;
```

MESS provides programs for detecting the pixel geometries of the cameras of an experiment and for creating lookups for finding neighbors of a pixel quickly.

3.5.5. Events

The `Event` data structure contains the data of the telescope events (televents) that participated in the event. A televent structure can contain high/low gain ADC raw data, which can be sampled or not. It can be stored either as full camera, pixel list, ROI (region of interest (see chapter 4, section 4.4) or square image. For the traces, a window may be defined and stored, but it must be the same window for all pixels (otherwise, it would not be space-efficient). Integrated charges, times of maximum and other pixel value types that were derived from ADC raw data are stored either as full camera, pixel list, ROI or square image.
Events can be passed between modules, plotted or written to files. As all MESS data structures, events can also be used in C/C++ programs. When writing files, the events can be compressed on-the-fly, optimized for small size or for speed (see 4).

3.5.6. Parameter sets

A `Parset` is a set of multidimensional parameters, and can be used for storing calibration parameters, parameterizations of events (e.g. Hillas parameters), subsystem data (e.g. environmental measurements, tracking or pointing), and most other data in $gamma$-ray astronomy that are not camera data They can be passed between modules, histogrammed and plotted or written to MES files and then read back to memory. When writing files, the data of each parameter in a parameter set can be stored as 32 or 64 bit floating point values or as 8 or 16 bit integer after quantization, which is a practical way of lossy floating point number compression. Here are some examples for typical parameters sets of an experiment with N telescopes:

Parameter Set	Parameter names	Dimensions
Hillas parameters	width, length and sum	n_tel, n_tel, ...
Event parameters	particle_type, energy, impact distance, ...	1, 1, n_tel, ...
Broken Pixels	bp1, bp2, ..., bpN	npix1, npix2, ..., npixN

3.5.7. Histograms

MESS provides an `n`-dimensional multi-histogram set data structure, called `Histset`, where:

- *n-dimensional* means that the histogram has `n` axes, each with a label, a description, a unit, ranges and bins. Example: a 3-dimensional histogram of Hillas width, length and log(sum).
- *multi-histograms* exist, because it is helpful to have inherent support for several histograms of the same parameter. When, for example, parameters are telescope-wise, there will be as many histograms as there are telescopes: A histogram of the Hillas width can be calculated for each telescope in the array and kept in a multihist structure. Note that the number of histograms in a multihist can be any number, not only the number of telescopes in the experiment.
- *sets* exist, because often, histograms of different parameters are compared and plotted next to or on top of each other. Also, when they are written to a file, their affiliation should be conserved. In such cases, it makes sense to have these histograms combined in a set. Example: Histograms of Hillas parameters.

On first look rather complex, this extra effort solves the major part of the problems in data analysis, data inspection, plotting and I/O. It is especially convenient within a MESS pipeline, where a `Histset` is a message and can be passed around modules, without thinking about how many telescopes there are, how to plot them and how to store everything to file.
The bins of an n-dimensional histogram are stored flat in memory in row-major order. The associated errors of the bins can also be stored.

Regular and irregular axes A regular axis of a histogram is defined by the number of bins, the minimum value and the maximum value. It is also possible to define an irregular axis of a histogram, which is done by passing the bin edges.

Axis transformations A regular axis can be set to have a transformation function, like `log`, `log10`, `sqrt` etc., so during creation of the axis, the bin ranges are calculated accordingly. Later, when values are filled into the histogram, the transformation is also applied in order to find the correct bin fast (in $\mathcal{O}(1)$).
Irregular axes cannot have transformation functions, and to find a bin on an irregular axis, a binary search is done.

From several Parsets to a Histset Let `pss` be an array of `Parsets` that need to be histogrammed against each other, so its length is the number of axes (dimensions) of each resulting `Hist`. Example: `Parset *pss[2];`
The number of parameters in the two `Parsets` must be equal and so must be the number of dimensions must be equal for each parameter: `|h1.a| == |h2.a|`, or in code:
`pss[0]->p[k].n == pss[1]->p[k].n`, for each `k`.
The Histset suitable for histogramming those Parsets must contain 3 Multihists (for `a`, `b`, and `c`), each containing a number of 2D Hists:

pss[0]	pss[1]
h1.a	h2.a
h1.b	h2.b
h1.c	h2.c

- h1.a vs. h2.a
- h1.b vs. h2.b
- h1.c vs. h2.c

The three histograms are independent of each other and can have different axis ranges, number of bins and axis transformation functions.

Creating a Histset A Histset can be created manually in a C program by calling the functions:

```
Histset *histset_create(int type, int n)
Multihist *multihist_create(int n)
Hist *hist_create(int axis_n)
Axis *axis_create(int n, double range_min, double range_max, int transform)
```

Obviously this involves loops and can be quite tedious if the histograms have different axes.
That is why in most cases the hist module should be used, which can be invoked from the command line in a mess pipeline:

Example 1 Let h be a hillas module that sends a Parset with Hillas parameters to a Hist module hist0 in a mess pipeline. 1D histograms of the three Hillas parameters width, length and log(sum) can then be created:

```
hist.hist0:h -axis
    h.width:10,0,0.25 h.length:10,0,0.4 h.sum:50,0.03,100,log
```

For each telescope, Hillas width is histogrammed in a range from 0 to 0.25 degrees with 10 bins, Hillas length accordingly. In the histogram of the Hillas sum, the x-axis is transformed with the log function before it is divided into 50 bins from 0.03 to 100.

Example 2 2D histograms can also be created easily. From h in the above example, three 2D histograms can be created: width vs. length, width vs. log(sum) and length vs. log(sum):

```
hist.hist0:h -axis
    h.width:10,0,0.25/h.length:20,0,0.4
    h.width:10,0,0.25/h.sum:50,0.03,100,log
    h.length:20,0,0.4/h.sum:80,0.03,100,log
```

As can be seen, each axis can have different binning and transformation functions.

Example 3 Histogramming across several Parsets is also supported. As an example, assume that the effect of image cleaning on the Hillas parameters is studied: In a mess pipeline, images are cleaned with 3-6 cleaning in one branch, and in another branch they are cleaned with 5-10 cleaning. If the Hillas parameters of both cleaned images are calculated and hillas_3_6.width vs. hillas_5_10.width, hillas_3_6.length vs. hillas_5_10.length and hillas_3_6.sum vs. hillas_5_10.sum should be histogrammed, it is enough to write:

```
hist.hist0:hillas_3_6,hillas_5_10 -axis
    hillas_3_6.width:10,0,0.25/hillas_5_10.width:10,0,0.25
    hillas_3_6.length:20,0,0.4/hillas_5_10.length:20,0,0.4
    hillas_3_6.sum:50,0.03,100,log/hillas_5_10.sum:50,0.03,100,log
```

gsl compatibility It is straightforward to use the histograms with the GNU Scientific Library [39] (`gsl`), which offers many useful routines for 1D and 2D histograms. Since `gsl` and MESS use the same memory layout for histograms, the pointer to the histogram data can simply be given and no data needs to be copied: ‘

```
Hist *h = hs->mh[2]->h[4];  // from the Histset choose the 3. Multihist and the 5. Hist
gsl_histogram *g = malloc(sizeof(gsl_histogram));
g->n = h->axis[0]->n;
g->range = h->axis[0]->range;
g->bin = h->bin;
```

Listing 3.1: Example for preparing a 1D MESS histogram for use in gnuplot.

```
Hist *h = hs->mh[2]->h[4];  // from the Histset choose the 3. Multihist and the 5. Hist
gsl_histogram2d *g = malloc(sizeof(gsl_histogram2d));
g->nx = h->axis[0]->n;
g->ny = h->axis[1]->n;
g->xrange = h->axis[0]->range;
g->yrange = h->axis[1]->range;
g->bin = h->bin;
```

Listing 3.2: Example for preparing a 2D MESS histogram for use in gnuplot.

The histograms can then be used in gsl. To save writing the code, MESS provides the functions:

```
gsl_histogram *g = hist_create_gsl_1d(Hist *h);
gsl_histogram2d *g = hist_create_gsl_2d(Hist *h);
```

This example pipeline produces a plot of multiple histograms of Hillas parameters of 200.000 events with an average size of 3 kB (see figure 3.4) in less than 4 seconds. Like with all MESS data structures, the histogram structure and the corresponding functions can not only be used in modules in a pipeline on the command line, but also from within a C or C++ program.

```
mess -iter_max 200000 -micropipe
    read.r: -in phase1_gamma_zenith_20_azimuth_000.mes ,
    cleanmn.c:r ,
    hillas.h:c ,
    hist.hist:h -axis
        h.width::0,0.012,0.02,0.03,0.05,0.06,0.075,0.1,0.3,0.35
        h.length:20,0,0.75
        h.length:100,0,0.7/h.width:100,0,0.3
        h.width:200,0.05,2,1_x/h.length:100,0.15,0.55,pow10
        h.cx:100,-2.5,2.5/h.cy:100,-2.5,2.5
        h.log_sum:50,3,10/h.phi:50,-1.6,1.6 ,
    interval.hist_last:hist -last_only ,
    plothist.p:hist_last
```

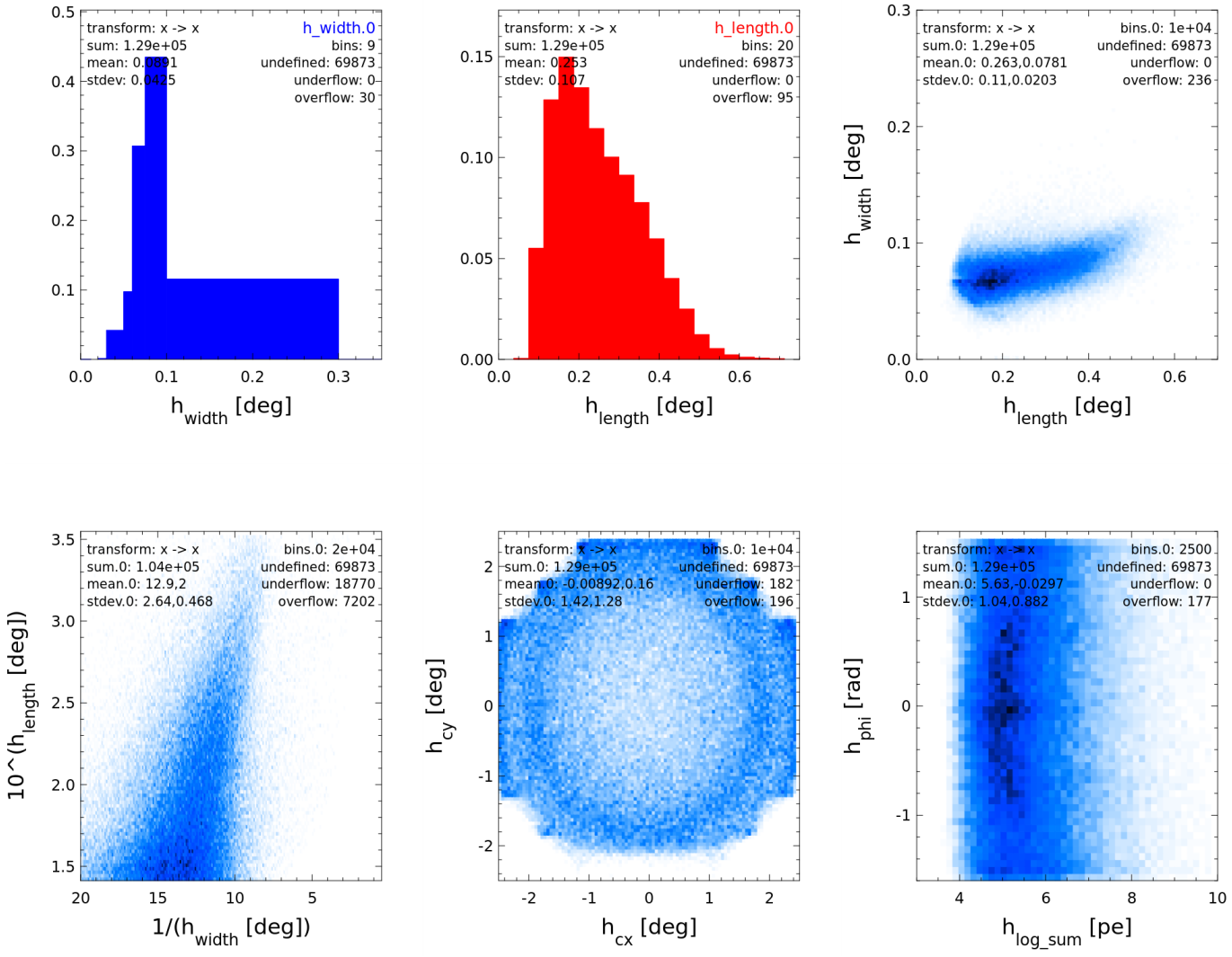


Fig. 3.4.: Example of a MESS pipeline (top) that produces 1D- and 2D-histograms (bottom), showing different axis definitions and transformations. The result of the definition of a irregular Hillas width axis with a large bin ranging from 0.1 to 0.3 can be seen in the upper left plot.
These plots are simply shown to illustrate the histogramming features of MESS.

3.5.8. Time and IDs

When recording events, they need to be tagged with an id, so they can be distinguished later. Additionally, their arrival time needs to be stored, which is necessary for applications such as light curves.
Here are some possibilities that were considered for storing seconds and nanoseconds.

32 bits + 32 bits The time stamps needs to be precise up to nanoseconds, so a single 64 bit integer is not enough, because using 32 bit for the nanoseconds leaves 32 bits for the seconds, which leads to the year 2038 problem. The 32 bits for the seconds are signed, so dates before 1970.01.01 0:00:00.0 can be represented. This is not necessary here, so instead of the signed integer, an unsigned integer could be used, which would extend the largest time possible to represent from year 2038 to after year 2200. The resulting incompatibilities with the system library's time functions (convert seconds to date etc.), which use signed integers, can be ignored, because the rare cases the system library needs to be invoked can be separately handled. However, if times far in the past or future occur, this scheme cannot represent them.

34 bits + 30 bits Using 30 bits is enough to represent nanoseconds ($2^{30} = 1073741824$, but $2^{29} = 536870912$), so one could use a 64 bit integer and reserve 30 bits for the nanoseconds and 34 bits for the seconds, but since sizes of data types are multiples of 8 bit, having 34 bits for the seconds and 30 bits for the nanoseconds requires several bit level operations whenever time stamps need to be accessed.
Also, in case two events arrive at exactly the same time, there needs to be an additional number to distinguish them.

64 bits + 32 bits From the above follows that the seconds should be 64 bit integers, the nanoseconds should be 32 bit integers, and the remaining 32 bit with least significance for distinguishing events with the same time. It could be used for the run number and/or subarray number, so if two subarrays record events at exactly the same time, the events get different `time_id`s and can be distinguished.
Usually, seconds and nanoseconds are stored separately, which requires two comparisons whenever the arrival time of two events needs to be compared. If a `union` is used, the numbers can be merged into a bit field which can then be accessed as a larger data type, e.g. a 64 bit or a 128 bit integer. 128 bit integers are supported on 64 bit systems with modern compilers.

In MESS, the third possibility is used for storing seconds, nanoseconds and run number:

```
typedef union {
    __uint128_t id;
    struct {
        unsigned int run;
        unsigned int ns;
        unsigned long s;
    } time;
} TimeID;
```

It allows to access these three numbers as a single number, with the seconds being the most significant part of it, the nanoseconds coming next, and the run number last. Having a single number makes event IDs obsolete and simplifies comparisons, which is convenient and error prone.
Every message in MESS (event, parameter set, histogram, ...) has such a `time_id` and it is used for synchronization of input streams, for example.

3.6. Non-pipelined programming with MESS libraries

There are cases where the pipeline approach does not work or where it is simply convenient to write a program than to launch a pipeline on the command line. For these cases, the MESS library exists, which combines the complete functionality of all MESS modules in a single file that can be used within C and C++ programs. There is no need to write or maintain the library, because it is created automatically from the modules: All functions and data structures of all MESS modules that are not `static` (C) or `private` (C++) are listed in the header file `MESS.h` and compiled into the library `libMESS.so`.
Listing 3.3 shows an example of a program rather that reads a given file with shower images, cleans the images with M, N-cleaning and the given thresholds, and then prints the Hillas parameters to the console.

```
#include "MESS.h"

int main(int argc, char **argv)
{
    system_init(argc, argv); // check if little endian, int=4bytes, ..
    WORK_DIR = argv[1];
    config_init(); // load telescope lookups
    Parset *ps = hillas_parset_create("hillas");
    Fileinfo *fi = file_open_read(NULL, argv[2], "mes");
    float m=atof(argv[3]), n=atof(argv[4]);
    Message_Record *mr = read_toc_filter(fi, "EVENT");
    Event *e = mr->msg[0];
    printf("---> Hillas parameters for each telescope:\n");
    while (!read_message_record(fi, mr)) {
        parset_reset(ps);
        for (int i = 0; i < telarray->telescope_n; i++) {
            Televent *t = e->tel[i];
            cleanmn_televent(t, PIXEL_VT_INTEGRATED_CHARGE, m, n, 1, 0, 0, 0, 0, 0);
            hillas_televent(t, PIXEL_VT_INTEGRATED_CHARGE, ps, 0);
        }
        event_print(stdout, e);
        parset_print(stdout, ps);
    }
    file_closedown(fi);
    return 0;
}
```

Listing 3.3: Example of a program using libMESS.so to read a .mes file clean the images and calculate their Hillas parameters.

Since there are no dependencies, it is simple to build a program using libMESS.so from within the MESS directory:

```
gcc -march=native -O2 -std=c99 -Iinclude -o abc abc.c -Llib -lMESS
```

For programs that only require the core functionality of MESS, the lighter library `libmess.so` (size: 157 kB) and the header file `mess.h` can be used instead of `libMESS.so` (size: 1.3 MB) and `MESS.h`. It provides the basic data structures `Event` (array event), `Televent` (telescope event), `Camera` (including camera lookups, like pixel positions, neighbor ids), `Parset` (parameter set), `Histset` (histogram set) and the corresponding functions to handle them, including compressed I/O from/to .mes files for all of them.
In this example, the program must be linked against `libMESS.so`, because `cleanmn_televent` and `hillas_televent` are not part of `libmess.so`,

3.7. Miscellaneous

3.7.1. Creating a module

Modules are created in a similar way as executables, with the difference that modules must be .so (shared object) files, so they can be dynamically loaded by the `mess` main program when a pipeline is created. Here is the command line to build a MESS module from within the MESS directory:

```
gcc -march=native -O2 -std=c99 -Iinclude -shared -o abc.so abc.c -Llib -lmess
```

In case some of the module's functions should be used by other programs, a header file is needed. It can be created with the program `c2h` from the developer tools:

```
c2h abc.c > abc.h
```

The MESS build system does all this automatically, only the name of the module has to be added to the list of modules in the `Makefile`.

3.7.2. Dependencies

The core library and most modules do not have any dependencies, whereas some optional modules depend on `gsl` for math, `dlib` for machine learning or `plplot` for plotting. The modules that read data of the different experiments or cameras depend on their respective libraries: For HESS raw data, calibrated data and Monte Carlo data the HESS libraries and `root` are needed, for the CTA Monte Carlo data `simhess`, and for CHEC raw data `FITS`, `TargetIO` and `TargetDrive`.
Despite these dependencies, MESS is simple to maintain, because its code base is small:

Part	Size	Build time
core library	7000 lines of code	4.0 s
core modules	8600 lines of code	4.8 s
complete package	29000 lines of code	102 s
documentation	382 kB	5.3 s

The complete package also includes testing modules, examples, and developer tools. and machine learning modules and readers for external formats (all written in C++). Compiling the complete package takes disproportionately longer, because the machine learning modules and the readers for external formats are written in C++, which takes much longer to compile. There are some modules that were used to test new algorithms, but they can be removed. This would reduce the core modules by ≈ 3000 lines of code, and and the complete package by ≈ 8000 lines of code.

3.7.3. Memory usage

Typical programs linked against the MESS library have a low memory footprint. For a HESS analysis pipeline, usually 4 MB of RAM are used, for a CTA analysis pipeline usually 200 MB. That is, if the data is read from MES files - otherwise, the modules `readhess` or `readsimtel` need to be involved, which require more resources than the `read` module of MESS. The differences in memory usage are due to the different number of telescopes and pixels and because the data of the CTA simulations is sampled.

3.7.4. Python

Python is a user-friendly alternative, if the required algorithms already exist as a C library compatible with the Python interface - otherwise, the code will be orders of magnitude slower. Also, since Python scripts require more CPU power and RAM than C programs, they are more expensive to run on a computing cluster. Nevertheless, Python is very popular at the moment, and many scientists do not want to learn C. MESS is user-friendly and efficient: it is written in C, uses simple data structures, and the core library and most modules do not have any dependencies, which makes it easy to create a wrapper and use it from within Python.

3.7.5. Parameter parsing

Programs and modules can have multiple parameters that need to be parsed, and some parameters have several arguments. Since parsing parameters is tedious and error-prone, MESS provides structures and functions for this (see the listings below for examples). With `mess -help <module>`, all parameters of a module and their descriptions are printed on the command line, which is useful for always up-to-date help text and documentation generation.

```
#include "mess.h"

static Input_Parameter in[] = {
  {"quiet",   "if flag is set, do not log"}, // log or not
  {"logfile", "name of log file", "stderr"}, // pipe or file (default: stderr)
  {"!out",    "output file name"},           // mandatory argument: !
  {"numbers", "a list of numbers"},          // several numbers
  {"*type",   "which type to use"},          // manual parsing: *
};

int main(int argc, char **argv)
{
  parse_input_parameters(argc, argv, ARRAY_SIZE(in), in);
  int i, n, *num, type, quiet = in[0].found;
  FILE *logfile = logfile_open(in[1].v[0]);
  char *out = in[2].v[0];
  n = in[3].n; // number of arguments of "-numbers"
  if (n) {
      num = MALLOC(int, n);
      for (i = 0; i < n; i++)
      num[i] = atoi(in[3].v[i]);
  }
  type = ...; // ..... manual parsing
  ...
}
```

Listing 3.4: Example of parameter parsing with MESS: source code.

```
> partest
[ERROR] src/libmess/util.c  get_param_values  L 486 : parameter `out` is needed for 'partest', but could not be
    found
> partest -out
[ERROR] src/libmess/util.c  get_param_values  L 483 : parameter `out` is needed for 'bin/example_partest', and it
     must have an argument
> partest -out abc
> partest -out abc
[ERROR] src/io/partest.c  main  L 19 : at least five arguments required for parameter 'numbers'
> partest -out abc -numbers -0.2 2.222 1 99 0.123 -13
```

Listing 3.5: Example of parameter parsing with MESS: command line test.

3.7.6. Advanced Vector Extensions 2

AVX2 (Advanced Vector Extensions 2) is an instruction set for parallel calculations, which is supported by all modern desktop and server CPUs. Parallel calculations can save time, when many similar independent calculations need to be performed. This does not mean that all algorithms can be implemented in AVX2 and will run faster. In this work, some algorithms have been be designed specifically for AVX2.
Processors that support AVX2 have several 256 bit registers per core, and the 256 bits of each register can be used in different ways:

bit mask	(un)signed integer	floating point
256×1 bit	32×8 bit	8×32 bit
	16×16 bit	4×64 bit
	8×32 bit	
	4×64 bit	

Depending on the instruction, the content of a register is interpreted differently. Since scientific raw data in this context consist mostly of 16 bit unsigned integers, operations on the other data types are not explained here.
Data is usually loaded from memory into one or more AVX2 registers and then, instructions that may involve several registers operate element-wise on the contents of the registers and produce results that can be written back to memory again. For each data type, there is a wide range of instructions, and each instruction has the target data type encoded in its name.

Example The instruction for adding two vectors of signed shorts is called `_mm256_add_epi16`, where the prefix `_mm256_` is the same for all AVX2 instructions, the `add` is the operation to be performed and `epi16` is the target data type (in this case signed 16 bit integers).
In order to load two vectors of 16 signed shorts $A_{0..15}$ and $B_{0..15}$ from memory to a register, add them and store the result back to memory, one could write the following code:

```
void add_vector(unsigned short *A, unsigned short *B, unsigned short *result)
{
    __m256i a, b;
    a = _mm256_loadu_si256((__m256i const *) A);
    b = _mm256_loadu_si256((__m256i const *) B);
    a = _mm256_add_epi16(a, b);
    _mm256_storeu_si256((__m256i *) result, a);
}
```

Unfortunately, not all AVX2 operations are that straightforward, but instead work separately on the low and on the high 128 bits of a register.

Example Consider the operation `_mm256_slli_si256(__m256i a, const int imm8)`, which shifts `a` left by `imm8` bytes, but - contrarily to what one would expect - not as a whole, but per 128 bit lane. Let v be eight 32 bit numbers: `(1,2,3,4,5,6,7,8)`. After shifting v left by 4 bytes with `_mm256_slli_si256`, it is not `v = (2,3,4,5,6,7,8,0)` but instead `v = (2,3,4,0,5,6,7,0)`. This behavior is common with many AVX2 instructions and requires a extra measures, such as shuffling and permuting.

The size and number of the AVX2 registers, the available operations, their latencies (how many CPU cycles are needed to perform the instruction) and their throughput (how many instructions can be performed in parallel) give some constraints on which algorithms can be implemented efficiently. The Intel Intrinsics Guide [29] is a valuable AVX2 reference, and Agner Fogs instruction tables [30] provide the latencies and throughputs of all instructions.

Usage in MESS In MESS, the trace integration and most compression algorithms use AVX2. Although compared to scalar CPU code, a theoretical speedup of factor 16 is possible, in practice a speedup of $\approx$ factor 10 is achieved. because data needs to be rearranged and because some operations are more difficult to implement with AVX2 than with the usual CPU instructions.

3.8. Components of MESS

MESS consists of several modules and programs for analyzing IACT data. The modules can be used to create MESS pipelines and the programs can be directly used. Currently, there are more than 70 modules and more than 20 programs. Additionally, there are several scripts that contain pipelines that are too complex to remember, and even whole calibration and analysis chains consisting of several programs and pipelines from lookup creation to sky map plotting. New scripts and programs can be easily created and added by filling the skeleton module and adding the name to the build system script.

3.8.1. List of 50 selected modules

bpseverity calculate the severity of a broken pixel pattern in different ways: number of BP, z (includes clustering and distance from camera center), mean reconstruction errors for MC events; see chapter 5

calibrate flat fielding and broken pixel identification

cleanmn image cleaning with many options; creates a mask to mark the cleaned pixels if cleaned pixels need to be kept

convert convert shower images between full camera image, pixel list, square grid image and ROI

diffevent check if events are the equal (from event header down to each sample)

dnnclassify classify shower image using a DNN (Deep Neural Network)

dnntrain train a DNN on shower images

dup duplicate input message (deep memory copy for all message types)

eventop different calibration-related operations on events, like adding and subtracting (sampled) events; also allows to calculate pedestal from non-shower pixels and calculate running average to subtract pedestal; simpler than the `calibrate` module and suitable for online analysis

fastbdtclassify classify based on trained model of a Boosted Decision Tree and input vector

fastbdttrain train a Boosted Decision Tree with input vectors of two classes

filterevent filter event based on properties: minimal or maximal amplitude, distance of centre of gravity from camera centre, minimal or maximal number of pixels, minimum/maximum intensity of a pixel, minimum/maximum mean intensity of all border pixels, minimum/maximum ROI size

filterps filter parameter sets based on AND/OR combination of conditions (e.g. Hillas width of telescope 5 must be larger than 0.1 and log(amplitude) of telescope 2 must be smaller than 3)

fitgaussiansalongaxis rotate the shower so that major Hillas axis is parallel to x-axis, then bin in x and in each bin project the pixel intensities onto the y-axis and histogram; finally fit a Gaussian and provide goodness of fit

fitlorentzalongaxis like `fitgaussiansalongaxis`, but with a Lorentzian instead of a Gaussian

generate toy Monte Carlo event generator for γ-ray, proton and muon shower images

hillas Hillas analysis

hist create n-dimensional histograms with or without irregular axes, axis transformation functions etc.

histop different operations on histograms, e.g. scaling, or multiplication, division, addition or subtraction of two histograms, Gaussian smoothing of 1D- or 2D-histograms, or n-times moving average of 1D-histograms

inspectevent print specified event, televent, pixel and sample or other data

integratesamples smooth waveform and integrate around maximum

interval after the pipeline has run n times, tag message for the next run, then wait again n times. Also works with time intervals, and with marking as empty instead of tagging.

islands find the connected components (islands) of a cleaned shower image

meantelevent calculate the mean telescope event; can be used for flat fielding and to find broken pixels

merge merge several messages to one; can be used to merge parameter sets before they are vectorized for a machine learning module; can also be used to merge events, parameter sets and histograms before writing combined message to file.

plotevent plot events; shortcuts for simple command (show traces, show pixel IDs, show pixel intensities, switch to raw data view, switch to array view, show Hillas parameters, show broken pixels, ...) and integrated command line for complex commands (change color palette, define color for pixels above/below some threshold, ...)

plothist plot multihistsets (histograms for several parameters and several telescopes); any of the histograms can be plotted in one plot; axes can be transformed with the standard functions (log, exp, sqr, sqrt, ..)

plotimage plot images, for example telescope events that have been converted to images of high resolution for use with image processing algorithms (e.g. Deep Neural Networks)

pointingcorrection apply pointing correction

print print the given message to the console (from header to deepest struct); useful for debugging

quality make q-factor or ROC curves from two histograms

random fills data structures with random values

read read `.mes` files; filters can be applied to read only certain message types or parameter and histogram sets with certain names should be read

readhess read HESS .root files; reads raw data, calibrated data, broken pixel patterns and pointing correction

readsimtel read `simtel` files; reads raw data and calibrated data

relay relay status of a message to other messages; useful if a module is affected by a filtering process in a different module. Example: Hillas parameters of events are calculated and written along with the event to a file; if the Hillas parameters cannot be calculated (e.g. too few pixels in event), no event must be written. This can be achieved, if the status of the Hillas parameters is relayed to the event. The `write` module will not write anything if all messages are empty.

ringbg calculate a Li-Ma significance map with the ring background method

select select a subsets of the input messages: if a module has several outputs, but only some of them are wanted; or select telescope events from events (e.g. only data of telescope 5 is wanted); or select parameters from parameter sets (only Hillas length is wanted); or select a dimension of a parameter set (only Hillas parameters of telescope 5 are wanted); or select subsets of multihistsets

showergeometry parameterize the input shower with 34 different geometry and histogram algorithms

smooth smooth the shower image in different ways

split split event into many events (each telescope event becomes a mono event), split parameter set into several parameter sets (each parameter becomes a parameter set - or each dimension becomes a parameter set)

statevent calculate statistics about events (telescope multiplicities and participation fractions, mean number of pixels/event, ...

statps calculate statistics about parameter sets (mean and standard deviation of each parameter and each dimension)

svmclassify classify based on trained model of a Support Vector Machine and input vector

svmtrain train a Support Vector Machine with input vectors of two classes

sync synchronize several input streams; useful if one streams provides messages with high frequency and another stream with low frequency. Example: events arrive with 10 kHz, broken pixel patterns with 1 Hz.

txt dump parameter sets and histograms to text files readable by gnuplot for more complicated plots

vectorize concatenate parameter sets to one single vector so it can for example be fed to machine learning algorithms

watershed calculate the watershed transform of a shower image

write write events, parameter sets, histograms and other MESS data structures to `.mes` files (regular file or named pipe) with selected compression level *none*, *fast* or *small*. Parameter sets can be written in sparse format, if they are expected to contain data for only few telescopes.

3.8.2. List of selected MESS programs

adc2pe single photoelectron calibration
c2h create C header files from C source files
checcam determine camera configuration of CHEC camera from data file
codestats provide code statistics (number of codes, comment, punctuation and alphanumeric characters, number of statementsm, branches and loops
concat concatenate `mes` files
h2txt create text files fom C header files by extracting all comments
hesscam determine camera configurations of HESS cameras from data file
mess create complex analysis pipelines on the command line
onlinecalibration do an online analysis with trace analysis, pedestal estimation, image cleaning, calculation of Hillas parameter, and plotting or writing to file of calibrated events
runlist create run list based on broken pixel filter criteria
simtelcam determine camera configuration of a camera from a simtel MC file
splitfile split file into two files with specified ratio of messages in each file
tellookup creates neighbour lookups for cameras, taking into account different geometries
txt2html create HTML file from `mes` file, by interpreting MESS markup
txt2latex create Latex file from `mes` file, by interpreting MESS markup
txtdir2htmldir create HTML file with menus and subpages from directory with text files
txtdir2latexbook create Latex book with chapters and sections from directory with text files

Furthermore, there are several test and example programs and more than 30 bash scripts for creation of lookups, file conversions etc.

3.9. Summary

The software framework MESS has been developed to make data handling in IACT experiments simpler and more efficient. It provides scientists analyzing IACT data with robust algorithms and useful data structures, and can be used to create pipeline of modules on the command line or as a library.
A pipeline is the recommended approach for simple problems, since many of them can be solved on the command line with the already existing modules. More complex problems can be solved by writing a new module that is then integrated into the pipeline or by writing a standalone program that uses the library. With more than 70 modules and dozens of programs, MESS can solve most of the calibration and analysis problems of IACT experiments on the command line.
In the end, it is the scientific ideas and the algorithms that count. However, the right tools can make testing and developing the ideas and algorithms a lot faster. With MESS, scientists can spend more time on science and less time on waiting for their data files to be read and processed or on searching through hierarchies of nested classes just to learn how to access a data structure.
Many parts of MESS are general and can be directly used in other experiments, for example the developer tools, the message system, the machine learning modules, the histogram plotter or the compression algorithms.

4. The MES File Format for IACT Data

MES (Modular Efficient Storage) is a file format for IACT experiments and is tailored to the MESS data structures, which are mainly Cherenkov camera images, parameter sets and multi-dimensional histograms. Since these data structures allow to describe raw data, calibrated data, parameterizations, subsystem data, histograms, sky maps, spectra and more, the MES file format can be used in all stages of an IACT experiment. It offers fast lossless compression for the camera raw data and optional lossy compression for the calibrated camera data and the subsystem data.
This chapter describes the requirements of a file format of an IACT experiment and presents the MES file format, which has been proposed as a prototype for the CTA file format and which fulfills all the requirements.

Requirements for an IACT Data Format The expected data rate in CTA is 70 GB/s, which exceeds the data rates of current IACT experiments by $\approx$ two orders of magnitude. The ≈ 100 telescopes will deliver huge data rates of up to 3 GB/s per telescope. The data undergoes several reduction procedures and it is necessary to be able to store it at each step. A file format that can be used in such an experiment needs strong compression and high (de)compression speed.
It must also be flexible in terms of data storage, because an IACT experiment produces diverse data sets. The I/O routines should to be easy to maintain and consist of only few lines of code and have almost no dependencies. Since these requirements are not independent of each other, there is always a trade-off during optimization.

Currently used file formats The file formats currently used in IACT experiments have several disadvantages. ROOT is a very large and complex software package with many dependencies. It requires to use C++ and the I/O is not very fast. FITS does not allow to efficiently store lists of arbitrary length, which are needed for cleaned shower images. Both formats offer compression, but the algorithms are not optimized for IACT data and the I/O will become even slower.

4.1. Description of the MES File Format

The MES (Modular Efficient Storage) file format has been developed as a prototype for the CTA file format and fulfills the above requirements. It has already been successfully used to store HESS and CTA data. The MESS data structures (see chapter 3) and the MES file format handle data in a coherent way and make it accessible by I/O routines that are easy to use.

4.1.1. Header

A MES file starts with a header that defines the content of the file, so when a reader opens the file it knows what memory structures to allocate.
The header is also necessary to prevent data blocks from writing their headers every time the data is written. For example, when parameter sets are written, the names of the parameters, their units and their dimensions do not change, so it would be a waste of space to store them every time the data changes. For large data blocks, like camera images, it would not make much difference, but for small data blocks, like parameter sets, the header can be larger than the data.

4.1.2. Camera data

The file format is able to handle data of experiments with different cameras. In CTA, for example, there are six different types of cameras, which have different numbers of pixels, samples per pixel, gain channels and different

pixel geometries. It is possible to:

- store the raw camera data of both gain channels, either simple ADC values or sampled data; if sampled, either the full trace or just a window
- store calibrated camera data and arbitrary other pixel values, like time of maximum or time over threshold; most of these values must be stored as floating point numbers (it would be desirable to be able to give the desired precision before writing to disk, in order to save space)
- store full camera images, pixel lists or regions of interest (ROI, see section 4.4)
- store versioning, tagging and size information for all important data structures, so that new code can read old data and old code can read (parts of the) new data

The content can vary from event to event and from telescope event to telescope event.
Such flexibility is necessary for the different use cases in an experiment, where different data sets need to be stored, for example:

- calibration events: full camera, all traces and all gain channels
- data taking: full camera, all traces, all gain channels
- raw MC events for trace analysis: full camera, all traces
- calibrated MC events for event reconstruction: pixel list, integrated charge, time of maximum
- calibrated data: full camera, pixel list or ROI intensities and times of maximum
- data that will be archived: pixel list or ROI, some pixels with intensity and time of maximum, some with trace

Lossless compression of raw data To reduce the enormous data rates efficiently, the file format supports fast lossless compression, which is explained in the next section.

Lossy compression of derived data After trace analysis, the derived pixel values are usually floating point numbers. Most of the time, not the full range and precision are needed, so the values can be quantized and then compressed in order to consume less space. The quantization and compression algorithm that is used in the MES file format is explain in the next section.

4.1.3. Parameter sets

Subsystem data (e.g. environmental measurements, drive system monitoring), event parameterizations (e.g. Hillas parameters), and many other data sets in IACT experiments consist of time series of several sets of multidimensional parameters. The MES file format offers support for storing them.

Different dimensions These parameter sets are not necessarily a data cube, because the parameters can have different dimensions. Event parameters, for example, have several scalar attributes (like energy or direction), and also some vector attributes (like the impact distance). It is also important to keep some important meta information in the header, such as `name`, `description` and `unit`.

Lossy compression The parameters are real numbers and it is possible to store them in the desired range and precision. Since not all parameters require the same precision, the MES data format offers the choice between 32/64 bit floating point numbers or quantization and compression.
Example: For storing Hillas parameters, 32 bit floats are sufficient, while for storing supernova explosion energies, 64 bit floats are needed. And for storing the status of a pixel (whether is is broken etc.), a small integer number is enough.
The quantization and compression algorithm is the same as

Sparse data If a parameter has high dimensionality, but most of the time only few dimensions have valid data, the data is called *sparse*. When it is stored, space can be saved if the valid dimensions are marked with indices.
Example: In an array with 50 telescopes, the Hillas parameters are to be saved for each event. Usually, an event triggers only few telescopes. Thus, a sparse storage method should be used for the Hillas parameters, so only the

indices of the triggered telescopes and their Hillas parameters will be saved. However, if it is expected that almost all dimensions contain a valid value, all values should be saved in order to save the space for the indices., The few invalid values are set to a special value (e.g. `-FLT_MAX` or `-DBL_MAX`).

4.1.4. Histogram sets

Multidimensional sets of histograms are supported by the MES file format, because they are ubiquitous in calibration and analysis (e.g. single PE histograms, lookups, sky maps). Since histograms are usually stored only once, and not in a time series, no compression algorithms are applied when they are saved.

4.1.5. Other data

Other necessary data types are supported as well, for example the telescope definitions (position, mirror size, ..) and the camera lookups (pixel locations, list of neighbors for each pixel). Other data types can be easily added to the file format.

4.1.6. Synchronization

When reading from several streams simultaneously, it is possible to synchronize them by comparing their time stamps. This is explained in 4.5.

4.1.7. Versioning

The MES file format supports versions, so that when an old file is opened with new software, it can still be read, and when a new file is opened with old software, at least some parts of it can be read.

4.1.8. Message records

The MES file format uses message records as containers for messages during file I/O. It allows to store different synchronized data streams inside one file, which is useful, because it greatly simplifies the I/O. For example, when events are read, it is convenient to have their Monte Carlo and Hillas parameters attached to them (see figure 4.1).
A message record provides a global header that applies to all the messages it contains:

- the version information
- a `time_id` (see 3) with seconds (64 bit), nanoseconds (32 bit) and another 32 bit value, accessible as a 128 bit value
- its size, for fast skipping if the version is unknown or during synchronization with other files

Since the individual headers of the messages are written once in the header of the message record, they do not have to be written each time a message is written. Thus, the relative savings for small messages whose payload size $\approx$ header size are significant, because only the data are written.
The structure of the message record is also defined in the header and must not change during I/O. If data are added or removed, a new message record block must be written. Only messages with the same timestamp can be inside the same message record. This allows processing and synchronizing multiple streams efficiently, without any memory allocation during I/O.

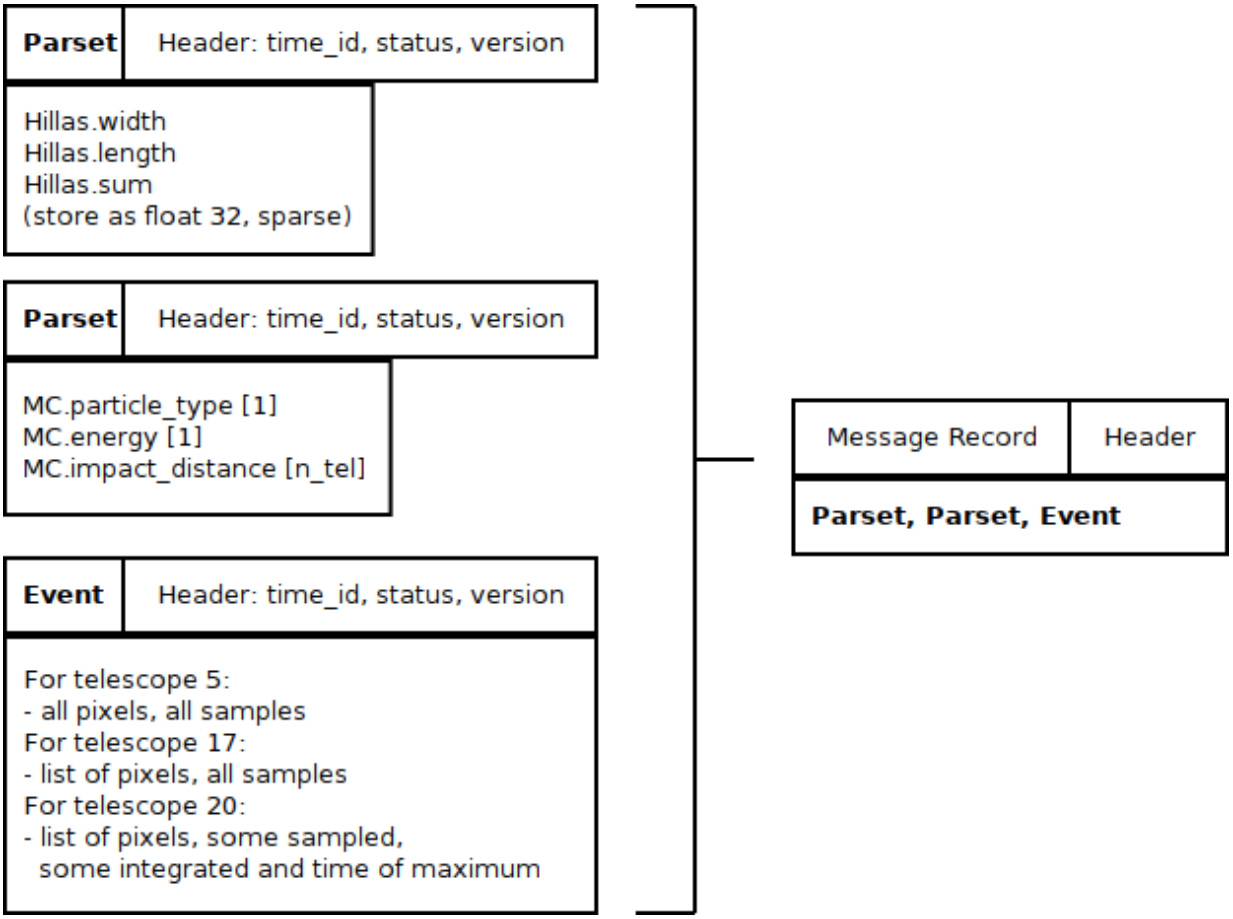


Fig. 4.1.: Example that shows how several messages are combined into a message record.

4.2. Built-in Compression Algorithms for Camera Raw Data

When the 16 bit unsigned integer ADC values are written to disk, lossless compression is applied to all values. If a camera does not provide sampled data, the single ADC values, which are store consecutively, are compressed in the same way the samples are compressed. This is done for each gain channel. The pseudo-code for storing the ADC values of full camera images is:

```
for gain = 0 to gain_n do
    if sample_n[gain] == 1
        store(televent.adc[gain][0], camera_pix_n)
    else
        ws = televent.win_start
        wl = televent.win_length
        for i = 0 to camera_pix_n do
            store(&televent.adc[gain][i][ws], wl)
```

If `televent.win_start == 0 && televent.win_length == sample_n[gain]`, the whole trace is stored. For other values, only a window of the trace is stored.
The MES file format supports different ways of storing camera raw data. Depending on the value of the `compression` variable in the telescope event header, one of these three compression algorithms is used in the `store` routine.

`none` If no compression is applied, the memory structures are simply dumped to disk.

`fast` In order to exploit the vectorization capabilities of modern CPUs, block-wise compression using AVX2 is applied to 16 consecutive ADC values. It is explained in more detail in chapter 11, section 11.1.

`small` This option offers slightly better compression. It uses wavelets and a hardcoded Huffman tree, and is described in chapter 11, section 11.2.

4.3. Built-in Quantization and Compression for Derived Pixel Values

Calibrated data and other derived pixel values are stored as 32 bit floating point numbers (`floats`) and cannot be compressed well. However, neither the whole range nor the full precision of a float is needed. For example, in HESS the pixel intensities in a calibrated image have a resolution of ≈ 0.07 photo-electrons, and they are almost always inside the range $[-100, 10000]$.
Therefore, the I/O routines of the MES file format quantize the derived pixel values and convert the results to integers (see listing 4.1).
With a defined resolution r of the pixel value and the minimum value v_{min} of a block of 16 values, the floating point value v is converted to an integer value I with:

$$I = \text{round}\left(\frac{v - v_{min}}{r}\right)$$

These integers are then compressed in blocks of 16 values, e.g. 16 samples of a trace, 16 pixel values: First, the minimum and maximum values are found. If their difference is < 16, each value in the block can be encoded using 4 bits. Otherwise, if their difference is < 256, each value in the block can be encoded using 8 bits. Otherwise, if their difference is < 65536, each value in the block can be encoded using 16 bits. Otherwise, each value is encoded using 32 bits.
Which of the four encodings was used is marked with 2 bits in the header. The minimum is also encoded in the header. Then, the minimum is subtracted from all values and they are stored using the minimal number of bits (4, 8, 16 or 32). Undefined values are supported as well.

```
static inline void encode_diff_and_min(unsigned int diff, int mini)
{
    // largest number that fits into 13-1 bits,-1 because signed
    if (abs(mini) < 4096) {  // 4096 = 2^(5+8)/2: 1 bit | 2 bits | 5+8 bits
        *(unsigned short *) &BUF[BP] = 1 | diff << 1 |
                                       ((unsigned short) (mini + 4096)) << 3; BP += 2;
    } else { // 268435456 = 2^(5+24)/2  :  2 bits  |  5+24 bits
        *(unsigned int *) &BUF[BP] = 0 | diff << 1 |
                                       ((unsigned int) (mini + 268435456)) << 3; BP += 4;
    }
}

void compress_floats(float *v, long nv, float res)
{
  int i, j, k, s1, s2;
  long p=0; // pad so that no large artificial difference is found
  for (j = nv; j < ((nv-1)/16+1)*16; v[j++] = v[nv-1]);
  while (nv > 0) {
    int mini, maxi, diff;    long l = nv > 16 ? 16 : nv;    float minf=FLT_MAX, maxf=-FLT_MAX, f;
    for (i = 0; i < l; i++) {
      f = v[p+i]; if (f == -FLT_MAX) continue;
      if (f < minf) minf = f; if (f > maxf) maxf = f;
    }
    if (minf == FLT_MAX) // if all values undefined
      diff = mini = 0;
    else {
      mini = (int) roundf(minf/res)-1; maxi = (int) roundf(maxf/res);
      diff = maxi-mini; // Store the minimum intensity-1;
    }     // -1 is needed because int-min would be 0, but 0 is used for -FLT_MAX
    if (diff < 256) {
      if (diff < 16) {  // 2 values in 1 byte
        encode_diff_and_min(0, mini);
        for (i = 0; i < l; i+=2) {
          f = v[p+i]; s1 = f == -FLT_MAX ? 0 : roundf(f/res) - mini;
          f = v[p+i+1]; s2 = f == -FLT_MAX ? 0 : roundf(f/res) - mini;
          BUF[BP++] = (s1 << 4) | s2;
        }
      } else { // 1 value per byte
        encode_diff_and_min(1, mini);
        for (i = 0; i < l; i++) {
          f = v[p+i]; BUF[BP++] = f == -FLT_MAX ? 0 : roundf(f/res) - mini;
        }
      }
    } else {
      if (diff < 65536) { // 2 bytes per value
        encode_diff_and_min(2, mini);
        for (i = 0; i < l; i++) {
          f = v[p+i];
          *(unsigned short *) &BUF[BP] = f == -FLT_MAX ? 0 : roundf(f/res) - mini; BP += 2;
        }
      } else { // 4 bytes per value
        encode_diff_and_min(3, mini);
        for (i = 0; i < l; i++) {
          f = v[p+i];
          *(unsigned int *) &BUF[BP] = f == -FLT_MAX ? 0 : roundf(f/res) - mini; BP += 4;
        }
      }
    }
    p += 16; nv -= l;
  }
}
```

Listing 4.1: Quantization and compression algorithm used in the MES file format for storing floating point numbers.

4.3.1. Compression ratio for calibrated HESS data

Table 4.1 shows the compression ratios that are achieved with the quantization and compression algorithm of the MES file format. The results are compatible with the expectations: full-camera images that contain many background pixels can be better compressed than camera images that contain mostly signal pixels.

Particle	Full Camera	ROI	Pixel List
Gamma	0.31	0.40	0.67
Proton	0.30	0.33	0.56

Table 4.1.: Compression ratio for different events and storage types.

4.3.2. Reduced resolution vs. reconstruction error

This test checks if a reduced resolution has an impact on the image parameters Hillas width and Hillas orientation. If the Hillas parameters of quantized shower images deviate significantly from the Hillas parameters of the original shower images, it is likely that there is a large negative impact on the performance of gamma/hadron separation and direction reconstruction.
The test files are a CTA MC prod 2 `simtel` file with full-camera γ-ray shower images, and one with full-camera proton shower images.
Both files are then read and stored with a varying PE resolution of the camera (from 0.004 to 1 PE), which results in several files, each containing pixel intensities stored with different accuracy. For example, a PE resolution of 0.1 means that the digits after the first comma are not significant and do not need to be stored.
Then, for each PE resolution in the test interval (0.004 - 1 PE) read the corresponding file, and compare its Hillas parameters to the original file with the true values. For each telescope event t_i and T_i in both files, the two Hillas widths w_i and W_i are calculated and their difference d_i used to calculate the RMS. The same is done for the Hillas orientation.
Figure 4.2 shows these plots for Hillas width and Hillas orientation. The RMS of Hillas orientation can be easily determined, because the position of the source in the camera is known for Monte Carlo showers. It is typically ~ 10 degrees, so the errors introduced by the reduced resolution are very small and can be neglected. Determining the RMS of Hillas width is more difficult and not done here.

4.3.3. Summary

The quantization and compression scheme of the MES file format that has been developed for derived pixel values reduces the data considerably. Since the precision of the values can be set for each telescope event individually before it is stored, the scheme works in many different scenarios and can be used to store many different kinds of events. If a correct resolution is specified, there is no significant impact on the reconstruction quality.
However, it is possible that sophisticated analyses are sensitive to small variations in the pixel values. To be sure, it is recommended to set the resolution to conservative values (e.g. in MESS, the standard resolution for intensities is 0.07 PE).

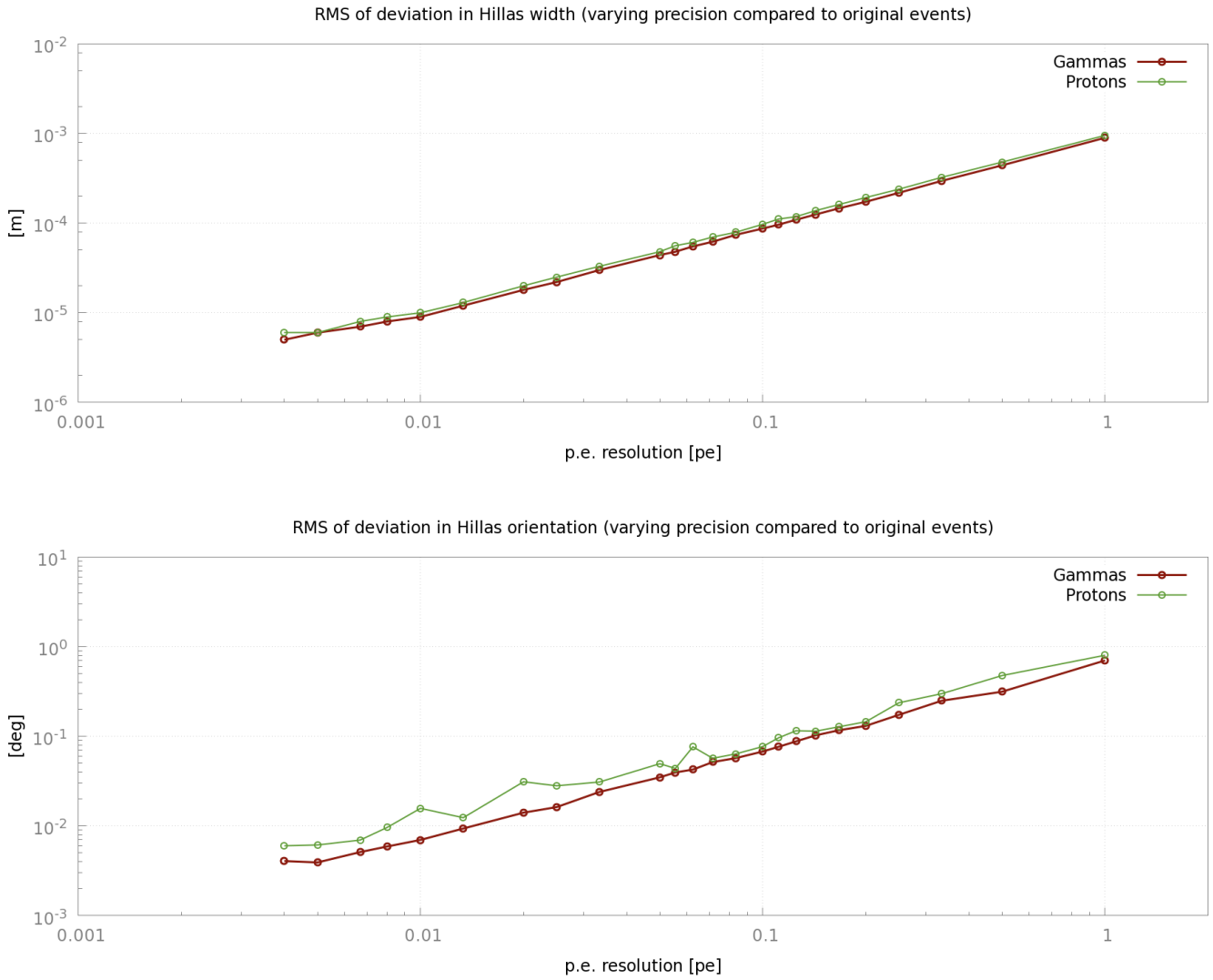


Fig. 4.2.: Reconstruction error for different quantizations of pixel intensities. Top: Deviation in Hillas width. Bottom: Deviation in Hillas orientation (note: RMS of Hillas orientation is 10 deg).

4.4. Regions of Interest in IACT Camera Images

The ROI (Region Of Interest) of a shower image is defined as the smallest circle around the center of gravity of the cleaned image that contains all pixels that survived the image cleaning (see figure 4.3).

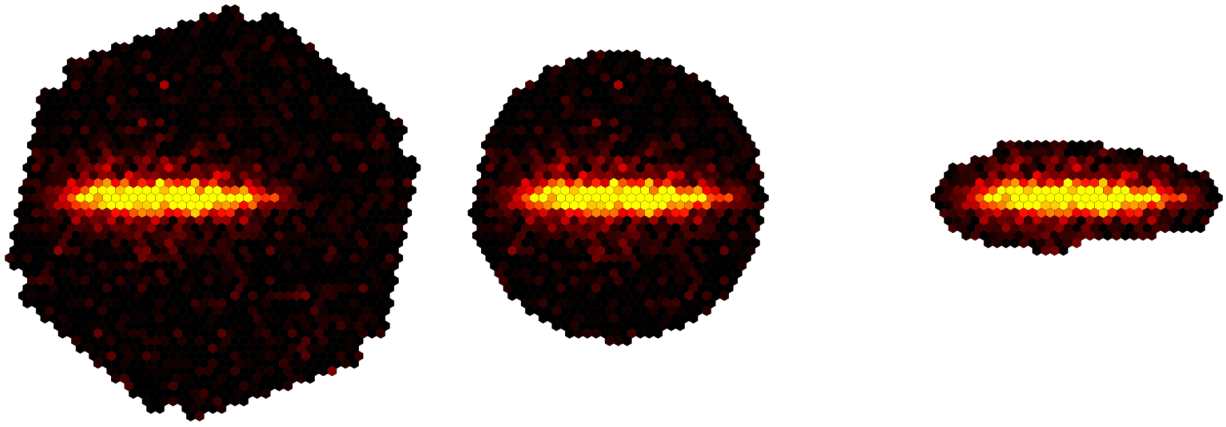


Fig. 4.3.: Full camera image (left), ROI (center) and pixel list (right).

This implies that the ROI strongly depends on the image cleaning, so the NSB level and the electronic noise of the camera must be known well. In order to lose as few significant pixels as possible, some extra rows of neighbors should be included in the cleaned image.

Regions of interest vs. pixel lists Since the ROI must include them as well, it is clear that the ROI never loses any pixels that are included in the cleaned image. On the contrary, the ROI even includes pixels below the cleaning threshold and outside of the neighborhood of the pixel list, so the probability that all shower and sub-shower pixels are captured is higher than if the pixel list was used. Figure 4.3 shows that the ROI includes some pixels that look like NSB pixels, but might be shower pixels. However, the pixel list does not include these pixels, although the list includes three rows of neighbor pixels beyond the cleaned image. These additional pixels can be used for an improved shower reconstruction [25] or for calibration. They can also be useful if a later calibration results in different cleaning thresholds. The circular shape was chosen because it is easy to parameterize and because it matches the form of the dominating low-energy showers.

Regions of interest for camera raw data ROIs are especially interesting for storing camera raw data. During data taking there might not be enough time for a sophisticated waveform analysis. However, a fast waveform analysis is good enough for finding the most significant pixels. If all pixels in a ROI around these significant pixels are stored (with their traces), it is very likely that all shower pixels are kept, even those that only turn out to be significant after a sophisticated waveform analysis.

Regions of interest in MES Besides full camera images and pixel lists, the MES file format also supports storing the ROI of a shower. The regions of interest can be extracted with MESS:

```
mess -micropipe read.r: -in gamma_fullcam.mes , dup.d:*r , cleanmn.c:d -m 3 -n 6 -nb_rows 2 ,
    hillas.h:c , convert.conv:r,c,h -type roi -pixvals , write.w:conv -out gamma_roi.mes
```

The `convert` module needs the uncleaned image, the cleaned image, and its Hillas parameters as input, It then returns an ROI with the center of gravity of the cleaned shower (included in the Hillas parameter set) and a radius just large enough to contain all pixels of the cleaned image.

4.4.1. Benchmark

In order to quantify the benefit of regions of interest, the full-camera images of a HESS Monte Carlo file containing γ-ray showers, and one containing protons are stored as regions of interest and as pixel lists. Thus, there are six files:

gamma_fullcam.mes gamma_roi.mes gamma_list.mes
proton_fullcam.mes proton_roi.mes proton_list.mes

The files with the pixel lists are obtained by cleaning the full-camera images with 3-6 cleaning and extending the resulting set of pixels by two rows. The files with the ROIs are obtained with the pipeline shown above.
The results are shown in table 4.2.

	File Size [%]		Pixels kept [%]		Bytes/Pixel		Signal pixels kept[%]	
Storage type	Gamma	Proton	Gamma	Proton	Gamma	Proton	Gamma	Proton
full camera	100	100	100	100	1.2	1.2	100	100
ROI	26	57	20	53	1.6	1.3	94	97
pixel list	18	27	8	14	2.7	2.3	92	91

Table 4.2.: Comparison of full-camera images, ROIs and pixel lists after selection and compression.

Obviously, pixel lists produce smaller files than ROIs, but they also contain much less pixels. In an IACT experiment, as much data as possible must be kept, consuming as little space as possible. Thus, the number of bytes per pixel must be taken into account. In this regard, full camera images are most efficient, however, they contain many noise pixels with low intensities, which can be better compressed by the built-in MES compression.
Therefore, the number of pixels that are probably signal pixels is counted in the ROI and pixel list files and compared to the number of probable signal pixels in the full-camera file. Here, a signal pixel is defined as a pixel that survives 3-4 cleaning.
The full camera image contains all pixels and thus all signal pixels. The ROI contains almost all signal pixels (94% for γ-ray showers and 97% for proton showers). The pixel list a little less signal pixels (92% for γ-ray showers and 91% for proton showers).

4.4.2. Summary

Regions of interest are a compromise between pixel lists and full-camera images. While they contain significantly more pixels than a pixel list, the resulting files are much smaller than an extrapolation would suggest. Although in average, the ROI contains almost all pixels in the camera that are probably signal pixels, only a relatively small subset (20-50%) of the full camera is stored.
Even if future tests should reveal that the additional pixels in a ROI do not lead to an improved analysis, they might still be useful, for example a calibration crosscheck.
The algorithm guarantees that all pixels that are included in the cleaned image are preserved in the ROI. Thus, for storing ROI or pixel lists, accurate image cleaning is mandatory, like for all other methods that store only a selection of pixels.

4.5. Reading and Synchronization

MESS has inherent support for synchronization of several input streams, which is useful if data of subsystems is stored in different files and at different times than the camera events. For example, a camera might write 1000 events per second, and a temperature sensor only 1 value per second. When later during the calibration or analysis these data are needed, a new temperature must only be read when the time stamp of the current event is larger than the time stamp of the current temperature. The easiest way to synchronize them is to use the `sync` module in a mess pipeline, where it must be invoked as parent of several readers, so it can synchronizes them based on their `time_id` (see 3).
Consider the following example:

```
mess -micropipe
    sync.s ,
    read.event:s -in /data/event.mes ,
    read.hillas:s -in /data/hillas.mes ,
    read.mc:s -in /data/mc.mes ,
    read.pedestal.s -in /data/pedestal.mes -slave ,
    write.w:r1 -out events_hillas_mc.mes ,
```

The four readers read events, Hillas parameters, MC parameters of the events and pedestal events. All readers are master readers, except the pedestal event reader (`-slave`). For a message to be valid, all masters must have the same `time_id`. The `time_id` of the slaves is the expiration time of the message and must therefore be $\geq$ the masters' `time_id`.
In each step of the pipeline, messages are read from all three master readers until all of their `time_id`s are equal. Looking at the type of messages they provide, it makes sense: Hillas parameters and MC parameters must belong to an originating event. If one of them is missing, the rest of the pipeline is undefined, so all three messages must be valid and have the same `time_id`. Pedestal events, however, arrive with much lower frequency, and the current pedestal event is used until it is expired.

event	hillas	mc	pedestal	synchronized	comment
1	1	1	2	yes	
2	2	2	2	yes	
3	3	3	2	no	slave reader must catch up
3	3	3	10	yes	(2 was expired)
4	4	4	10	yes	
...	...	...	...	...	

In this case, everything went smooth and the three master readers were already in sync. Now if some master readers miss some messages the other master readers have, they must read until they reach the same `time_id`:

event	hillas	mc	pedestal	synchronized	comment
...	...	...	...	...	
45	45	45	85	yes	
55	62	55	85	no	Hillas reader is ahead

Here, the Hillas parameter reader misses one or more messages, and with this combination of `time_id`s, the synchronization is not finished yet! So the Hillas parameter reader is paused, until the other readers have caught up:

event	hillas	mc	pedestal	synchronized	comment
55	62	55	85	no	
57	62	57	85	no	
57	62	57	85	no	
58	62	58	85	no	
61	62	61	85	no	
62	62	63	85	no	the MC reader missed a message
63	63	63	85	yes	

What is shown with all details here, is more efficient in reality. Assuming the readers have the previous state:

event	hillas	mc	pedestal	synchronized
55	62	55	85	no

In this case, the `sync` module calls the event reader with the information to move on in the file until an event with a `time_id >= 62` appears. As can be seen in the tables above, there is an event with `time_id == 62`, so the state is:

event	hillas	mc	pedestal	synchronized
62	62	55	85	no

Then, then `sync` module calls the MC reader the same way it called the event reader. State:

event	hillas	mc	pedestal	synchronized
62	62	63	85	no

Now, the event reader and Hillas parameter reader need to be called, which completes the synchronization:

event	hillas	mc	pedestal	synchronized
63	63	63	85	yes

Expiration time for slave messages This scheme with the expiration time was chosen to prevent a reading module in slave mode from having to peek into the input stream at every step in the pipeline, even if no new data has arrived yet. For offline analyses, this is no problem, because when the data was written to disk, the expiration time of each slave message can be set to the time of arrival of its successor. However, if the data analysis happens during data acquisition, the expiration time of a slave message (e.g. pedestal event) is not known in advance. This problem can be solved, if the module that feeds the slave message into the pipeline always sets the `time_id` of the message to the current time, i.e. the `time_id` of the masters. This ensures that the `sync` module will call the module at each step in the pipeline. If new slave data is available, the module reads it into memory and sets the `changed` attribute of the message to 1. The new message will be used, and if the data stream is to be written to file, this message will be included, because `changed == 1`. If no new slave data is available, `changed` is set to 0, and the slave message will be used, as if it had not expired yet. If the data stream is written to file, this message is omitted, because `changed == 0`.

Advantage of built-in compression The MES file format has built-in compression for data blocks. The header of the message record and its messages remain uncompressed. This allows fast synchronization of different input streams, since the `sync` module only compares the `time_id`s, which are in the headers. If during synchronization a message must be skipped, the file pointer simply seeks behind the data block (for pipes, a `read` is performed) without having to decode the data. External compressors, like `fc16` or `gzip` [42], ignore the internal file structure and compress everything. In order to access the `time_id` in the header, the complete message must be decompressed. Therefore, synchronization with built-in compression is in general much faster than with an external compressor.

5. Data Quality

5.1. Run Selection

In HESS, data is taken in separate runs of 28 minutes duration, so if a problem with data integrity occurs, at most one run is lost. Also, the position of the source in the sky does not change much during the run, which allows to use only one set of calibration parameters and zenith angle lookups per run. Shorter runs would not provide enough statistics for calculating calibration coefficients.
During a run, several subsystems monitor the experiment and fill a summary of the recorded data into a database. Later, when the data of a certain position in the sky is to be analyzed, all runs that had the position in the field of view are selected, and the corresponding monitoring data is loaded from the database and checked for problems.
A run is rejected, for example, if the atmosphere was too opaque, if it was too short (< 600 s) or if it is a HESS-I run and less than two telescopes participate in the run. The remaining runs can then be calibrated and analyzed.
It is also possible that a telescope is excluded from a run, for example if the telescope pointing was inaccurate, if the drive system was behaving incorrectly or if the camera has too many broken pixels. If after exclusion less than two telescopes remain, the whole run is rejected.
MESS uses all runs that remain after the standard HESS data quality cuts are applied, except for the broken pixel cut (because MESS uses a different strategy, see below) and the cut on run duration (because MESS uses a different calibration that works with low statistics).

5.2. Data Quality Explorer

The data quality cuts only take the most obvious and severe problems with the data into account. However, the quality of the analysis might still be affected by minor problems. Since it can be important to quickly find anomalies in the subsystem data or to find correlations of the quality of the data and environmental conditions easily, the program *Data Quality Explorer* has been developed as a standalone program outside of MESS. It queries the HESS database, which contains all monitoring and subsystem data, and then processes and plots selected values in different ways. Each table header is assigned a variable name, so that a formula can be created, parsed and plotted.
Figure 5.1 shows a simple example where $F = x_{16}/x_3$ (= Events/Duration = mean trigger rate) and $G = x_{92}$ (= sky temperature, measured by the radiometer) are plotted as time series (top), histogrammed (center) and against each other (bottom). It can be seen that trigger rate and temperature are weakly anti-correlated, which is expected because the radiometer temperature is correlated with the cloudiness of the sky.
The *transparency coefficient* t has been developed to quantify the opacity of the atmosphere [21]. Based on a cut on t, the run is classified as *spectral quality* run or as *detection quality* run. *Spectral quality* means that the atmosphere was so clear that the run may be used for calculating the energy spectrum of a source, where *detection quality* means that the run must only be used for detecting sources. Figure 5.2 shows the spectral and detection quality runs in during the April shift in 2012. The available dark time can never be completely used, because two minutes are needed for run transitions. Also, if during some nights the weather was too bad for operating the telescopes, significantly more dark time is lost.
After 2012.04.15, for example, from the available 360 minutes of dark time, only 140 minutes could be used for observations; the rest of the time, the sky was cloudy. And during the observation, the atmosphere was not clear enough for the runs to be of spectral quality. Probably, because there were still some clouds.

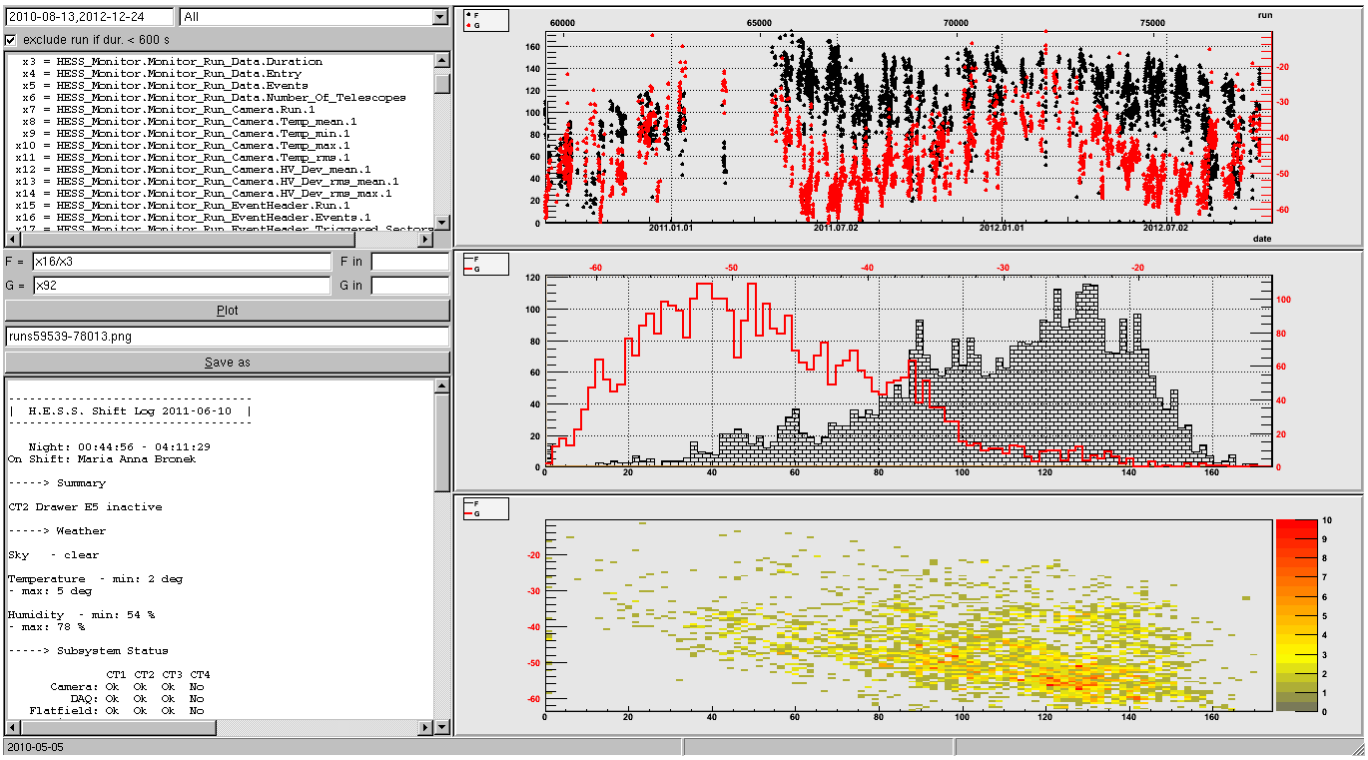


Fig. 5.1.: Data Quality Explorer: Database entries can be combined in formulas and plotted.

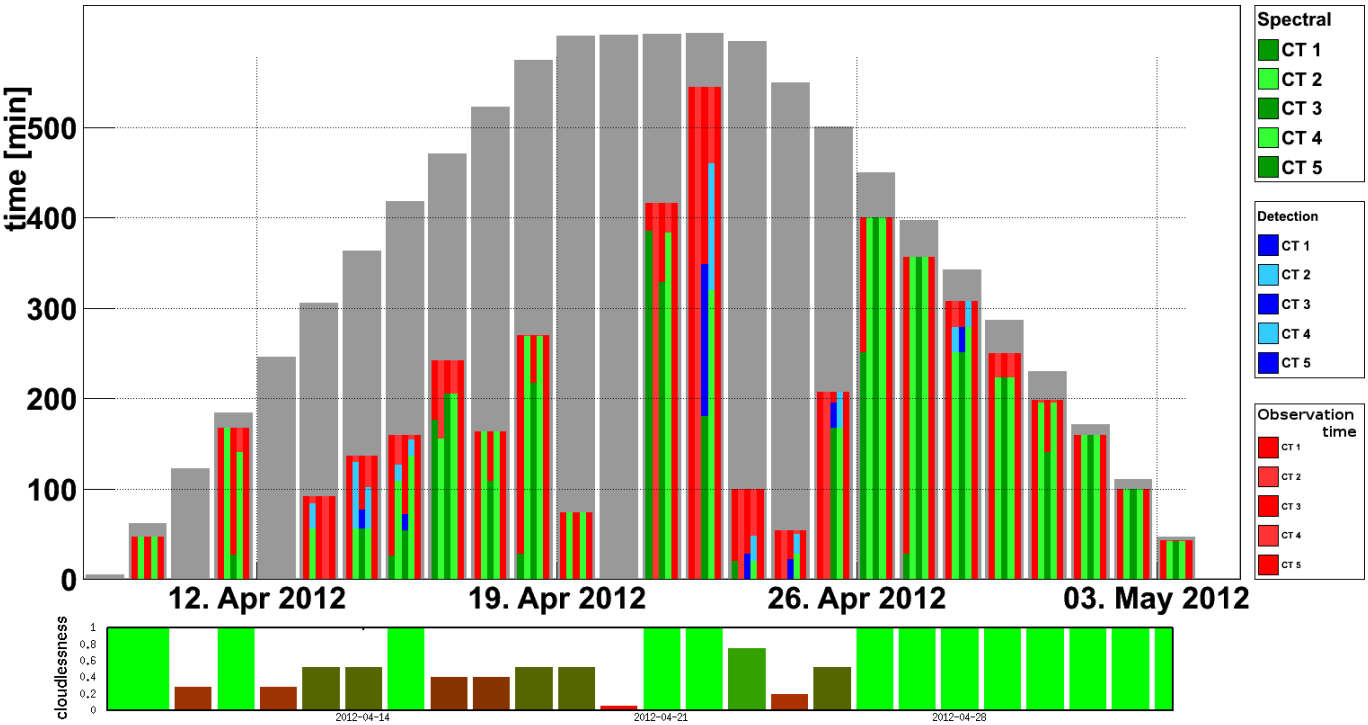


Fig. 5.2.: Top: Usage of observation time for spectral and detection quality runs. Each of the five thin bars within a thick bar stands for a telescope. The dark time is shown in gray and the observation time in red. The green and blue bars show how much of the recorded data has spectral and detection quality. If a red bar is not overlapped by green or blue bars, it means that the observation time could not be used because of hardware issues with the telescopes.
Bottom: Weather quality (cloudlessness) for the same period, estimated by the shifters by looking at the sky.

5.3. Broken Pixels

Besides the weather, the data quality also depends on the hardware status of the telescopes. Thus, it is important to notice and fix any issues with the telescopes early. Upon deeper inspection, figure 5.2 reveals that CT1 is missing in nearly all runs. A frequent reason for excluding a telescope from a run are too many broken pixels, which can occur, if a star passes the field of view, if cables are loose or if there are problems with the electronics. Since a broken pixel is ignored in the analysis, the camera image is incomplete and the reconstructed direction and energy will less accurate. A pixel is considered broken for a complete run, if the following condition is met:

- the high gain ARS and the low gain ARS were not working or
- there was no signal at all or
- something bad with high gain and low gain or
- the high voltage was off or
- there was a miscellaneous problem with the high gain and the low gain

Figure 5.3 shows the number of broken pixels in different telescopes. Over several months, CT1 has significantly more broken pixels than the other telescopes.

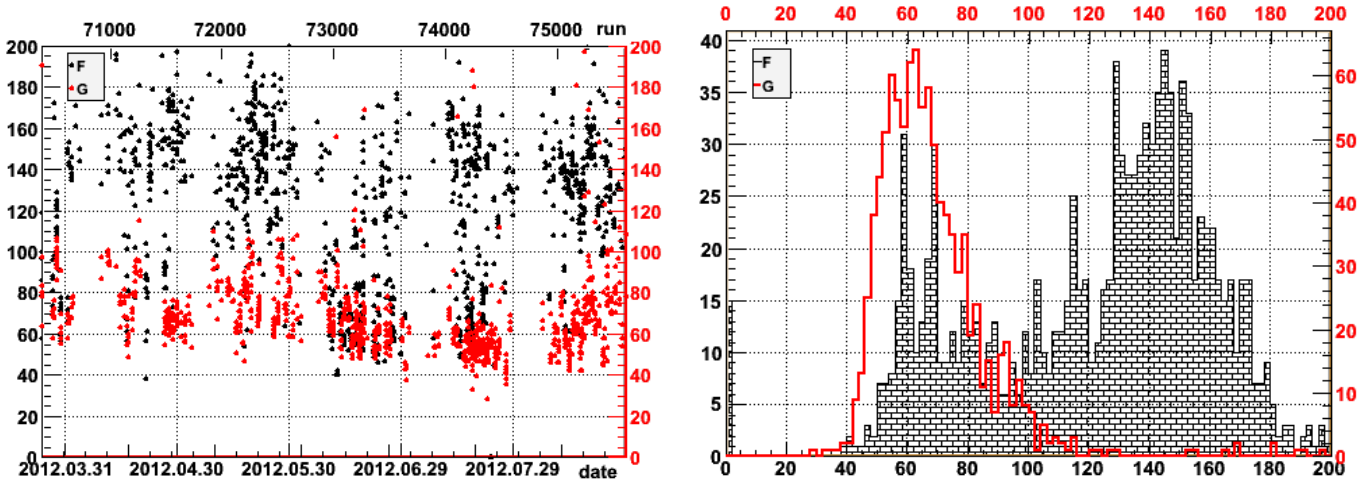


Fig. 5.3.: Broken pixels in CT1 (F, black) vs. mean number of broken pixels in CT2-4 (G, red). Left: time series. Right: Histograms.

Impact on shower reconstruction In order to investigate the problem of broken pixels further, their impact on shower reconstruction was investigated. Figure 5.4 shows typical broken pixel patterns for HESS. The camera on the left picture has broken drawers (4x4 broken pixels) to the left of the shower and in the camera center, one broken analog ring sampler (four horizontally adjacent pixels) at the lower left of the camera, as well as few individual broken pixels. The right camera has many distributed pixels, some broken analog ring samplers and a broken drawer at the bottom of the camera.
The more shower photons land on broken pixels, the more the reconstructed parameters, like direction and energy, are affected. That is why HESS introduced a broken pixel cut: telescopes are not included in runs, if they have more than 120 broken pixels (out of 960 pixels).
However, if there are only two working telescopes available to begin with, and one is rejected because of the broken pixel cut, the complete run is lost, because there is no reliable way in HESS to analyze single telescope data (if it is a HESS1 telescope). Also, even if a telescope has many broken pixels, it is often still better to keep it, because many showers do not fall on the broken pixels (or only an insignificant part of the shower), and an additional telescope helps enormously in reconstructing the direction. So since the impact of broken pixels on the shower reconstruction not only depends on their number, but also on their distribution in the camera, a different approach than the cut on 120 broken pixels was investigated.

Another idea is to parameterize the broken pixel patterns such that clusters of broken pixels are penalized, as well as broken pixels close to the camera center:

$$z = \sum_{i \in BP} {\rho_i}^2 e^{\frac{-d_i^2}{\sigma^2}} \tag{5.1}$$

Here, d is the distance from the camera center in meters, σ is a constant (currently 0.3) and ρ is the local density of broken pixels, in this case simply the number of neighbors plus one.
This way, even telescopes with more than 120 broken pixels could remain in a run, if they were well distributed and mostly at the camera borders. Conversely, a telescope with less than 120 broken pixels could be excluded from a run, if they were clustered and mostly close to the camera center. Figure 5.5 shows the parameterizations of the broken pixel patterns in figure 5.4, compared to the parameterizations of all broken pixel patterns ever recorded.

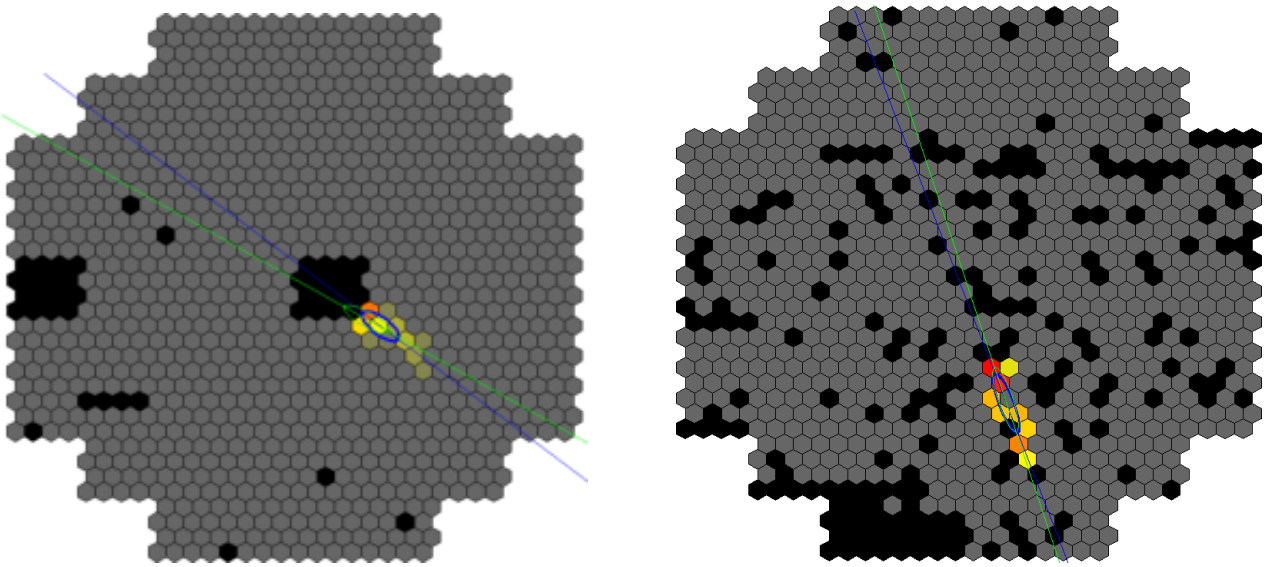


Fig. 5.4.: MC shower images seen with cameras with different broken pixel patterns; broken pixels drawn black. Hillas ellipse and orientation are drawn for the shower with broken pixels (blue) and for the same shower if it had landed on an intact camera (green). Left: Few broken pixels, but clustered and close to the camera center. Right: Many broken pixels, but distributed, except one cluster, but this is at the bottom, close to the camera border.

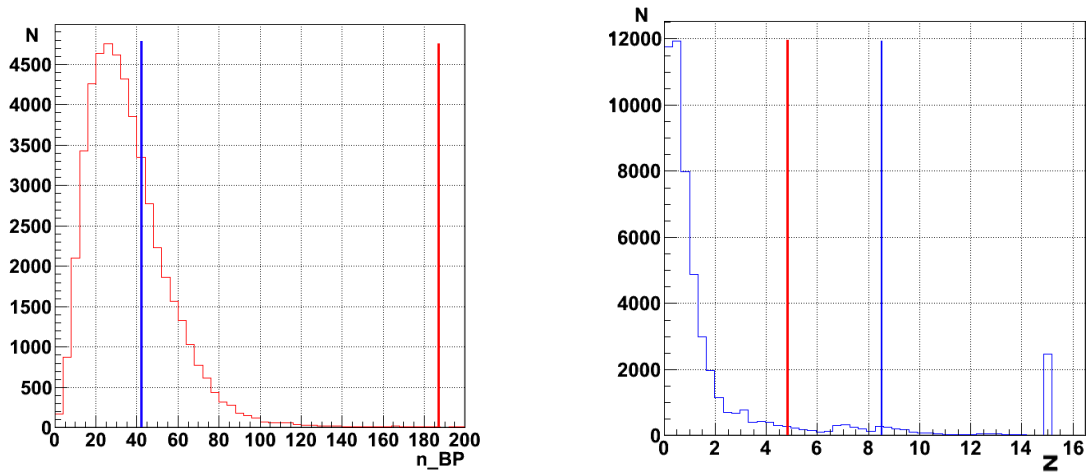


Fig. 5.5.: Left: Distribution of number of broken pixels for all HESS runs. Right: Distribution of z for all HESS runs. The parameterization of the broken pixel pattern in figure 5.4 is marked in each plot:
Left pattern: blue, n_BP=42 and z=8.5. Right pattern: red, n_BP=186 and z=4.8.

Broken pixel pattern parameterizations vs. reconstruction error Correlating the parameterizations of the broken pixel patterns, z and n_{BP} (number of broken pixels), with mean reconstruction errors yields figures 5.6 and 5.7. Here, for each broken pixel pattern that is applied to a virtual camera, 10.000 Monte Carlo γ-ray events are reconstructed (Hillas orientation and Hillas amplitude) and their mean reconstruction error calculated. It can be seen that z correlates better with the error than n_{BP}, for the direction as well as the amplitude. If a cut was be made at some z value, more good broken pixel patterns would be kept and more bad broken pixels would be discarded than with a cut on some n_{BP} value.

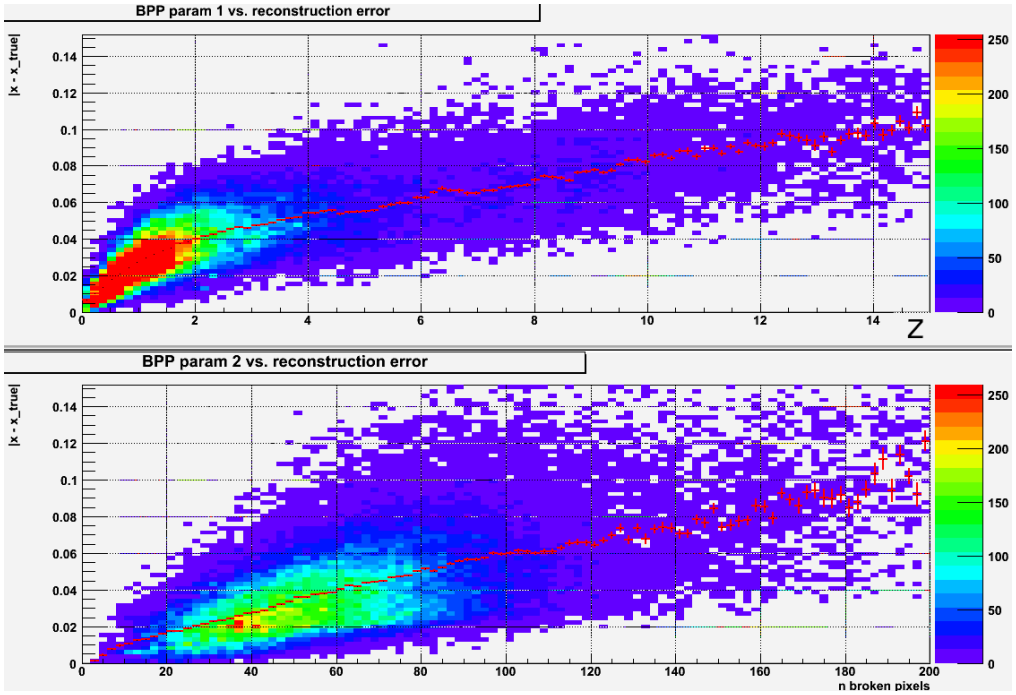


Fig. 5.6.: Mean error in Hillas orientation vs. parameterization of broken pixel pattern.
Top: z. Bottom: number of broken pixels.

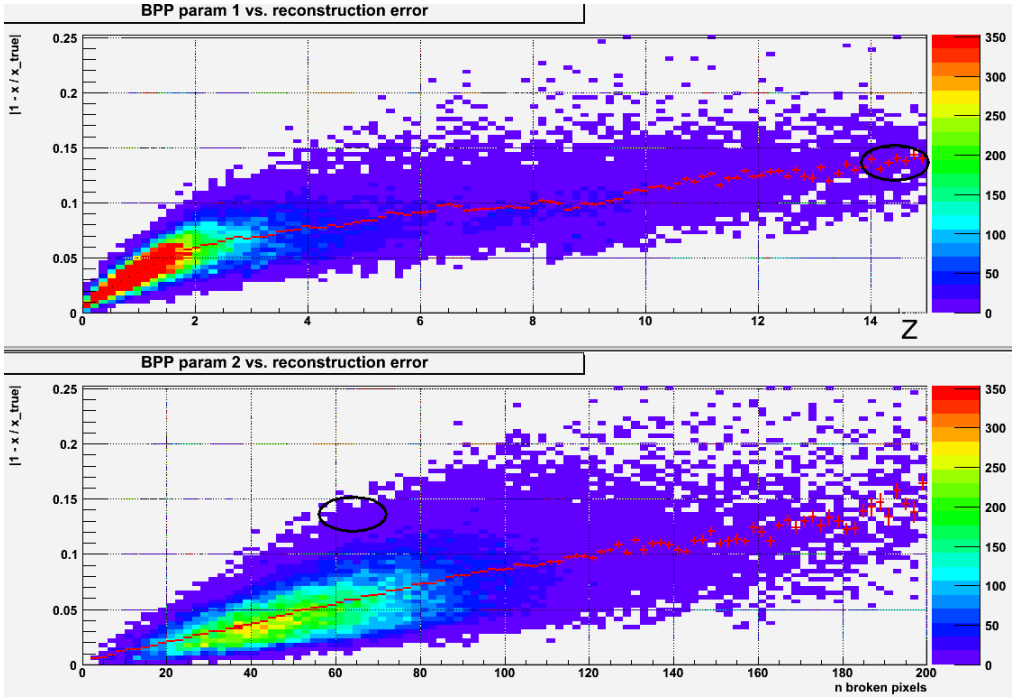


Fig. 5.7.: Mean relative error in Hillas amplitude vs. parameterization of broken pixel pattern.
Top: z. Bottom: number of broken pixels.

Quantifying the improvement To quantify the improvement, the broken pixel patterns are divided into two classes: *signal* (good) if mean reconstruction error $< e$ and *background* (bad) if mean reconstruction error $\geq e$, where the value of e depends on the kind of reconstruction error being calculated: for the error in Hillas orientation it is $e_\alpha = 5°$ (see equation 5.2) and for the error in Hillas amplitude $e_A = 0.08$ (see equation 5.3). Now, for different parameterizations and cut values (x-axis), the number of good broken pixel patterns that pass the cut is divided by the number of all good broken pixel patterns. For the background, the same is done, i.e. the number of bad broken pixel patterns that pass the cut is divided by the number of all bad broken pixel patterns.
Figure 5.8 shows that the signal and background ratios are improved for both direction and amplitude, if z is used instead of n_{BP}. In the corresponding ROC curves in figure 5.9, the signal ratio is plotted against the background ratio. It can be seen that the new parameter keeps significantly more good broken pixel patterns and discards significantly more bad broken pixel patterns. *Example:* A broken pixel pattern is parameterized with $z = 8$, and the mean reconstruction error e_A is larger than 0.08, which makes it a bad broken pixel pattern.

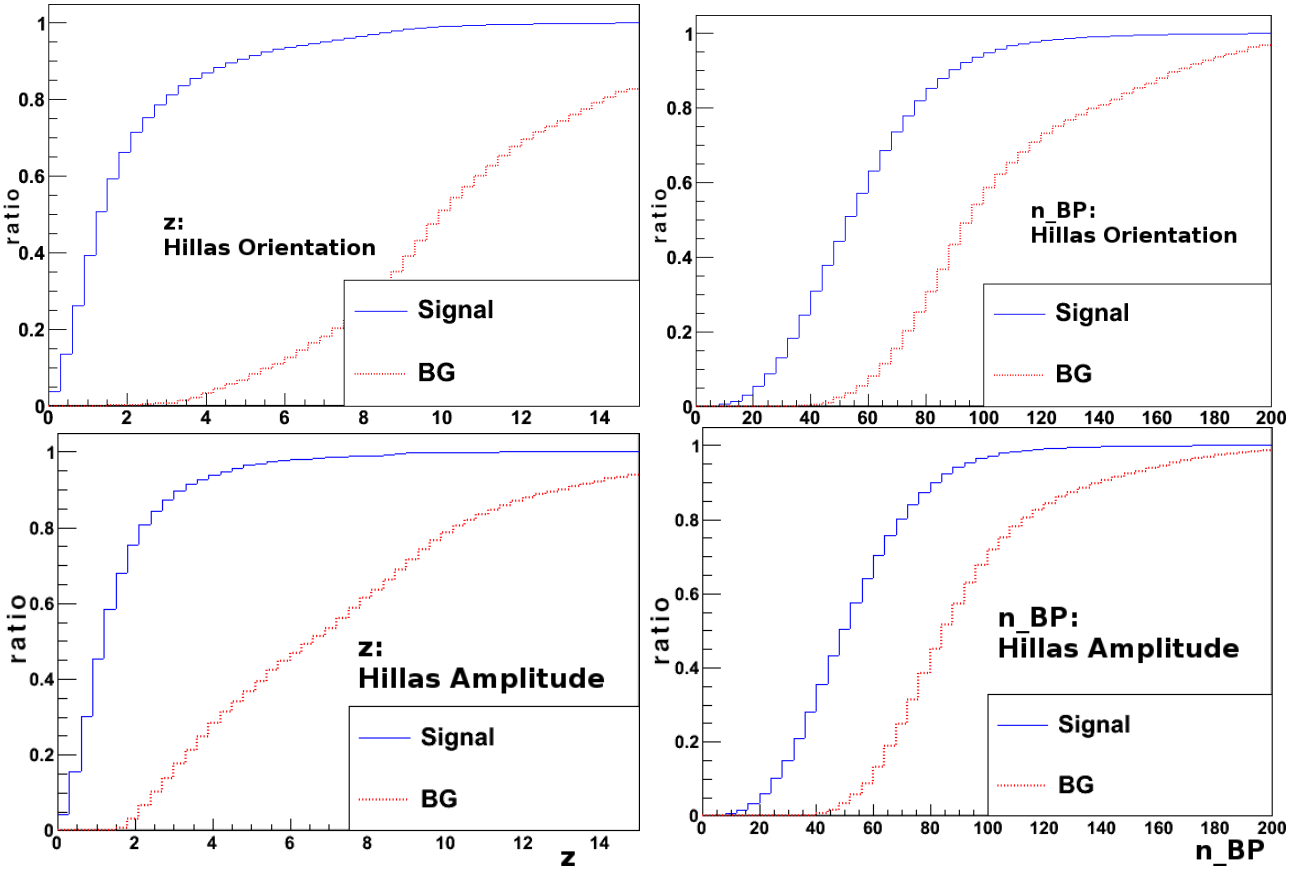


Fig. 5.8.: Background and signal ratios. Left: z. Right: number of broken pixels. Top: Hillas Orientation. Bottom: Hillas Amplitude.

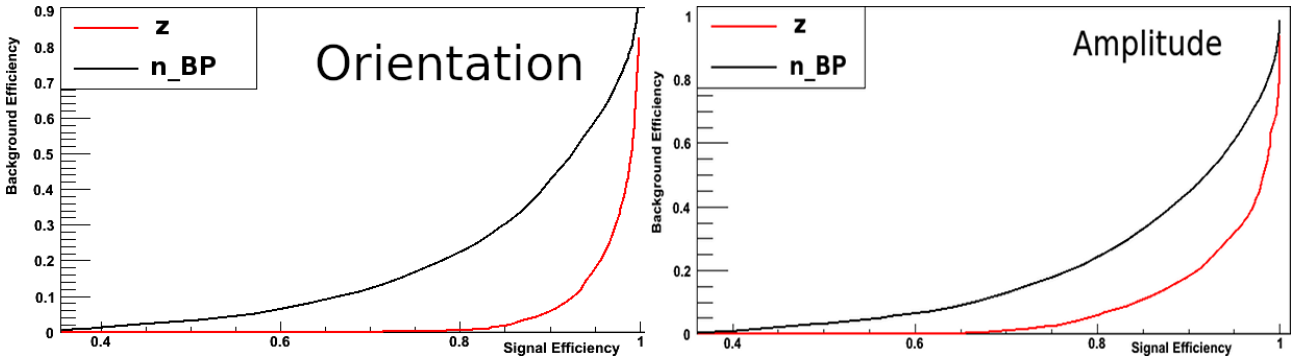


Fig. 5.9.: Signal vs. background efficiency for z and n_{BP}. Left: Hillas Orientation. Right: Hillas Amplitude.

Fig 5.10 shows how many of the runs that were excluded using the old cut can be kept using the new cut at $z = 8$, and how many runs are now excluded that were previously kept. The changes are sometimes significant and would - according to this study - lead to better data quality. The value $z = 8$ was arbitrarily chosen and should be optimized for different analysis use cases (detection, spectrum or morphology).

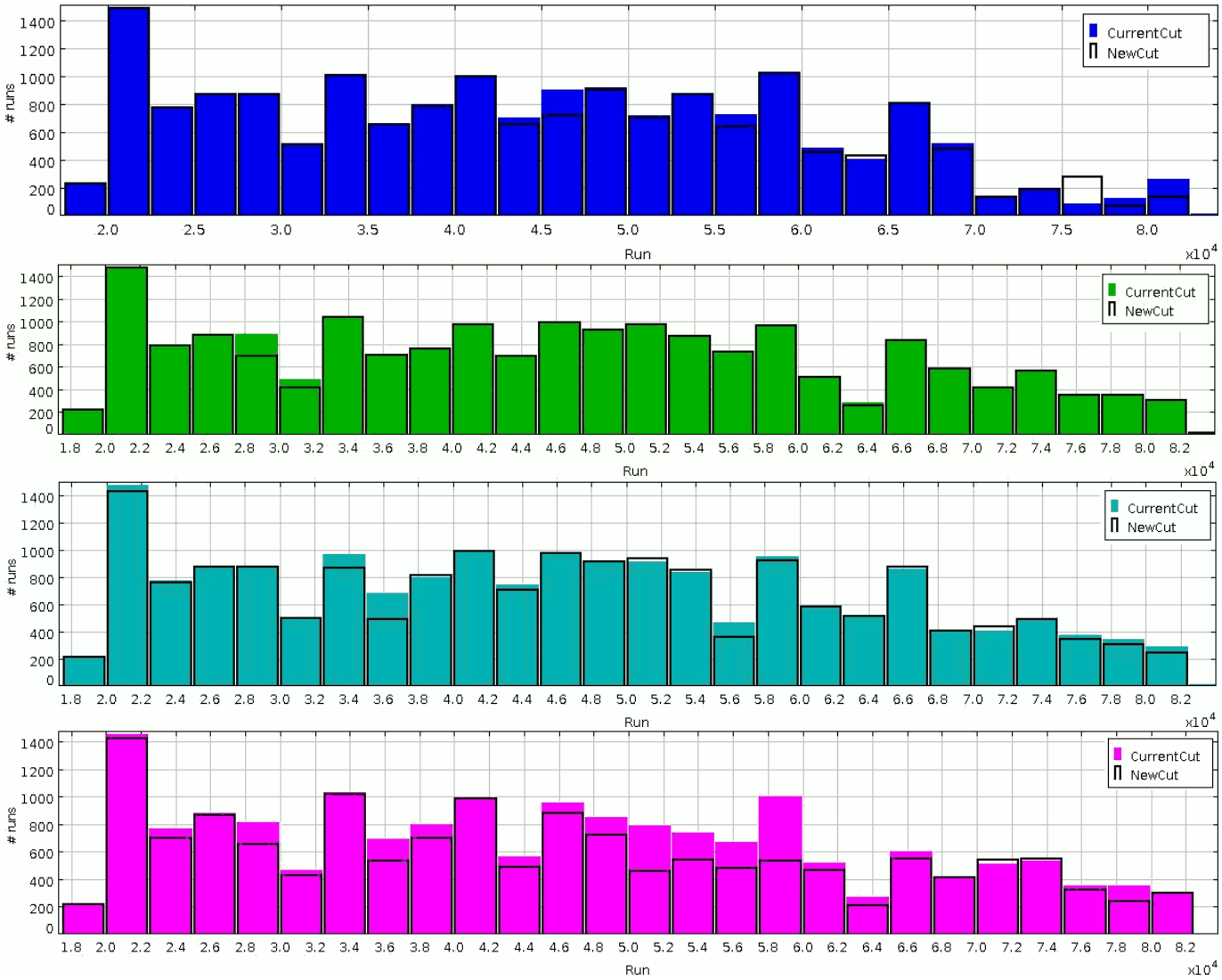


Fig. 5.10.: Telescopes kept/excluded in runs using the new/old broken pixel cut.
From top to bottom: CT1-CT4.

5.3.1. Broken pixel cut on mean reconstruction error

While a cut on the introduced broken pixel parameter is an improvement over the cut on a fixed number of broken pixels used in HESS so far, it is even better not to parameterize the broken pixel patterns at all, but instead to calculate the mean reconstruction error of a broken pixel pattern. For this, a representative set of Monte Carlo gamma shower images ($n \approx 10^5$) needs to be reconstructed; first with, then without the broken pixel pattern applied. The absolute difference in direction:

$$e_\alpha = |\alpha_{\text{noBP}} - \alpha_{\text{BP}}| \tag{5.2}$$

or the relative difference in amplitude:

$$e_A = |1 - \frac{A_{\text{noBP}}}{A_{\text{BP}}}| \tag{5.3}$$

are then used as cut parameter.

This is feasible, because for each run, there is only one broken pixel pattern per telescope, and calculating the mean reconstruction error only takes a few seconds.
The cut on reconstruction error could depend on the type of analysis to be performed, for example a cut for spectral quality and another cut for detection quality, like the cut on the atmospheric transparency coefficient. The complete solution would include the gamma/hadron separation performance and do a stereo analysis.

5.3.2. Weighting broken pixel patterns during reconstruction

The described cuts exclude a telescope from a run, if the broken pixel pattern fulfills certain criteria. However, excluding a telescope from a run means losing a lot of events.

Impact of losing telescopes A test with MC γ-ray showers was done to estimate the impact of losing telescopes. At first, four telescopes were available, and only events that triggered at least two telescopes were recorded. The number of recorded events was set as reference (**100%**). If one telescope was removed, it dropped to **75%**, if two neighboring telescopes were removed, it dropped to **53%**, and if two opposite telescopes were removed, it dropped to **41%**.
Note: Removing two neighboring telescopes leaves two neighboring telescopes and removing two opposite telescopes leaves two opposite telescopes. Since neighboring telescopes are closer to each other than opposite telescopes, it is more like that they capture a shower.

Impact of the broken pixel cut on the number of available telescopes in all HESS runs Because of technical problems, there are often less than four telescopes to begin with, even before the broken pixel cut. From the ≈15400 spectral observation runs that pass all data quality criteria (like atmospheric transparency, tracking errors, participation fraction etc.), 81% are four-telescope runs and the rest are three- and two-telescope runs. This amounts to ≈ 58000 participating telescopes. The broken pixel cut used in HESS ($n < 120$) then removes additional ≈3500 telescopes from the 58000 telescopes. In 18 runs, less than two telescopes remain and the run is lost. This is not much, but keeping the removed telescopes would keep many more events and could improve the reconstruction for showers that are not impaired by the broken pixels.

Shower Impairment and Weighting Function For direction reconstruction, [9] proposes to weight the intersection of two shower axes with the sine of the angle between the axes and errors associated to the showers. In their work, an intact camera is assumed and the associated errors are related to the shower geometry and amplitude.
Here, the idea is to define a function that estimates the impairment of the current shower S, inflicted by the broken pixel pattern B, by calculating the ratio of measured light and a probable upper limit of the amount of light that would have been measured if all broken pixels had been intact. For each broken pixel in B that has neighbors in S, the maximum neighbor is selected and added to a global sum G. For each broken pixel that lies inside of the convex hull of S and has no neighbors in S (either because it lies outside of the shower or because all neighbors are broken pixels), the maximum intensity in S is added to G. The ratio of the sum of all intensities in S and G lies between 0 and 1 and is an estimate of how much the measured shower deviates from the real shower in a worst case scenario, where much light would have landed on the broken pixels. $1 - \frac{S}{G}$ can be interpreted as the shower quality in terms of broken pixel impairment.
The reconstructed position for an N-telescope event is then:

$$p = sum_{i=1}^{N} sum_{j=i+1}^{N} sin(\alpha_i - \alpha_j)(1 - \frac{S_i}{G_i})(1 - \frac{S_j}{G_j}) \tag{5.4}$$

The global sum G is in most cases larger than the real amount of light that has landed on the camera, resulting in a smaller weight for all telescope pairs it is involved in, but since a small weight for a shower is similar to excluding the telescope from the reconstruction (of this event), the results should in most cases not be worse than those of the classical cuts that exclude telescopes from the analysis of the whole run.

5.4. Summary

A visual data exploration program for the HESS subsystem and monitoring database has been developed, which can be used to find anomalies or correlations. Since arbitrary parameters can be combined in formulas and then plotted, this tool is very flexible and can be used instead of other diagnostic and exploration tools.
Different parameterizations of broken pixel patterns have been developed and tested in order to only remove telescopes from a run if their broken pixel patterns would lead to bad shower reconstructions. Finally, it was proposed not to apply any broken pixel cut at all, but instead calculate the effect of a broken pixel pattern on each shower individually and use it as weight during reconstruction.

6. Calibration

Calibration in ground-based γ-ray astronomy is mainly needed to understand how the instrument responds to Cherenkov light. This means that the mechanics, optics and electronics of the instrument must be understood in detail.
The calibration described here focuses on the camera raw data (ADC values) and provides calibrated events. Malfunctioning pixels must be marked, the night sky background and electronic noise must be determined, and the number of ADC values that correspond to a single photoelectron. The calibrated event will then consist of pixels whose values are photoelectron counts.
This calibration needs some pedestal runs to extract the ADC-to-p.e. ratio from and a data run with the ADC values of the high gain and (if available) the low gain channel. A chunk of the data (a fraction of a data run or the whole run) is read into memory, so that N_{run} events can be randomly accessed and the following calibration parameters can be determined:

- unusually behaving pixels
- high gain/low gain ratio
- running pedestal mean starting value
- running pedestal standard deviation starting value

Allowing to split the run into several chunks can be helpful if the run is so long that calibration parameters might change over time.
If the experiment is designed such that some calibration parameters are directly provided instead of having to inspect the data, these parameters can be used. For example, if the camera sends a broken pixel mask every second, it can be used instead of the broken pixel mask calculated by the calibration described here. For the software, it is no problem if calibration parameters arrive at random times or different frequencies.
In this chapter, HESS-I raw data is used. Each event consists of two, three or four telescope events, whereof each consists of two 960-pixel images, one with high gain and one with low gain ADC values. Since most of this calibration is telescope-wise, only data of one telescope is shown in the examples.

6.1. Statistical Methods

Cherenkov cameras have many identical pixels and it is useful to compare them to each other in order to detect odd behavior. A pixel that produces a signal when it should not or that does not produce any signal when it should can lead to wrong results. While *mean* and *standard deviation* are usually useful for finding outliers, they become less meaningful if the outliers participate in their calculation. Robust statistical methods would be helpful for obtaining more reliable results.

6.1.1. Robust statistics

Here, the robust mean of N numbers is defined as the mean of the inner numbers $r1{*}n \ldots r2{*}n$ after sorting, where n is the number of values that are not undefined and $r1$ and $r2$ are the fractions on the left and the right end of the distribution that are not taken into account when calculating the mean. The robust standard deviation is calculated accordingly.
Example: When calculating the robust mean and standard deviation of

$$v = (u, u, 2, 5, 3, u, 0, u, -200, 200) \tag{6.1}$$

(u means undefined) with $r1 = \frac{1}{3}$ and $r2 = \frac{2}{3}$, v is first sorted

$$v = (u, u, u, u, -200, 0, 2, 3, 5, 200) \tag{6.2}$$

then the undefined values are removed

$$v = (-200, 0, 2, 3, 5, 200) \tag{6.3}$$

and finally the mean of the inner $\frac{1}{3}$ numbers (2 and 3) is calculated.
Due to the sorting, the method is slower than the standard way of calculating mean and standard deviation. In order to find useful values for $r1$ and $r2$, a rough model of the distribution and an estimate of the number of outliers should be known.

6.2. ADC-to-PE Ratio

The response of the photomultiplier tube to a single photon is obtained from pedestal runs, where the lids of the camera is closed and only the electronic noise of the system and the responses of occasional photons coming from an LED are measured. Since the multiplication of photoelectrons within two dynodes is a statistical counting process, the resulting distribution is the sum of several Poisson distributions, where the most significant component is the number of electrons produced at the first dynode. Sometimes, it is also possible to see the less frequent response of two photoelectrons.
The ADC-to-PE ratio is the distance between two neighboring n-p.e. peaks, and since the first two peaks (0 pe and 1 pe) are the most prominent peaks (see figure 6.1), their distance will be calculated. The first peak is the electronic noise distribution and the second peak is the single-p.e. distribution. However, the two distributions (and all other distributions mentioned) overlap and thus, the two peaks of the combined distribution are not identical with the peaks of the individual distributions. Furthermore, the underamplification of electrons needs to be taken into account.
Modeling this whole process in detail and fitting the resulting distribution is complicated, but the following procedure yields a good approximation that is compatible with the current HESS calibration, MC and PMT measurements inside the lab:
A histogram of high gain ADC values is created for each pixel during a pedestal run ($\approx$ 80.000 events for HESS I), then it is smoothed with a ten-times left-right right-left moving average and finally, the distance between the first two local maxima is calculated and divided by 1.25. Sometimes, the two maxima cannot be found, which can be the case if the pedestal runs are not performed correctly (e.g. lid not closed, LED not turned on) and there is no useful data for some or all pixels of a telescope. Therefore, when calibrating a HESS data run, the closest n pedestal runs are used and for each pixel k, the median ADC-to-p.e. ratio of all n runs is used as final value v_k. If v_k is more than 10 robust standard deviations RS away from the robust mean RM (with $r_1 = 0.1$ and $r_2 = 0.9$) of the ADC-to-p.e. ratios of the other pixels in the camera, v_k is replaced with RM.
Figure 6.1 shows a typical single PE histogram of a pixel of a HESS telescope during calibration.
The pedestal estimation and the broken pixel finder in MESS are scale-invariant, so the conversion from ADC values to intensities is already done at the beginning. However, since it is common to run these algorithms on ADC values, the text and the plots use ADC values most of the time.

6.3. Online pedestal estimation

The pedestal is the signal the photomultiplier tube (PMT) produces when the telescope is pointed to the sky and no air showers hit the telescope.
The pedestal mean of a pixel is a random offset without meaning for the experiment, and the width is a combination of the electronic noise and the night sky background (NSB). Knowing the mean is necessary to adjust all pixels to the same offset (0) and knowing the width is important for distinguishing between air showers (signal) and NSB plus electronic noise (background). It is important to ignore the outliers, because they would distort the mean and the width estimate of the pedestal.

Baseline The mean pedestal is called *baseline* here. While it would be desirable to define the baseline as the mean of the electronic noise and then measure the number of NSB photons that hit each pixel separately from the Cherenkov light, there are two problems with this approach: First, the mean of the electronic noise is obtained from

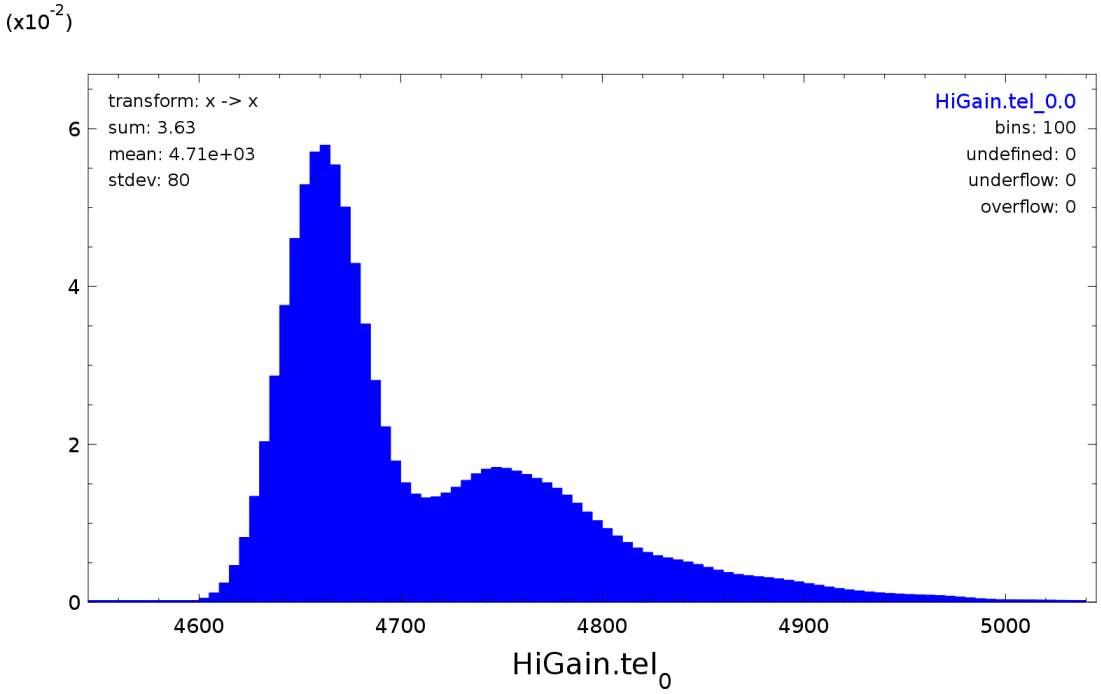


Fig. 6.1.: Single photoelectron distribution of the signal in the high gain channel of the HESS telescope CT1. The first peak is approximately the baseline and results from the electronic noise. The second peak is approximately the signal of the single photoelectrons. Thus, the distance between the two peaks is approximately the number of ADC values per photoelectron.

the pedestal runs, which have taken place some(times a long) time before the data run and might have changed. Second, in HESS, it is not possible to measure the NSB independently from the Cherenkov light of the shower. So in this scheme, the baseline always includes the mean NSB, and if it is subtracted from a shower pixel, sometimes too much is subtracted and sometimes too little. Since the error is additive, this is a problem only for small signals, and since it averages out if several pixels are considered, the resulting error on the estimated energy of the primary particle should not be too large. The effect of this error is not investigated here.

Variable pedestal In HESS, a run lasts 28 minutes, and in this time it is possible that the pedestal changes significantly. Since a false estimation of the pedestal makes it more difficult to distinguish between NSB light and Cherenkov light, it is important to always have a good estimate of the pedestal, so an *online* pedestal estimation algorithm is needed that can detect and deal with variations in the pedestal.
In the following, some simple known online pedestal estimation algorithms are shown. Later, a new method is presented and the results are compared. Note that this section deals with *online* pedestal estimation, the methods described here do not see the complete time series of the data. Instead, they only see the current sample and already have to provide an estimate of the pedestal, so it can be subtracted and the event immediately analyzed.

6.3.1. Known algorithms for online pedestal estimation

Exponential smoothing

This well-known method adds a new sample x_i with weight α and combines it with the current estimate y_i, weighted with $1 - \alpha$.

$$y_{i+1} = (1 - \alpha)\, y_i + \alpha\, x_{i+1} \tag{6.4}$$

If the y_i on the right side of the equation are recursively substituted, it can be seen that earlier x_i will always con-

tribute to the current estimate, but their weight decreases exponentially:

$$(1-\alpha)\,y_i + \alpha\,x_{i+1} = \alpha \sum_{k=0}^{i} (1-\alpha)^{k+1}\,x_{i-k} \tag{6.5}$$

If the mean of the data is constant, a small α can be chosen and good results are achieved, but if there is a trend in the data, the estimates will lag behind. A smaller α will decrease the lag, but it will make the y less smooth. Another problem are outliers, because they affect the estimate linearly.

Simple non-linear smoothing

Another way to estimate the baseline is to start with the mean of the first samples and then increase y by a small constant if $x > y$ and decrease y if $x < y$:

$$y_{i+1} = y_i + \begin{cases} c & x_i > y_i \\ -c & x_i < y_i \end{cases} \tag{6.6}$$

This algorithm is robust to outliers, but if the constant c is too small, y will lag behind changing baselines. If the true baseline is nearly constant and c is too large, y will fluctuate around the baseline.

Performance of these known methods

The plots in figure 6.2 show how exponential smoothing and simple non-linear smoothing perform in different situations. The exponential smoothing algorithm is tested with 3 different parameters ($\alpha = 0.1$, $\alpha = 0.01$, $\alpha = 0.001$) that provide different regularization/attachment to the data, and the non-linear smoothing algorithm is tested with $c = 1$, so there are four plots (one per algorithm). Below the four plots an enlarged view of the four plots is shown, so the data and the jittering baseline are better visible. On page two The data consists of the high gain ADC values of a pixel of a real HESS camera during an observation run. In order to study the behavior of the algorithms during changes of the baseline, artificial functions (a jump, sine waves with different frequencies and a linear slope) are overlaid.
Figure 6.5) is similar to the previous figure, but it has a sawtooth function overlaid.

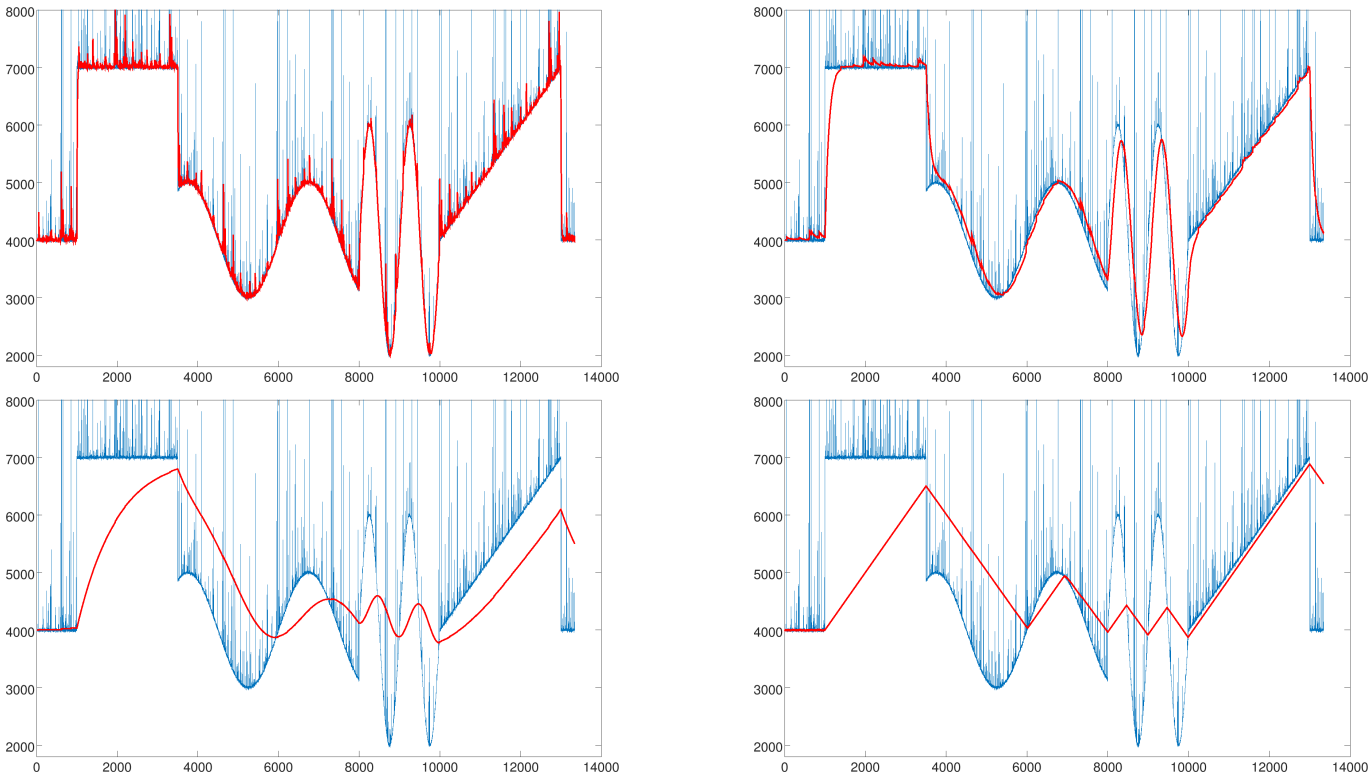


Fig. 6.2.: ADC values (Monte Carlo gammas) of a pixel of a HESS telescope, added to artificial jump, sines and slope, smoothed with different algorithms. Top left, top right, bottom left: exponential smoothing with $\alpha = 0.1$, 0.01 and 0.001. Bottom right: simple non-linear smoothing.

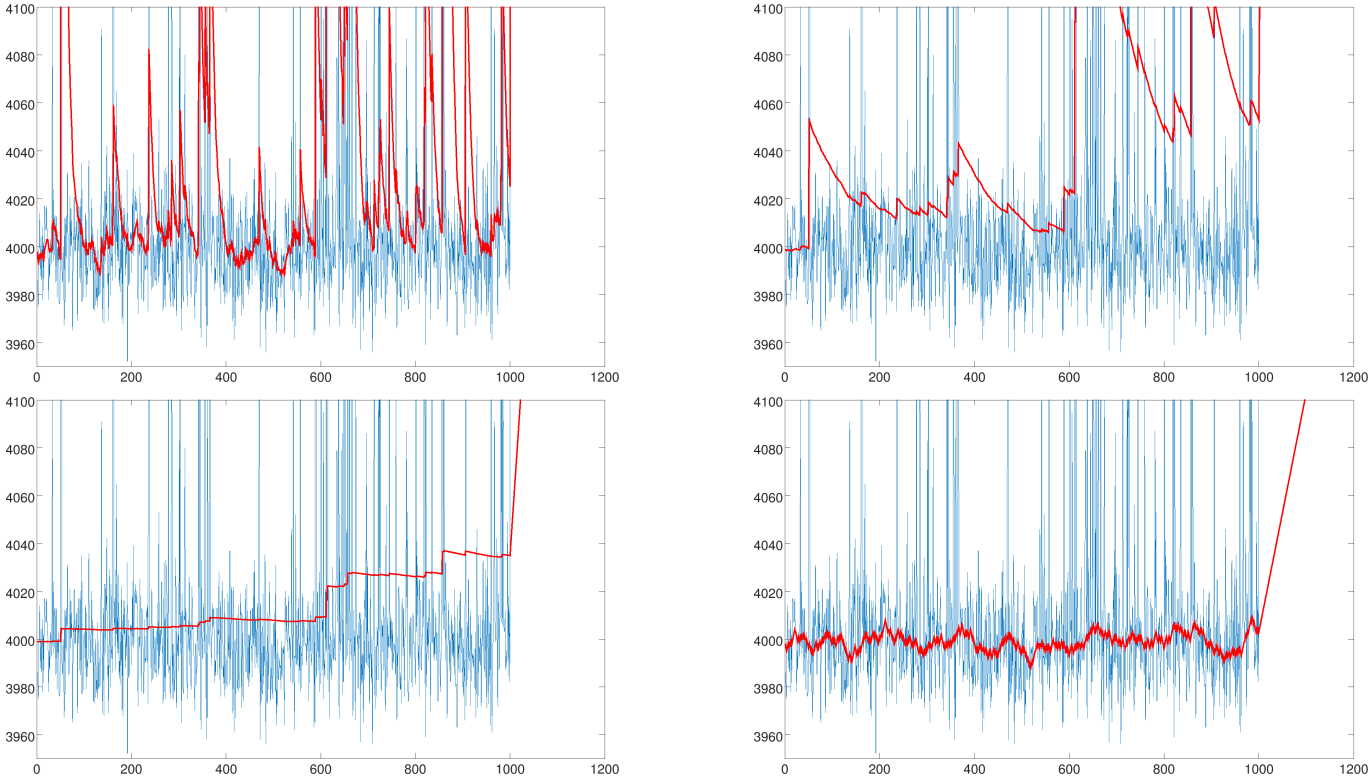


Fig. 6.3.: Same plots as above, with an enlarged view of the first 1200 samples, where the baseline is supposed to be at 4000. Top left, top right, bottom left: exponential smoothing with $\alpha = 0.1$, 0.01 and 0.001. Bottom right: simple non-linear smoothing (note the different scale of the y-axis).

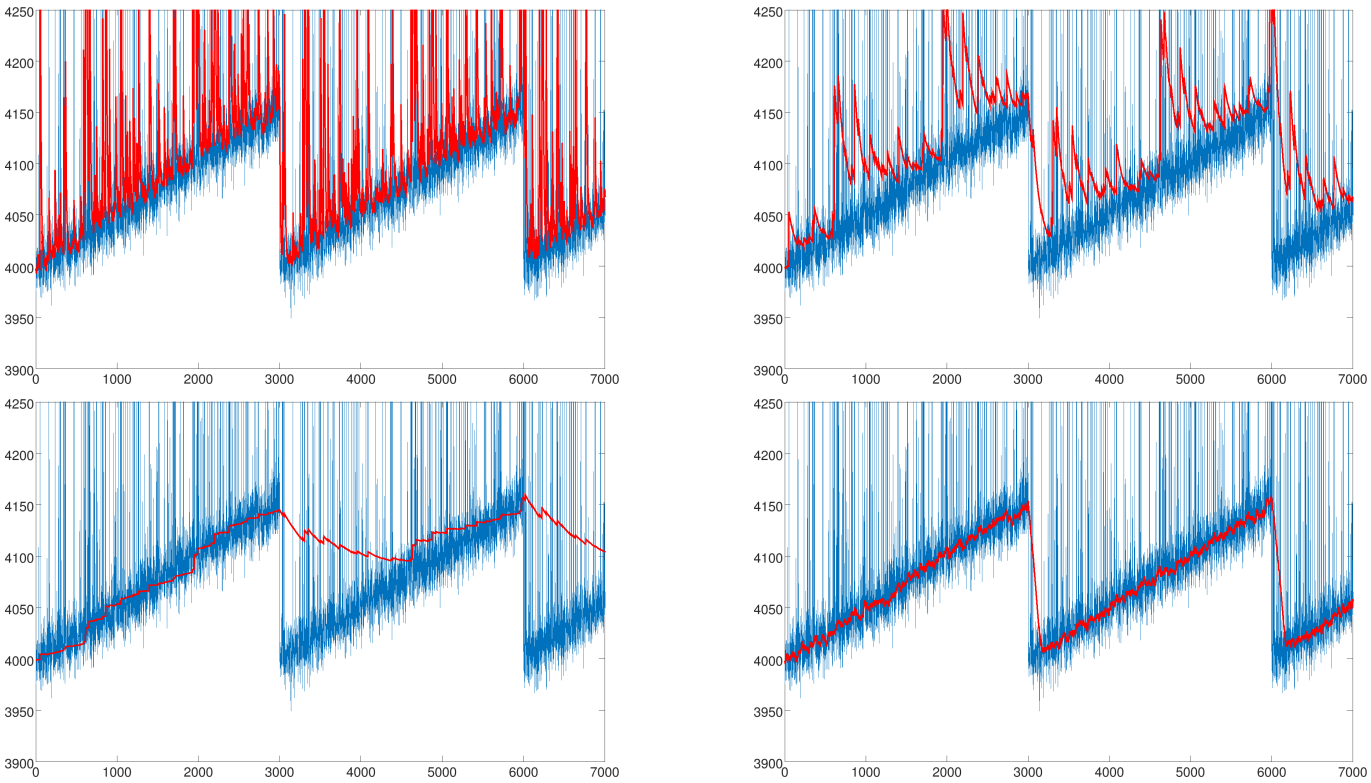


Fig. 6.4.: ADC values (Monte Carlo gammas) of a pixel of a HESS telescope, added to a sawtooth function ($x_i+ = 0.05\,(i\%3000)$), smoothed with different algorithms. Top left, top right, bottom left: exponential smoothing with $\alpha = 0.1$, 0.01 and 0.001. Bottom right: simple non-linear smoothing.

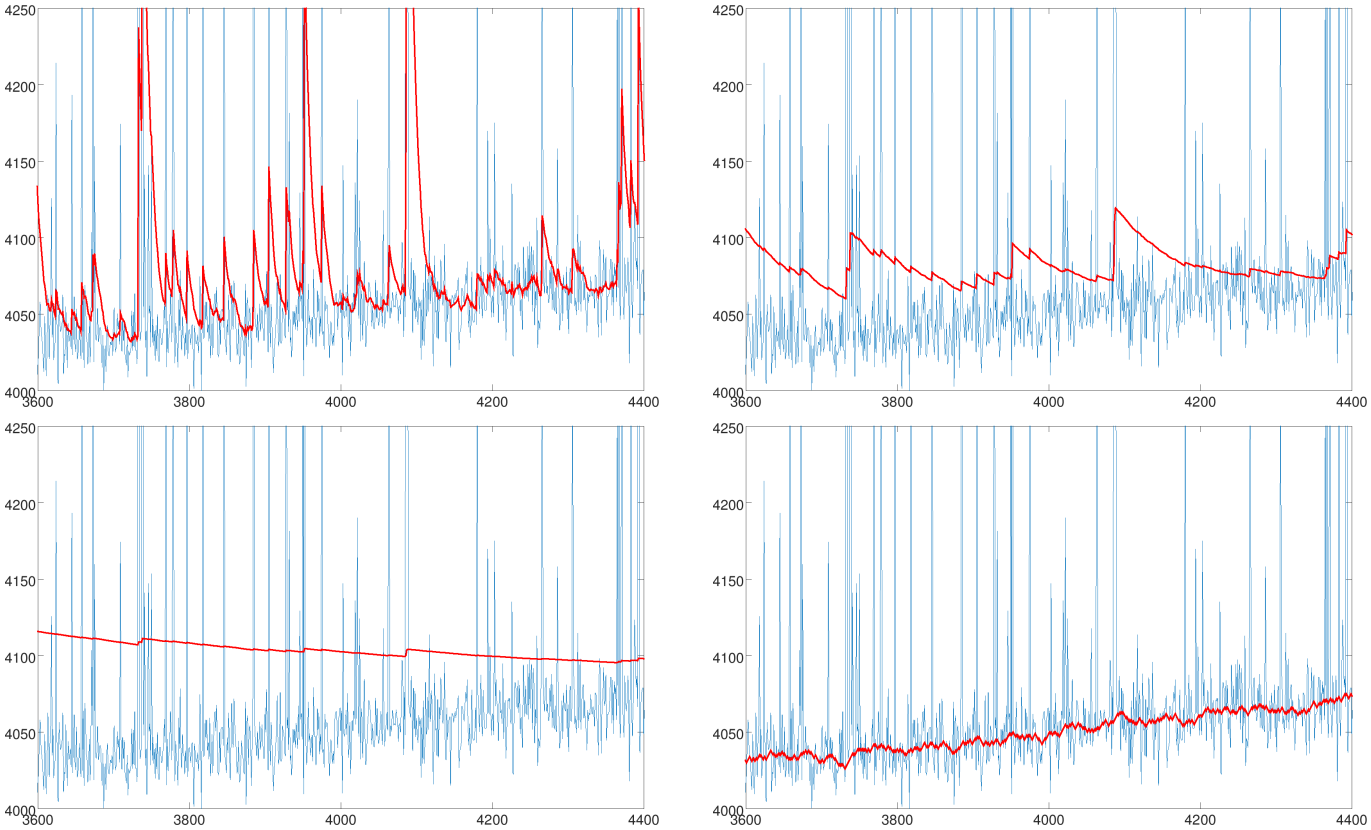


Fig. 6.5.: Same plots as above, with an enlarged view of a slope. Top left, top right, bottom left: exponential smoothing with $\alpha = 0.1$, 0.01 and 0.001. Bottom right: simple non-linear smoothing.

6.3.2. Robust adaptive baseline estimation (RABE)

It can be seen that the standard algorithms either adapt too slowly or are too noisy.
A new algorithm that is not affected by outliers and adapts to trends and jumps in the data was developed and is presented here. It maintains a running estimate of the standard deviation and performs all further calculations relative to it, which makes it scale independent.
The idea is to use two curves, one (y) that follows the data (x) closely, but has no outliers and is less noisy, and another curve (z) that takes the first curve as input and tries to smooth it without producing too much lag.
The first curve is given by:

$$y_{i+1} = y_i + v_i + \arctan(\frac{x_{i+1} - (y_i + v_i)}{s_i})\frac{s_i}{\pi} + (x_i - (y_i + v_i))|j_i| \tag{6.7}$$

Since $\arctan(x)$ converges to $\frac{\pi}{2}$ for $x \to \infty$ (see figure 6.6), the scaling factor $\frac{s_i}{\pi}$ is introduced to make it converge to $\frac{s_i}{2}$. Scale-independence is achieved by normalizing the difference between x and y to units of standard deviation.

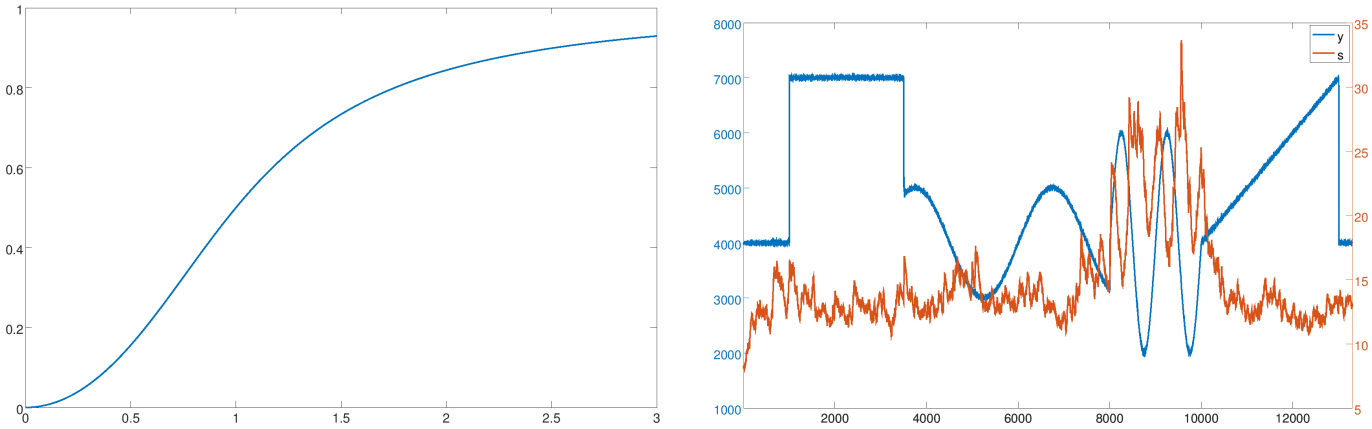


Fig. 6.6.: Left: $\arctan(x^2)\frac{2}{\pi}$ with x being multiples of the standard deviation. This transformation is used to make y robust to outliers. Right: s (approximation of standard deviation of x without outliers) and y (corresponding data without outliers). s should be constant, but it can be seen that s becomes incorrect when the trend in the data is too large.

With the difference between samples, corrected for trends and limited to $3s_i \approx 3$ standard deviations,

$$d_i = \min(|v_{i-1} + x_{i-1} - x_i|, 3s_i \tag{6.8}$$

the standard deviation of x is approximated with an exponential smoothing filter $(\alpha = 0.01)$:

$$s_{i+1} = 0.99s_i + 0.01(y_i - d_i) \tag{6.9}$$

A constant and relatively small α is justified, because it can be assumed that the noise level is constant over short periods of time.
The trend of y is calculated by applying an exponential smoothing filter $(\alpha = 0.1)$ to the difference of two samples, limited to $1s_i \approx 1$ standard deviation:

$$v_{i+1} = 0.9v_i + 0.1(\min(s_{i+1}, \max(-s_{i+1}, y_{i+1} - y_i))) \tag{6.10}$$

The term $(x_i - (y_i + v_i))|j_i|$ is used to quickly catch up if there are jumps in the baseline. This is done by initializing j_0 to 2^{-20} and multiplying it by 2 every time $x_i > y_i$. If the trend term v does not grow fast enough to help y reach x, j will make y approach x with exponential speed: after at most 20 iterations, y will have reached x. After y has overtaken x, j_0 is set to -2^{-20} and multiplied by 2 every time $x_i < y_i$ until x is reached. Then the procedure restarts. If there is no rapidly changing trend or jump in the data, j will flip its sign every few iterations, and its absolute value will be very small and have almost no influence when calculating y_{i+1}. The influence of j will become significant only after the estimate y is below the data x for many iterations: after 10 such iterations $j_i = 2^{-10} \approx \frac{1}{1000}$, after 17 such iterations $j_i = 0.125$ and after 20 such iterations $j_i = 1$.

The second and final curve is calculated by applying an exponential smoothing filter to y ($\alpha = a_i$):

$$z_{i+1} = w_i + (1 - a_i)\, z_i + a_i y_i \tag{6.11}$$

with the trend estimate

$$w_{i+1} = w_i + a_i\, (z_{i+1} - z_i) \tag{6.12}$$

and the variable smoothing factor

$$a_{i+1} = (1 - c_0)\, a_i + \frac{2c_0}{\pi} \arctan(c_1\, (y_i - z_i)^2 / {s_i}^2) \tag{6.13}$$

Making the smoothing factor a_i dependent on the current difference (in units of standard deviations) between the two curves y (data without outliers) and z (smoothed data with possible delay) allows z to follow changes in trends. Replacing a with the second derivative of z should give similar results, but was found to be unstable.
The constants c_0 and c_1 should be between 0 and 0.1 and control how smooth z becomes and how well it follows y. In particular, c_0 is the constant of the exponential smoothing filter and c_1 controls how much influence a single difference has. Since the constants only control the speed of change of z and its trend w, it is not obvious how to set them in order to achieve a certain result. The default values $c_0 = 0.01$ and $c_1 = 0.02$ produce good results for all curves tested. Smoother results can be achieved by lowering c_1 and faster adaption to changes can be achieved by increasing c_2. Attempts to determine the constants algorithmically from the data resulted in unstable systems, but this does not mean that it is impossible.
Instead of using y for tracking x without the outliers, the running minimum could be used. This was done and yielded good results, but since this approach is limited to data with upwards outliers only, it is not used.

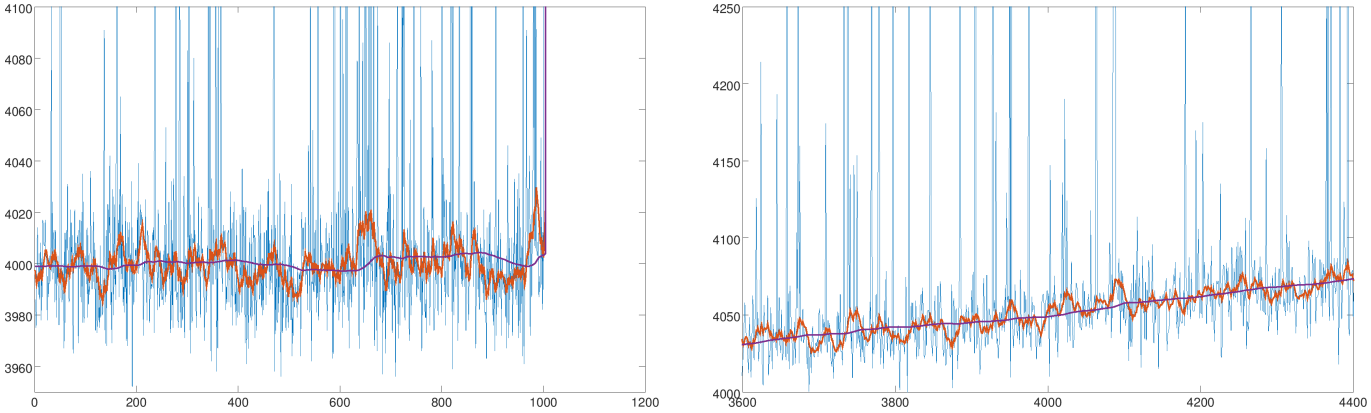


Fig. 6.7.: x (data) in blue, y (approximation without outliers) in red, and z (smoothed y) in purple. Left: constant baseline. Right: variable baseline.

6.3.3. Overshooting and the final RABE algorithm

As can be seen in the top left plot of figure 6.8, this algorithm overshoots after jumps. The problem is that during the jump, the velocity is continued to be estimated, resulting in huge values, which are then used as the prediction for the next sample. This can be prevented by handling the jumps differently: If more than 10 consecutive samples are more than 5 σ away from the current estimate, they are interpreted as a jump of the baseline. After the jump, the mean of those samples is assigned to y and z. Furthermore, for the next 10 samples after the jump, z is set to be equal to y.
Figure 6.9 shows that the final RABE algorithm does not suffer from overshooting. In the other plots, which do not contain any jumps, the improved RABE algorithm produces the same results as the normal RABE algorithm. An open problem are fast changes of the baseline, like in the sine curves with higher frequency, but since this is not a relevant problem in calibration of IACT data, this is not further investigated.

Possible different solution to overshooting problem Another way to prevent overshooting after jumps is to extend the algorithm: The difference between the estimates y_i and z_i, normalized by the standard deviation s_i, can be used to decide whether y_i or z_i should be used as final estimate for a sample. It is also possible to combine the two values by non-linearly weighting them, for example again with the arctan function or with the exponential function:

$$Z(y,z) = y\,(1 - e^{-0.05\,(y-z)^2/s^2}) + z\,e^{-0.05\,(y-z)^2/s^2} \tag{6.14}$$

If Z is not smooth enough, it can be run through an exponential smoothing filter with a small α of e.g. 0.1.
This different solution to the overshooting problem is **not** applied, because the results produced by the improved RABE algorithm are satisfactory and because this solution produces less smooth curves.

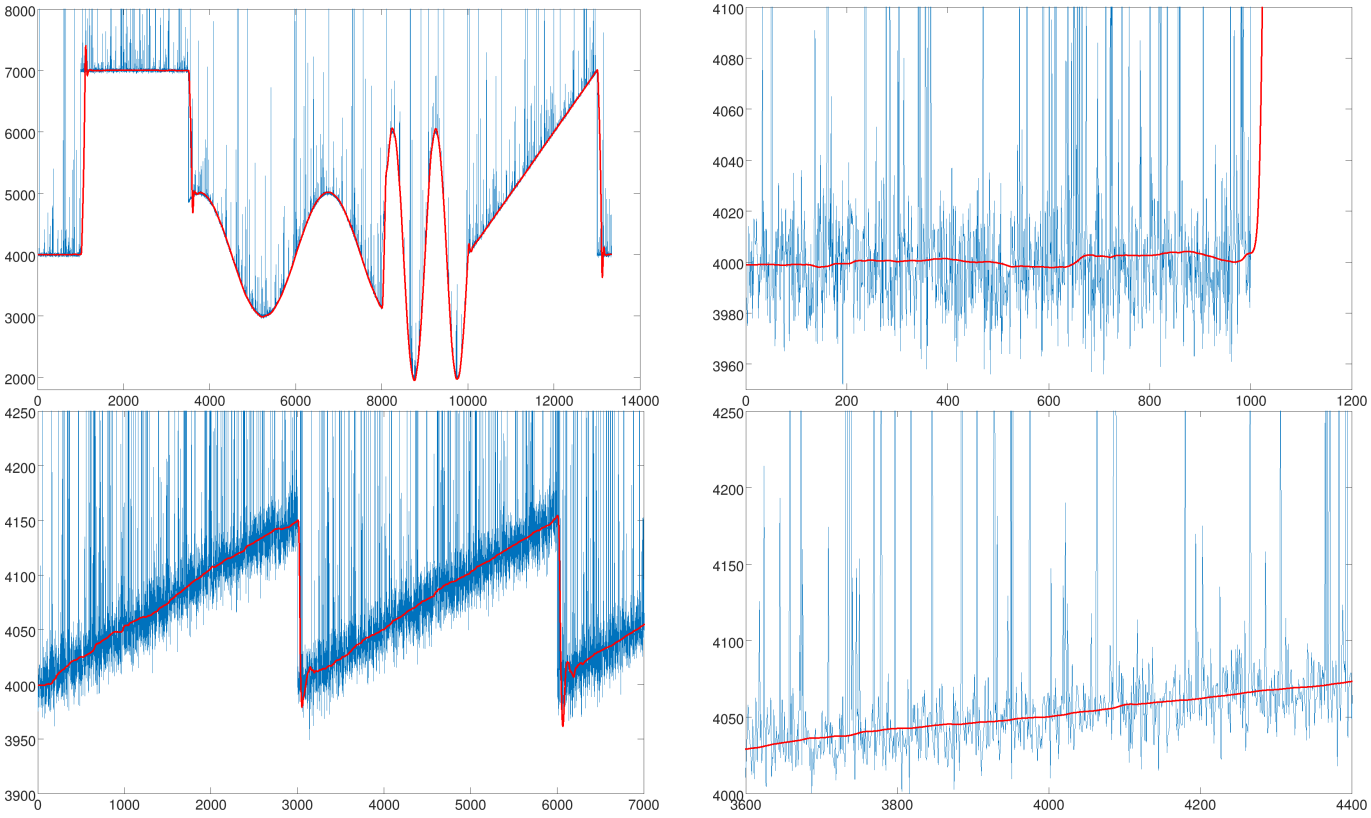


Fig. 6.8.: Baseline estimation using normal RABE for the same data that was used for exponential smoothing and simple non-linear smoothing. x is blue, y is red.

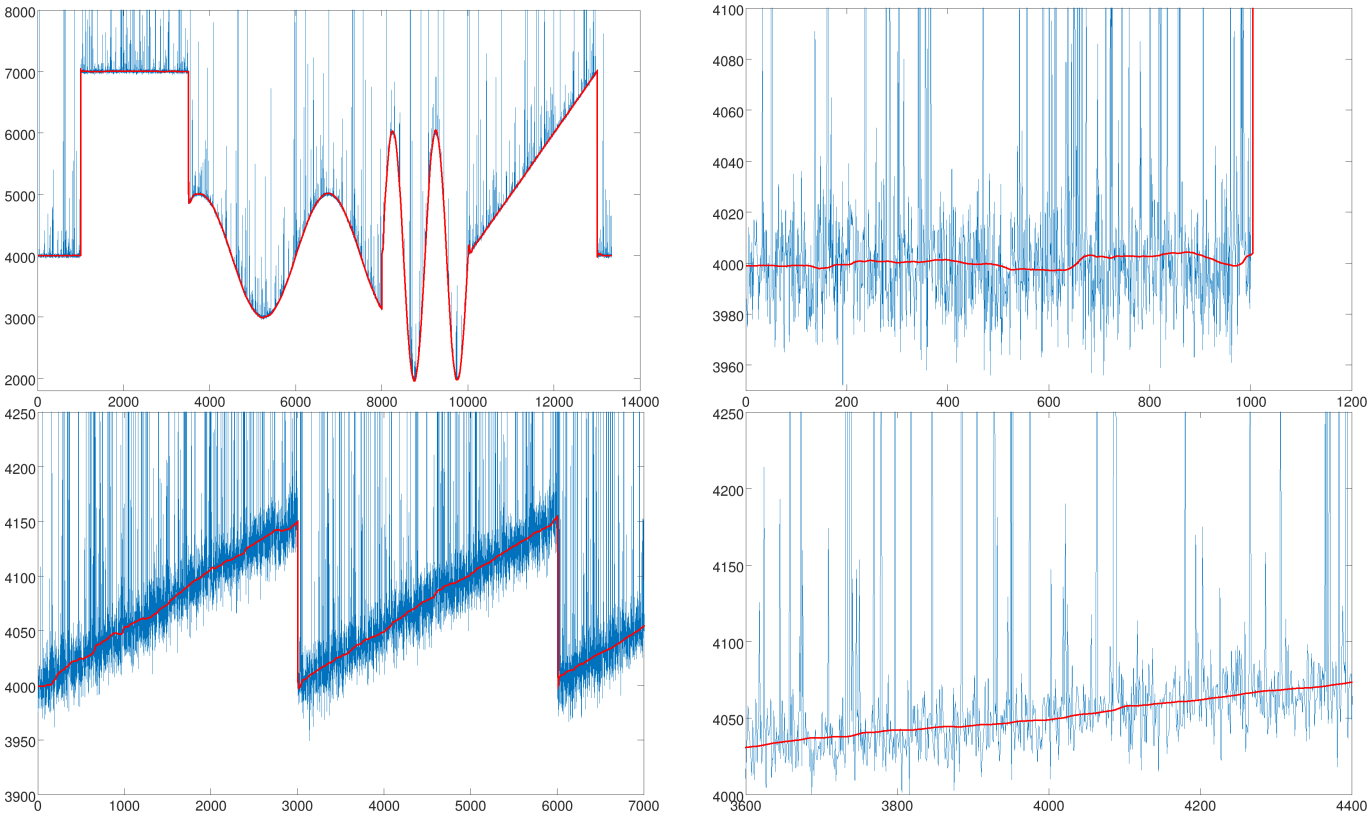
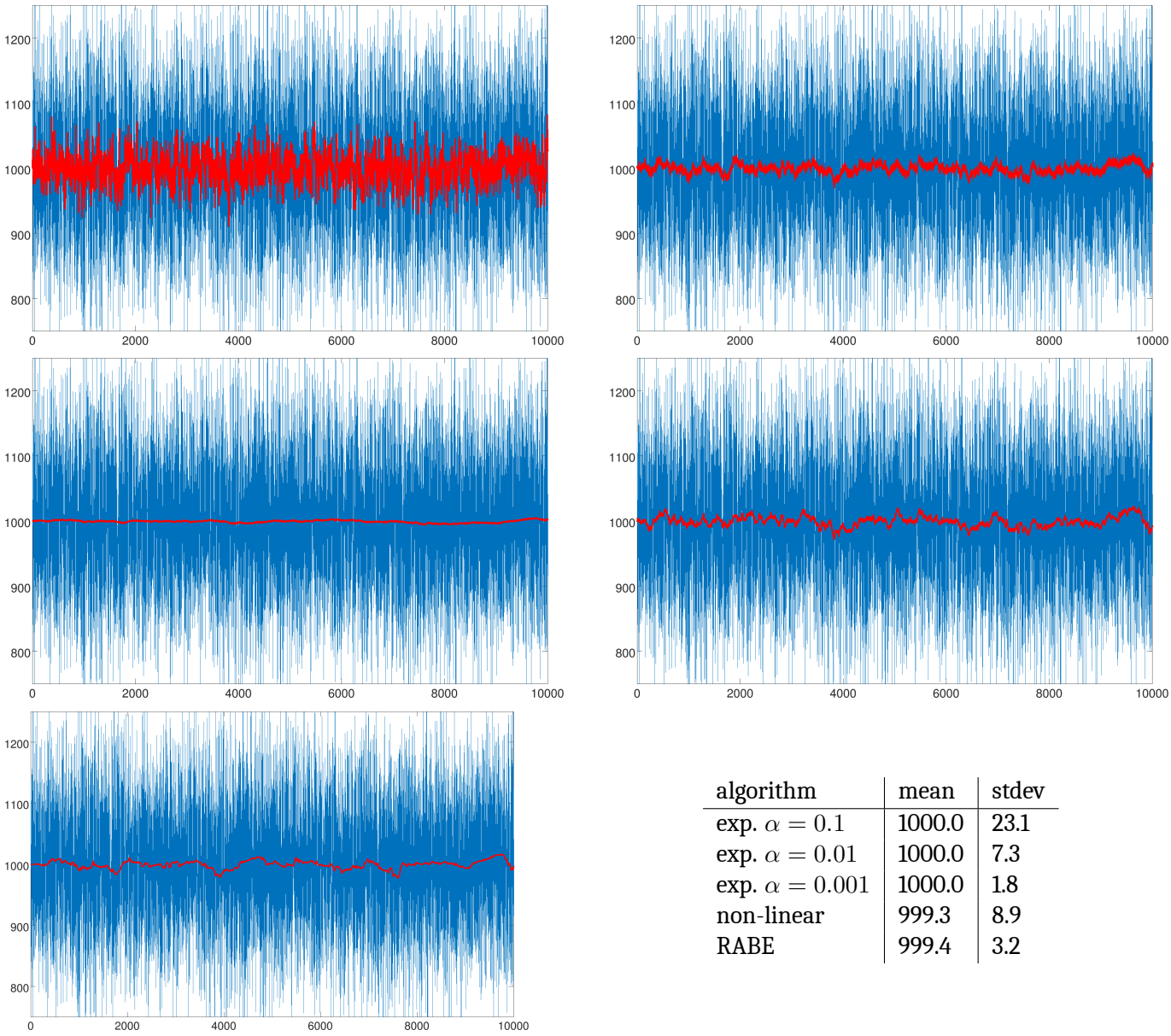


Fig. 6.9.: Baseline estimation using improved RABE with overshooting mitigation for the same data that was used for exponential smoothing and simple non-linear smoothing. x is blue, y is red.

Results on Gaussian noise Applying the algorithms to normally distributed noise ($\mu = 1000, \sigma = 100$) yields the results shown in figure 6.10. Exponential smoothing with $\alpha = 0.001$ performs best, followed by RABE with parameters set to default values.



algorithm	mean	stdev
exp. $\alpha = 0.1$	1000.0	23.1
exp. $\alpha = 0.01$	1000.0	7.3
exp. $\alpha = 0.001$	1000.0	1.8
non-linear	999.3	8.9
RABE	999.4	3.2

Fig. 6.10.: Normally distributed noise ($\mu = 1000$, $\sigma = 100$). Top left, top right, center left: exponential smoothing with $\alpha = 0.1$, 0.01 and 0.001. Center right: simple non-linear smoothing. Bottom left: RABE.
The table shows the mean of the estimated baseline (red curve) for each algorithm. The closer the mean to the real mean ($\mu = 1000$), the better. The standard deviation of the estimated baseline, however, should be as close to 0 as possible.

6.3.4. Conclusion

Given that RABE adapts to changes of the baseline much faster than simple non-linear smoothing or exponential smoothing with small α, is robust to outliers (both upwards and downwards), scale-invariant, adapts to different noise levels, and produces a smoother baseline than simple non-linear smoothing and exponential smoothing with large α, it is the preferred baseline algorithm in MESS. Possible improvements could be faster detection of changing slopes and a more intuitive parameter for the trade-off between being smooth and being close to the data.

Source code

```
float x=0, y=0, z=0, s=0, v=0, w=0, a=0, j=1./1048576, q, y0, z0;
while (1) {
    x = get_next_sample();    y = y0 = y+v;    q = x-y;
    if (j < 0 && x > y) j =  1./1048576; else
    if (j > 0 && x < y) j = -1./1048576;
    if (s < 1e-37) { y += q * 0.001;    z += (y-z) * 0.01; } // if s == 0
    else y += atanf(q/s)/M_PI * s + q * fabsf(j);
    j *= 2;
    q = fabsf(q); if (q > 3*s) q = 3*s;    s += (q - s)*0.01;
    v += (y-y0 - v)*0.1; // y-y0 is max. 0.5*s
    z = z0 = z+w;    z += (y-z)*a;
    w += (z-z0)*a; // because: (w+(z-z0) - w)*a
    a += (atanf(0.02*SQR((y-z)/s))*2/M_PI - a) * 0.01;
}
```

Listing 6.1: Robust Adaptive Baseline Estimation Algorithm (approaching jumps with exponential speed).

```
float x=0, y=0, z=0, s=0, v=0, w=0, a=0, q, y0, z0, xn=0;
int y_above=0, jmp=0, just_jumped=0;
while (1) {
    x = get_next_sample();    y = y0 = y+v;    q = x-y;
    if (fabsf(q) < 5*s) jmp = xn = 0;
      else if (y_above*q > 5*s) { y_above = -y_above;    jmp = 1;    xn = x; }
      else { jmp++;    xn += x; }
    if (s < 1e-37) { y += q*0.001;    z += (y-z)*0.01; } // if s == 0
      else if (jmp > 10) { y = z = xn/jmp;    jmp = xn = 0;   just_jumped = 1; }
      else {
        y += atanf((x-y)/s)/M_PI * s;
        q = fabsf(x-y); if (q > 3*s) q = 3*s;
        s += (q - s)*0.01;
        v += (y-y0 - v)*0.1; // y-y0 is max. 0.5*s
        if (just_jumped) {
            if (++just_jumped == 10) just_jumped = 0;
            else z = y;
        } else {
            z = z0 = z+w;    z += (y-z)*a;
            w += (z-z0)*a; // because: (w+(z-z0) - w)*a
            a += (atanf(0.02*SQR((y-z)/s))*2/M_PI - a) * 0.01;
        }
    }
}
```

Listing 6.2: Robust Adaptive Baseline Estimation Algorithm (improved version that mitigates undesired behaviour after jumps).

The code can be changed to use double precision variables by replacing *float* with *double*. The precision of the arcus tangens (*atanf*)and the calculation of the absolute value (*fabsf*) are not important, and in almost any case, their argument will no exceed the range of a single float, so these two functions do not need to be changed to their double precision counterparts.

6.4. NSB level estimation

The NSB level is here defined as the mean pedestal width of the high gain channel across all pixels and all telescopes. It is calculated on-the-fly for each event. Although the electronic noise is also included in the pedestal, it is not mentioned here, because the NSB always dominates, especially for regions in the sky that are bright in optical light. The current NSB level is important for other parts of the analysis, such as γ/hadron separation or energy estimation. A good estimate is needed, so the correct lookups can be loaded.

6.5. Broken pixel (per gain channel) identification

It is important to identify unusual numbers of ADC values, because they might introduce unwanted responses later in the analysis. For the time this calibration will be applied, i.e. for the N_{run} events, a gain channel of a pixel with unusual behavior is marked as broken and its data will be ignored.
It is not trivial to identify broken pixels and it is not enough to merely look at the distributions of the standard deviation, as these counter examples show:

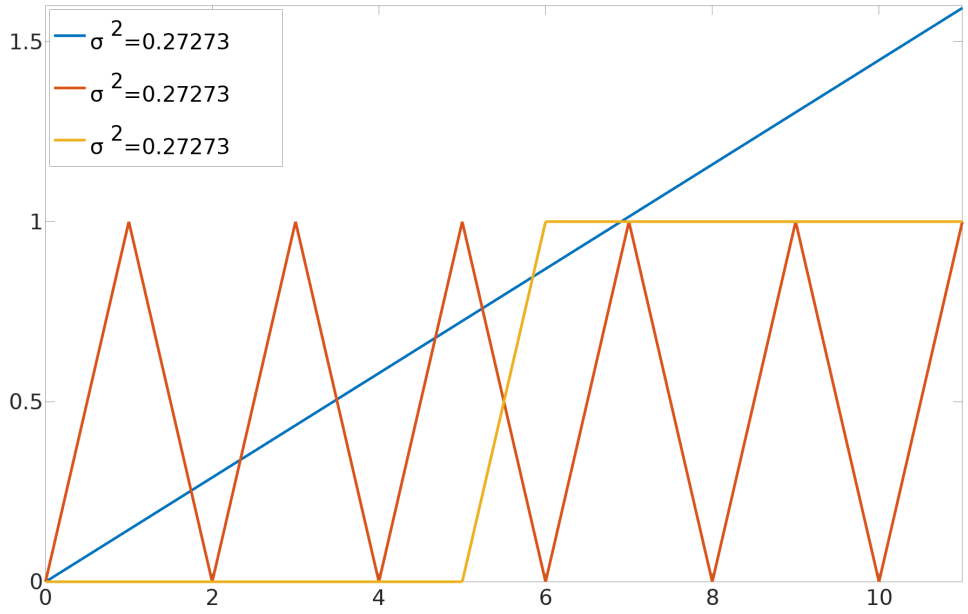


Fig. 6.11.: Same variance.

Sometimes, the ADC values of a pixel are nearly constant for a long time, with little noise, then the ADC values are suddenly at a different value and stay there. The variance in such a case can be similar to the variance of a healthy pixel.
A good way for checking which pixels are broken is to manually inspect the data, preferably the large events, because for them, it is obvious if a pixel contains signal or not. If after several ten thousand events the same pixels get the wrong amount of light, they should be marked as broken.
These are the requirements on an algorithm that identifies broken pixels:

- it must take into account that events that land on the part of the camera that is closest to the array center are more likely to trigger
- it must be able to detect clusters of broken pixels, like broken drawers. This is not easy, because some methods compare the pixel to its neighbors in order to find irregularities. Clusters introduce two kinds of neighborhoods: inside the cluster and at the border. In the first case, the algorithm must not be mislead by the homogeneity of the neighborhood and in the second case, the algorithm must not flag a healthy pixel, which is outside of the cluster, but next to its border, as broken because of the heterogeneity of the neighborhood.
- it must work reliably also for low statistics. This is important for online analysis and for temporarily broken pixels. A longer calibration period gives better statistics, but if a pixel is broken only for the fraction of a that period, it will be either marked broken for all that time or healthy, which is both not correct. Here, it was found that several ten thousand events ($\approx$ 5 minutes of HESS I data taking) are enough.
- it must not produce false positives, so for Monte Carlo data, no broken pixels must be found
- the intensities of the mean calibrated telescope event must be flat and normally distributed

The following checks are applied to each telescope, gain channel and pixel separately. After a check, the value of each pixel is compared to the other pixels in order to identify unusual behavior.

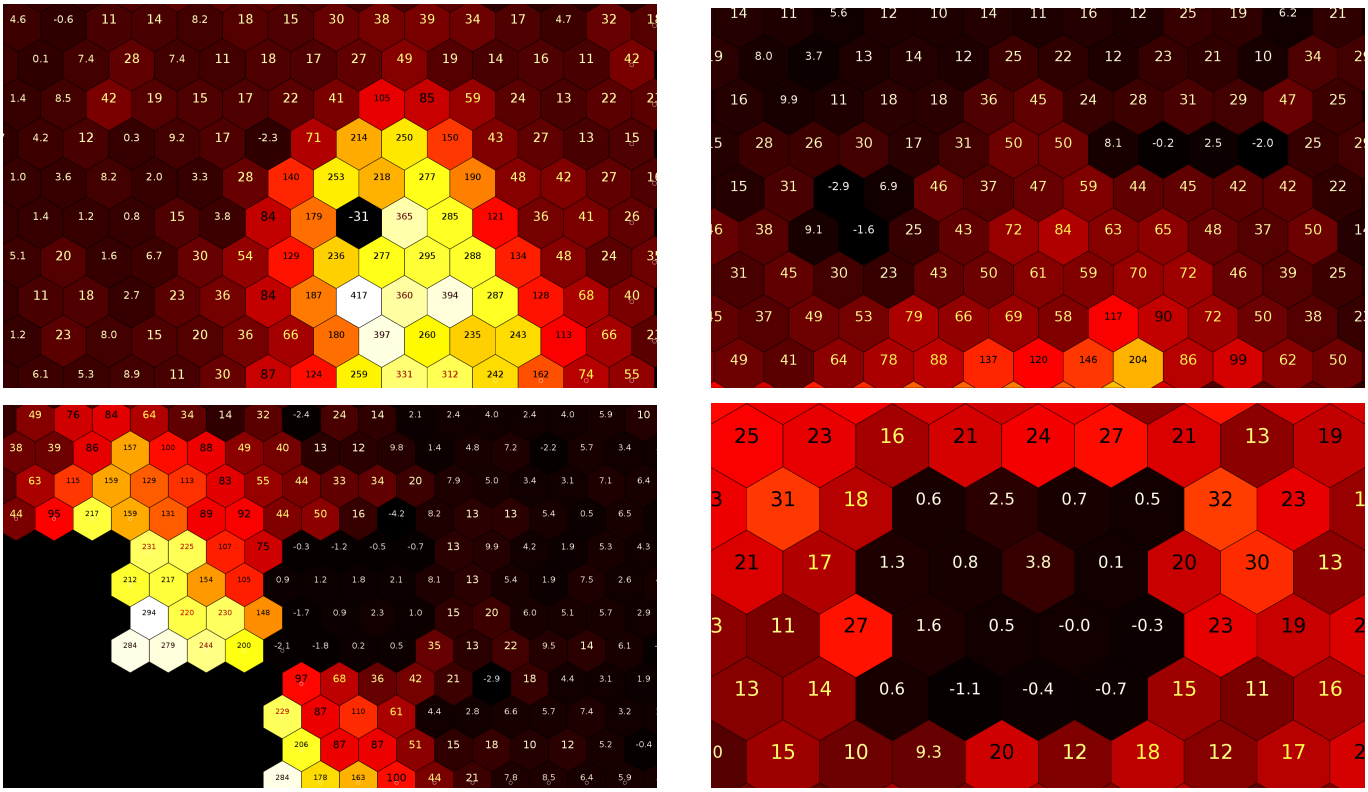


Fig. 6.12.: Examples of how broken pixels can look like when inspecting data. The images show the ADC values of the high gain channel for the current shower. The values are normalized per pixel, i.e. the value of a pixel is the number of standard deviations from its mean ADC value of the current run. Top left: a single broken pixel. Top right: a broken ARS. Bottom: broken drawers.

6.5.1. Too often no signal

It is counted how often a pixel has zero ADC values within the calibration period. The expected value is 0, with almost no variation for healthy pixels, so the distribution usually consists of some spikes: one at zero and some at other values. For such a distribution, mean and standard deviation are not helpful for defining cuts. Instead, all pixel gain channels (i.e. all gain channels of all pixels) that have no signal in more than 5% of the events of the calibration period are marked as broken.

6.5.2. Unusual differences in time

If pixels fluctuate significantly less or more than the rest of the pixels in the camera, they are marked as broken. Here, the standard deviation σ of a pixel value during a calibration period is calculated on-the-fly. This value is provided by the baseline algorithm and since signal pixels are not included in the calculation, this value approximates the current noise level of the pixel. Then, the number of pixels with a difference in two consecutive events of more than 1σ (current standard deviation) is counted and divided by the number of times the pixel participated in events in the calibration period.

$$D^k = {N_k}^{-1} \sum_i |v_i^k - v_{i-1}^k| > \sigma_i^k \tag{6.15}$$

Then, the outliers are found by calculating the robust mean RM and standard deviation RS of D for $r1 = 0.2$ and $r2 = 0.8$. All pixel gain channels whose D^k is smaller than $RM - 8RS$ are marked as broken.
On MC protons, the parameter is normally distributed, and in data it finds the pixels with unusual differences in consecutive events.

6.5.3. Signal not often enough

Sometimes, a pixel has the same noise level as the other pixels, but it is not able to detect any signal. This check marks pixels as broken that do not measure enough signal during the calibration period.
For each pixel k it is counted how often there is signal with more than 5 running standard deviations away from the mean.
Figure 6.13 shows the signal of a healthy pixel (left) and an unhealthy pixel (right). It can be seen that the unhealthy pixel almost never detects any signal outside the 5σ band.

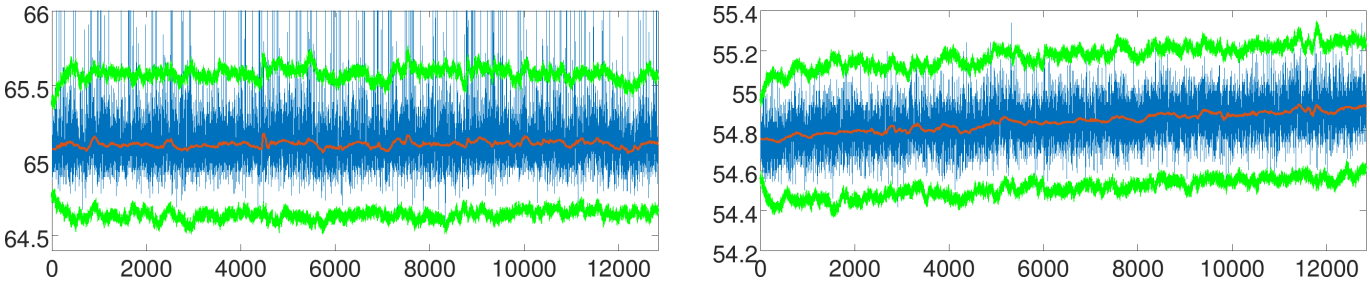


Fig. 6.13.: ADC values (multiplied by ADC-to-PE ratio) of a healthy pixel (left) and an unhealthy pixel (right). The running pedestal estimation is shown in red and the bands that mark 5 running standard deviations is shown in green.

6.5.4. Unusual standard deviations

If the standard deviation of a pixel during the calibration period significantly deviates from the standard deviations of the mean standard deviation of all pixels, it is marked as broken.
For each pixel k, the mean s^k of its running standard deviation is calculated $S^k = \sum_k s_i^k$ and compared to the other S by calculating the robust mean RM and standard deviation RS of S for $r1 = 0.2$ and $r2 = 0.8$ and marking all pixels that are more than 10σ away as broken.

6.5.5. Pixels with no signal that have pixels with signal as neighbors

This parameter is calculated for each gain channel, but after conversion from ADC values to photoelectron counts. It is unlikely that a pixel v^k with no signal has several neighbors v^j with signal. Specifically, it was found that a good separation parameter can be created by having three conditions

$$c_1(i,k,j) = \begin{cases} 1 & v_i^k < 0.2 \text{ and } v_i^j > 8 \\ 0 & \text{otherwise} \end{cases} \tag{6.16}$$

$$c_2(i,k,j) = \begin{cases} 1 & v_i^k < 1 \text{ and } v_i^j > 15 \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

$$c_3(i,k,j) = \begin{cases} 1 & v_i^k < 1.5 \text{ and } v_i^j > 30 \\ 0 & \text{otherwise} \end{cases} \tag{6.18}$$

from which at least one has to be true

$$c_4(i,k,j) = \begin{cases} 1 & c_1(i,k,j) + c_2(i,k,j) + c_3(i,k,j) \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.19}$$

for at least two neighbors v^j:

$$c_5(i,k) = \begin{cases} s & s = \sum_j c_4(i,k,j) > 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.20}$$

The number of neighbors for which $c_5(i,k) > 1$ is summed up across all events in the calibration period:

$$N^k = \sum_i c_5(i,k) \tag{6.21}$$

The robust mean and standard deviation are calculated ($r1 = 0$ and $r2 = 0.8$), and pixel gain channels that are more than 8σ away from the mean are marked as broken. Here, $r1 = 0$ is chosen because it is not abnormal for a pixel to have $N^k = 0$ and thus be at the very left end of the distribution.
The order of the checks is important, because if a pixel with unusual behavior is included in a check, it might bias it. If, for example, a pixel with constant fake signal participates in a method that uses the neighbors as comparison, its neighbor pixels might get identified as unusual, because they have too little signal. But if it is marked as broken beforehand, it will not participate in the check.
Here, the simple and robust checks are done first, and the more complex checks later. The last check is the one that involves the neighborhood.
In most cases, the identified broken pixels match those found by the calibration used by HESS. For a random Monte Carlo sample of 50.000 proton showers, the above procedure misidentified two pixels in the low gain channel of a telescope as broken. Ideally, no broken pixels would have been found, however, 2 false positives in $960 \times 4 \times 2$ pixel gain channels is acceptable.
A better scheme could include more checks, and the final decision (if the pixel is broken or not) could combine the responses of all checks to one single number to cut on.

6.6. High Gain/Low Gain Ratio and Conversion to Intensities

By now, the high gain or/and the low gain channel of some pixels is/are probably marked as broken. For all other pixels (those that still have both gain channels), their mean high gain/low gain ratio is calculated:

$$\mathrm{HL}^k = N^{-1} \sum_{i \in N_{\mathrm{run}}} \frac{H_i^k}{L_i^k}, \text{ if } H_i^k > 5 \text{ and } L_i^k > 5 \tag{6.22}$$

so that in case the signal in the high gain channel is too high to be accurate, the signal of the low gain channel - multiplied by the high gain/low gain ratio - can be used instead. The conversion from high gain and low gain to intensity is then done as follows:

- if only the low gain channel is broken, the value of the high gain channel is used
- if only the high gain channel is broken, the value of the low gain channel is multiplied by HL^k
- if both channels are available, the value of the high gain channel is used if its ADC value is less than 4000 above the noise mean, otherwise the value of the low gain channel multiplied by HL^k is used
- if both gain channels are broken, the pixel is marked as broken and the intensity is undefined

Figure 6.14 shows a plot of the high gain/low gain ratio of all pixels in HESS run 60000, which is obtained with the pipeline in figure 6.3.

```
mess -micropipe readhess.r: -in 60000 , integratesamples.i:r ,
    pixelstoparset.HiGain:i -pix_vt PIXEL_VT_ADCSUM_HI ,
    pixelstoparset.LoGain:i -pix_vt PIXEL_VT_ADCSUM_LO ,
    hist.hist:HiGain,LoGain -axis HiGain.tel0:601,4200,4800/LoGain.tel0:151,3800,3950 ,
    plothist.plo:hist
```

Listing 6.3: MESS pipeline for creating a 2-dimensional high gain/low gain histogram.

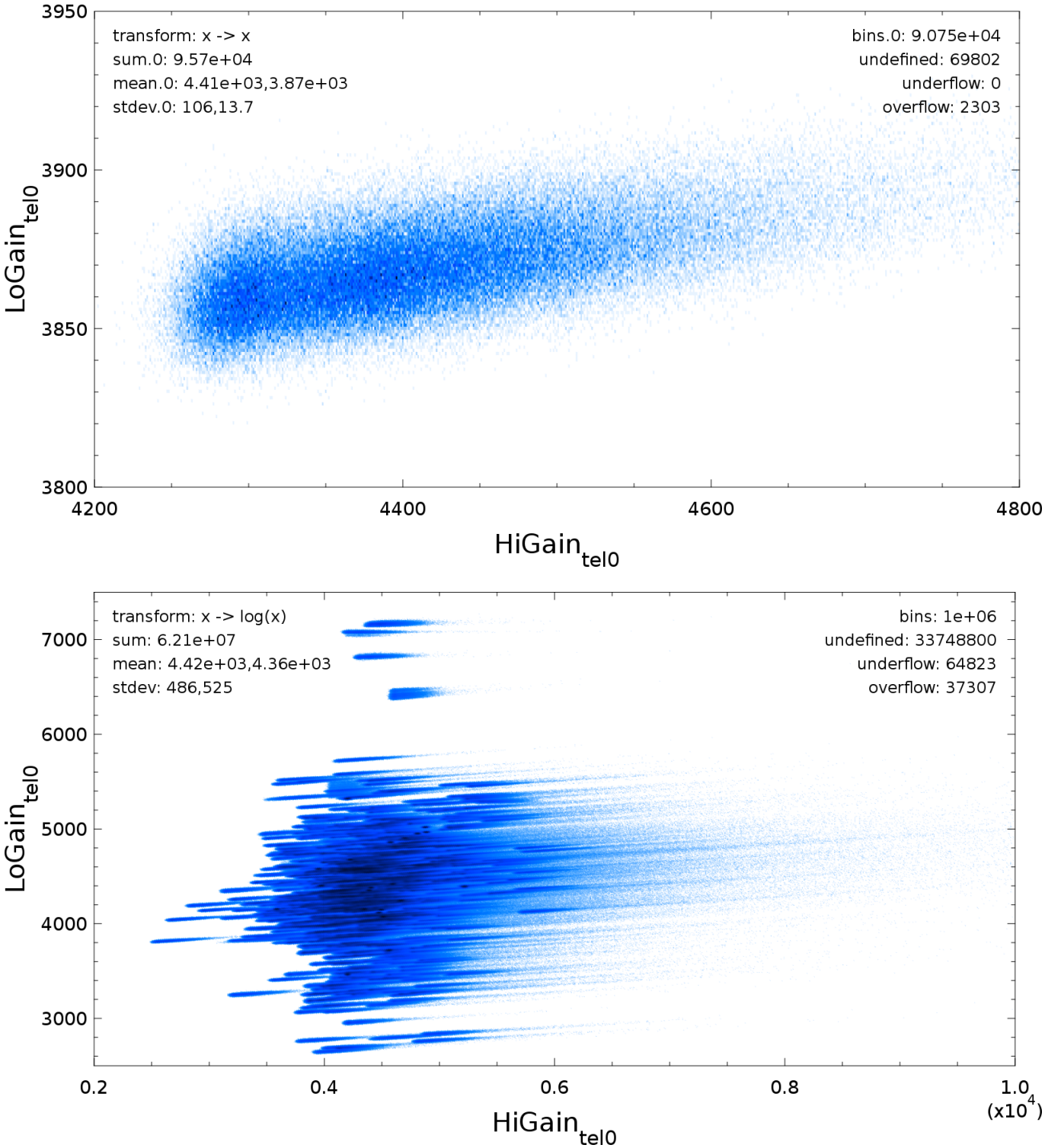


Fig. 6.14.: High gain vs. low gain ADC values for HESS run 60000 before pedestal subtraction. Top: For the first pixel in the camera. Bottom: For all pixels in the camera.

6.7. Comparison to *sim_telarray* calibration

In order to check the quality of the calibration, it is compared to the *sim_telarray* calibration of Monte Carlo data, which knows the exact pedestal, high gain/low gain ratio and ADC-to-PE ratio. Since there are no Monte Carlo pedestal runs, the ADC-to-PE ratio is directly read from the Monte Carlo file. The rest of the calibration (pedestal estimation, calculation of high gain/low gain ratio, and conversion to intensities) is done exactly like the calibration of real data. This means that no Monte Carlo ground truth is used anywhere, and it also means that the broken pixel detection algorithm is run, which is a good way to check if it produces false positives.
The results of the comparison can be seen in figure 6.19. Although the true calibration parameters are unknown to MESS, its calibration yields results that are very similar to the results of the *sim_telarray* calibration. Note that for this test, a calibration period of only 20.000 events was used to determine the calibration parameters.

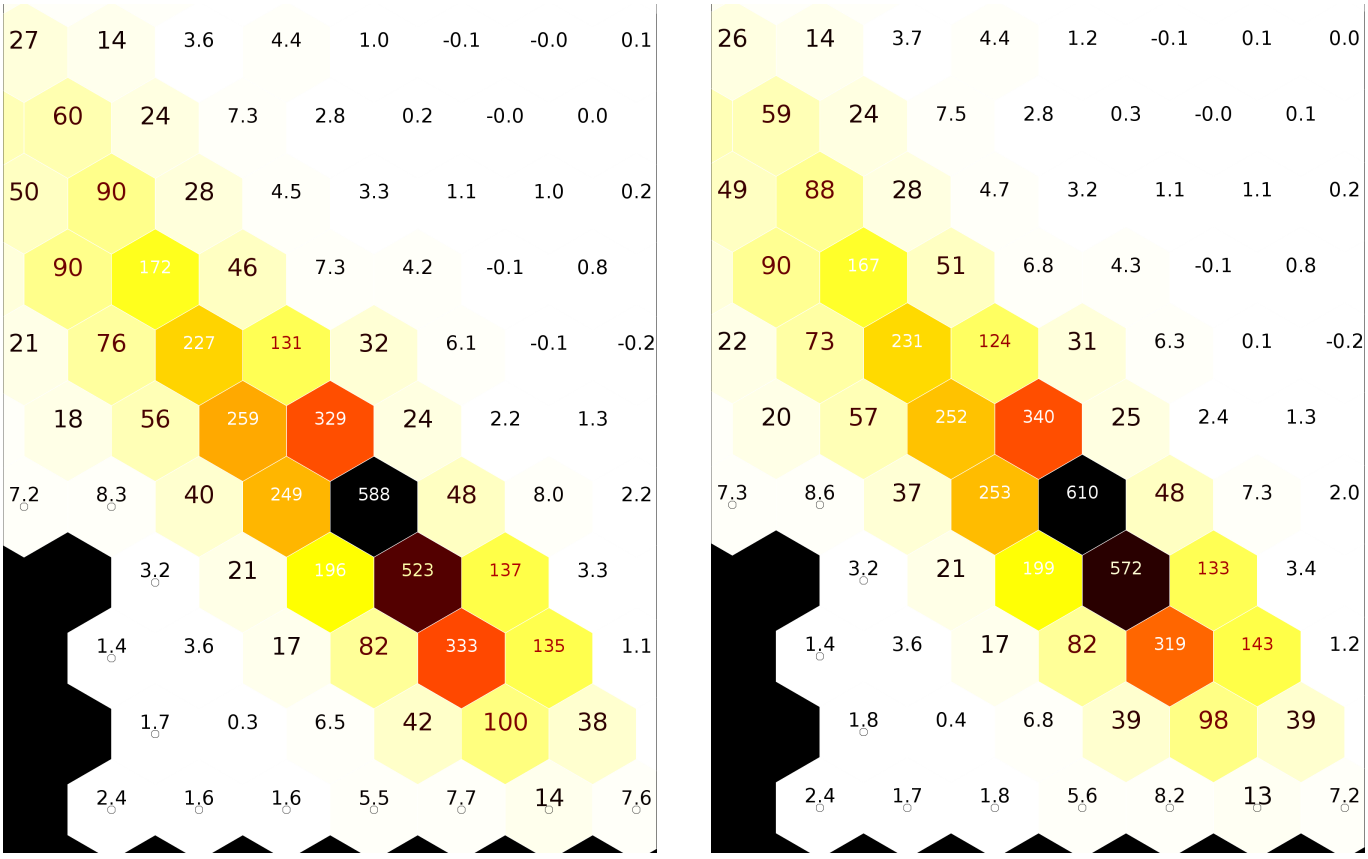


Fig. 6.15.: Random Monte Carlo shower calibrated with MESS (left, with calibration parameters derived from showers) and *sim_telarray* (right, using the true calibration parameters).

The distribution in figure 6.16 (left) shows the relative difference of intensities $(I_{\text{MESS}} - I_{\text{sim\_telarray}})/I_{\text{sim\_telarray}}$ for all pixels where $I_{\text{MESS}} > 5$ pe or $I_{\text{sim\_telarray}} > 5$ pe, across 5000 events. It can be seen that the error distribution is not ideal; its mean is 1% off and its standard deviation is 4.5%. Mean and standard deviation improve, if a higher intensity threshold is chosen, which can be seen in the left and right plot.
The same plots are shown for the absolute difference of intensities in figure 6.17.
Since in average $(I_{\text{MESS}} - I_{\text{sim\_telarray}}) < 0$, it can be concluded that the MESS calibration underestimates the signal slightly. One might think that there are errors in the high gain/low gain ratio calculation, but as this bias also exists at low intensities (where the low gain channel is ignored) and as the ADC-to-PE ratio is directly read from the MC file„ the bias must come from the baseline estimation.
Figure 6.18 shows the error on complete showers $S_{\text{sim\_telarray}}$ and S_{MESS}, where S is the sum of all intensies > 5 pe.

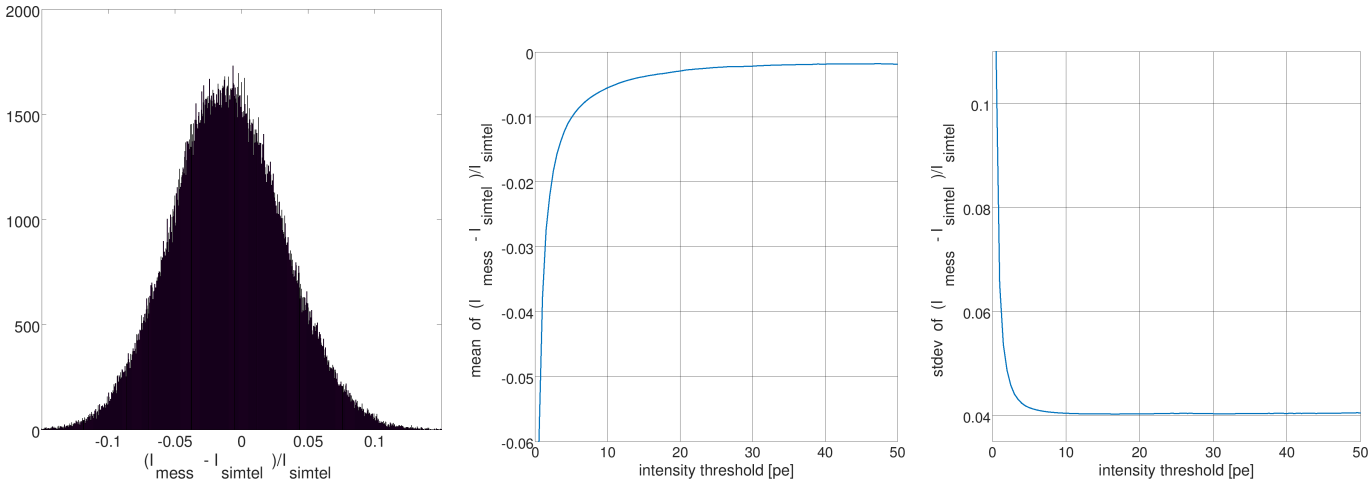


Fig. 6.16.: Relative per-pixel difference of the sim_telarray and the MESS calibration.

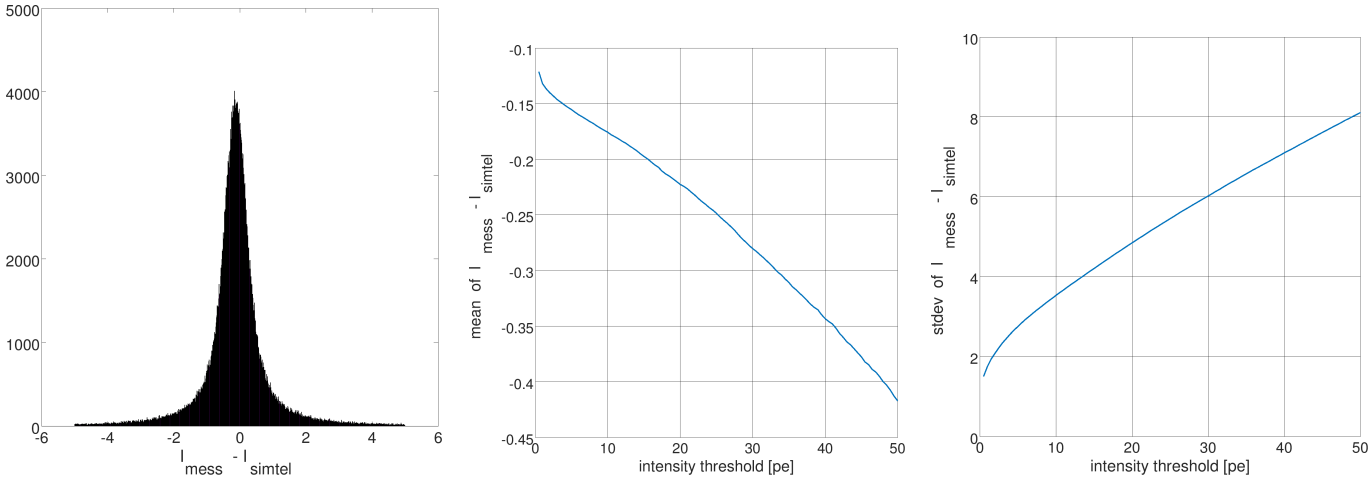


Fig. 6.17.: Absolute per-pixel difference of the sim_telarray and the MESS calibration.

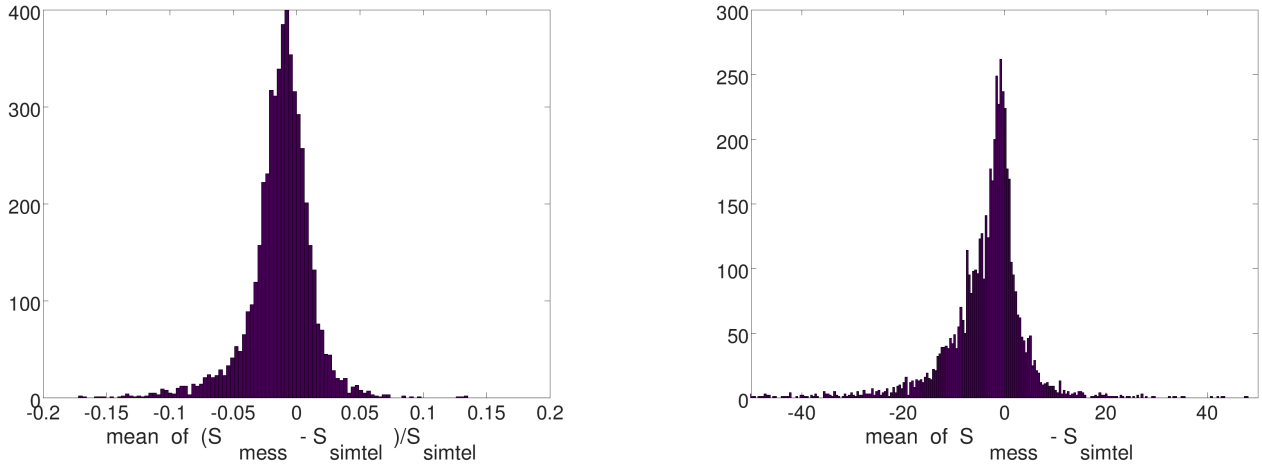


Fig. 6.18.: Absolute per-shower difference of the sim_telarray and the MESS calibration.

6.8. Comparison to *HESS* calibration

The MESS calibration is also compared to the HESS (Heidelberg version) calibration.
It is not trivial to find out which calibration is closer to the truth, but it can be used as a consistency check for the MESS calibration. There should be a huge overlap of the indentified broken pixels and the shower images should look similar.

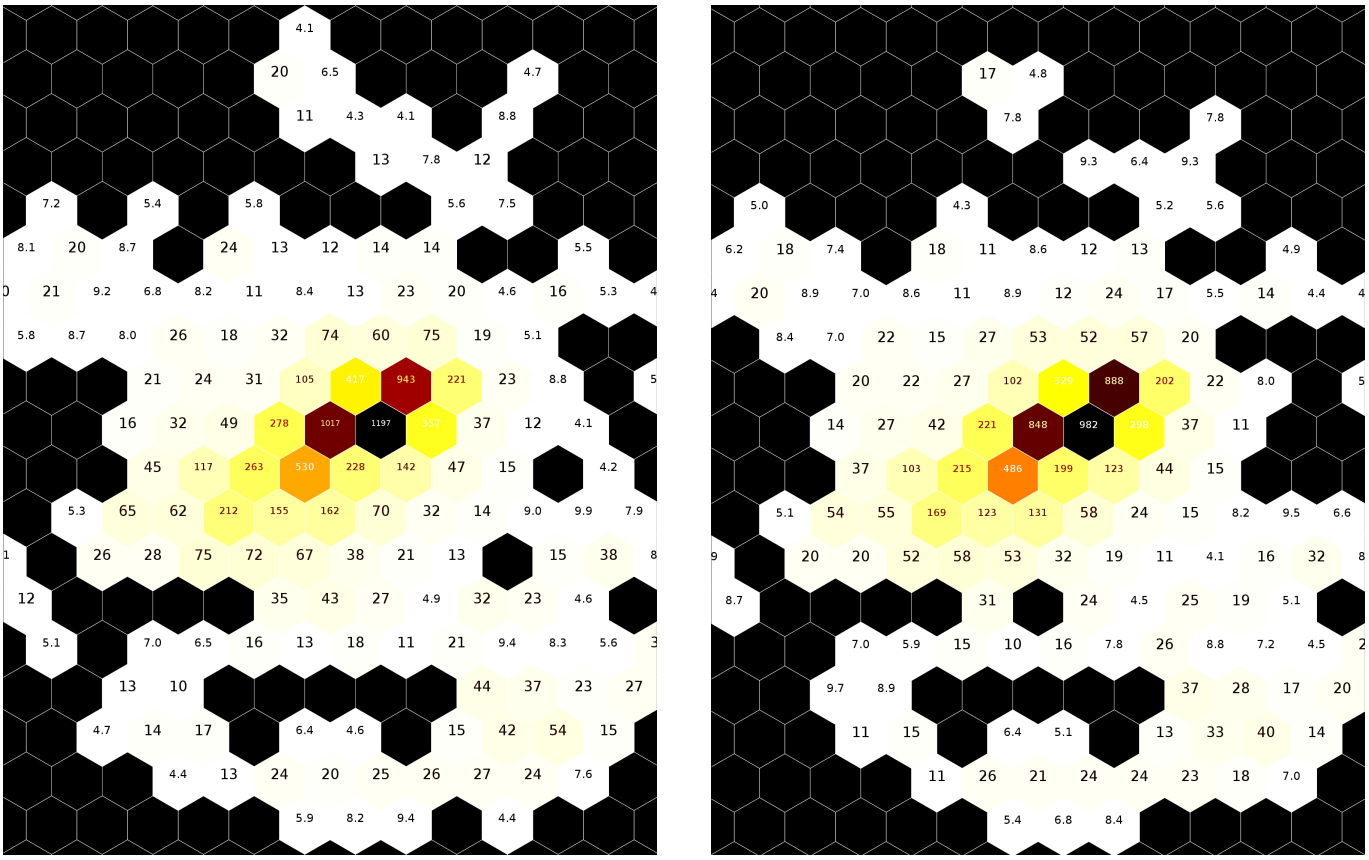


Fig. 6.19.: Shower from HESS run 48717 calibrated with MESS (left) and HESS (right).

6.8.1. Number of events

Figure 6.19 suggests that the MESS calibration results in more pixels being above the cleaning threshold, which means that small events are more likely to be kept. This is investigated, and table 6.20 shows how many useful events remain after the HESS and after the MESS calibration. Here, a useful event is defined as an event in which at least 2 telescopes participate, and in which each telescope event must contain not less than 4 pixels after 4,7-cleaning.
It can be seen that with MESS, a few more percent of the data are kept, but it is not clear whether the MESS calibration is more sensitive or simply has a bias towards higher intensities. Although manual inspection shows that the retained pixels are all clustered and seem to belong to showers, it is more reliable to look at the significance of a source. It would also be interesting to see how the HESS calibration performs on Monte Carlo events.

6.9. Interpolation

Broken pixels (pixels with undefined intensities) can be interpolated. Per default, all pixels that have at least one neighbor will be assigned the mean of their neighbors. If desired, multipass-interpolation can be chosen, which interpolates until there are no more undefined intensities. This can be useful if there are large areas with undefined intensities, such as broken drawers.

calibration	HESS	MESS	change HESS → MESS [%]
number of events	222790	230894	3.6
number of telescope events	647471	668511	3.2
number of pixels	11637850	13484524	15.9
number of pixels per telescope event	17.97	20.17	12.2
multiplicity 2	94715	99665	5.2
multiplicity 3	54259	55735	2.7
multiplicity 4	73816	75494	2.3
participation CT1	156156	160455	2.8
participation CT2	154368	158583	2.7
participation CT3	160900	167611	4.2
participation CT4	176047	181862	3.3

Fig. 6.20.: Statistics of useful events after calibration of HESS run 48717.

6.10. Normalization with Broken Pixel Parameters of Monte Carlo Simulations

Depending on the azimuth angle, some parts of the camera will record more light than other parts. This leads to an anisotropic distribution of the broken pixel parameters in the camera (see figure 6.21), which can result in some pixels being incorrectly flagged as broken pixels.

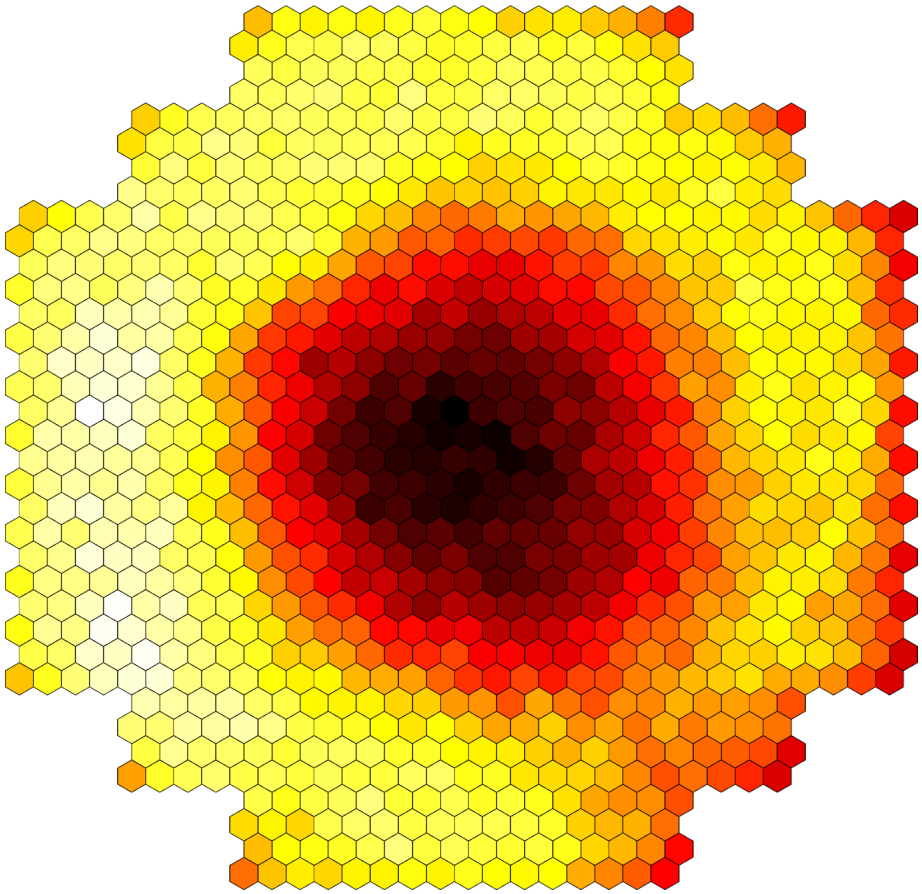


Fig. 6.21.: Anisotropic distribution of broken pixel parameters depending on zenith and azimuth angle. The events are from a Monte Carlo run, which by definition never has any broken pixels, so the distribution should be flat.

Therefore, the broken pixel parameters of for a certain observation run are normalized by the the broken pixel parameters calculated for Monte Carlo data with the same zenith and azimuth angles. Afterwards, any gradients of a broken pixel parameter in the camera is removed by the normalization. This is similar to the acceptance correction

for reconstructed event positions.

6.11. Applying Calibration Parameters to Monte Carlo Data

The extracted calibration parameters of a run can be applied to Monte Carlo events, which can be useful for optimizing the training of γ/hadron separation machine learning algorithms for a specific run.
The one-time preparatory step for this scheme is to create γ-ray and proton events for different zenith angles, azimuth angles, and NSB levels:

zenith angles	0, 20, 30, 35, 40, 45, 50, 55, 60, 65, 70	degrees
azimuth angles	0,30,60,90,120,150,180,210,240,270,300,330,360	degrees
NSB levels	100, 200, 300, 400, 500, 600, 700	MHz

This is straightforward for broken pixel patterns: the Monte Carlo data with the same zenith angle, azimuth angle and NSB level as the data is loaded and the broken pixel pattern applied.
Before the training, the occurring zenith angles and NSB levels (which are estimated by the online pedestal algorithm, see parameter s) are collected and the appropriate MC data sets selected. Then, the broken pixel pattern of the run is applied and a machine learning algorithm is trained, so the algorithm has to learn the differences between γ-ray- and hadron-induced air shower images, under almost realistic conditions. Each trained model is saved so that whenever during the analysis of the real data, the zenith angle or the NSB level change, the best suited model can be loaded and used. This scheme is only an approximation, but it is robust, simple and general.
The change of mirror reflectivities (called *phases* in HESS) has been taken into account in the infrastructure. With the same pipelines run-wise Monte Carlos can be created for HESS II or CTA.

Sampling from data A different scheme is to create the Monte Carlos without electronic noise and NSB and for each data run to analyze, determine the noise distribution for each pixel and gain channel, and apply it to the Monte Carlo events by sampling from it. Some tests were done, and it is possible that this scheme leads to an improvement of the reconstruction quality, but since *sim_telarray* would have to be configured to not add electronic noise, the signal from the data would have to be removed reliably, and sampling random numbers from a histogram costs too much time, it is not implemented.

Run-wise Monte Carlos Another improvement would be to create new air shower simulations for each run and take the atmospheric and environmental conditions into account. This requires $\approx 10^5 - 10^6$ air shower simulations per run, which is a significant effort, because rerunning the cosmic ray simulator (*CORSIKA*) and the telescope array simulation (*sim_telarray*) costs much time.

6.12. HESS Calibration

The presented methods have been implemented as MESS modules. The bash script in listing 6.4 shows how they can be used to calibrate a HESS run.

1. the 5 pedestal runs that are closest to the run that is to be calibrated are found
2. from these pedestal runs ADC value histograms are created for each pixel and each gain channel
3. the ADC-to-PE ratio is calculated from the histograms
4. the HESS raw data is converted to the mes format
5. the HESS auxiliary data (pointing correction, run header, tracking position, ...) is converted to the mes format
6. raw and auxiliary data are merged into one file
7. the pedestal is calculated and the broken pixels are determined
8. the calibration is applied to the raw data and the intensities are written to a mes file

```
#!/bin/bash

export MESS=$(dirname `readlink -e "$0"`)
hess=$1   # directory with HESS data, mc and lookups
run=$2
phase=phase1

if [ ${run:0:1} == "0" ]; then run=${run:1}; fi # if there is a '0' at the beginning, remove it
if [ ${#run} == 5 ]; then
    runA="0$((${run:0:1}-1))"
    runB="0$((${run:0:1}+1))"
elif [ ${#run} == 6 ]; then
    runA="$((${run:0:2}-1))"
    runB="$((${run:0:2}+1))"
fi
run0=`printf %06d $run`

log=${run0}.log
if [ -z "$2" ] || [ "$2" == "verbose" ]; then verbose=""; else verbose="-intro -outro"; fi

echo "Reading and calibrating HESS run ${run}. Logfile: ${log}
Logfile for HESS run ${run0}.
Time of execution of script: `date`
User: `whoami`@`hostname`
" > $log

n=2
echo "Step 1: Finding the (max.) $n closest pedestal runs" | tee -a $log
rm -f __tmp_pedestal_runs.txt

for i in `ls /lfs/l2/hess/hddata/run${run0:0:2}*/run${run0:0:2}*/*_ADCtoPe_*.root \
    /lfs/l2/hess/hddata/run${runA:0:2}*/run${runA}*/*_ADCtoPe_*.root \
    /lfs/l2/hess/hddata/run${runB:0:2}*/run${runB}*/*_ADCtoPe_*.root`; do
    f=`find $(dirname $i) -name "*_Camera_*.root"`
    if [ -z "$f" ]; then
        continue
    fi
    f=`basename $f`
    f=${f#*run_}
    if [ "${f:0:1}" == "0" ]; then
        f=${f:1} # if there is a '0' at the beginning, remove it
    fi
    f=${f%_Camera*}
    d=`echo "sqrt(($f-$run)*($f-$run))" | bc`
    echo "$d $f" >> __tmp_pedestal_runs.txt
done

sort -n __tmp_pedestal_runs.txt | head -n $n | cut -d' ' -f2 > __tmp_pedestal_runs_closest.txt
cat __tmp_pedestal_runs_closest.txt | tee -a $log
cat __tmp_pedestal_runs_closest.txt

echo "Step 2: Creating per-pixel ADC value histograms from pedestal runs" | tee -a $log
i=0
while IFS='' read -r pedrun || [[ -n "$pedrun" ]]; do
    echo "run: $pedrun"
    $MESS/mess -micropipe \
        readhess.r: -in $pedrun ,
        integratesamples.adc:r ,
        interval.int:adc -last_only -set_tag 1 ,
        calibrate.cal:int -pedestal_hist -filter_tag 1 ,
        write.w:cal -merge -out ${run0}_pedestal_${i}.mes &>> $log || { echo 'ERROR' ; exit 1; }
    i=$((i+1))
done < __tmp_pedestal_runs_closest.txt
```

```
echo "Step 3: Calculating the ADC-to-p.e. ratio from the best runs" | tee -a $log
$MESS/adc2pe . ${run0}_adc2pe.mes ${run0}_pedestal_*.mes &>> $log || { echo 'ERROR' ; exit 1; }

echo "Step 4: Converting HESS raw data of run $run"
$MESS/mess $verbose -micropipe \
    readhess.r: -in $run ,
    integratesamples.adc:r ,
    select.sel:adc -remove_raw ,
    write.w:sel -merge -out tmp_${run0}_raw.mes &>> $log || { echo 'ERROR' ; exit 1; }

echo "Step 5: Converting HESS auxiliary data (pointing position and correction, run header, ...) of run $run" |
     tee -a $log
$MESS/mess $verbose -micropipe \
    readhess.dst: -in /lfs/l2/hess/dstdata/run*/run_${run0}*.root ,
    select.run_pc:dst -idx dst.1 dst.2 dst.4 ,
    write.w:run_pc -merge -out tmp_${run0}_meta.mes &>> $log || { echo 'ERROR' ; exit 1; }

echo "Step 6: Merging all data into one file" | tee -a $log
$MESS/mess $verbose -progress 10000 -micropipe \
    sync.s: ,
    read.rdata:s -in tmp_${run0}_raw.mes ,
    read.rmeta:s -in tmp_${run0}_meta.mes -slave ,
    write.w:rdata,rmeta -merge -out ${run0}_raw.mes  &>> $log || { echo 'ERROR' ; exit 1; }

echo "Step 7: Calculating calibration coefficients" | tee -a $log
$MESS/mess -intro -outro -micropipe read.r0: -in ${run0}_raw.mes ,
    read.r1: -in ${run0}_adc2pe.mes -slave ,
    read.r2: -in $hess/$phase/lookups/${phase}_brokenpixel_parameters.mes -slave ,
    interval.int:r0 -set_tag 1 -last_only ,
    calibrate.cal:int,r1,r2 -get -filter_tag 1 ,
    write.w:cal -merge -out ${run0}_calibration.mes  &>> $log || { echo 'ERROR' ; exit 1; }

echo "Step 8: Calibrating data" | tee -a $log
$MESS/mess -micropipe sync.s: ,
 read.rcal:s -in ${run0}_calibration.mes -slave ,
 read.r:s -in ${run0}_raw.mes ,
 calibrate.cal:r,rcal -apply ,
 select.rsel:r -idx r.2 r.3 ,
 write.w:cal,rsel -merge -out ${run0}.mes  &>> $log || { echo 'ERROR' ; exit 1; }

mv ${run0}.mes ${run0}_calibration.mes ${run0}_raw.mes ${run0}_adc2pe.mes $hess/$phase/data/

#clean up
rm -f tmp_${run0}_raw.mes tmp_${run0}_meta.mes
i=0
while IFS='' read -r pedrun || [[ -n "$pedrun" ]]; do
    echo "run: $pedrun"
    rm -f ${run0}_pedestal_${i}.mes
    i=$((i+1))
done < __tmp_pedestal_runs_closest.txt
rm -f __tmp_pedestal_runs*
```

Listing 6.4: A bash script that involves several MESS pipelines in order to calibrate HESS data.

6.13. Summary

A baseline estimation algorithm has been developed that could be of interest for the calibration of IACT and other experiments. It is able to find the baseline for real data and even for complicated artificial data. A broken pixel identification scheme with Monte Carlo normalization has also been developed and successfully tested. HESS raw data was successfully calibrated.

7. Analysis

The standard procedure for a simple HESS analysis of a single run is as follows:

1. image cleaning: all pixels that do not belong to the shower are removed
2. event selection: all shower images are removed that have less than 3 pixels or whose center of gravity is too far away from the center of the camera
3. Hillas parameters: the Hillas parameters of all shower images are calculated
4. background reduction: the Hillas parameters are used to distinguish proton induced imaged from γ-ray induced images
5. direction reconstruction: the intersections of the orientations of all pairs of shower images are calculated, weighted with the sine of the angle between them
6. the direction of the shower is determined by calculating the center of gravity of all intersections
7. all reconstructed directions are filled into a 2-dimensional histogram, which is then divided by the acceptance of the camera. The acceptance is a 2-dimensional map that depends on the camera (geometry and size) and the zenith and azimuth angle of the current run. It contains the probability of an event being detected, reconstructed and identified as γ-ray. At the borders, for example, the acceptance is lower, because events are truncated.
8. significance map: the ring background method (see [11]) is used to calculate the Li-Ma significance (see [6]).

This chapter shows how a basic analysis can be done with MESS. It introduces some MESS modules that are needed for the analysis and explains their features. For some modules, benchmarks were done on an `Intel(R) Core(TM) i5-6600T CPU @ 2.70GHz` (35W low power) with 32 GB RAM and a 512 GB Samsung 950 Pro SSD. In the benchmarks and examples, several different data sets are used:

- CTA MC prod 3 proton raw data for the waveform analysis
- HESS MC calibrated mono events of CT5 only, which were originally uncleaned full-camera events. For each event the ROI (region of interest) was calculated and all other pixels removed. Then, the events were split into different files, according to their ROI size. This way, the effect of event size on the algorithms for image cleaning, Hillas parameter calculation and γ/hadron separation can be investigated.
- Real HESS data is used for the final analysis.
- 10 TB of HESS I MC data to create lookups for different zenith angles, azimuth angles and NSB levels

7.1. Waveform Analysis Algorithm

In order to extract the signal, digital processing of the traces is required. Depending on the camera, this can involve smoothing or deconvolution of the waveform.
For the analysis of sampled data, a fast waveform integration algorithm was developed. It does not apply necessary deconvolutions and other camera-specific functions, but instead simply smoothes the samples s of all gain channels of a pixel with a filter approximating a Gaussian:

$$S_i = (s_i + 3s_{i+1} + 3s_{i+2} + s_{i+3})/8$$

After finding the global maximum S_m, the integrated charge for this pixel is set to S_m and the time of maximum to $m + 1.5$.
The algorithm was implemented using AVX2 (Intel Advanced Vector Instructions 2), which is supported by all modern CPUs, and can easily be extended to future vector instruction sets (like AVX512). In case of AVX2, the registers are 256 bit wide, so 16 unsigned shorts can fit into one register. The key idea of the algorithm is to load 16 samples three times: the first time with offset 0 into variable `a`, the second time with offset 1 into variable `b`, and the last time with

offset 2 into variable c. Calculating $(a_i + a_{i+1}), (a_{i+1} + a_{i+2}), ...$ can then be achieved by simply calculating $a + b$, which is done in parallel for all 16 values. By using this principle, the smoothed trace in a window of 16 samples is calculated and then the maximum is found, also in parallel. If this maximum is larger than the global maximum, it will be stored as the global maximum. If there are still samples left, the window advances 15 samples and the algorithm repeats itself. An example of a result produced by the algorithm can be seen in figure 7.1.
The algorithm is shown in code listing 7.1, for better overview with some checks omitted and for the high gain channel only. Also, the equivalent C code is not presented, which is used as fallback in case the CPU does not support AVX2.
A simple benchmark with 7.5 GB of CTA prod3 Monte Carlo data was done. In the file, there are 4.8×10^9 samples and it takes 34 s to read and decode the events and process all of them 10 times (for better statistics). So a rate of $\approx 1.4 \times 10^9$ **samples/s** is achieved, which is close to the maximum rate of what a single FlashCam can deliver. If traces are longer, edge cases appear less frequent and the rate is higher. On CHEC data (see 10.0.3) with 128 samples, for example, the rate is $\approx 3.7 \times 10^9$ **samples/s**.
These results were achieved on an old and inexpensive computer, and only a single computing core was used, so the algorithm is well suited for quick analyses on low-end hardware.

```
void integratesamples(unsigned short *data, int len, unsigned short *adcsum, float *tom)
{
  __m256i a, b, c, m, s, r;
  int p=0, adcsum_tmp;
  *adcsum = 0;   *tom = 1.5; // offset to center of filter window
  memset(data+len, 0, ((len/16+2)*16-len)*2); // pad with 0 => last samples won't be peak
  while (len > 0) {
    // loadu has latency=3 and reciprocal throughput=0.5 (page 224. Agner Fog),
    // so better use 9 cycles than 4.5+x for shuffling etc.
    a = _mm256_loadu_si256((__m256i const *) &data[p]);   // load samples 0-15
    b = _mm256_loadu_si256((__m256i const *) &data[p+1]); // load samples 1-16
    c = _mm256_loadu_si256((__m256i const *) &data[p+2]); // load samples 2-17
    // can add traces several times without overflow (they are 12 bit)
    a = _mm256_adds_epu16(a, b); // smooth 0+1,         1+2, ...
    b = _mm256_adds_epu16(b, c); // smooth 1+2,         2+3, ...
    c = _mm256_adds_epu16(a, b); // smooth 0+1+1+2,     1+2+2+3, ...
    // horizontal add
    a = _mm256_permute2x128_si256(c, c, _MM_SHUFFLE(0, 2, 0, 1));
    b = _mm256_alignr_epi8(a, c, 2);
    c = _mm256_adds_epu16(c, b); // smooth: 0+1+1+2+1+2+2+3, ..
    // find maximum among neighbours several times
    m = _mm256_max_epi16(a, _mm256_alignr_epi8(c, c, 2));
    m = _mm256_max_epi16(m, _mm256_alignr_epi8(m, m, 4));
    m = _mm256_max_epi16(m, _mm256_alignr_epi8(m, m, 6));
    m = _mm256_max_epi16(m, _mm256_alignr_epi8(m, m, 8));
    m = _mm256_max_epi16(m, _mm256_permute2x128_si256(m, m, 1));
    adcsum_tmp = _mm256_extract_epi16(m, 0); // get maximum
    if (adcsum_tmp > adcsum) { // find maximum in block of 16 samples
      adcsum = adcsum_tmp;
      r = _mm256_cmpeq_epi16(c, m); // compare with smoothed trace sets all bits to 0, except max
      *tom = p;
    }
    p += 15;  len -= len > 15 ? 15 : len;
  }
  *tom += __builtin_ctz(_mm256_movemask_epi8(r))/2; // 2 bytes per short
  *adcsum >>= 3;
}
```

Listing 7.1: `integratesamples` routine (simplified) using AVX2.

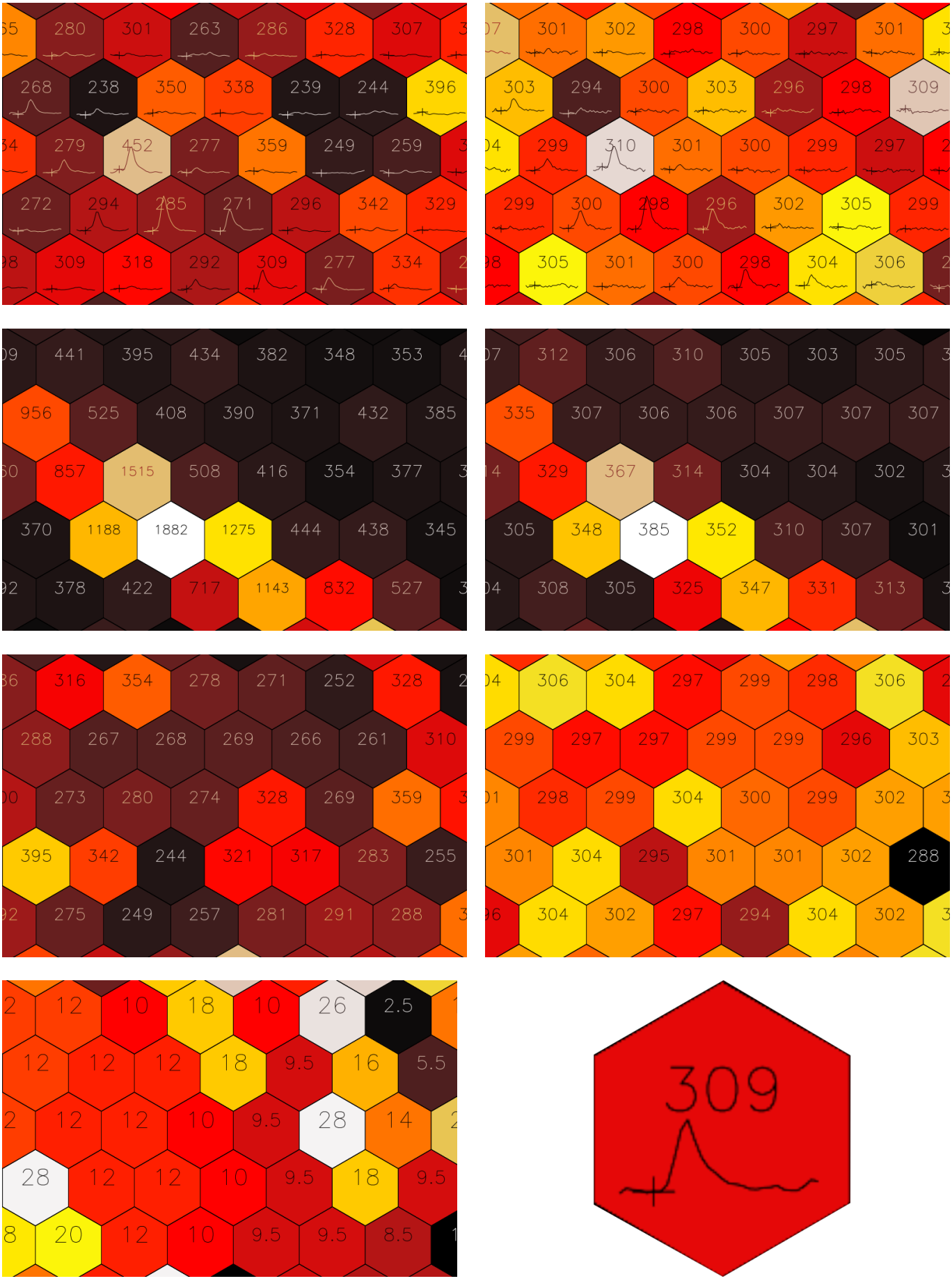


Fig. 7.1.: Example of a result produced by the `integratesamples` routine. Left: High gain channel. Right: Low gain channel. From top to bottom: Incoming sampled raw data with trace (current sample of trace marked with a + sign and value represented by color and displayed in each pixel); result of the filter that calculates a normalized ADC sum around the time of maximum of the smoothed trace; pedestal estimate.
Bottom left: Time of maximum (only stored for high gain channel).
Bottom right: Zoom of a pixel with trace.

7.2. Image Cleaning

Most image analysis and reconstruction algorithms require the shower image to be cleaned from the NSB noise around the shower.
M, N cleaning is the standard method for cleaning IACT images and has been used in IACT experiments for several decades. It requires every pixel that survives the cleaning to have an intensity of at least M photo electrons (pe). If it has less than N pe, it must have at least one neighbor with at least N pe. If it has N pe or more, it must have a neighbor with at least M pe, and that neighbor is also included in the list of surviving pixels.
The parameters M and N need to be chosen according to the NSB level and the desired application. In HESS, for an NSB level of 100 MHz, it is common to use $5, 10$ cleaning for telescopes CT1-4 and $4, 7$ cleaning for telescope CT5.
MESS contains the module *cleanmn*, which cleans shower images with M,N-cleaning. It is not especially optimized, just C code that uses the MESS data structures.

Speed of the cleaning module With a file of 494.935 CT5 full camera γ-ray shower images, its speed is benchmarked. Since reading the file from disk and decompressing the data takes a significant amount of time, each event is read once, then cleaned 100 times, and the average time is given. Because the image cleaning is done in-place, before each cleaning, the event is copied to a different memory location, and after the cleaning, the original event is restored. The memory copies are also time expensive and must be measured separately and later subtracted.
When cleaning full-camera images, the pixels can be iterated the way they are stored in memory, and a cleaning rate of **482 MegaPixels/s** is achieved, if the surviving and the cleaned pixels are kept and the surviving pixels are marked in a list. If they are also set to `undefined`, the speed drops to 236 MegaPixels/s. Setting them to `undefined` is useful for algorithms that need to examine the neighborhood very often, because doing this with a pixel list is slow. For the online analysis and on-site analysis, this is the most realistic scenario, because the cameras will provide full-camera images.
When cleaning images that are stored as pixel lists or ROIs, the speed will be lower, because the location in memory of each pixel value must be found first.
A faster function for image cleaning is provided with `cleanmn_televent_fast`, which is similar to `cleanmn_televent`, but does not provide all the options. It is used in the data reduction example later.
The *cleanmn* module offers several options, such as:

- cleaning on arbitrary pixel value types, such as intensity, time of maximum or ADC value
- defining the minimal number of neighbor pixels that have to fulfill the condition for M, N-cleaning
- inverting the shower, i.e. keep the pixels that would be removed and remove the pixels that would be kept
- marking the cleaned pixels in a mask and keeping the intensities of all pixels
- clean with cleaning thresholds M and N relative to a given pixel distribution (used in online calibration)
- safe cleaning, which reduces the parameters M and N gradually, until at least 4 pixels remain after cleaning
- extending the cleaned shower by a given number of neighboring rows and take the added pixels from the original shower (see figure 7.2)
- apply the cleaning of the image of one pixel value type (e.g. intensity) to another (e.g. samples). figure 7.2 shows an example. This can be useful in an online data reduction scenario, where a preliminary calibration, trace integration and image cleaning yield a list of shower pixels to be saved. In order to allow a more thorough calibration and trace integration when more time is available (for example at daytime or off-site) the traces of all pixels in the list need to be saved. This is done in the section *Data reduction*. It is also possible to only keep the pixels of the neighboring rows (see figure 7.2).

Probabilistic image cleaning The probability distribution of a background pixel depends on the electronic noise of the camera and the NSB. Based on this distribution, an image cleaning has been developed that calculates the probability that a given pixel has intensity p and its six neighbors have intensities $p_1 \dots p_6$. This cleaning retains clusters of faint pixels that would have been removed by M, N cleaning. It was supposed to improve γ-hadron separation, but it only had a marginal effect on the performance. Due to its slowness, it will not be used.

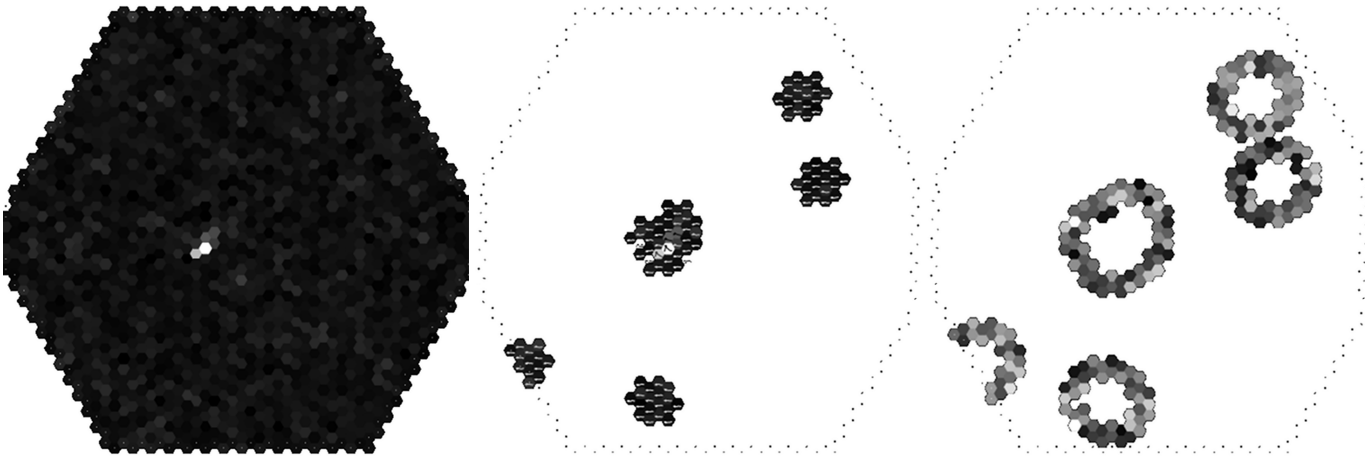


Fig. 7.2.: Example of a shower cleaned with the *cleanmn* module. Left: original shower, intensities. Center: cleaned shower, samples of remaining pixels kept. Right: two extra rows around the cleaned shower.

7.3. Hillas Analysis

The Hillas analysis[19] is an essential tool for the analysis of IACT events. It can be used for γ/hadron separation and for direction reconstruction. Although nowadays there are more powerful methods available, such as Deep Neural Networks for γ/hadron separation and template analyses for direction reconstruction, the Hillas analysis is still important, because it does not require much CPU power or a lot of RAM, and because it is robust and serves as a benchmark.
Like the image cleaning module, the *hillas* module is simple C code that uses the MESS data structures. The same input file was used for the benchmark, although here, it is already cleaned with $4, 7$-cleaning plus 2 extra rows of neighbors, and split into different event sizes.

ROI radius [m]	events	pixels	pixels/event	read	read+hillas	hillas (in-memory)
0.0-0.2	170255	6261754	36.8	0.58 s	0.64 s	0.041 s (152 MPix/s)
0.2-0.3	146374	8526027	58.3	0.54 s	0.60 s	0.047 s (181 MPix/s)
0.3-0.4	74854	6434363	85.9	0.36 s	0.41 s	0.032 s (201 MPix/s)
0.4-0.5	46057	5327570	115.7	0.27 s	0.36 s	0.024 s (221 MPix/s)
0.5-0.6	26725	4036923	151.1	0.22 s	0.28 s	0.020 s (201 MPix/s)
0.6-1.0	26434	5163429	195.3	0.22 s	0.30 s	0.022 s (234 MPix/s)

Table 7.1.: Statistics of the cleaned files (different ROI sizes of CT5 mono events) and performance of the MESS *hillas* module. The times to read and to perform the Hillas analysis are shown, and for the in-memory benchmark the speed is given in MegaPixels per second.

7.3.1. Square grid

It was tested to do the image cleaning and Hillas analysis on a square grid instead of a hexagonal grid. To convert a hexagonal grid to a square grid, every other row needs to be shifted by half a pixel (see figure 7.3). This leads to a speedup, because on a hexagonal grid, the list of neighbors of a pixel must be obtained from a lookup table, which is time-consuming, whereas on a square grid, it can directly be determined by simply adding or subtracting 1 to the current x and y position.
The image cleaning on the square grid must yield the same result as the image cleaning on the hexagonal grid, so it must be considered that the neighborhood changes every other row. For example, in figure 7.3, pixel 4 in the hexagonal grid has the two Northern neighbors 0 and 1, which become a Northwestern and a Northern neighbor in the square grid, whereas pixel 7 in the hexagonal grid has the two Northern neighbors 4 and 5, which become a Northeastern and a Northern neighbor in the square grid.
During the Hillas analysis, the (integer) grid coordinates are used instead of the (floating point) camera positions. If the pixel intensities were also stored as integers, the Hillas analysis could be done completely in integer arithmetic.

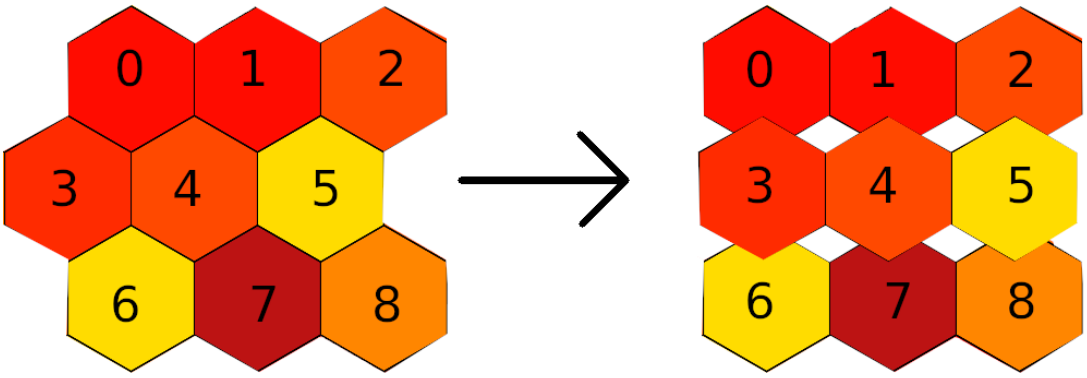


Fig. 7.3.: Converting a hexagonal grid to a square grid by simply shifting every other row half a pixel.

After calculation of the Hillas parameters, the results in x and y are simply scaled with the pixel distances in both directions. So although the event topology is messed up in the square grid, after this operation the correct Hillas parameters are obtained. This has been tested in detail on a per-event basis.
When both the image cleaning and the Hillas analysis are done on a square grid, it is **twice** as fast as on a hexagonal grid. However, for many other algorithms and tasks, not using the original pixel layout is unintuitive and leads to complications, like having to scale and taking care of integer overflows and integer divisions. Like in the image cleaning benchmark, higher pixel rates are achieved for higher ROI sizes, for the same reasons.

7.4. γ/Hadron Separation

For reducing the hadronic background, γ/hadron separation is necessary. Several new methods are presented in chapter 9.

7.5. Direction Reconstruction

For this analysis, algorithm 2 from [9] was implemented, with a minor extension: The quality of the shower (proximity to border of camera, irregularity, ...), the γ-likeness returned from the γ/hadron separation algorithm, and the severity of the broken pixel pattern for each individual shower are included in the weighting of the reconstructed directions of telescope pairs.

7.6. Monte Carlo Simulations

For calibration and machine learning lookups, Monte Carlo events are produced with *CORSIKA/sim_telarray* for zenith angles 0, 10, 20, 30, 35, 40, 45, 50, 55, 60, 65 and 70 degrees and azimuth angles 0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300 and 330 degrees. As primary particles, γ-rays and protons are simulated. The simulated telescopes are HESS I telescopes of phase 1 (before the exchange of mirrors). The minimum and maximum energy of the primary particle are set depending on the particle type and the zenith angle. The spectral index is set to 2 in CORSIKA, which makes higher energetic events more likely to occur. However, in order to keep the computation feasible, the maximum energy is set to 40 TeV.

7.7. Complete Analysis Pipeline

Before a run can be analyzed, it must be calibrated (see previous chapter). Then, a model for γ/hadron separation must be learned. Here, a Boosted Decision Tree (provided by [48]) is trained on the Hillas parameters *width*, *length* and $\log(sum)$ of the shower cleaned with 4,7-cleaning, 5,10-cleaning and 7,14-cleaning, resulting in an input vector for the BDT of length 9:

```
mess -micropipe
 read.r1: -in ~/l2/hess/mc/phase1/gamma/$zen/phase1_gamma_zen_${zen}_az_${az}_nsb_${nsb}.mes ,
 read.r2: -in ~/l2/hess/mc/phase1/proton/$zen/phase1_proton_zen_${zen}_az_${az}_nsb_${nsb}.mes ,
 filterevent.f1:r1 -borderpixel_int_max 5 , filterevent.f2:r2 -borderpixel_int_max 5 ,
 dup.d1a:f1 , dup.d1b:f1 , dup.d1c:f1 , dup.d2a:f2 , dup.d2b:f2 , dup.d2c:f2 ,
 cleanmn.c1a:d1a -m 4 -n 7 -safe , cleanmn.c1b:d1b -m 5 -n 10 -safe , cleanmn.c1c:d1c -m 7 -n 14 -safe ,
 cleanmn.c2a:d2a -m 4 -n 7 -safe , cleanmn.c2b:d2b -m 5 -n 10 -safe , cleanmn.c2c:d2c -m 7 -n 14 -safe ,
 hillas.h1a:c1a , hillas.h1b:c1b , hillas.h1c:c1c , hillas.h2a:c2a , hillas.h2b:c2b , hillas.h2c:c2c ,
 select.s1:h1a,h1b,h1c -par h1a.width h1a.length h1a.log_sum
                           h1b.width h1b.length h1b.log_sum
                           h1c.width h1c.length h1c.log_sum ,
 select.s2:h2a,h2b,h2c -par h2a.width h2a.length h2a.log_sum
                           h2b.width h2b.length h2b.log_sum
                           h2c.width h2c.length h2c.log_sum ,
 vectorize.v1:s1 , vectorize.v2:s2 ,
 fastbdttrain.bdt:v1,r1,v2,r2 -model model_zen_${zen}_az_${az}_nsb_${nsb} -ntrees 400 -depth 9
```

Telescope events that have border pixels with more than 5 p.e. are discarded, which is supposed to ensure that no truncated events are included. Safe image cleaning (explained in section 7.2) is used to ensure that at least 3 pixels remain after image cleaning. The analysis pipeline then looks like this:

```
mess -micropipe
 read.r: -in $hess/$phase/data/${run}.mes ,
 filterevent.fil:r -borderpixel_int_max 15 -n_tel -2 ,
 select.ev:r -idx r.0 ,
 select.ps_ev:r -idx r.1 ,
 select.pc:r -idx r.3 ,
 dup.d1a:fil , dup.d1b:fil , dup.d1c:fil ,
 cleanmn.c1a:d1a -m 4 -n 7 -safe , cleanmn.c1b:d1b -m 5 -n 10 -safe , cleanmn.c1c:d1c -m 7 -n 12 -safe ,
 hillas.h1a:c1a , hillas.h1b:c1b , hillas.h1c:c1c ,
 select.s1:h1a,h1b,h1c -par h1a.width h1a.length h1a.log_sum
                           h1b.width h1b.length h1b.log_sum
                           h1c.width h1c.length h1c.log_sum ,
 vectorize.v1:s1 ,
 fastbdtclassify.bdt:v1,ps_ev -model_dir $hess/$phase/lookups/gamma_hadron_separation/$name -cut $cut ,
 reco.reco:ev,h1a,pc,bdt ,
 hist.hist:reco -axis reco.nom_x:100,-2,2/reco.nom_y:100,-2,2 ,
 interval.lasthist:hist -last_only ,
 ringbg.ring:*lasthist -radius 0.12 0.4 0.7 ,
 histop.smooth:lasthist -smooth 0.15 ,
 merge.merged:lasthist,smooth,ring -type HISTSET ,
 write.wr:merged -out skymap_showergeometry_${run}.mes ,
 plothist.p:merged
```

The `fastbdtclassify` module separates the hadronic air shower images from the γ-ray air shower images. It also monitors the zenith and azimuth angle and the NSB level, and loads the best suited model.
For the end user, the procedure for obtaining sky maps from HESS raw data is:

```
hess_analyze  DATA_DIR  RUN  GAMMA_HADRON_CUT
```

7.8. Results

The MESS analysis pipeline from the previous section is applied to a point source (Crab Nebula) and an extended source (RX J1713-3946). Using standard event selection, image cleaning and γ/hadron separation, the Crab nebula is detected and the morphology of RX J1713-3946 is clearly visible. Better results can be achieved by optimizing the parameters of the `filterevent` and the `cleanmn` module, and by using one of the more sophisticated γ/hadron separation algorithms shown in chapter 9.
Because the count map is not acceptance corrected, and the sources are only shown in camera coordinates (and not in sky coordinates), it is difficult to compare the results of MESS to the results of the HESS software. However, for a visual comparison both results are shown in figures 7.4 and 7.5. Note that since analyzing a run of RX J1713-3946

with the HESS software was not possible out-of-the-box, an image by the HESS collaboration shown at the ICRC 2015 is used for the comparison instead. It shows an excess map that was obtained by analyzing many HESS runs until 2015 (> 150 hours). So clearly that map is much more detailed than the map the MESS analysis yielded (28 min of data).

7.9. Summary

This chapter demonstrated that HESS data can be analyzed with MESS. The data structures and the module system are well suited for data processing in γ-ray astronomy. Several new and fast algorithms have been developed that allow going from raw data to sky maps with minimal amount of code and dependencies. New algorithms can be developed quickly and without caring about technical problems such as reading the data.
Although the analysis shown here is very basic, it is still able to provide results comparable to the HESS standard analysis. The Crab Nebula is detected with a significance of 21 σ using the data of a single run, and the extended source RX J1713-3946 is also detected using data of a single run. CTA data can be analyzed in the same way, except that the *integratetraces* module needs to be called and that the pedestal is estimated differently. Its flexibility and speed make it a candidate for the CTA analysis pipeline, which will need to handle a heterogeneous telescope array and huge amounts of data. If the trace integration is done, one single machine should be enough to handle the image cleaning and Hillas analysis for the CTA array. γ/hadron separation with a trained model, direction reconstruction and sky map plots could be done on a different core on the same machine.

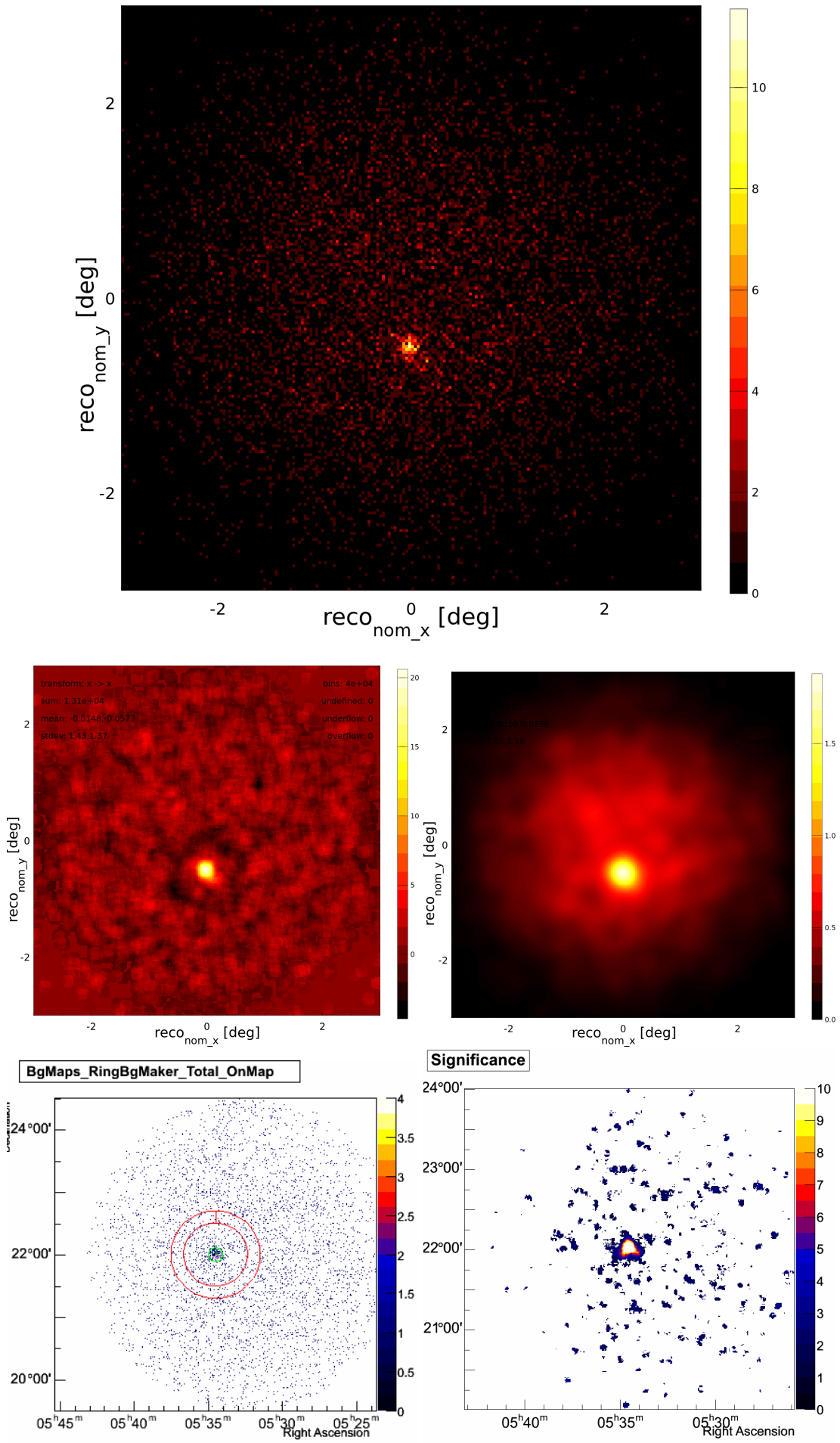


Fig. 7.4.: HESS run 48717 (Crab nebula) calibrated and analyzed with MESS. Top: Count map. Center left: significance map. Center Right: Count map smoothed with a Gaussian filter of width 0.15°. The maps are not acceptance corrected. Bottom left: HESS count map. Bottom right: HESS significance map.

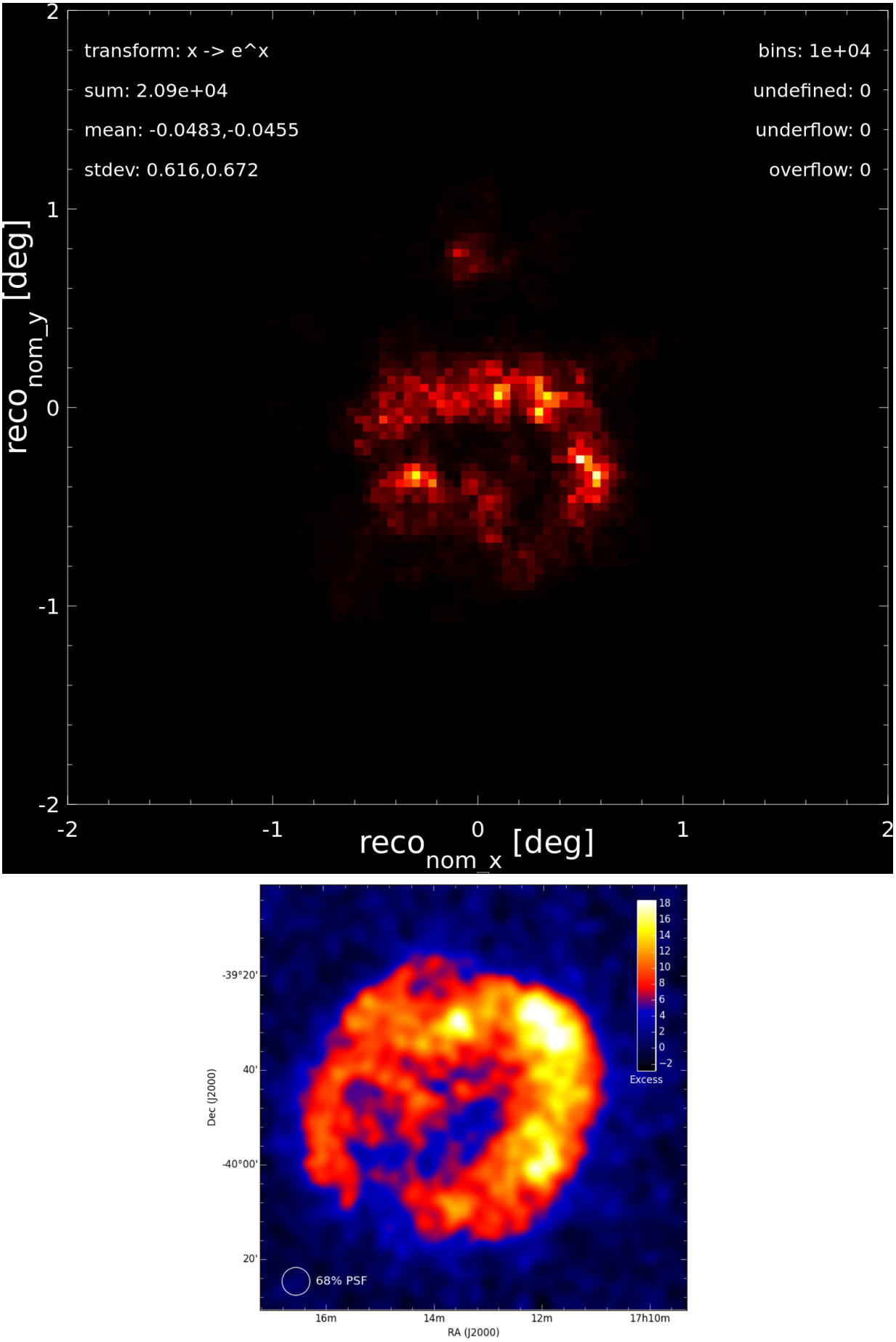


Fig. 7.5.: Top: HESS run 21164 (RX J1713-3946) calibrated and analyzed with MESS. The count map is shown in camera coordinates and is not correctly rotated (unlike the HESS sky map). It has been smoothed and transformed ($x \rightarrow e^x$) to emphasize the morphology of the source. A nearby source is visible on top.
Bottom: The same run analyzed with the HESS software, using not 28 min, but more than 150 hours of data.

8. Online Analysis and Data Reduction

8.1. Online Analysis

An *online analysis* is the preliminary analysis of the data during or a short time after data acquisition. HESS has an online analysis, and it is used to find problems during observation early and to decide whether to continue observing that source or to point the telescopes to another source. CTA is required to have an online analysis that provides an approximation of the final analysis. The online analysis needs to be done on the observation site, where it is usually too expensive to maintain a large computing cluster.
MESS provides modules that allow to run a complete analysis on the command line. In general, the normal MESS analysis is very fast and suitable for the use as online analysis. However, since the calibration parameters are not available at the beginning of a run, the calibration parameters of the last calibration period (e.g. the last run) must be used, or a very short calibration period (e.g. the last 1000 events) of the current run is used and it always lags this time behind. While both is possible with MESS, another, mixed approach for a very simple online analysis is presented here.

8.1.1. Example pipeline

In the following example, the current pedestal is estimated from the non-shower pixels of the camera.
See figure 8.1 for the command line, a graphical overview of the module I/O, and the resulting significance map. γ/hadron separation is not done here, but the module can be easily plugged in (see previous chapter).

Reading the input file A *mes* file with shower images is read with the `read` module. The incoming full-camera events are sampled and contain NSB and electronic noise.

Trace integration For each event, the `integratesamples` module calculates the ADC sums (divided by the number of samples) for each pixel and all available gain channels and stores them as pedestal images. Note: Despite of the parameter `-pixelvalue_type PIXEL_VT_PEDESTAL_HI`, both gain channels are filled, although for an online analysis it can be justified to only consider the high gain channel.

Pedestal estimation using all ADC sums From this ADC sum image, the mean and standard deviation of all pixels are calculated by the `pedestal` module, and passed to the image cleaning module.

Removing the significant shower pixels from the camera image When the *cleanmn* module, which usually only gets a shower event as input, has the mean and standard deviation of the shower pixels as an additional input, its parameters M and N are interpreted relative to the given mean and standard deviation:

$$M' = \mu + M\sigma \qquad N' = \mu + N\sigma \tag{8.1}$$

So if the mean pixel value in the camera is $\mu = 100$ and the standard deviation of all pixels is $\sigma = 20$, the `cleanmn` module called with μ, σ and M=5 and N=10 would do an M',N' cleaning with $M' = 100 + 5 \times 20 = 200$ and $N' = 100 + 10 \times 20 = 300$. Safe cleaning ensures that not all pixels are removed by reducing the cleaning thresholds by 10% until at least 3 pixels remain after cleaning. With the `-invert` option, the significant shower pixels that remain after cleaning are discarded and the removed background pixels restored.

Pedestal estimation using the background pixels only The mean and the standard deviation of the background pixels in the ADC sum image are calculated again. This time, the shower pixels are not included, which results in better pedestal estimation.

Removing the background pixels from the camera With the better pedestal estimate, the original camera image is cleaned.

Direction reconstruction and plotting of sky map The Hillas orientation of the cleaned showers is used to reconstruct the direction of the incoming particles. The direction is histogrammed, a significance map is created and plotted.

8.1.2. Results

In this test, CTA prod 3 Monte Carlo data (NectarCam and FlashCam) was analyzed on-the-fly following the above procedure. This example pipeline is only supposed to give an idea of the reduction that can be achieved, and to show that the modules are flexible and have many options, so they can be quickly adjusted to a different pipeline.
The camera image in figure 8.1 shows the result of the inverted image cleaning; the shower is cut out, so it does not bias the pedestal estimate. This is a very simple method of extracting the shower pixels, much less powerful than the methods in the calibration chapter. However, since in CTA it is foreseen that the cameras provide the pedestal, the sophisticated methods will probably not be necessary. The final online analysis will include the `filterevent` module to remove events that are too close to the border, a γ/hadron separation module, and the `ringbg` module to produce a significance map.
The pipeline is bottlenecked by the I/O speed, and its in-memory speed is not measured here. However, the following section about data reduction and the descriptions of the individual modules in the analysis chapter provide such measurements.

```
mess -micropipe
 read.r: -in gamma_20deg_180deg_run29___cta-prod3-lapalma-2147m-LaPalma.mes ,
 integratesamples.adcsums:r -method smooth8 -pixelvalue_type PIXEL_VT_INTENSITY ,
 dup.dup:*adcsums ,
 eventop.meanstdev_all:adcsums -meanstdev PIXEL_VT_INTENSITY ,
 cleanmn.cl:adcsums,meanstdev_all -m 3 -n 6 -nb_rows 2 -safe -invert -keep_cleaned ,
 eventop.meanstdev_ped:cl -meanstdev PIXEL_VT_INTENSITY ,
 cleanmn.cl2:dup,meanstdev_ped -m 4 -n 7 , hillas.hill:cl2 , reco.reco:hill,r ,
 hist.hist:reco -axis reco.nom_x:100,-3,3/reco.nom_y:100,-3,3 , plothist.plo:hist
```

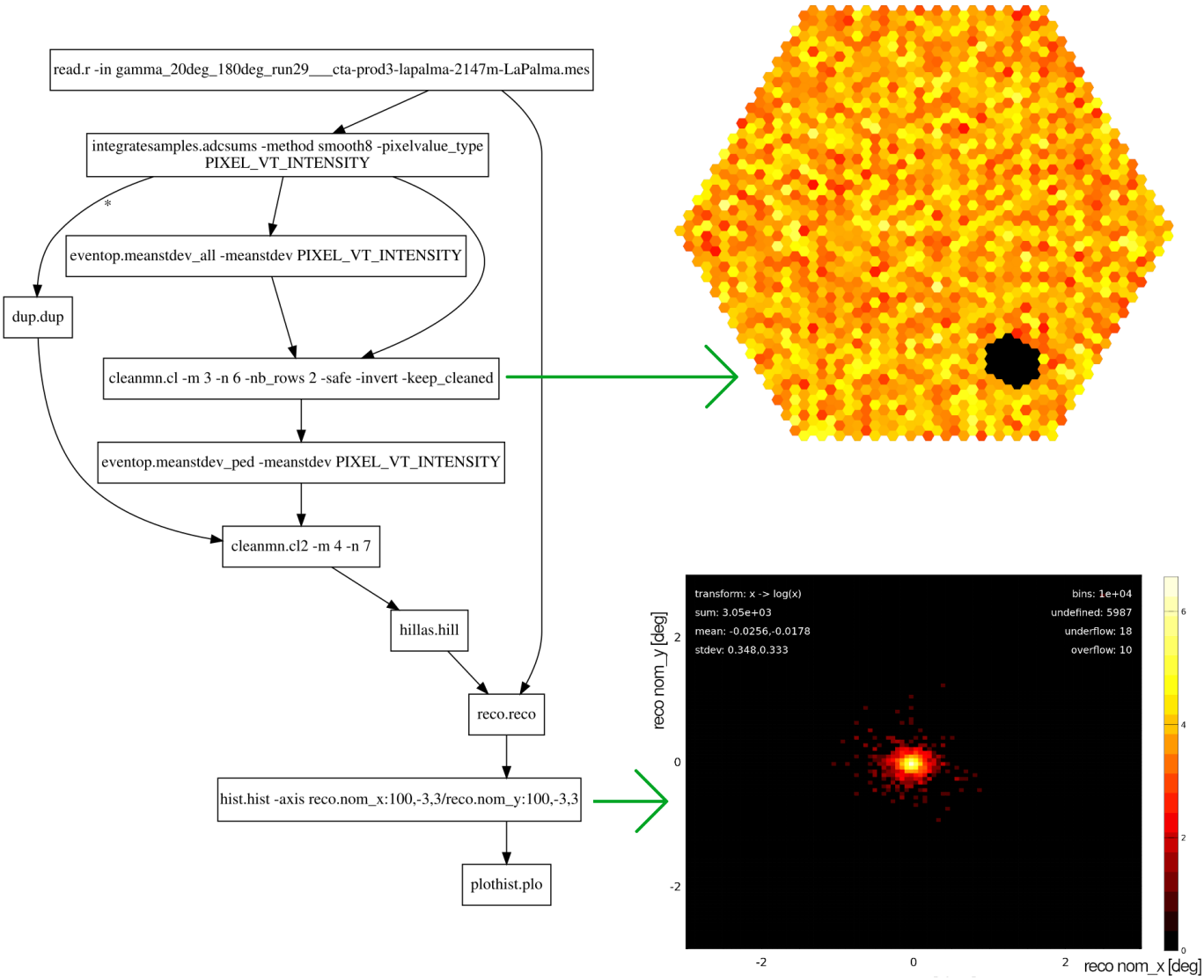


Fig. 8.1.: Top: Example of a pipeline for a simple online analysis. Left: Graphical overview of module I/O. Right: Pedestal estimation using the pixels that are not part of the shower (top), significance map (right).

8.2. Data Reduction

CTA is expected to produce a huge amount of data, somewhere between 30 and 70 GB/s. Even if the telescope arrays were located closer to a large computing facility, it would still not be easily possible to store all the data. Therefore, the data has to be reduced on site and transferred to a computing center affiliated with CTA, through the 1 Gigabit line that is specified to connect to the CTA sites.
The only way to reduce the data significantly is by zero suppression and then further compression of the remaining data. The zero suppression must only remove background from the data but no signal, so it requires a good calibration and a robust algorithm.
Here, a simple algorithm is presented that reduces the CTA data significantly. The input is a stream of raw data events, so for each pixel there are one or two gain channels with sampled ADC data. The traces of the high gain channel are then integrated around the time of maximum and the first sample of the trace is used as pedestal estimate. The pedestal is subtracted and the resulting value multiplied by the ADC-to-PE ratio. The image with pixel intensities is then cleaned and stored, along with the traces and times of maximum for these pixels. In CTA, the pedestal and the ADC-to-PE ratio will be provided by the camera, so this will not change the run time of the algorithm, only improve its quality.

This algorithm has been implemented in less than 30 lines of C using the MESS library:

```
#include "MESS.h"
int main(int argc, char **argv)
{
    system_init(argc, argv);
    if (argc != 7) die("USAGE: ./onlinecalibration WORK_DIR in.mes out.mes M N adc2pe");
    WORK_DIR = argv[1];
    telarray = config_read(WORK_DIR, TELARRAY);
    Fileinfo *fi = file_open_read(NULL, argv[2], "mes"), *fo = file_open_write(NULL, argv[3], "mes");
    double m=atof(argv[4]), n=atof(argv[5]), adc2pe=atof(argv[6]);
    Message_Record *mr = read_toc_filter(fi, "EVENT");
    if (!mr->msg_n) die("no events found in file '%s'", fi->fn);
    write_toc(fo, mr);
    Event *e = mr->msg[0]; e->h.status = DATA_VALID;
    while (!read_message_record(fi, mr)) {
        for (int i = 0; i < telarray->telescope_n; i++) {
            Televent *t = e->tel[i];    if (t->status != DATA_VALID) continue;
            integratesamples_smooth8_calibrate(t, adc2pe);
            cleanmn_televent_fast(t, m, n, 2);  // clean intensities and time of max.
            t->compression = COMPRESS_U16_FAST; // with M,N cleaning and 2 rows of NBs
        }
        write_message_record(fo, mr);
    }
    file_closedown(fi);    file_closedown(fo);
    return 0;
}
```

The program has been tested with a CTA MC prod 3 FlashCam file containing γ-ray shower images, M,N cleaning with $M = 1.5$ and $N = 2.5$, and an ADC-to-PE value of 0.017:

```
onlinecalibration . gamma.mes gamma_reduced.mes 1.5 2.5 0.017
```

8.2.1. Results

The in-memory speed (complete algorithm, but without I/O) is 570 MegaSamples/s (= **1.14 GB/s**) or **15.800** FlashCam telescope events per second.

	size (MB)	size (%)	pixels per televent	events	telescope events
original	718	100.0	1764	5426	10668
cleaned, traces + intensities	12	1.7	87	5425	10446
cleaned, intensities only	3	0.4	87	5425	10446

8.3. Summary

With two examples it was shown that the MESS software is probably suitable for the CTA online analysis and data reduction.

Online Analysis The first example demonstrated the simplicity and flexibility of MESS pipelines. Compared to the pipeline used in the analysis chapter, only few modules needed to be replaced and rearranged in order to yield an online analysis. Different ways of calibrating and analyzing the data can be implemented as modules and inserted into the pipeline.

Data Reduction In the second example, simulated CTA raw data with traces was reduced with a small program that uses MESS library functions. After removing all background pixels and keeping all shower pixels with their traces, intensities and times of maximum, the size of the data is only 1.7% of the size of the original data. When only the intensities and times of maximum are stored, the size of the data is 0.4% of the size of the original data. Assuming an estimated CTA data rate of 30 GB/s (after trigger), the data rate after reduction would be 0.12 GB/s, which could be sent with the planned 1 Gigabit line.

The data processing speed in memory was measured to be 1.1 GB/s with a single core, which means that a small computing cluster would be enough for the data reduction.

Visual inspection of the reduced data showed that almost all showers were found. Sometimes, islands with NSB fluctuations were identified as significant pixels, survived the image cleaning and were included in the reduced data set. This is expected, because the cleaning thresholds were very low in order to prevent faint showers and hadronic sub-showers from being removed.
2% of all telescope events were lost, because the image cleaning removed all pixels. This can be attributed to using a fixed ADC-to-PE ratio for all pixels and for using the first sample in the trace as pedestal estimate. In CTA it is foreseen that the cameras will provide correct calibration parameters, which will lead to correct pixel intensities, which will lead to better results after image cleaning.

9. γ/Hadron Separation

This chapter describes some methods for γ/hadron separation that have been developed in MESS.

9.1. Introduction

Hadrons are deflected by intergalactic magnetic fields, so their directional information becomes useless and they are regarded as background. Since the hadron flux is orders of magnitude higher than the gamma flux, the data taken with IACTs is dominated by hadron events, so good hadron rejection is required. Since most of the hadrons are protons and since the larger the nuclei, the easier they are to separate from gammas, this study focuses on γ/proton separation.
The physics of the shower can only be rudimentally reverse-calculated from the camera image, so the identification of the primary particle, which induced the air shower, must be performed based on the different image features. It can be exploited that proton shower images are usually wider than gamma shower images of the same length and often have several local maxima. There are several parameterizations that emphasize such differences between gammas and protons, but some parameterizations often yield the same values for gamma and proton shower images. Combining these parameterizations into a single response (whether the primary particle was a gamma ray or not) is best done with an machine learning algorithm, because the parameter space is too large and complex for finding a simple cut.
If the shower is recorded with several telescopes (stereo event), the performance of gamma/hadron separation increases dramatically, mostly because the reconstructed direction can be compared with the source position. However, this does neither work for extended sources, which are very common in gamma-ray astronomy, and that is why this work focuses on mono events (events that triggered only one telescope). Since the classification performance on mono events scales well to stereo events, the results are also relevant there.

9.2. Input data

It is not straightforward how the input for the classification algorithm should be chosen, because:

- a shower can trigger a different number of telescopes
- CT1-CT4 have 960 pixels each and CT5 has 2048 pixels. The length of the input vector must be fixed for most machine learning algorithms, so if only some telescopes triggered, it is not clear how to set the cells of the input vector that correspond to the other telescopes.
  Some parameterization algorithms allow empty cells in their input vector, but for an imaging algorithm it is not easy. Possible solutions:
  - squeeze all images into one large image and if a telescope has no data, fill its part of the image with zeros (it is not clear how the algorithm works with telescopes randomly appearing)
  - train for each multiplicity (this would already be complicated for five telescopes - because it makes a difference if telescope 1 and 2 triggered or telescope 1 and 3 - and would get much more complicated for CTA)
  - split each event into its `n` telescope events, train on them with an imaging algorithm and feed the `n` responses into a parameterized algorithm. This is the most flexible and feasible approach, although it does not give the imaging algorithm access to all the images (array view) at the same time.
- CT1-CT4 have a different energy threshold than CT5. CT5 participates in 85% of all events, while each of the others only in 30%. Half of all events are CT5 mono events.
- IACTs often have hexagonal pixels. While this is not a huge problem for parameterizations, most imaging algorithms require rectangular input images. Possible solutions:
  - shift every other row half a pixel (the image will be slightly distorted)

ROI_0	ROI_1	number of γ events	number of proton events
0.0	0.2	217994	195624
0.2	0.3	184243	137062
0.3	0.4	93942	68120
0.4	0.5	57765	45681
0.5	0.6	33426	30252
0.6	1.0	33138	37565

Table 9.1.: Intervals of ROI radii and the number of showers that fall into each bin

  - interpolate into rectangular grid (information might be smeared out)
  - upsample to higher resolution (this keeps the information, but it is not clear how the algorithm will react to the many neighboring pixels with same color)
  - adapt the algorithm to work on hexagonal grid (this is problematic, because the libraries are complex and their code is optimized for GPUs)
- the input images are very noisy due to the night sky background and the electronic noise of the cameras. While the algorithms that work on images might be able to learn to ignore that noise, for most of the parameterization algorithms this is a problem. The usual method is to clean the shower image, parameterize it and feed the parameters into a machine learning algorithm, which then learns to classify the images. Possible solution: clean the image (remove all pixels that are not significant and do not have a significant neighborhood)
- the classification performance depends on the size of the shower in the camera and of the sum of intensities of the shower, which in turn depend on the energy of the primary particle and its impact distance, i.e. the distance from the telescope to the point on earth where the primary particle would have landed, if it hadn't reacted with the atmosphere. Thus, training and testing should be done in different energy and impact distance bands - however, the problem with this approach is that the energy estimation cannot be done accurately for small showers, so the binning should be based on image features alone. Binning in sums of pixel intensities or number of signal pixels in the shower does not need such extra knowledge.

For all gamma-ray experiments, a huge number of MC air shower simulations with different observation conditions are available. Here, all HESS Phase 2b5 Monte Carlo events are used that either triggered CT5 only or CT5 and another telescope. They are calibrated using *simtelarray* and then converted to the *MES* file format. There are 1.3 million gamma and 1 million proton showers, and for each shower all camera pixels are stored. Only CT5 events are considered, and with its participation fraction of 0.83 for the gammas and 0.89 for the protons, there are ≈ 1.1 million gammas and ≈ 0.9 million protons after discarding all non-CT5 showers.
The events are *diffuse*, which means that they were simulated such that the γ-rays arrive from random positions.
Note: If γ-rays were simulated to be originated from a point source and protons isotropic, that position in the camera could be learned from the shower orientation and the center of gravity (COG), and the results would be much better. This method of reducing the background is called *direction cut*, and is especially effective in stereo analysis, where the number of telescopes is larger than 1. However, this additional knowledge is not always given, and in order to function on extended sources as well as on regions with several sources in a small neighborhood, this study is done using diffuse γ-rays.
Next, the regions of interest (ROI) around the showers are extracted and the rest of the pixels discarded:

- each image is cleaned with 3,6 cleaning (see chapter *Analysis*)
- the remaining pixels are marked in the original image and 3 extra rows of neighboring pixels are added around them
- from the COG of all these pixels, the smallest circle is calculated that includes all of them
- these pixels are extracted from the original, uncleaned image and are returned as ROI

Since larger showers exhibit more and different image features than smaller showers, it is likely that the separation algorithms will perform better if they are applied to showers of approximately the same size. Thus, the showers are split into five sets, where each set contains only showers with ROIs of certain sizes (see 9.1).
Figures 9.1 and 9.2 show examples of showers that fall in different ROI size bins.

All showers that have one or more border pixels (pixels at the border of the camera) with an intensity larger than 5 pe are excluded from the data set, because in these cases it is likely that the shower is not fully contained in the camera and thus, significant parts of it might be cut off, which could mislead the separation algorithm.
In the following, several γ/proton separation algorithms are presented. Except for the Support Vector Machines, the Deep Neural Networks and the Lorentzian fit, all algorithms have been implemented from scratch and are optimized for speed and a low memory footprint.

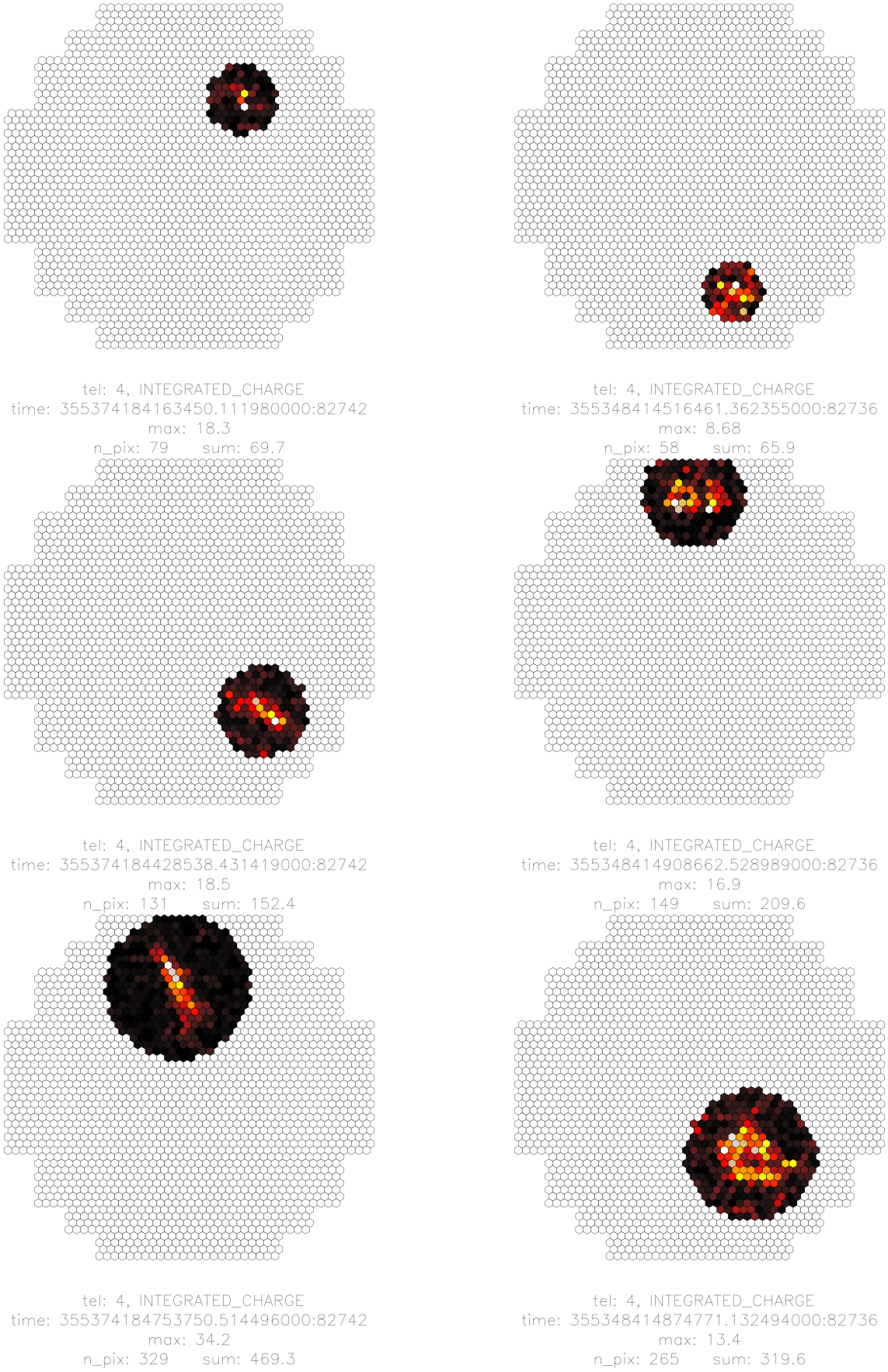


Fig. 9.1.: Examples of showers that fall into different ROI radius intervals. From top to bottom: $r = 0.0 - 0.2$ m, $r = 0.2 - 0.3$ m, $r = 0.3 - 0.4$ m. Left: Gamma. Right: Proton.

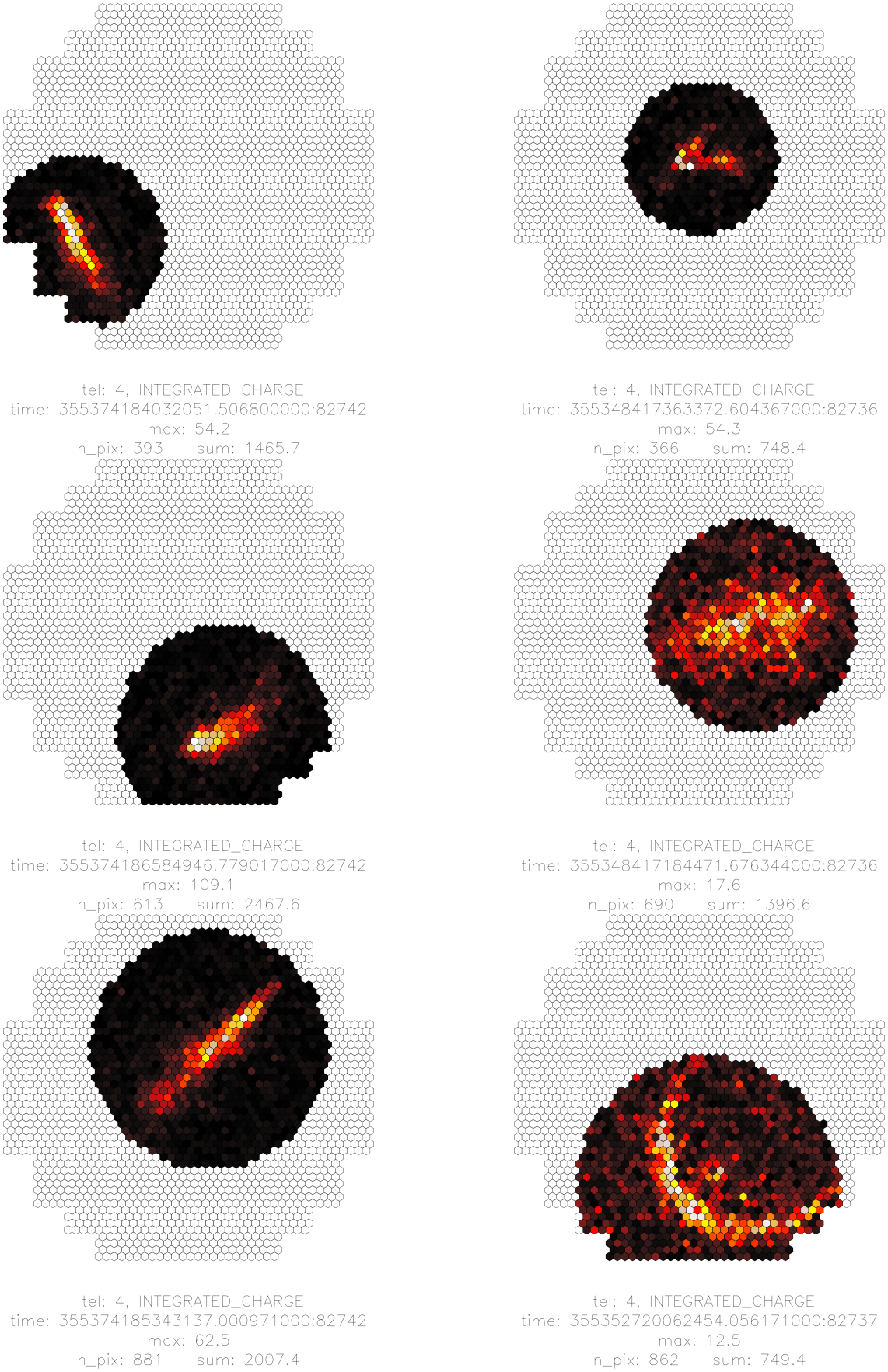


Fig. 9.2.: Examples of showers that fall into different ROI radius intervals. From top to bottom: $r = 0.4 - 0.5$ m, $r = 0.5 - 0.6$ m, $r = 0.6 - 1.0$ m. Left: Gamma. Right: Proton.

9.3. Hillas parameters

The standard method for γ-hadron separation is to calculate the Hillas parameters [19] *width*, *length*, *skewness* and *size* (also called *amplitude* or *sum*) of a cleaned shower image and then use machine learning to correlate the combination of these three parameters with the particle type. Here, *log(size)* is used instead of *size*. Also, the lengths in the camera are measured in meters instead of degrees.
Figure 9.3 shows the distributions of the Hillas parameters for gammas and for protons for a ROI size of 0.4-0.5 m. The complete set of plots for all ROI sizes can be found in the appendix.

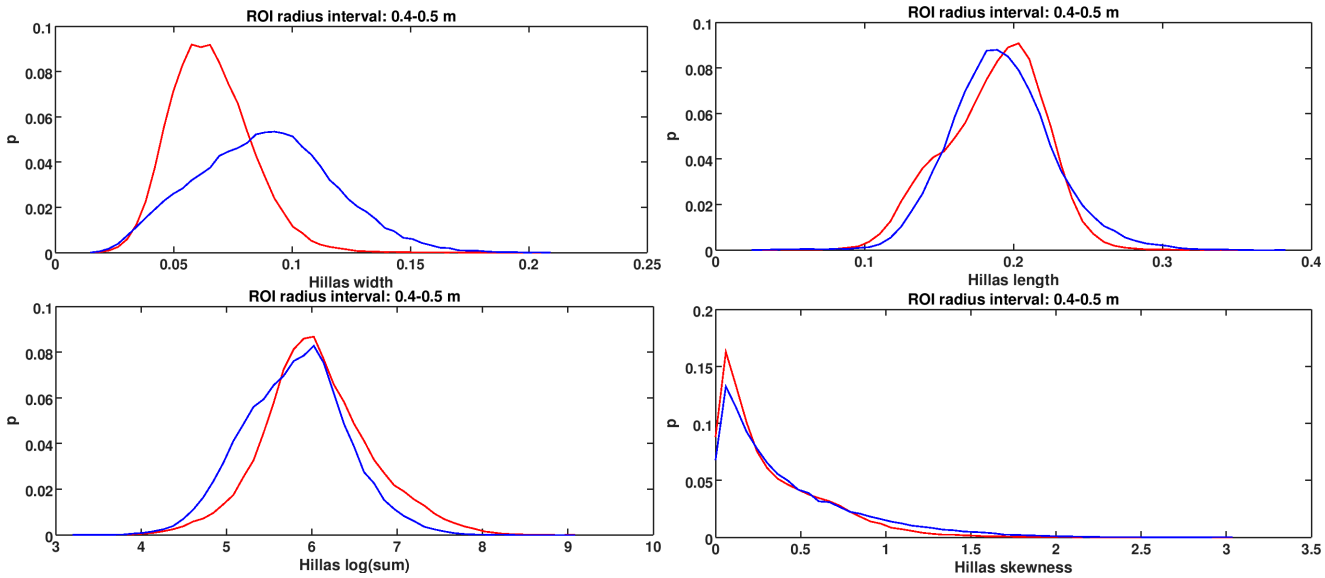


Fig. 9.3.: Hillas parameters of gamma and proton showers for a ROI size of 0.4-0.5 m. From top left to bottom right: Hillas width, length, log(sum) and skewness.

The plots show that the larger the shower, the better the discrimination performance, and that Hillas width is the best discriminating parameter. The spikes in the Hillas width and length distributions for the ROI size 0.0-0.2 m can be explained with only few possible combinations for pixel alignment of small showers. The first spike in the Hillas width distribution, for example, is located at ≈ 0.01 m, which is the variance of three pixels that form a small triangle (see figure 9.4).

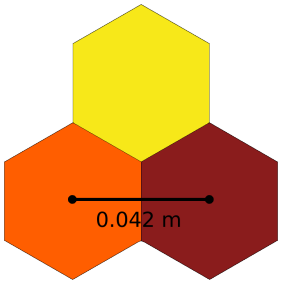


Fig. 9.4.: Example for a very small shower in the CT5 camera (pixel distance: 42 mm), Hillas width: 10.5 mm.

For this study, the reference for separation performance will be a support vector machine (SVM) trained with Hillas parameters. An SVM transforms the input data into a higher dimension so that a separating hyperplane can be constructed, which has a maximal margin to the closest data points of both input data sets. Heuristics allows training on huge high-dimensional input vectors within reasonable time, however since there is no concept of locality, they are not suited for training on raw images. Instead, the parameterized images should be fed into the SVM.
As will be seen later, this classification is better than classification based on any of the individual Hillas parameters. There are two common methods for quantifying the classification performance: the q-factor (quality factor) and the ROC (receiver operating characteristic) curve, which will be explained in the following.

9.4. Performance curves

Let p_γ and p_{proton} be normalized distributions of parameterizations of shower images. The *q-factor* gives the signal-to-noise ratio for a selected cut at x:

$$q(x) = \int_0^x p_\gamma(t)\,\mathrm{d}t \left(\int_0^x p_{\text{proton}}(t)\,\mathrm{d}t \right)^{-2} \tag{9.1}$$

The *ROC curve* gives for any fraction of signal that should be kept the fraction of background that will be kept. Let P be the desired fraction of events that will remain after a cut value a is set:

$$P(a) = \int_0^a p(x)\,\mathrm{d}x \tag{9.2}$$

With $a \in [0, 1]$, the ROC curve is then defined as:

$$r(a) = (P_\gamma(a), P_{\text{proton}}(a)) \tag{9.3}$$

The ROC and q-factor curves will be calculated for all classification methods and compared to the reference method for γ/hadron separation (SVM-Hillas).
Note 1: In HESS, training is done in energy bands (which requires estimating the energy of the primary particle), a distance cut is used to reject showers that are too close to the camera border (COG of cleaned shower must not be more than 0.525 m away from the camera center), and boosted decision trees are used as machine learning algorithm. However, since this study compares different parameterizations and classification algorithms to the Hillas based classification using an SVM, it is not required to use the same setup.
Note 2: In addition to the SVM, other machine learning algorithms (Random Forests and Neural Networks) were tried on all shower parameterizations. They did not provide any significant improvement over the SVM, so mostly only the results of the SVM are presented here.
Note 3: For all methods, only the result for the ROI size 0.5-0.6 m shown. The complete set of plots for all ROI sizes can be found in the appendix.

9.5. Benchmark

The separation power depends on the separation algorithm, the image size, the image cleaning, the type of SVM, and the regularization parameter C of the SVM. Another parameter is the desired signal-to-noise ratio (controlled by fixing the desired ratio of γ-events to be kept). For some scenarios, it is mandatory to have nearly no background events, even if the majority of the signal events is lost. For other scenarios, most of the signal events needs to be kept, even if more background events pass the cut.
Thus, for each ROI size bin, the optimal combination has to be found, and the benchmark is done as follows:

- split all events in different ROI sizes (r=0.0-0.2, r=0.2-0.3, r=0.3-0.4, r=0.4-0.5, r=0.5-0.6 and r=0.6-1.0)
- calculate the ROC curves for each M,N image cleaning, each SVM type and each SVM parameter.
  The image cleaning parameters are m, n, n_{nb} (number of significant neighbors a pixel must have) and n_{rows} (number of extra rows around cleaned shower):

$$\begin{array}{llll}
m = 2 & n = 3 & n_{nb} = 2 & n_{rows} = 0 \\
m = 1 & n = 4 & n_{nb} = 2 & n_{rows} = 0 \\
m = 2 & n = 5 & n_{nb} = 1 & n_{rows} = 0 \\
m = 3 & n = 6 & n_{nb} = 1 & n_{rows} = 0 \\
m = 4 & n = 7 & n_{nb} = 1 & n_{rows} = 0 \\
m = 4 & n = 7 & n_{nb} = 1 & n_{rows} = 1 \\
m = 5 & n = 10 & n_{nb} = 1 & n_{rows} = 1
\end{array}$$

  The SVM kernels are linear and radial basis functions. The SVM regularization parameter C goes from 10^-6 to 10^6 in steps of factor 10.

- for each fraction of signal to be kept (from 1% to 100% in 1%-steps), calculate how much background is kept for the best of all above parameter combinations

This results in a benchmark that finds for each separation algorithm and ROI sizes the best ROC or q-factor curve in the described parameter space. Then, each of the algorithms is compared to the Hillas analysis by dividing the performance curve (ROC or q-factor) of the respective algorithm by the performance curve of the Hillas-based SVM-classification.
This way of benchmarking tries to be as fair as possible to all methods by getting the best result out of each of them. If the parameters (SVM type and regularization parameter C, image cleaning and desired signal ratio) were fixed, they might be the best combination for one method, but not for the others. One could have selected the best ROC and q-factor curve among all parameter combinations, but that curve would not have been optimal for all signal efficiencies.
Figure 9.5 shows all ROC curves of the SVM-based classifier for the 0.3-0.4 m ROI size. The curve that will be chosen for the ROC curve plots in the benchmark is the best curve from the test set (black line). For the q-factor plots, the same procedure is applied.

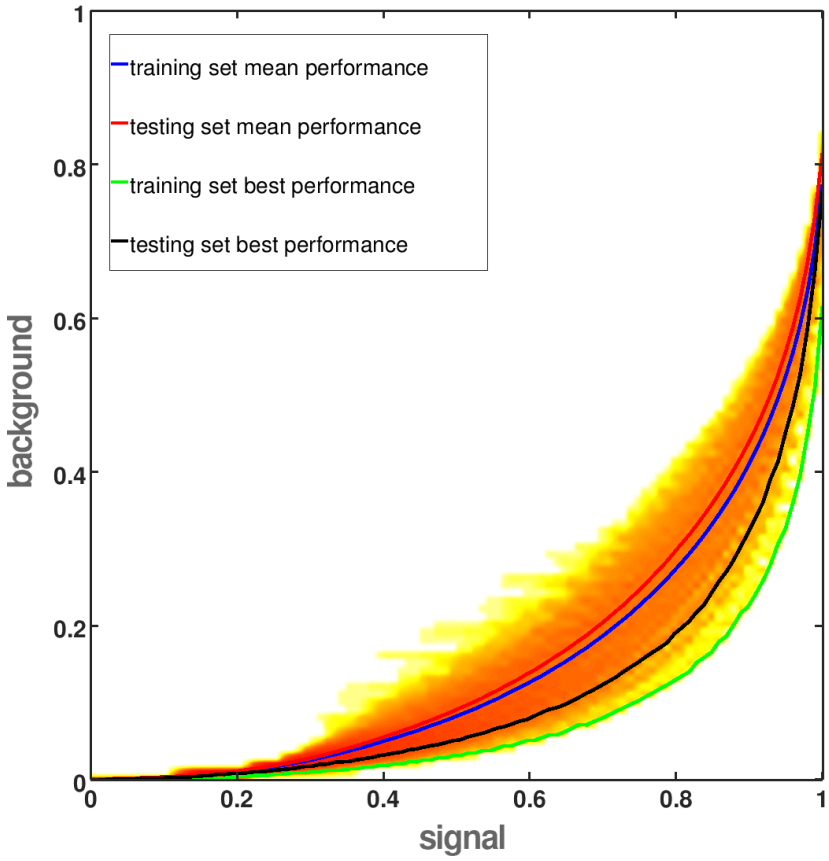


Fig. 9.5.: Several ROC curves overlaid. For each image cleaning, SVM function and regularization parameter C, a ROC curve is produced and drawn. The mean and best curves of training and test set are marked.

For BDT-based classification, the variables are:

- number of trees (50, 100, 200 or 400)
- depth of the trees (4, 6, 9)
- parameters of the algorithm, e.g. the number of bins in a histogram (see the methods described later)

9.6. Hillas analysis with different image cleanings

It is not clear which image cleaning yields the best Hillas parameters for classification. A harder cleaning, like 5-10 cleaning, removes upward fluctuations in the NSB more effectively than a softer cleaning, like 3-6 cleaning, and thus produces more accurate Hillas parameters. On the other hand, the harder cleaning often removes small sub-showers in proton showers, which are very helpful for distinguishing them from gamma showers. Feeding the parameterizations of different image cleanings into the ML algorithm gives it information about the significant part of the shower as well as information about sub-showers. In this benchmark, the Hillas parameters of differently cleaned images (3,6 and 5,10 and 7,14) are calculated and fed into an SVM.

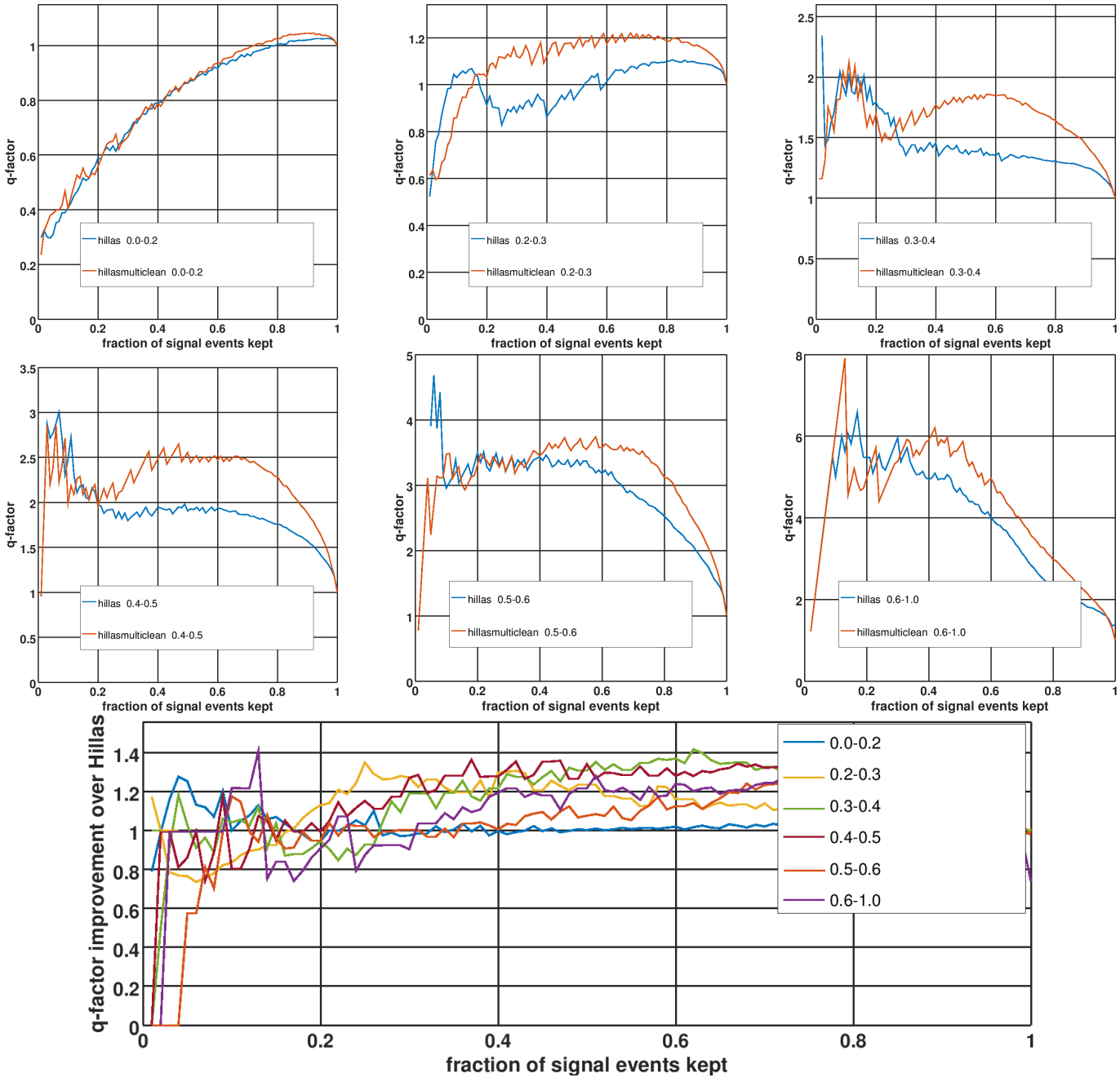


Fig. 9.6.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using Hillas parameters of 3,6 and 5,10 and 7,14 cleaned images. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

Results: Figure 9.6 shows that when more than 30% of all signal events must be kept, the q-factor is $\approx 20\%$ better for all shower sizes, except for the small ones. Figure 9.7 shows that for medium and large image sizes, the number

of background showers that pass as γ-ray showers is reduced by up to 50%.

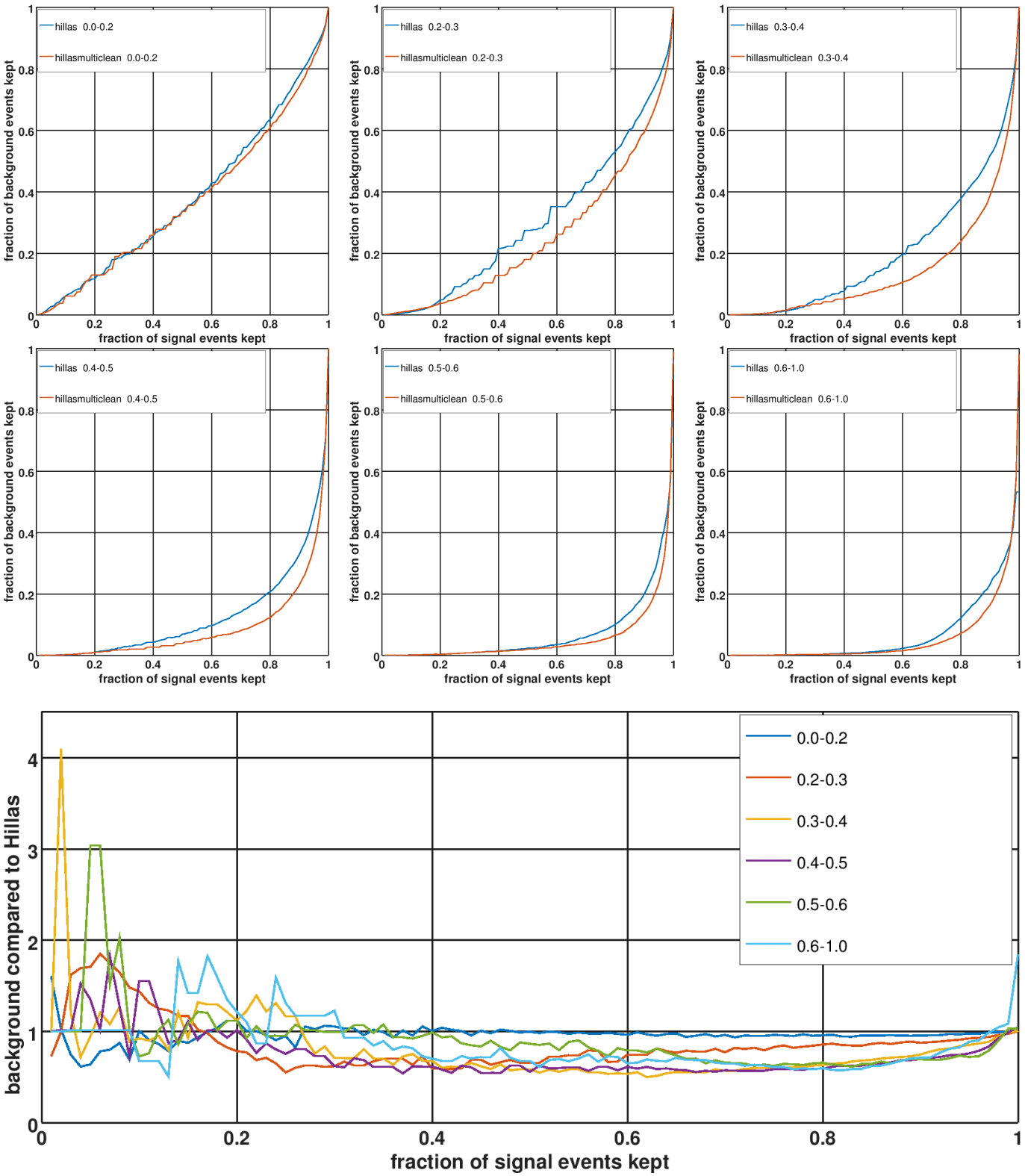


Fig. 9.7.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using Hillas parameters of 3,6 and 5,10 and 7,14 cleaned images. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.7. Projection along the Hillas axes

This method projects all pixels onto the minor Hillas axis, divides it into n bins (here $n = 16$) and provides the normalized histogram, so it can be used in a machine learning algorithm.
Motivation: The sub-showers in proton events lead to wider images, which is why the Hillas width separates so well. However, when the scattering is reduced to one number, information that could help separating might get lost: For γ-ray showers, pixels with high intensities lie close to the major Hillas axis. If some of these pixels are missing, and instead additional pixels with lower intensities are introduced farther away from the major axis, it might lead to the same Hillas width.
Example: When the intensities of a γ-ray shower image s_1 and a proton shower image $s_2$are projected onto the minor Hillas axis, their distributions of intensities are $v_1(x = -2 \dots 2) = (1, 2, 5, 2, 1)$ and $v_2(x = -2 \dots 2) = (4, 4, 3, 2, 0)$ (see figure 9.8), with standard deviations ($v_1 \approx 1.095$ and $v_1 \approx 1.092$). Although the distributions are different (one is symmetric and the other asymmetric), their standard deviations are almost the same, and it is not possible to distinguish the two showers just by looking at the Hillas width.

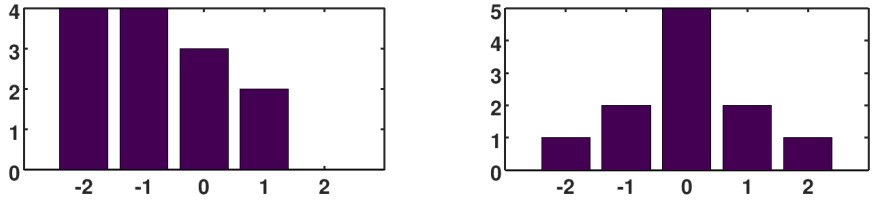


Fig. 9.8.: Example of two different distributions with almost the same variance. A distribution similar to the right distribution is expected if the intensities of a γ-like shower are projected along the major Hillas axis onto the minor Hillas axis. Asymmetric distributions like the distribution on the left are not expected.

Results: Figure 9.9 shows that for most image sizes, the q-factor gets worse. Figure 9.10 shows that for the images that lie within the 0.4-0.5 m ROI bin, this method yields slightly better results and reduces the background by 10-15%. The next section checks whether the separation power increases when the Hillas parameters are also fed to the SVM.

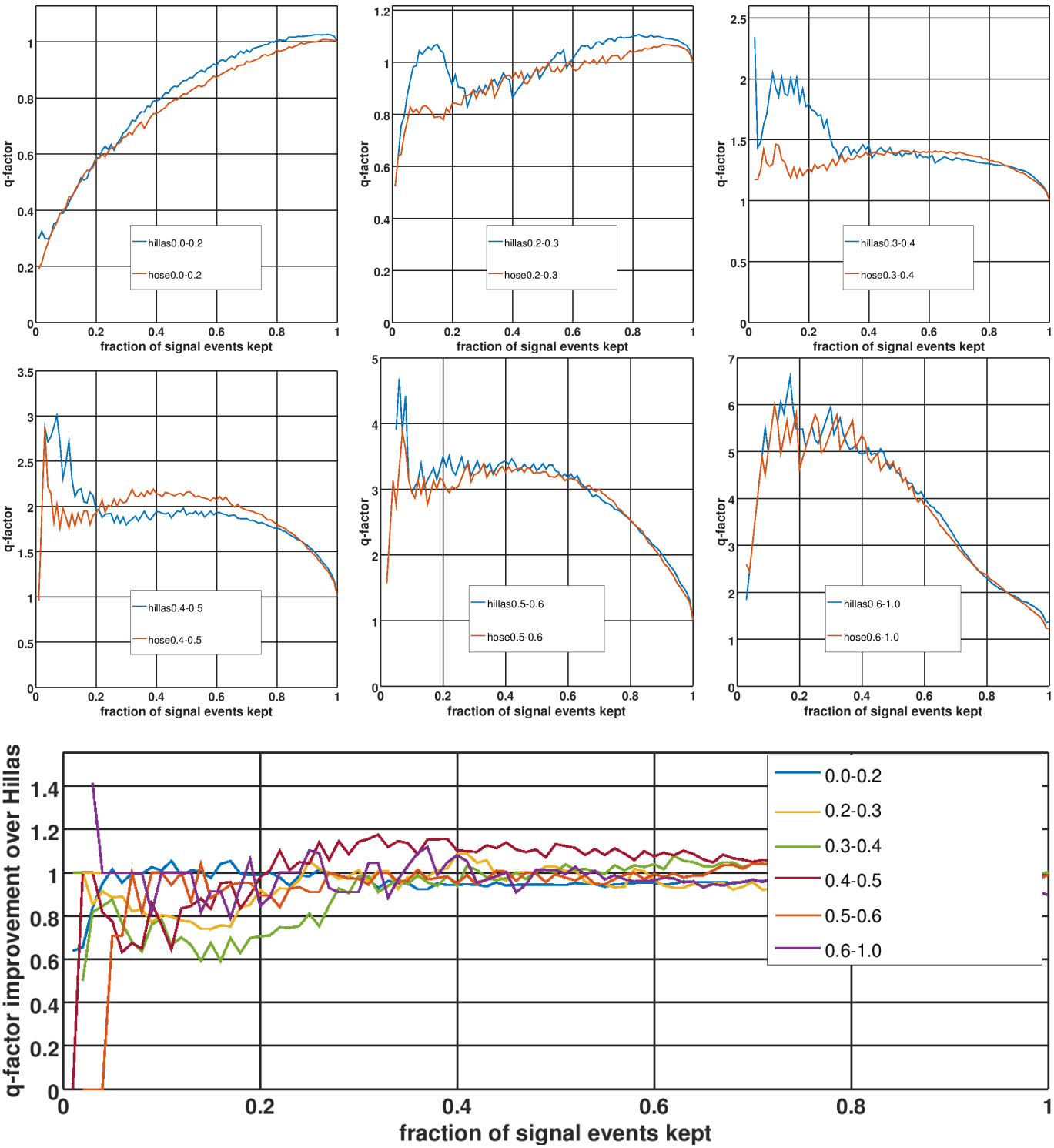


Fig. 9.9.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using Hillas parameters of 3,6 and 5,10 and 7,14 cleaned images. using the histogram of pixel values projected on the minor Hillas axis. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

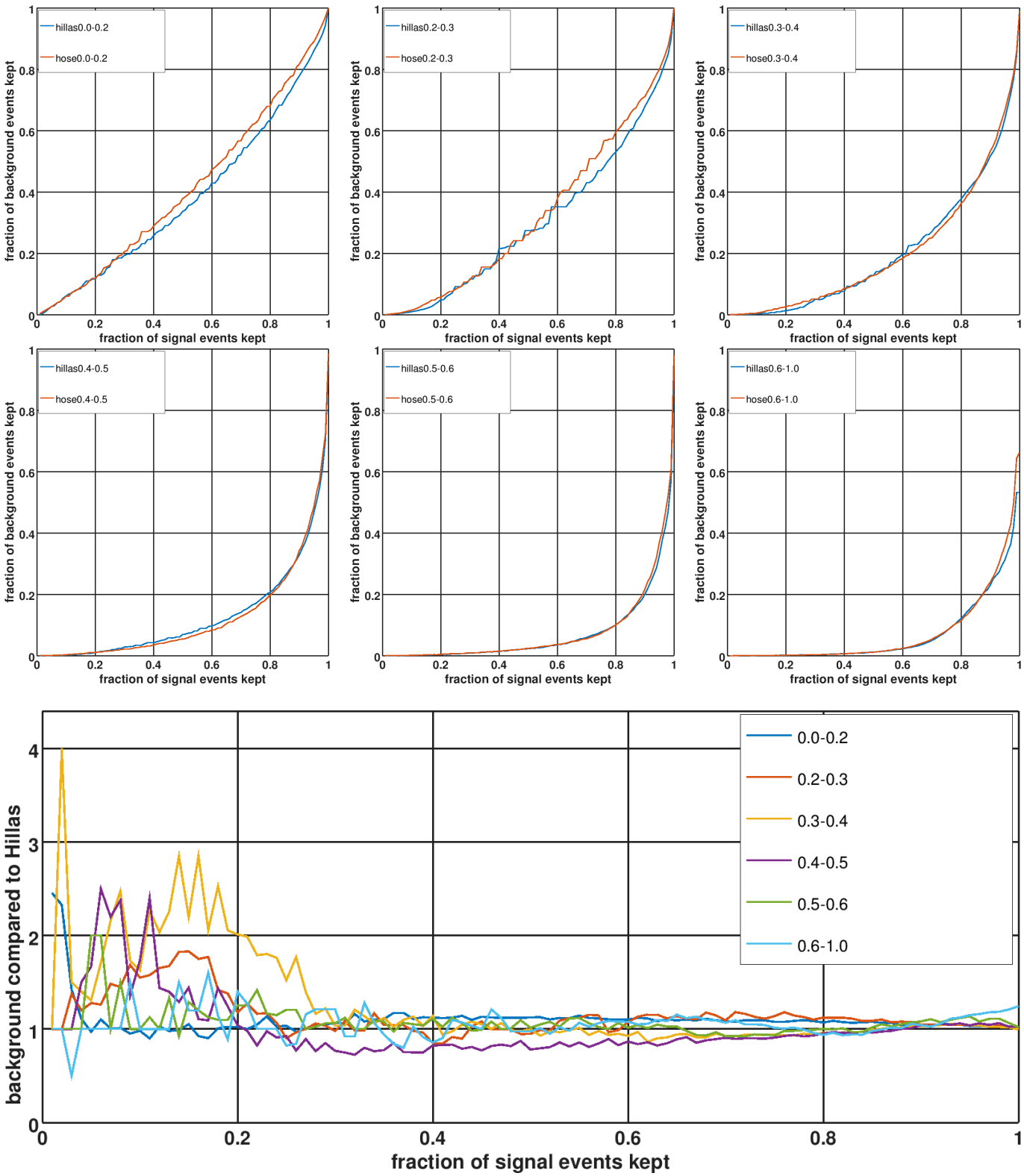


Fig. 9.10.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the histogram of pixel values projected on the minor Hillas axis. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.8. Projection along the Hillas axes plus Hillas parameters

This method is the same as the previous method, but in addition to the normalized histogram the Hillas parameters width, length and log(size) are provided as well.

Results: Figure 9.11 shows that except for very small and very large images, the q-factor increases by 5-20%. Figure 9.12 shows that for the 0.4-0.5 m bin, the background is reduced by 10%. Thus, this method is not significantly better than the pure Hillas-based classification.

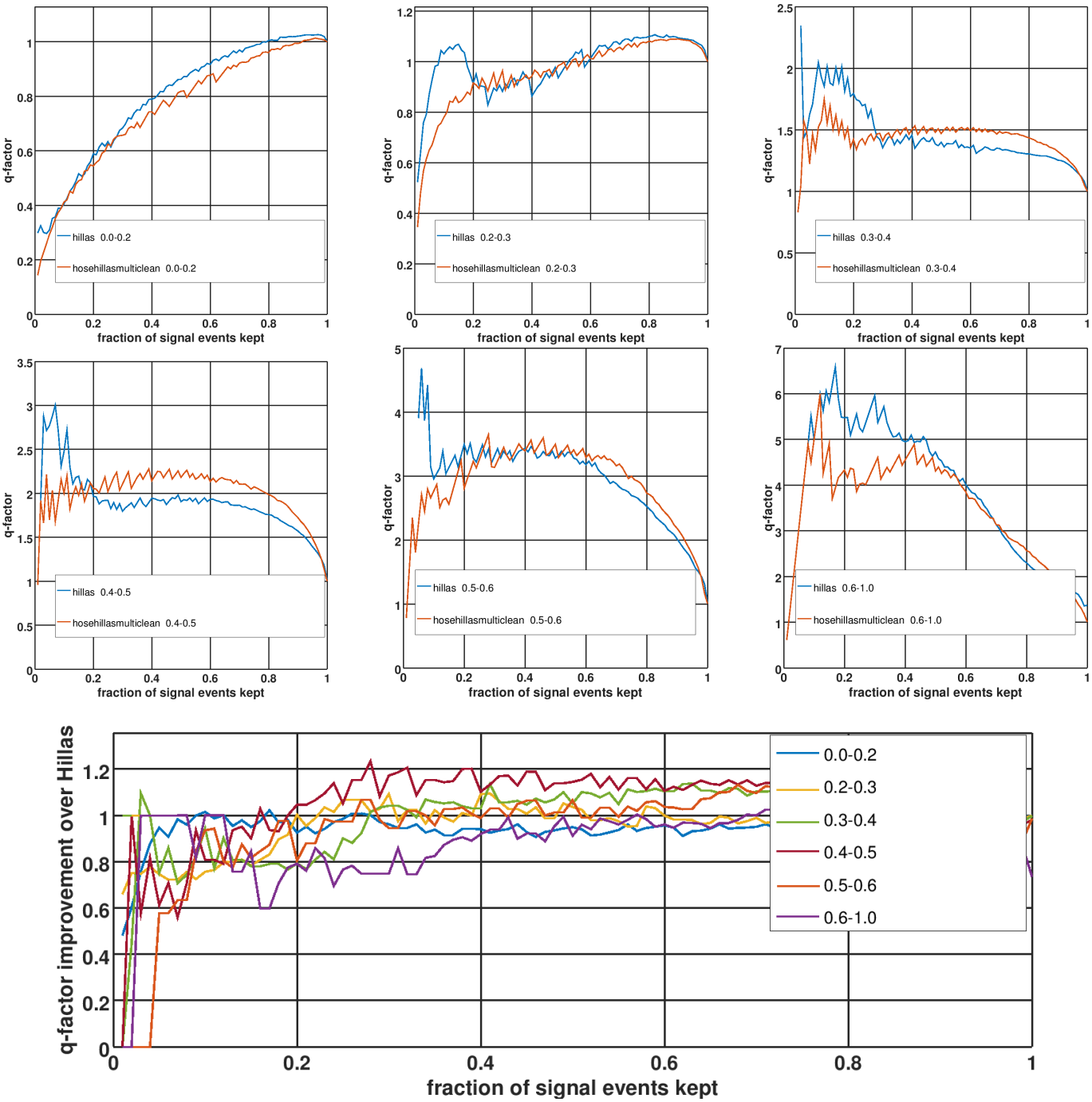


Fig. 9.11.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the histogram of pixel values projected on the minor Hillas axis plus the Hillas parameters. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

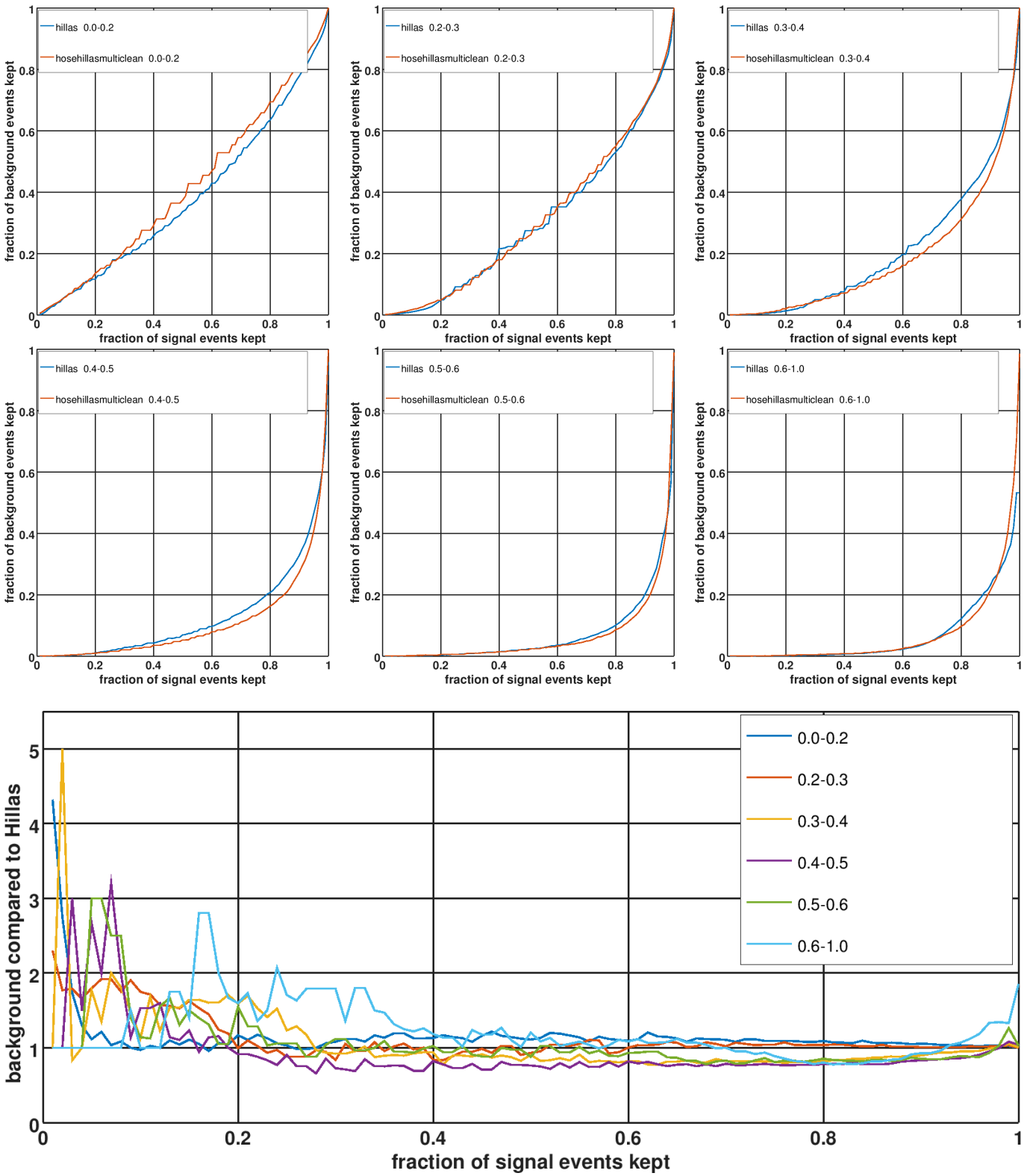


Fig. 9.12.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the histogram of pixel values projected on the minor Hillas axis plus the Hillas parameters. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.9. Lorentzian fit

It has been observed that in γ-ray shower images, the projection of pixels onto the minor Hillas axis follows a Lorentzian distribution [22]:

$$L(x) = \frac{1}{\gamma(1 + \frac{x-x_0}{\gamma}^2)} \tag{9.4}$$

where x_0 is the location and γ is the scaling factor. This can be used for γ/hadron separation. Here, the shower is rotated until the major Hillas axis is horizontal. Then, the pixel with the maximum intensity is found and all pixels that are 3 pixels or less away from it in x-direction are histogrammed (see figure 9.13). The Lorentzian distribution is fitted to the histogram and the parameters of the best fit (x_0, γ and the error) are fed to the SVM.

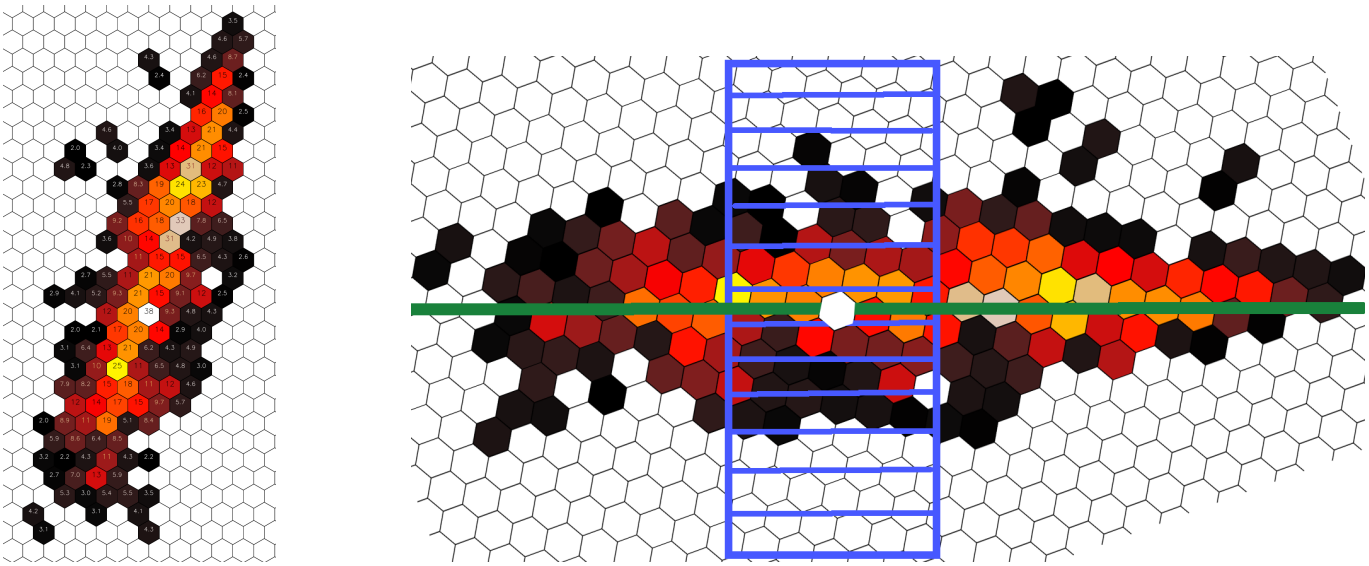


Fig. 9.13.: Schematic view of the histogramming procedure. Left: Original cleaned shower. Right: Same shower after rotation. The pixel with the maximum intensity (white) is in the center. The bins of the histogram are overlaid.

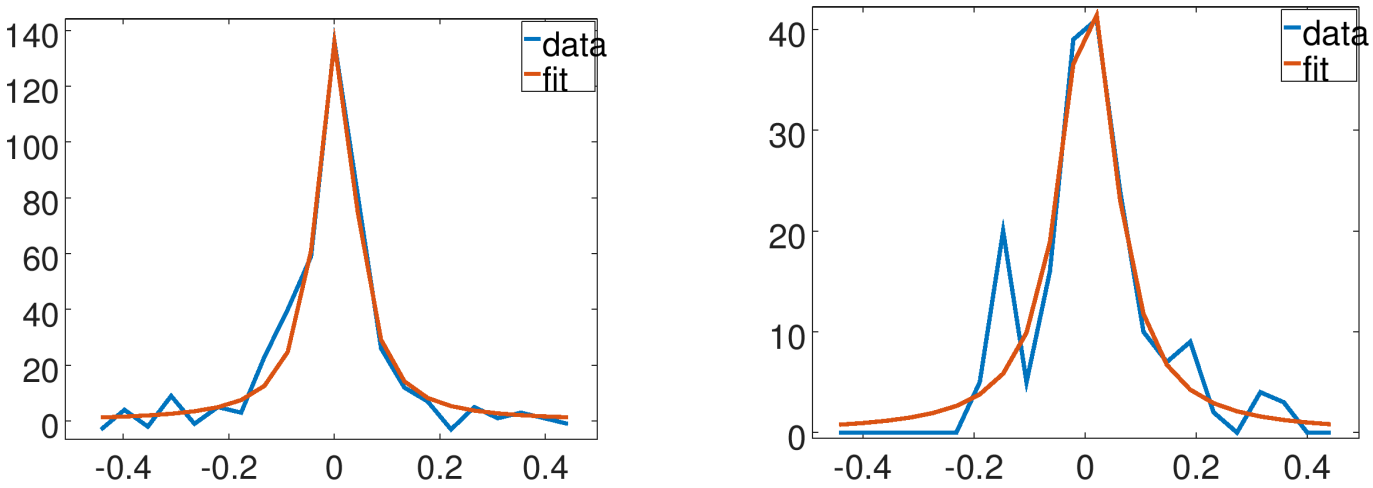


Fig. 9.14.: Fit results for two example shower images. Left: γ-ray. Right: proton.

Results: Figure 9.16 and 9.17 show that for all image sizes, this method is worse than the Hillas-based classification. The correctness of the fitting algorithm was verified and individual showers were checked (see figure 9.14). Then, different image cleanings and different ways of binning were tried. Maybe this method performs so poorly, because most sub-showers are too small to change the results of the fit significantly.

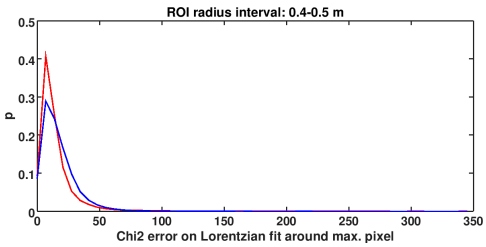


Fig. 9.15.: Error of Lorentzian fit around the maximum pixel for gamma-ray (red) and proton showers (blue). The ROI size is $0.4 - 0.5$ m, plots for other ROI sizes can be found in the appendix.

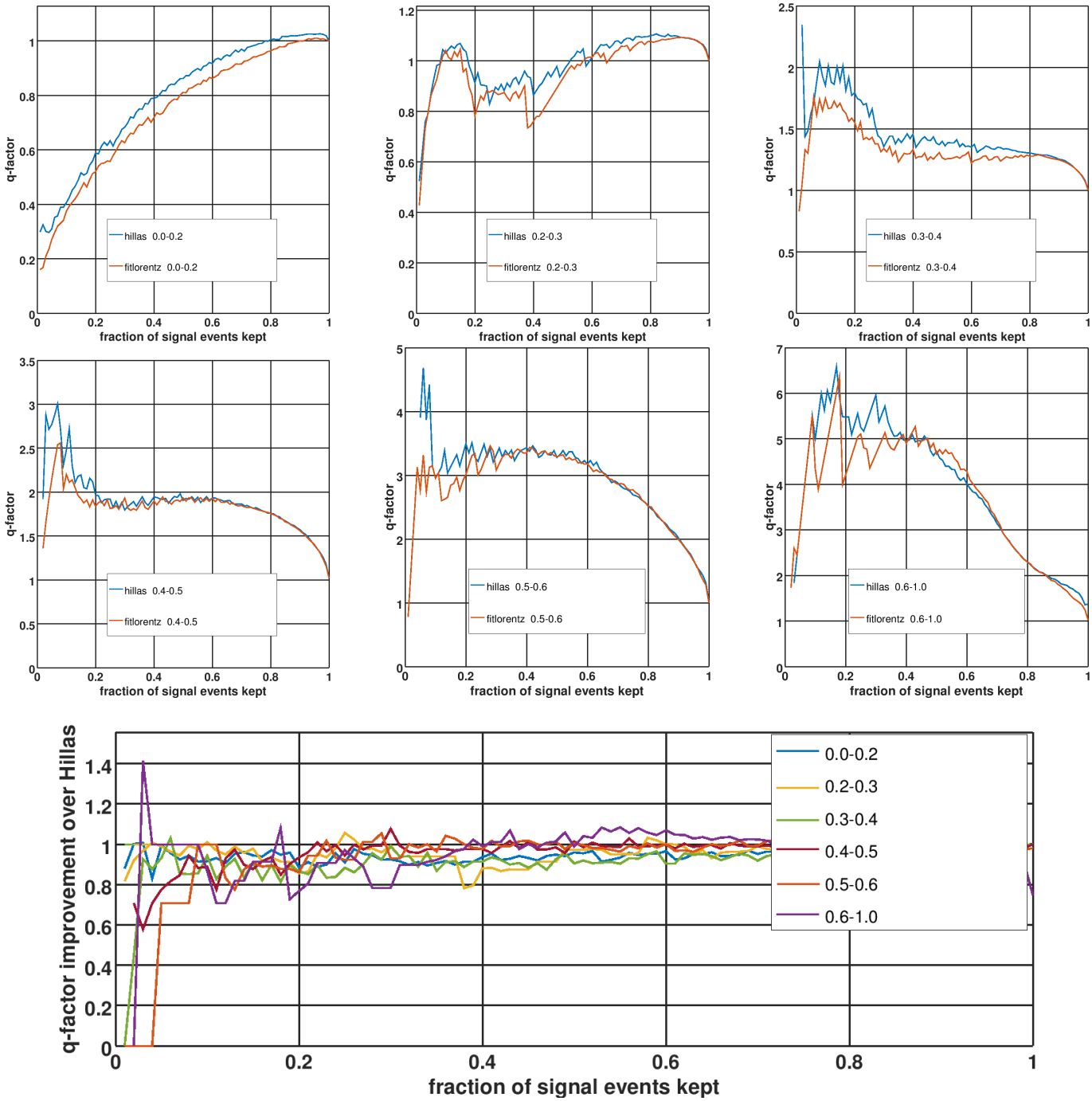


Fig. 9.16.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the Lorentz fit parameters. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

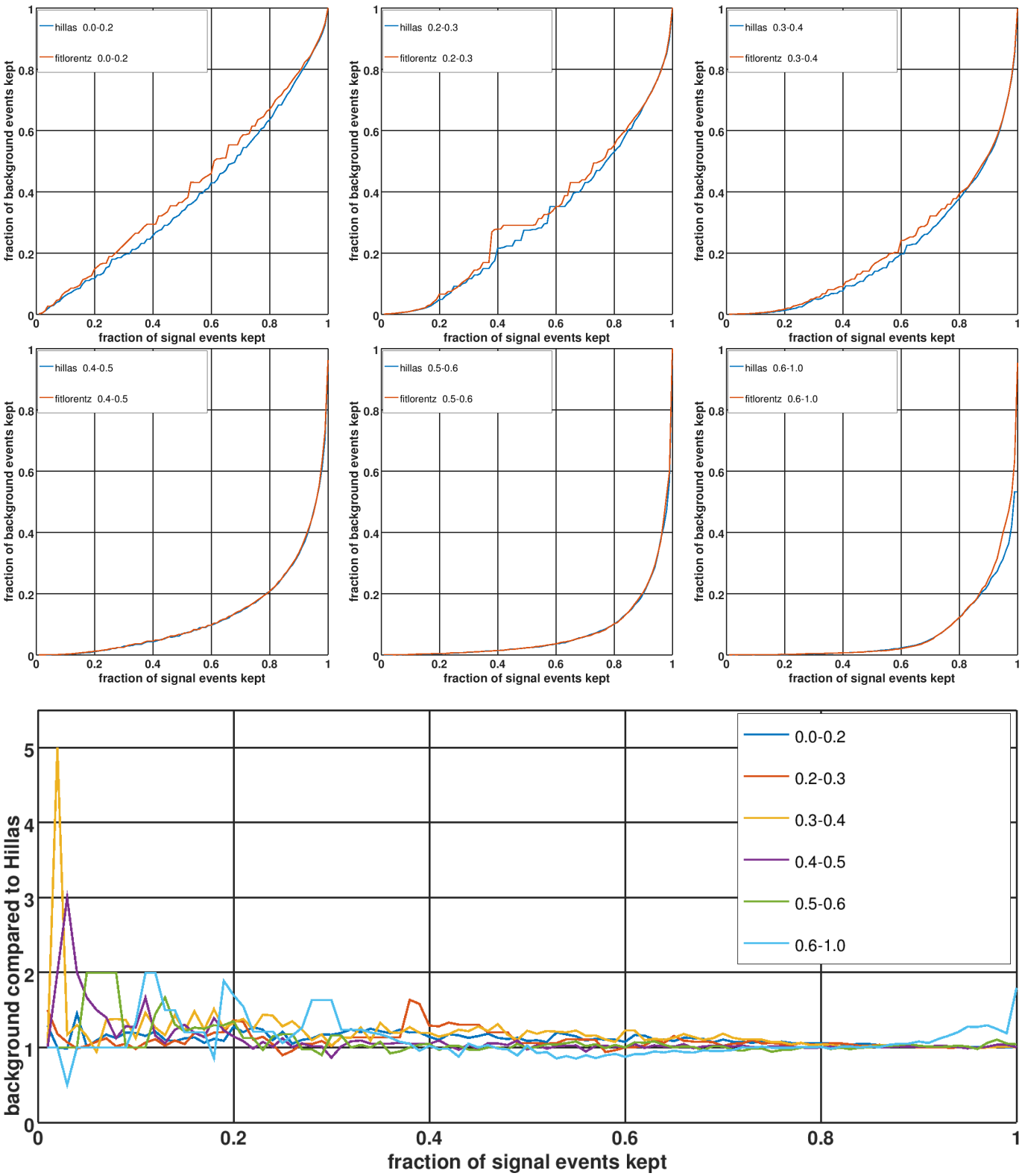


Fig. 9.17.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the Lorentz fit parameters. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.10. Geometrical and histogram features

Proton shower images are usually more fragmented than gamma shower images, which is well reflected by the Hillas width. Here, some additional, empirically developed methods for parameterizing shower images are presented. Some methods analyze the geometry of the shower image, often involving the direct neighbors of a pixel, while other methods focus on the histogram of intensities of a shower.
Each of the methods responds to some features of shower images. The idea is to find a subset of all methods that is not strongly correlated and, when fed to an SVM, gives good separation performance. The individual separation performances are shown in the appendix. For all methods:

- the shower image is called I
- the intensity of a pixel in the image I_i
- its coordinates x_i and y_i
- the list of neighbor pixels of pixel i is called B_i, and $|B_i|$ gives the number of neighbors of pixel i
- the vector with all parameterizations of the image p that are described in this section
- the number of pixels n
- the sum of all pixels $S = \sum_i I_i$
- the pixel with the maximum intensity M_1, and the pixel with the second highest intensity M_2.
- the major Hillas axis is called H_0 and the minor Hillas axis is called H_1
- the center of gravity is calculated as:

$$\texttt{COG}_x = \frac{1}{S}\sum_i I_i x_i \qquad \texttt{COG}_y = \frac{1}{S}\sum_i I_i y_i \tag{9.5}$$

- the center of gravity with equal weight for all pixels (COG1) as:

$$\texttt{COG1}_x = \frac{1}{n}\sum_i x_i \qquad \texttt{COG1}_y = \frac{1}{n}\sum_i y_i \tag{9.6}$$

  Compared to the normal center of gravity, this parameter penalizes sub-showers more.

9.10.1. Distances to Hillas axes

Let the distances from the maximum pixel to the Hillas axes be:

$$p_0 = \text{dist}(M_1, H_0) \qquad p_1 = \text{dist}(M_1, H_1) \tag{9.7}$$

For γ-ray showers, the maximum pixel is usually part of the shower head and lies in most cases close to the major Hillas axis, because γ-ray showers are symmetric across this axis (see figure 9.18). For protons, this is not true, so the distance can be used as discrimination parameter. So on average, p_0 is smaller for gammas than it is for protons (see figure 9.19). Along the minor Hillas axis, γ-ray showers are asymmetric, and a larger distance from the maximum pixel is expected. The same is true for protons, so this parameter alone does not discriminate well. However, in conjunction with p_0 it can be expected to increase the separation power.

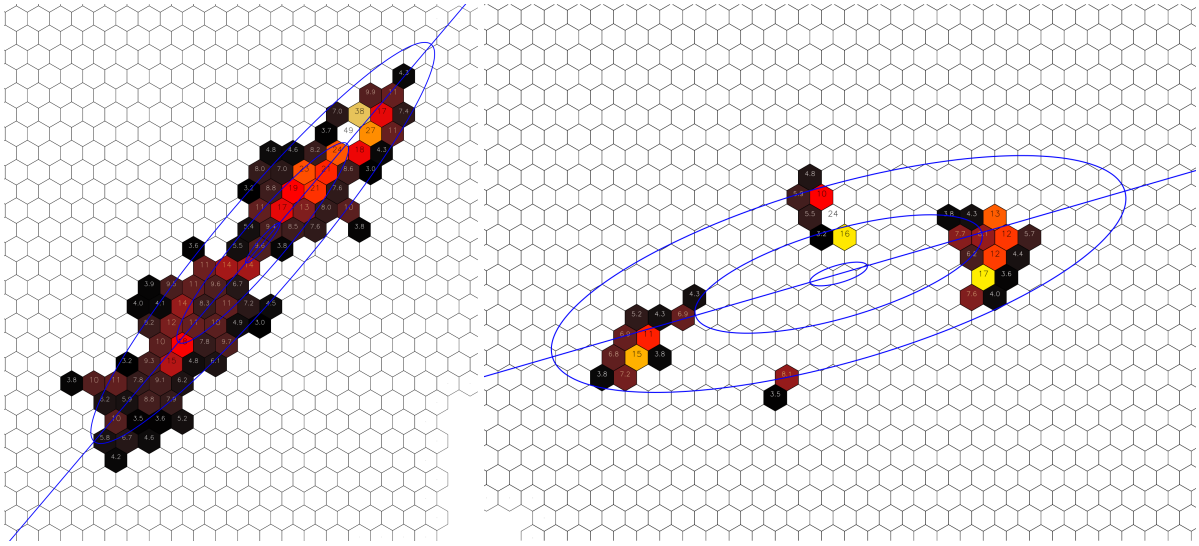


Fig. 9.18.: Left: A γ-ray shower that shows how the maximum pixel (white, intensity = 49 pe) is part of the shower head, and the shower head is on the major Hillas axis. Right: A proton shower, where the maximum pixel (white, intensity = 24 pe) is farther away from the major Hillas axis.

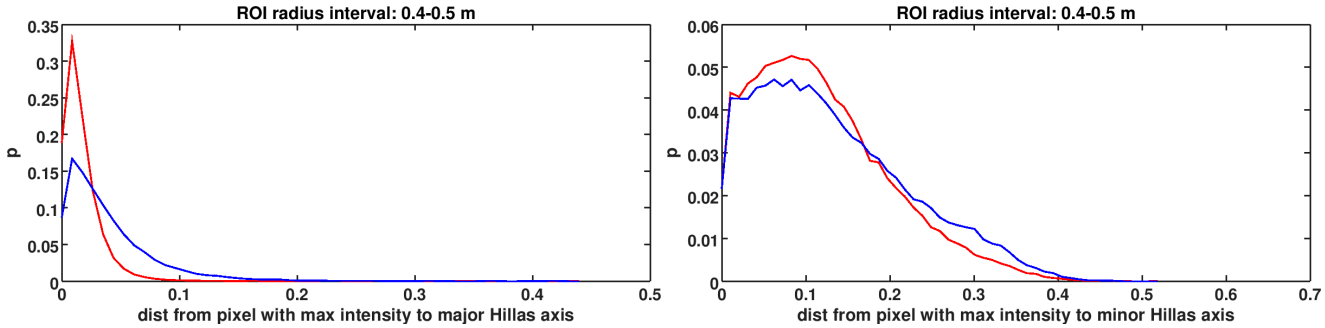


Fig. 9.19.: Distance from pixel with max. intensity to major (left plot) and minor (right plot) Hillas axis for a ROI size of $0.4 - 0.5$ m (the plots for the other ROI sizes can be found in the appendix). Red: gamma, blue: proton.

Depending on the image cleaning, the distance between a Hillas axis and the farthest pixel on each side of the axis can be a useful separation parameter. For example, one small sub-shower far away from the major Hillas axis will make this parameter respond, even if so many high intensity pixels are close to the axis that the Hillas width would be small. Let A_i be the pixels above the major Hillas axis, and B_i the pixels below. The distance of the farthest pixel of all A_i to the major Hillas axis gives one separation parameter, the distance of the farthest pixel of all B_i to the major Hillas axis gives another one.

$$p_2 = \max_i \left(\text{dist}(A_i, H_0)\right) \qquad p_3 = \max_i \left(\text{dist}(B_i, H_0)\right) \tag{9.8}$$

This is similar to Hillas width, except that only the pixel with the maximum distance is considered and two values are calculated, one for each side of the axis, so asymmetries are captured.
The same procedure is applied to the minor Hillas axis: Let C_i be the pixels above the minor Hillas axis, and D_i the pixels below. The distance of the farthest pixel of all C_i to the minor Hillas axis gives one separation parameter, the distance of the farthest pixel of all D_i to the minor Hillas axis gives another one.

$$p_4 = \max_i \left(\text{dist}(C_i, H_1)\right) \qquad p_5 = \max_i \left(\text{dist}(D_i, H_1)\right) \tag{9.9}$$

This is similar to Hillas width, except that only the pixel with the maximum distance is considered and two values are calculated, one for each side of the axis, so asymmetries are captured. The results of this method depend very much on the image cleaning used.

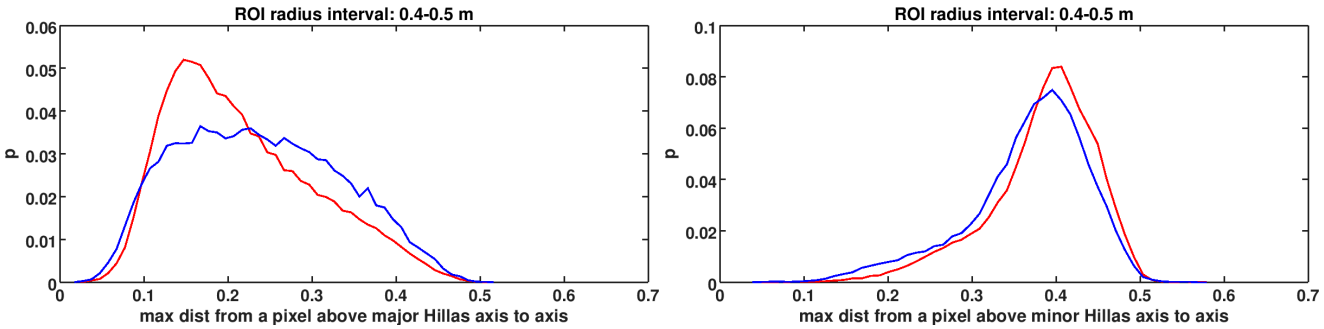


Fig. 9.20.: Distance from pixel with max. distance **above** (towards shower head) the major (left plot) and minor (right plot) Hillas axis to that axis for a ROI size of $0.4 - 0.5$ m (the plots for the other ROI sizes can be found in the appendix). Red: gamma, blue: proton. The distribution of the distance from pixel with max. distance **below** (away from shower head) the major and minor Hillas axis looks similar and is not shown here.

In order to measure the asymmetry across Hillas axes, the distance between a Hillas axis and the center of gravity of all pixels on one side of that axis can be calculated. The center of gravity is calculated with a weight of 1 for each pixel to give pixels far away from a Hillas axis with low intensities more impact. Let A_i be the pixels above the major Hillas axis, and B_i the pixels below. The mean distance of all A_i to the major Hillas axis gives one separation parameter, the mean distance of all B_i to the major Hillas axis gives another one.

$$p_6 = \sum_i \text{dist}(A_i, H_0) \qquad p_7 = \sum_i \text{dist}(B_i, H_0) \tag{9.10}$$

This is similar to Hillas width, except that the pixel intensities are ignored and two values are calculated, one for each side of the axis, so asymmetries are captured.
The same procedure is applied to the minor Hillas axis: Let C_i be the pixels above the minor Hillas axis, and D_i the pixels below. The mean distance of all C_i to the minor Hillas axis gives one separation parameter, the mean distance of all D_i to the minor Hillas axis gives another one.

$$p_8 = \sum_i \text{dist}(C_i, H_1) \qquad p_9 = \sum_i \text{dist}(D_i, H_1) \tag{9.11}$$

This is similar to Hillas length, except that the pixel intensities are ignored and two values are calculated, one for each side of the axis, so asymmetries are captured. Here, asymmetries are expected for γ-ray and for proton showers, but in conjunction with p_3 and p_4, these parameters are probably useful.

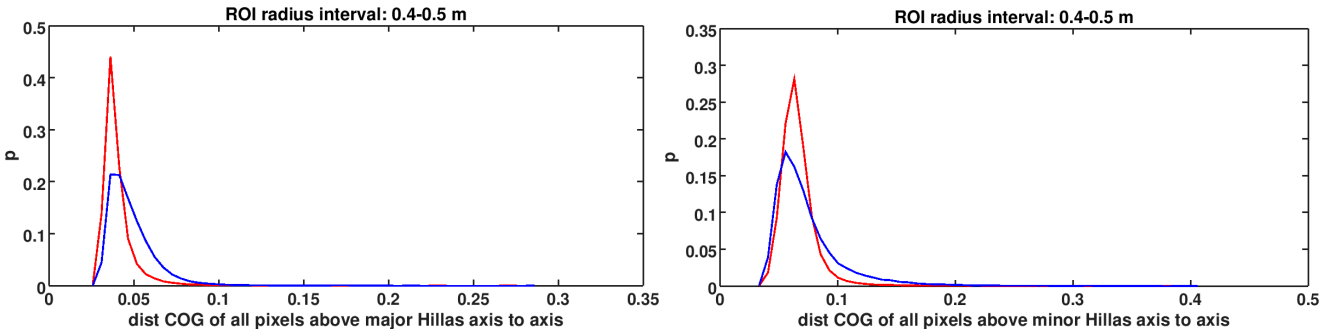


Fig. 9.21.: Distance from COG of all pixels **above** the major (left plot) and minor (right plot) Hillas axis to that axis for a ROI size of $0.4 - 0.5$ m (the plots for the other ROI sizes can be found in the appendix). Red: gamma, blue: proton. The distribution of the distance from COG of all pixels **below** the major and minor Hillas axis looks similar and is not shown here.

9.10.2. Other distance related parameters

COG1 is the center of gravity with weighting 1 for all pixels, so pixels far away from the other pixels have a large impact, even if their intensities are small. The distance between the COG (center of gravity) and COG1 is a heuristic

for detecting showers that have their pixels with high intensity concentrated on a small area, but the pixels with lower intensity asymmetrically spread across a larger area.

$$p_{10} = d(COG, COG1) \tag{9.12}$$

Calculating the distance between the three brightest pixels can detect events that have a second shower far away from the main shower.

$$p_{11} = \frac{d(M_1, M_2) + d(M_1, M_3) + d(M_2, M_3)}{3} \tag{9.13}$$

In γ-ray showers, the three brightest pixels are expected to lie close to each other, while this is not expected for all proton showers.

The *weighted auto-correlation* calculates the sum of all products of all intensity pairs weighted by the inverse of their distance. It is a heuristic for detecting showers that have several pixels with high intensities scattered across the image. As can be seen in figure 9.18, γ-ray showers usually have their pixels with intensity concentrated at the shower head, whereas in proton showers, these pixels are more scattered. In order to prevent low intensity pixels from diluting the response of the classifier, only pixels with an intensity of at least 10% of the maximum intensity in the image are considered.

$$x_{12} = \frac{1}{n_{\text{pairs}}} \sum_{i,j} \sqrt{\frac{I_i I_j}{\text{dist}(i,j)}} \tag{9.14}$$

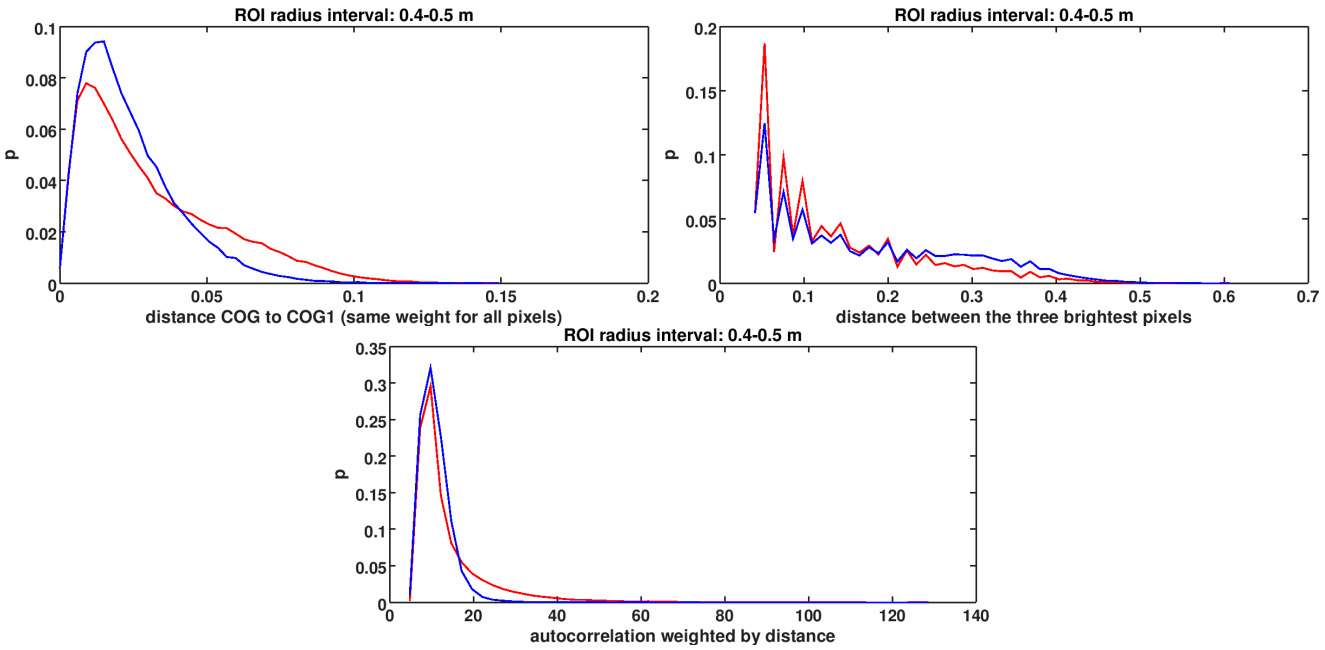


Fig. 9.22.: Top left: Distance from center of gravity (1-norm) to center of gravity.
Top right: Mean distance between the 3 brightest pixels.
Bottom: Auto-correlation. Sum of all products of all intensity pairs weighted by inverse distance.
In all plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

9.10.3. Neighborhood related parameters

The *local activity* is a heuristic for detecting high-frequency noise in an image. For each shower pixel i with more than four neighbors, it sums up the absolute differences between it and each of its neighbors B_{ik} and divides by the number of neighbors $|B_i|$. The global sum is then normalized by the shower amplitude.

$$p_{13} = \frac{1}{S} \sum_i \frac{|\text{B}_i| > 4}{|\text{B}_i|} \sum_{k \in \text{B}_i} |I_i - I_k| \tag{9.15}$$

It was expected that proton shower images would yield larger values than γ shower images because of the sub-showers, but figure 9.23 shows that the opposite is true - possibly because the Cherenkov light of γ-showers is more concentrated and results in steeper gradients to the maximum pixels.
Compactness is a heuristic for detecting showers whose border deviates from a circle. It counts the non-border pixels (pixels with 6 neighbors) and the border pixels (pixels with less than 6 neighbors), and then calculates the sphericity:

$$\text{border} = \sum_i (|\mathrm{B}_i| < 6) \tag{9.16}$$

$$\text{non-border} = n - \text{border} \tag{9.17}$$

$$p_{14} = \frac{\text{border}}{\text{non-border}^2} \tag{9.18}$$

This parameter responds to fragmented showers and showers with fragmented borders, so it should be suited to detect protons. However, it also responds to asymmetric showers, and γ-ray shower are usually elliptical. This means that this parameter will not be useful alone, but only in combination with other parameters.
Another way to detect fragmented borders is to count the *bays* (pixels with 4 or 5 neighbor pixels) and the *tongues* (pixels with 1, 2 or 3 neighbor pixels) and then calculate the ratio:

$$p_{15} = \frac{\sum_i 3 < |\mathrm{B}_i| < 6}{\sum_i 0 < |\mathrm{B}_i| < 4} \tag{9.19}$$

The next method counts the number of *local maxima*, where a local maximum is defined as a pixel that has at least 3 neighbors, and all of them have to have a smaller intensity. The result is then normalized by the number of pixels in the shower.

$$p_{16} = \frac{1}{n} \sum_i (|\mathrm{B}_i| > 2) \prod_{k \in \mathrm{B}_i} I_k < I_i \tag{9.20}$$

Each comparison returns either 0 or 1. In the product formula, if all comparisons return 1, the product equals 1, but if one or more comparisons return 0, the result is 0. The idea behind this parameter is that if a pixel is larger than all of its neighbors, it is a hint for a sub-shower.

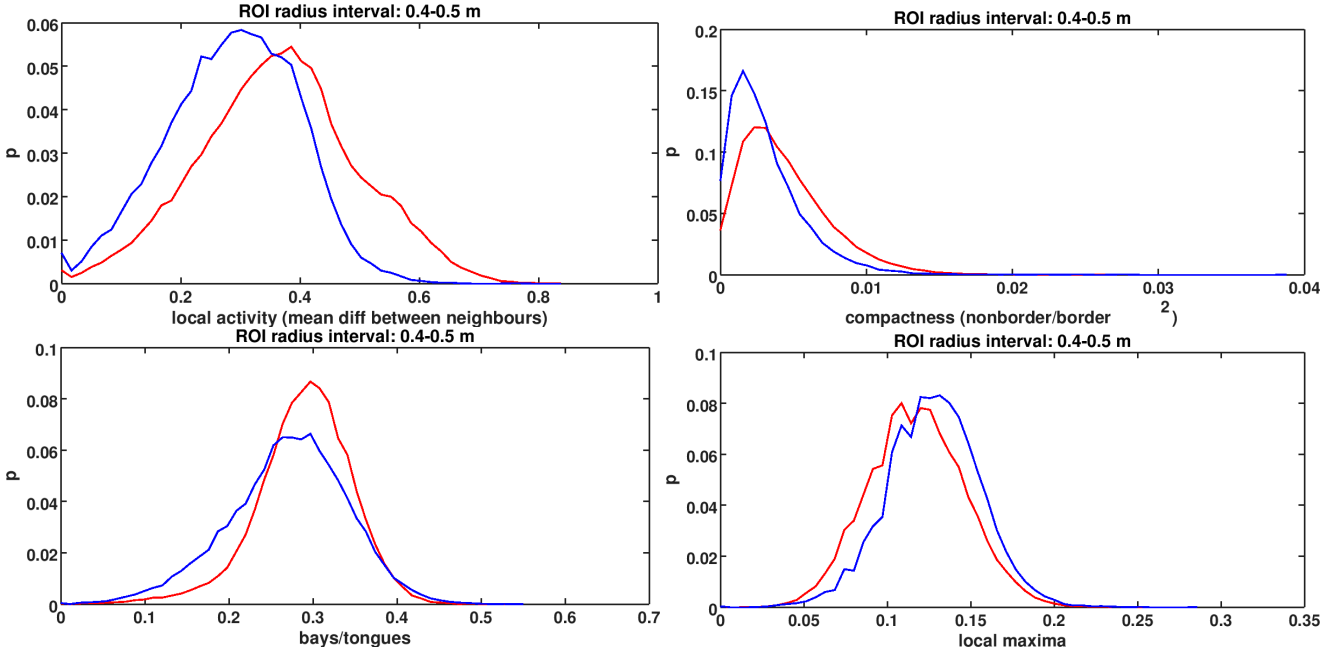


Fig. 9.23.: Top left: Local activity (mean difference between pixel and its neighbors). Top right: Compactness of shower image ($\frac{\text{pixels}_n}{\text{border}^2}$).
Bottom left: Bays (pixels with 4 or 5 neighbors) over tongues (pixels with 1, 2 or 3 neighbors). Bottom right: Local maxima.
In all plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

9.10.4. Convex hull

The convex hull of a set of points is the smallest convex set that contains all points. Figure 9.24 illustrates with an example how pixels far away from the main shower can have a huge impact on the area of the convex hull.

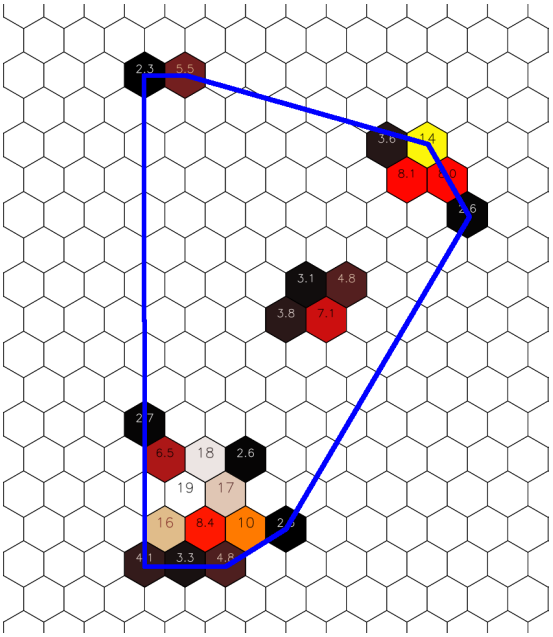


Fig. 9.24.: The convex hull of a proton shower cleaned with 2-5 cleaning.

The convex hull can be used to detect outspread or asymmetric showers. The first method calculates the area A of the convex hull and normalizes with the logarithm of the amplitude S:

$$x_{17} = \frac{A}{\log(S)} \tag{9.21}$$

The second method is a uses the area A and the border B of the convex hull to calculate its sphericity:

$$x_{18} = \frac{A}{B^2} \tag{9.22}$$

The sphericity allows to distinguish between circular and elliptical images.

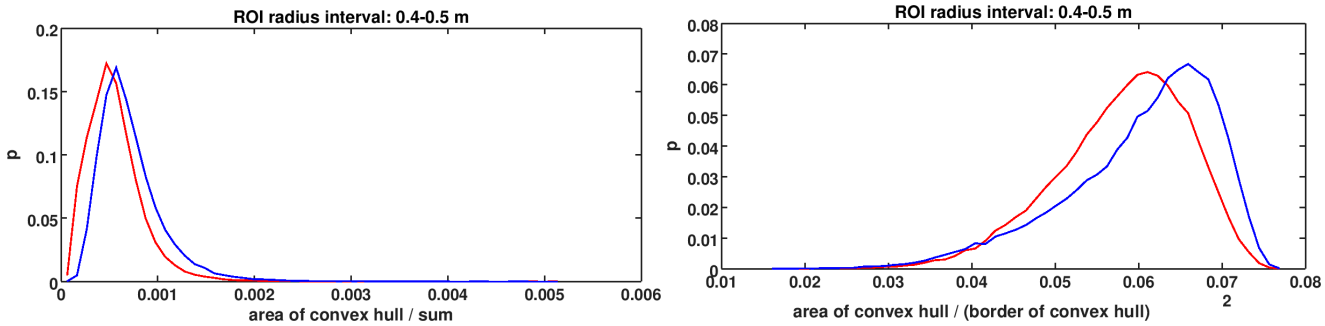


Fig. 9.25.: Left: Density of convex hull (area/amplitude). Right: Compactness of convex hull (area/border^2). for a ROI size of $0.4 - 0.5$ m (the plots for the other ROI sizes can be found in the appendix). Red: gamma, blue: proton.
In both plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

9.10.5. Segmentation related parameters

Analyzing the connected components [34] of a shower image is a heuristic for detecting sub-showers that are located so far away from the main shower that there are gaps between them after image cleaning (see figure 9.26). These

newly created islands can be found with the *connected component* algorithm, which assigns labels to all shower pixels, so that all pixels of the same island have the same label and pixels of different islands have different labels.

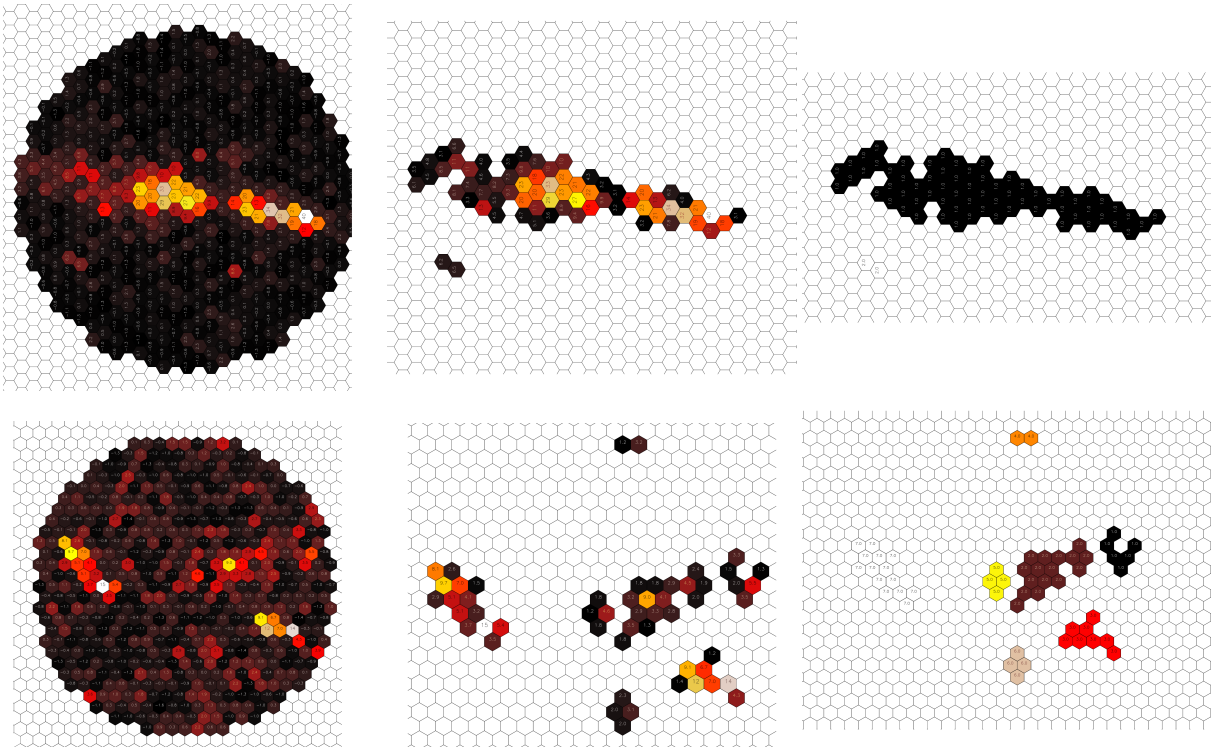


Fig. 9.26.: Top: γ-ray shower. Bottom: proton shower.
Left: original shower image. Center: image cleaned with 3,6 cleaning. Right: connected components of cleaned image.

After the algorithm has finished, the number of labels is the number of islands in the image. It is normalized by the number of pixels in the shower. Typical proton showers are expected to have more islands than γ-ray showers. Another useful parameter is the fraction of pixels that are located in the largest island, normalized by the number of pixels in the shower. Since γ-ray showers are usually more compact than proton showers, more pixels are expected to be located in the largest island.

$$x_{19} = \frac{\text{islands}_n}{n} \qquad x_{20} = \frac{\text{largestIsland}_n}{n} \tag{9.23}$$

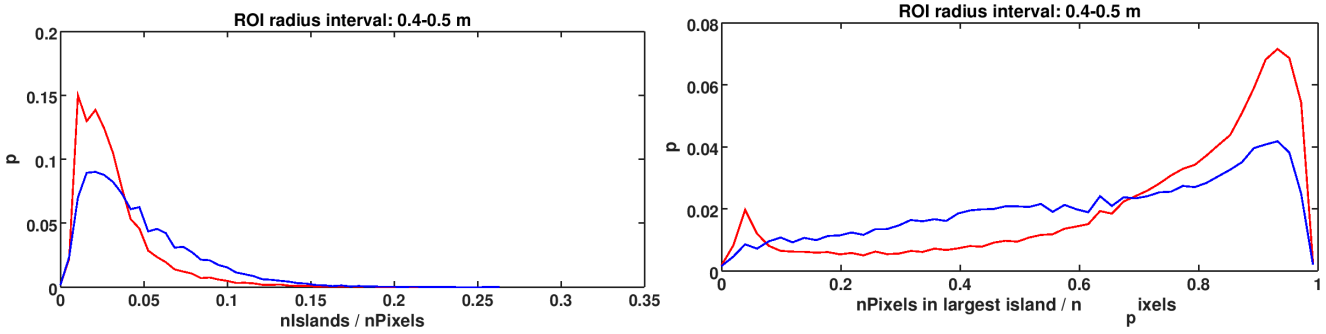


Fig. 9.27.: Left: Number of islands (connected components) in image divided by number of pixels. Right: Number of pixels in largest island (connected component) divided by number of pixels.
In both plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

The *watershed algorithm* [35] is a more general segmentation algorithm than the connected components algorithm. It does not only detect islands, but also segments around local maxima within the islands. This is useful for finding

small sub-showers that are on top of the main sub-shower. The basic idea of the watershed algorithm is to regard the image as relief, turn it around so that all local maxima become local minima, pierce holes into the relief at all local minima, and then drown the relief in water. During drowning, the water level has to be observed and whenever the water of two local minima meets, the so-called *watershed line* is recorded as the border of the two segments. When the relief is completely drowned, the segmentation of the image is complete. In order to prevent over-segmentation, the input image is smoothed.
Once the shower image is segmented, two parameters are derived:

$$x_{21} = \frac{\text{segments}_n}{S} \qquad x_{22} = \frac{\sum_{i \in \text{largestSegment}} I_i}{S} \tag{9.24}$$

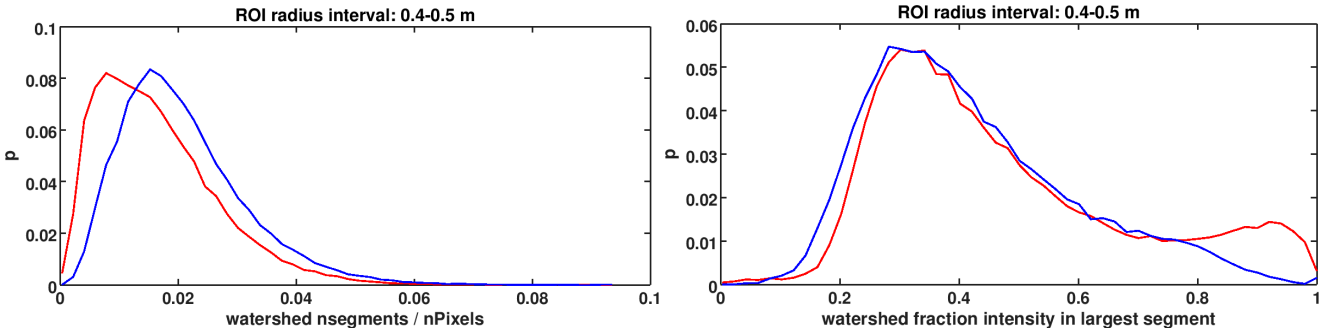


Fig. 9.28.: Watershed transform. Left: $\frac{\text{n\_segments}}{amplitude}$. Right: fraction of amplitude in largest segment.
In both plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

9.10.6. Histogram related methods

Since proton showers are usually broader than γ-ray showers for the same amplitude, the *density* can be calculated in order to detect showers that have their amplitude spread over a larger area, which is common for proton showers.

$$p_{23} = \frac{S}{n} \tag{9.25}$$

Integrating over certain intervals in the normalized histogram of intensities can be used as separation parameter. Let m be the minimum intensity in the shower image, and if $m < 1$ pe, then set $m = 1$. These four intervals provide good separation (values are in pe):

$$x_{24} = \frac{1}{S}\int_{m}^{m+3} t\,\mathrm{d}t \qquad x_{25} = \frac{1}{S}\int_{m+3}^{m+8} t\,\mathrm{d}t \qquad x_{26} = \frac{1}{S}\int_{m+8}^{m+12} t\,\mathrm{d}t \qquad x_{27} = \frac{1}{S}\int_{m+12}^{\max} t\,\mathrm{d}t \tag{9.26}$$

Since γ-ray showers are more collimated, they are expected to have less small and medium intensities.
In physics, *entropy* is a measure of the lack of knowledge of the state of all particles. Similarly, in image analysis, entropy is a measure of the mean information content of an image. Except for the missing Boltzmann constant, the definition is the same:

$$x_{28} = -\sum_i p_i \log(p_i) \tag{9.27}$$

where p is the normalized histogram of the shower image. For $p_i = 0$, $p_i \log(p_i)$ is defined to be 0. If all pixels in the image have the same value, the number of states is lowest. The histogram is then empty, except for one bin, and the entropy of the image is 0. If the image is a random image, the number of states is highest. All histogram bins are equally filled, and the entropy is maximal: for n bins, it is $\log(n)$.
This method measures how equally distributed the intensities of the shower pixels are. γ-ray showers are expected to have higher entropy, because the smooth gradients towards the shower lead to more histogram bins being filled. Proton shower have more low and medium intensities and thus less randomness in the histogram.

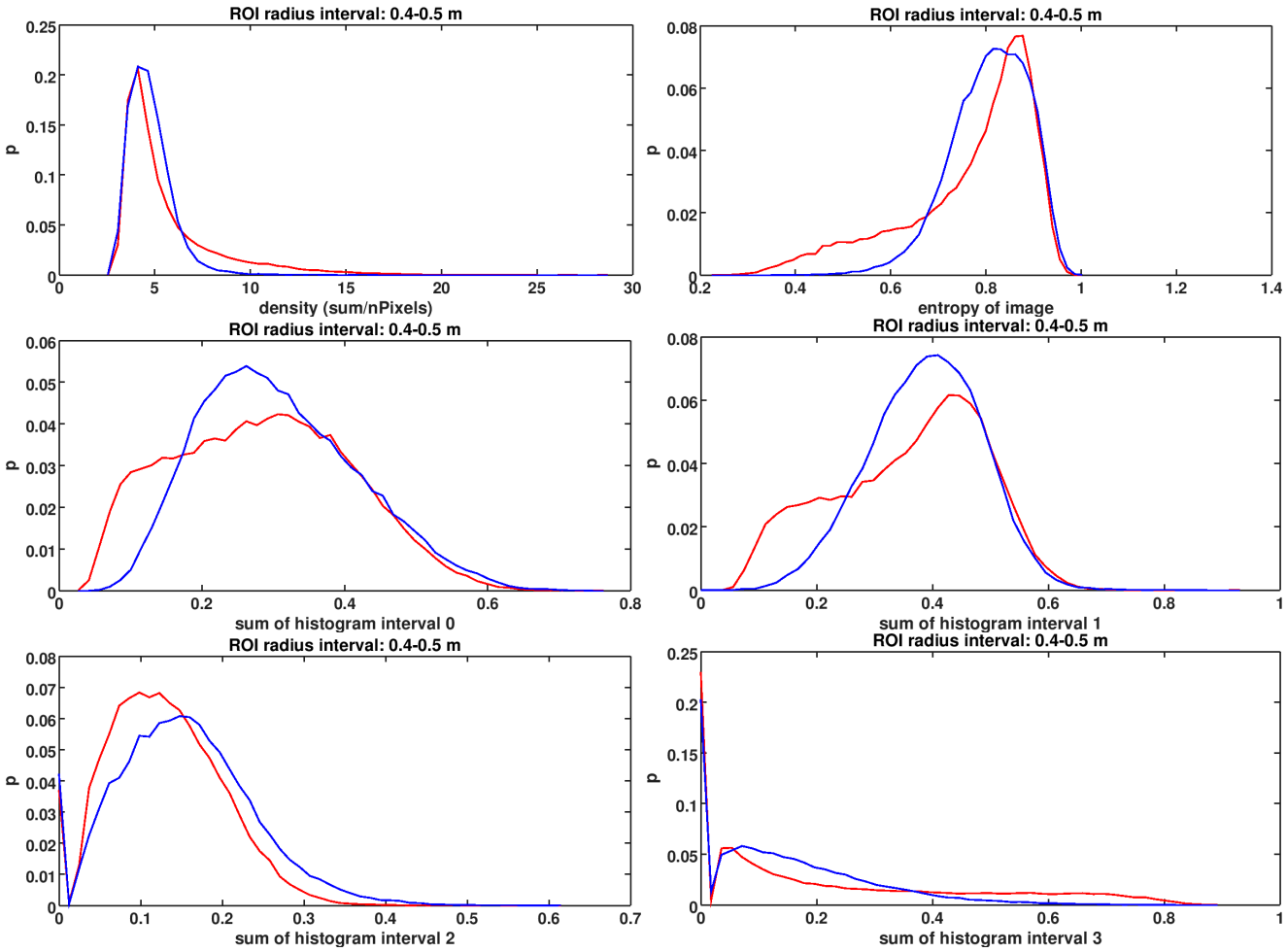


Fig. 9.29.: Top left: Density (amplitude/number of pixels). Top right: Image entropy.
Center left to bottom right: Sum of all pixels between 1 pe and 3 pe, between 3 pe and 8 pe, between 8 pe and 12 pe, and above 12 pe, always divided by amplitude.
In all plots, gamma is red and proton is blue, and the ROI size is $0.4 - 0.5$ m. The plots for the other ROI sizes can be found in the appendix.

Results: Figure 9.30 shows that for small and medium image sizes, the separation performance is significantly improved. Figure 9.31 shows that for the small and medium image sizes, the background is reduced by up to 50%.

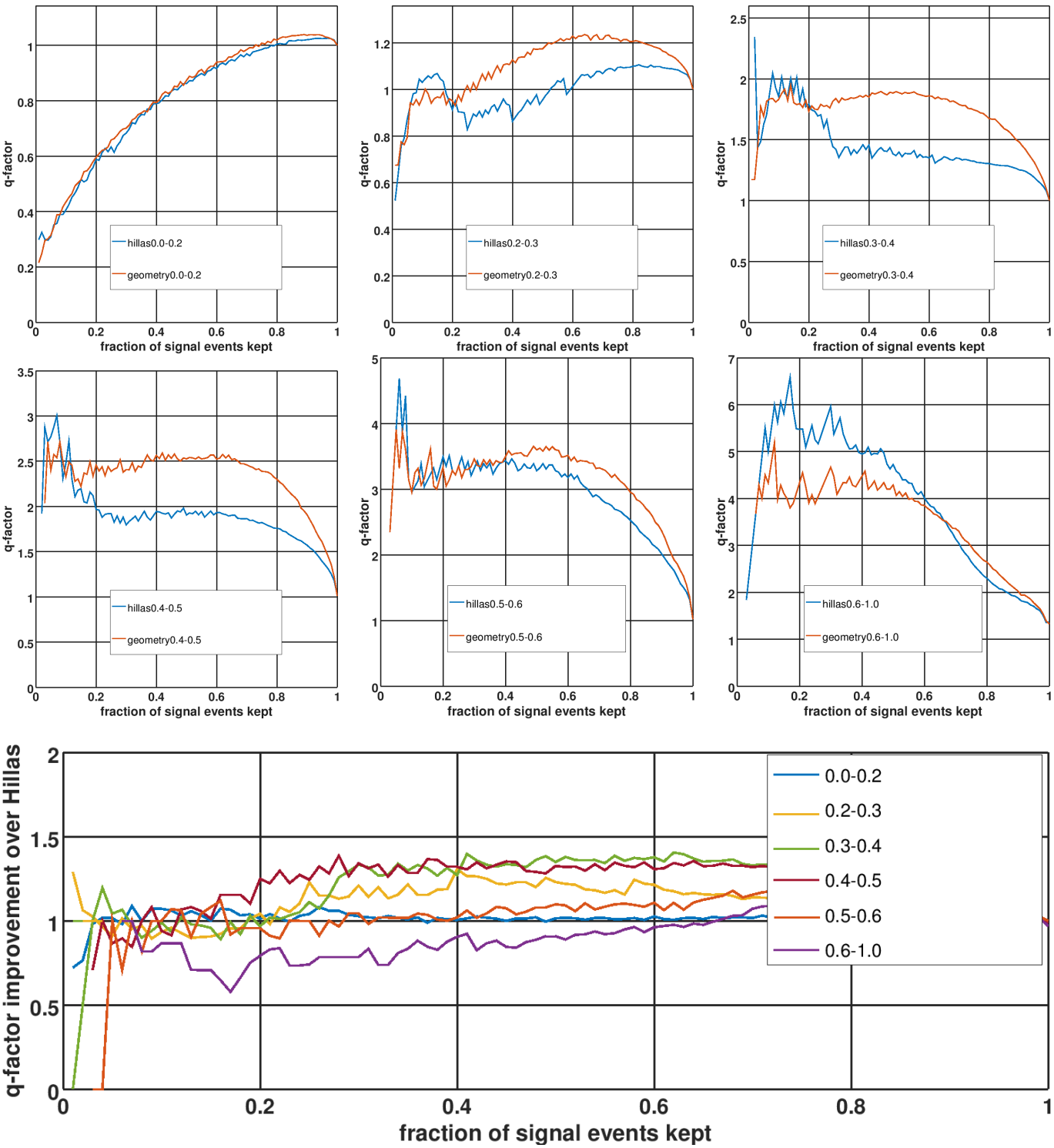


Fig. 9.30.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the geometry parameters described in this section. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

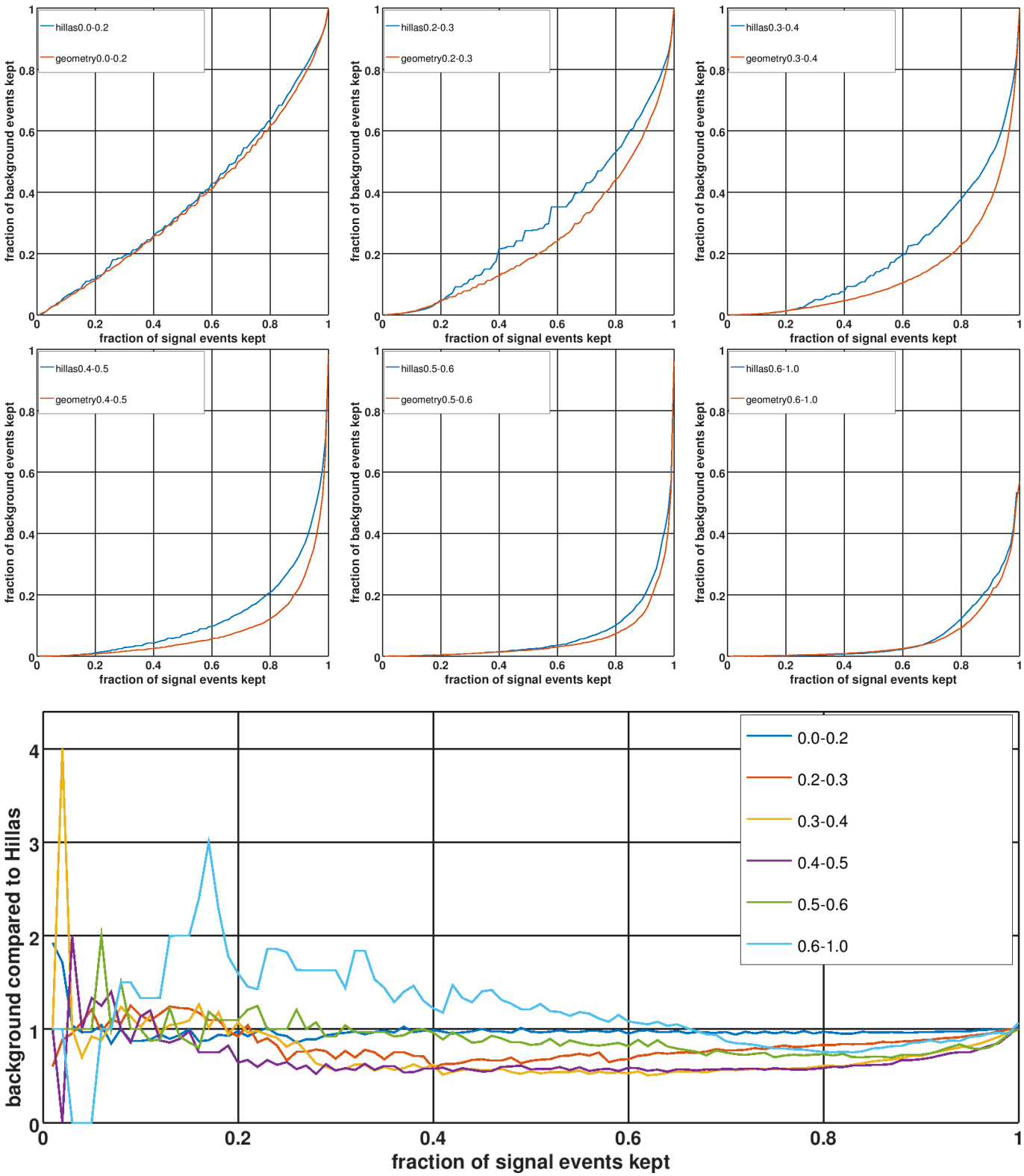


Fig. 9.31.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an SVM-classification using the geometry parameters described in this section. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.11. Correlation matrix

For two variables X and Y, the Pearson correlation coefficient is defined as:

$$\rho_{XY} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{(\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2)^{\frac{1}{2}}} \tag{9.28}$$

The following correlation matrices contain the absolute correlation coefficients for different parameterizations and thus provide an overview of their linear or anti-linear association. For each ROI size the correlation matrix for parameterizations of γ-ray and proton showers is shown. Since one half of a correlation matrix is always redundant, the γ and proton matrix can be merged into one matrix. The γ matrix is always shown top left, and the proton matrix bottom right. The correlation matrices reveal correlations between several variables, for example `Hillas length` and `autocorrelation`, which means that some variables are redundant to some extent.

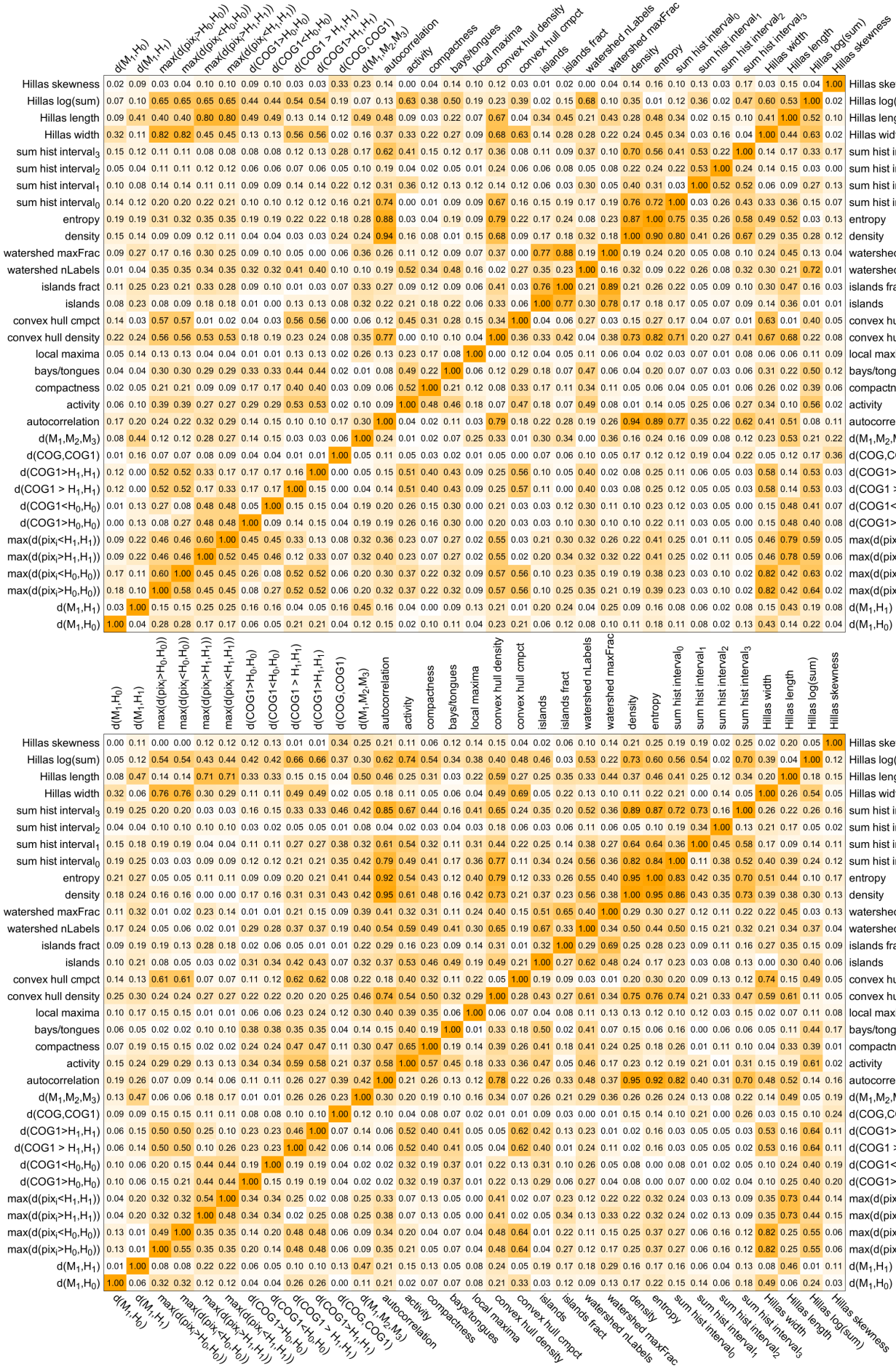


Fig. 9.32.: Correlation matrix for parameterizations of showers with ROI radius of 0.0-0.2 m (top) and 0.2-0.3 m (bottom). The upper left triangular matrix contains the correlation coefficients for γ-ray showers, the lower right triangular matrix contains the correlation coefficients for proton showers.

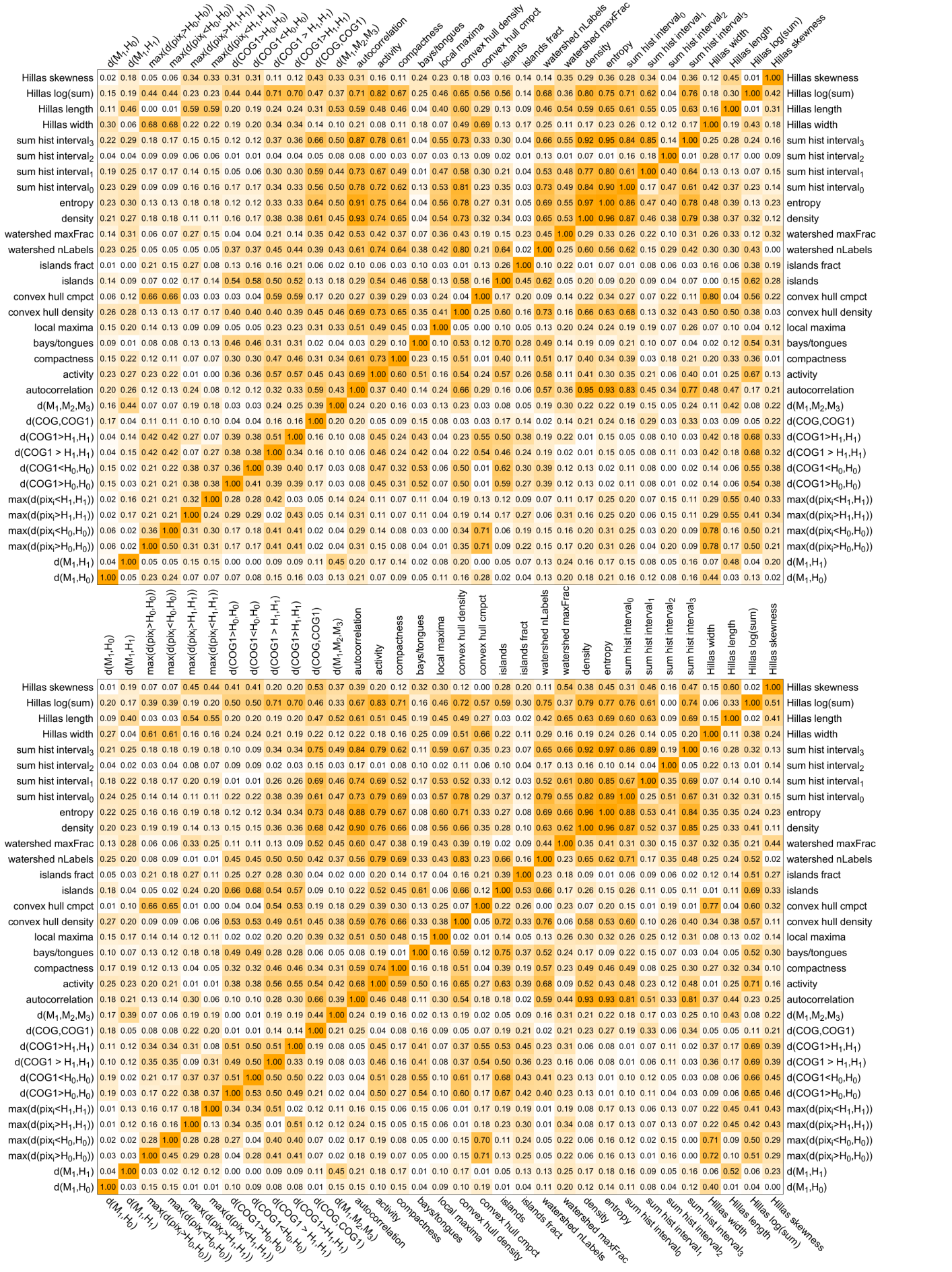


Fig. 9.33.: Correlation matrix for parameterizations of showers with ROI radius of 0.3-0.4 m (top) and 0.4-0.5 m (bottom). The upper left triangular matrix contains the correlation coefficients for γ-ray showers, the lower right triangular matrix contains the correlation coefficients for proton showers.

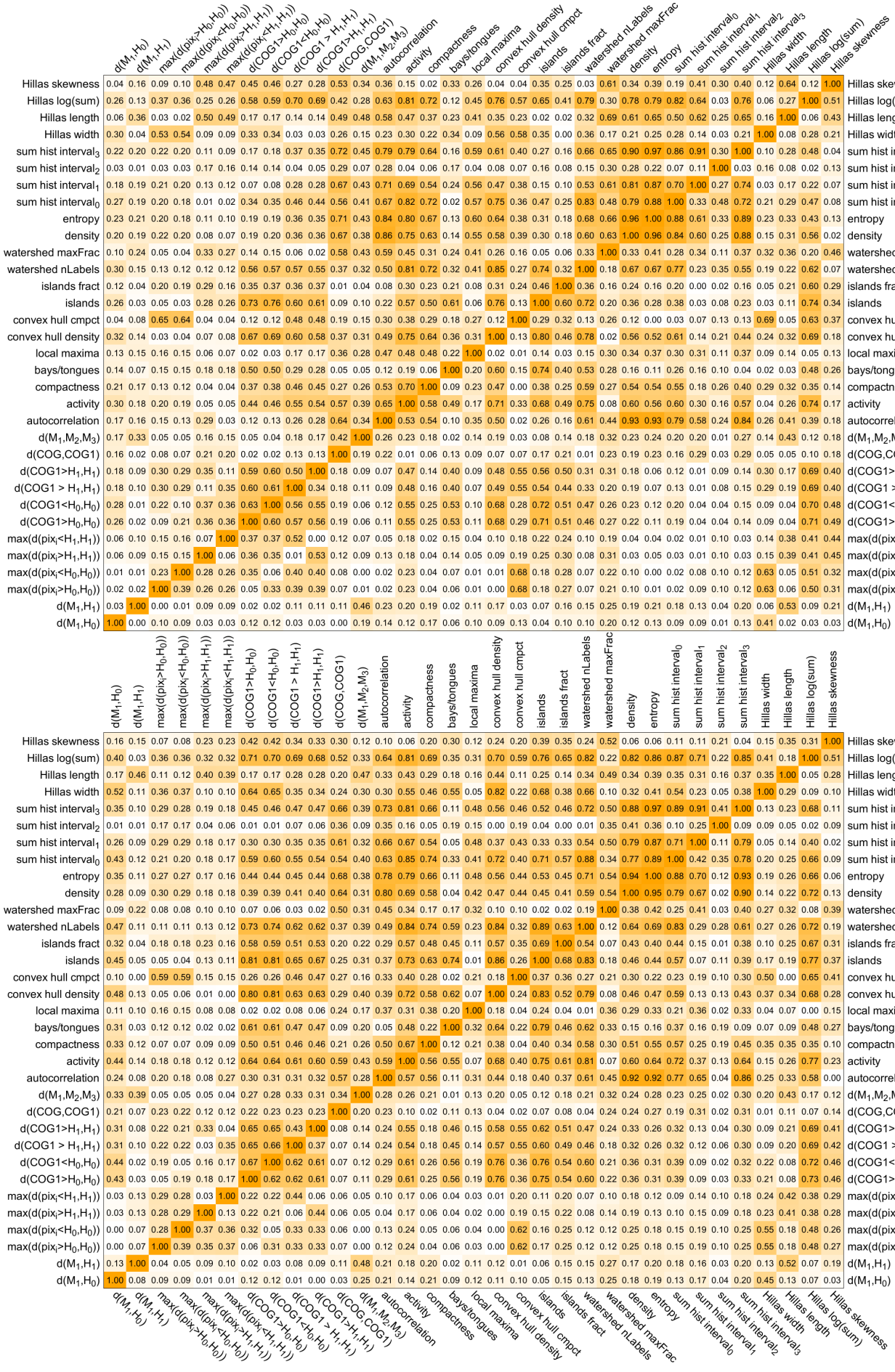


Fig. 9.34.: Correlation matrix for parameterizations of showers with ROI radius of 0.5-0.6 m (top) and 0.6-1.0 m (bottom). The upper left triangular matrix contains the correlation coefficients for γ-ray showers, the lower right triangular matrix contains the correlation coefficients for proton showers.

The singular values of the correlation matrix of parameterizations show how much of the combined class variance can be captured with fewer basis vectors of an optimal base. Figure 9.35 shows these singular values for the parameterizations of the γ-ray shower and the proton shower data set, for all ROI sizes. It can be seen that even with the optimal set of basis vectors, at least 15 dimensions would still be needed in order to reconstruct 95% of the information. The last 9 dimensions carry only $\approx 5\%$ of the information.

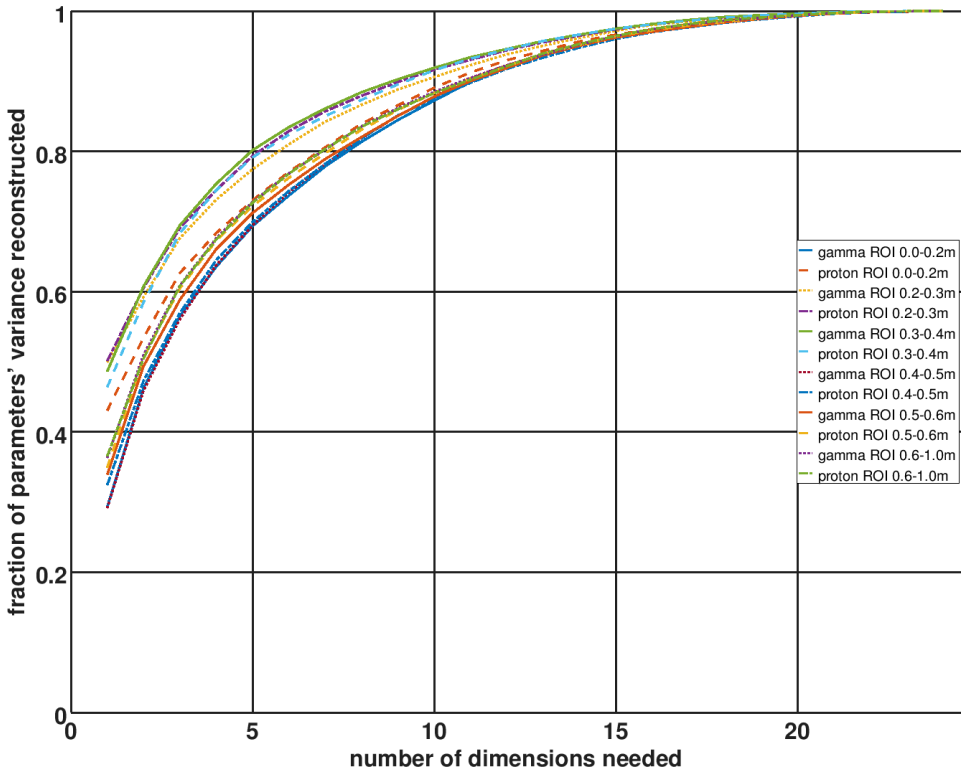


Fig. 9.35.: Fraction of the information that is contained in all variables, depending on the number of basis vectors used to reconstruct the information. The basis vectors are sorted by importance.

The SVD and correlation matrix only give hints about the combined separation power of all parameters.
It is possible that weakly correlated parameters with some separation power do not have much increased separation power if combined. This is the case if the parameters are uncorrelated for all samples that cannot be correctly classified, but uncorrelated for the samples that can be correctly classified. Combining such parameters would classify the same events correctly, and with higher certainty, but it would not help improving the signal-to-noise ratio. As the results will show, this is true for the geometry parameters.
It is also possible that parameters with weak individual separation power can separate classes if they are combined. This is the case for Hillas length, which by itself is not very powerful, but in conjunction with Hillas width and amplitude leads to very good separation performance.
Example: Given are two classes that can be described by two parameters:

$$c_1 = x, y | x \in [0,1], y = x \qquad c_2 = x, y | x \in [0,1], y = 1 - x \tag{9.29}$$

It is obvious that combined, the members of the two classes can be separated perfectly (see figure 9.36, left), but individually, none of the parameters x and y can separate c_1 from c_2 (see figure 9.36, right).

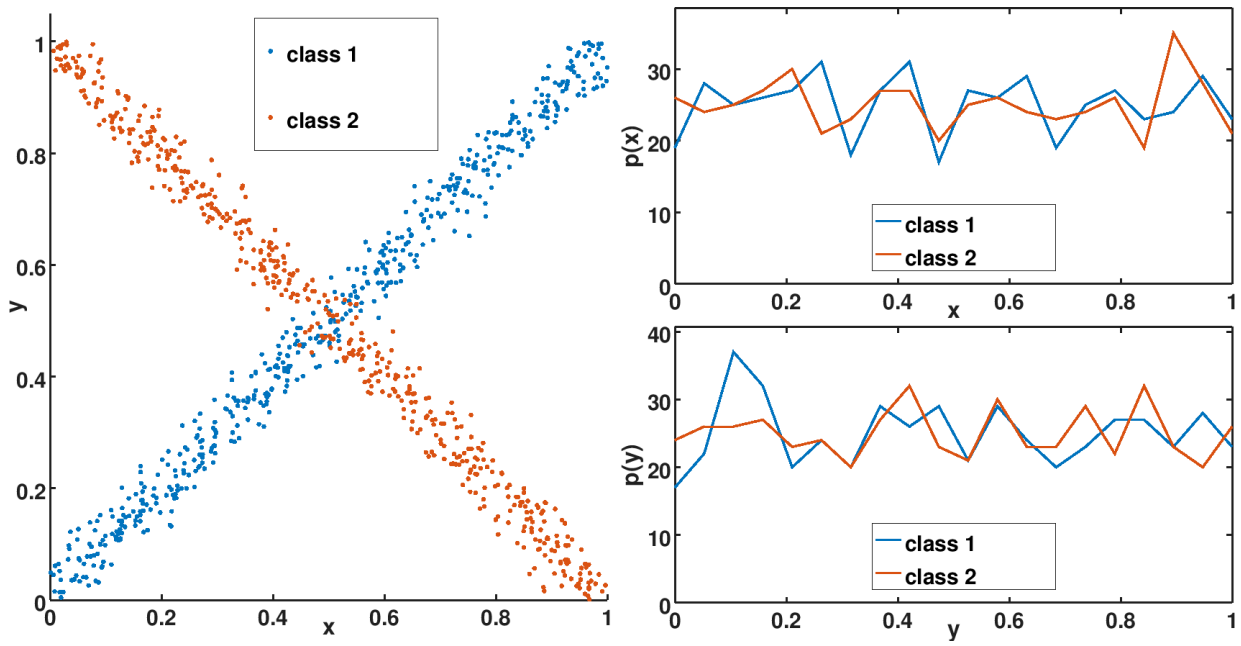


Fig. 9.36.: Examples of two classes described by x and y (left). The histograms (right) show that none of the parameters alone can separate the two classes.

9.12. All methods combined

All parameters from the previous sections (Hillas parameters of images with different cleanings, histogram of projection onto minor Hillas axis, fit of a Lorentzian, and the geometry parameters) can be concatenated to a long vector and fed into an SVM.

Results: Figure 9.37 shows that when more than 30% of all γ-ray showers must be kept, this classifier removes significantly more background events than the pure Hillas-based SVM-classifier. For medium-sized images the improvement is the largest.

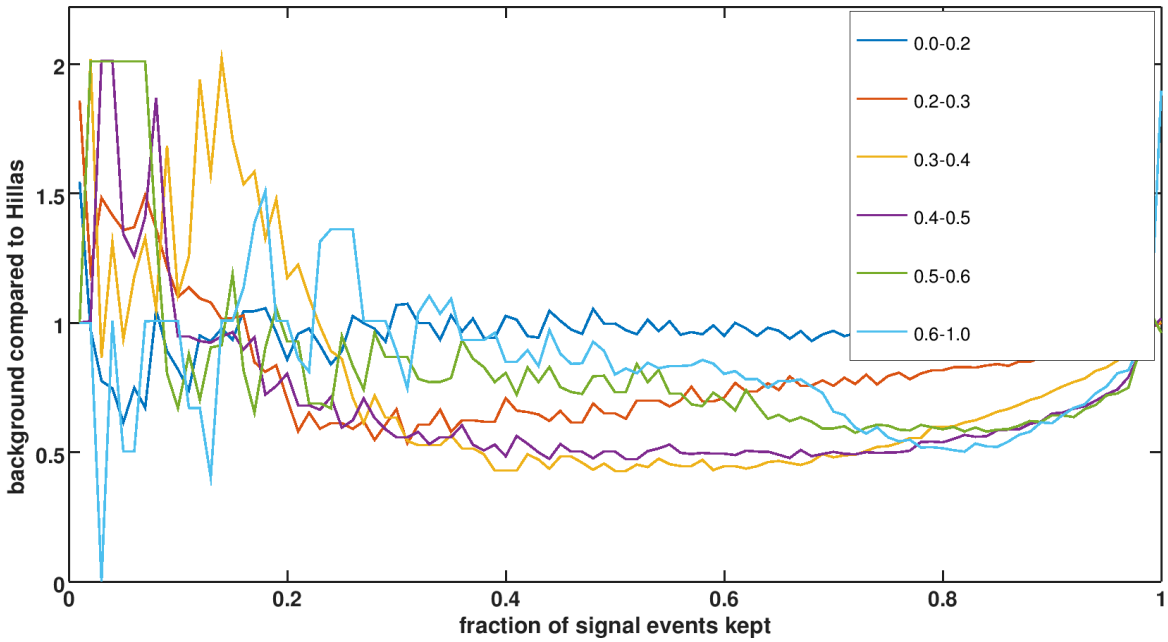


Fig. 9.37.: The ROC curves for different image sizes for an SVM-classification using the Hillas parameters of images with different cleanings, histogram of projection onto minor Hillas axis, fit of a Lorentzian, and the geometry parameters.

9.13. Calculating moments in each bin

Another method, which was later developed, is to rotate the shower until the major Hillas axis is horizontal, and then divide it into n vertical histograms (here $n = 5$) with m bins each (here $m = 13$, see figure 9.38).

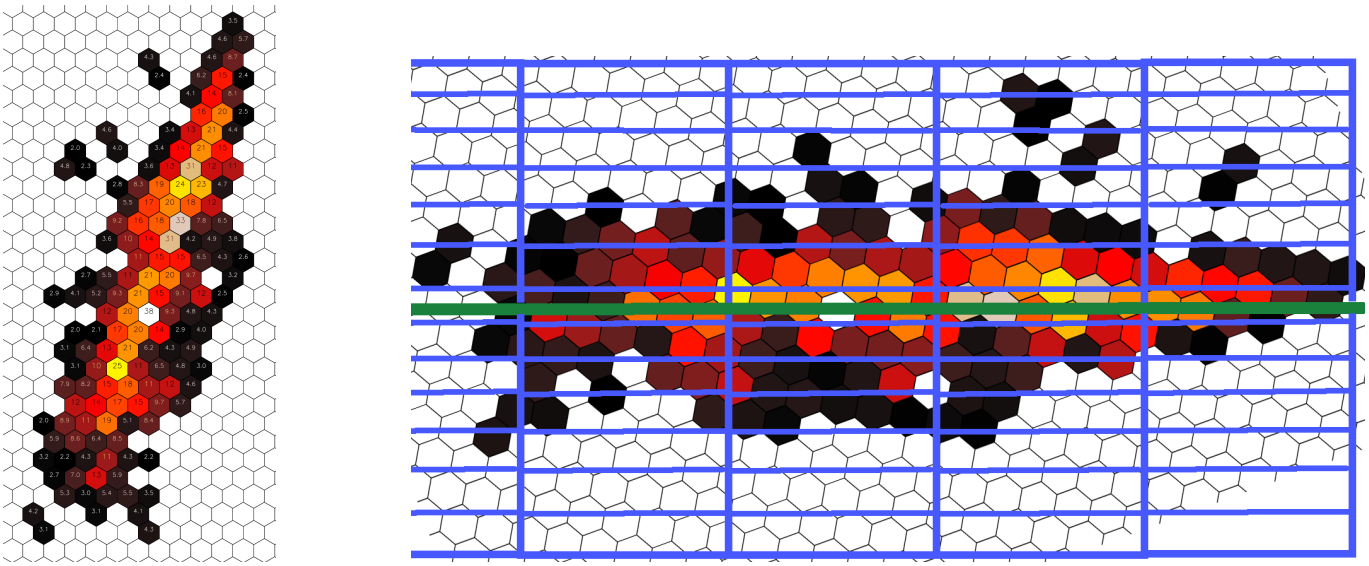


Fig. 9.38.: Schematic view of the histogramming procedure. Left: Original cleaned shower. Right: Same shower after rotation, with the bins of the histograms overlaid.

For each histogram, 3 parameters are calculated:

1. the logarithm of the sum of all bins
2. the center of gravity
3. the variance

Thus, in this example with $n = 5$ and $m = 13$, $3 \times 5 = 15$ parameters are extracted from a shower image and then fed into a machine learning algorithm. Here, a BDT (Boosted Decision Tree) was used.

Results: Figure 9.39 shows a significant improvement of the q-factor over the Hillas-based SVM-classifier for all image sizes and all fractions of signal events to be kept. For medium-sized images the q-factor is 30-40% higher. Figure 9.40 shows that for the same amount of γ-ray events kept, the amount of proton events that pass the cuts significantly drops. For medium-sized images, the background reduction is improved up to 50% compared to the Hillas-based SVM-classifier.

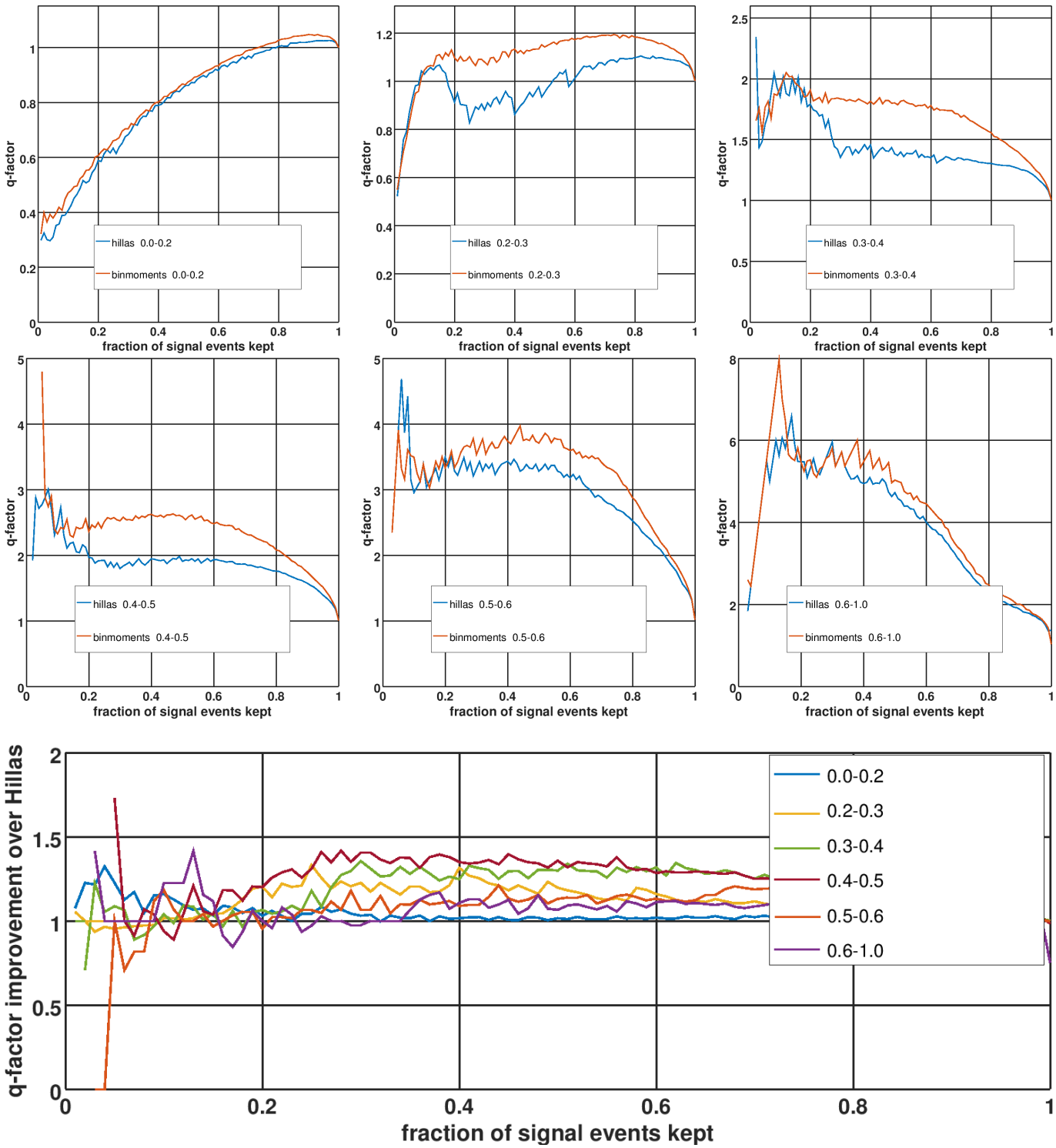


Fig. 9.39.: Top: q-factors for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and a BDT-classification using the bin moment analysis parameters described in this section. Bottom: q-factor ratio (red curve divided by blue curve) for different image sizes.

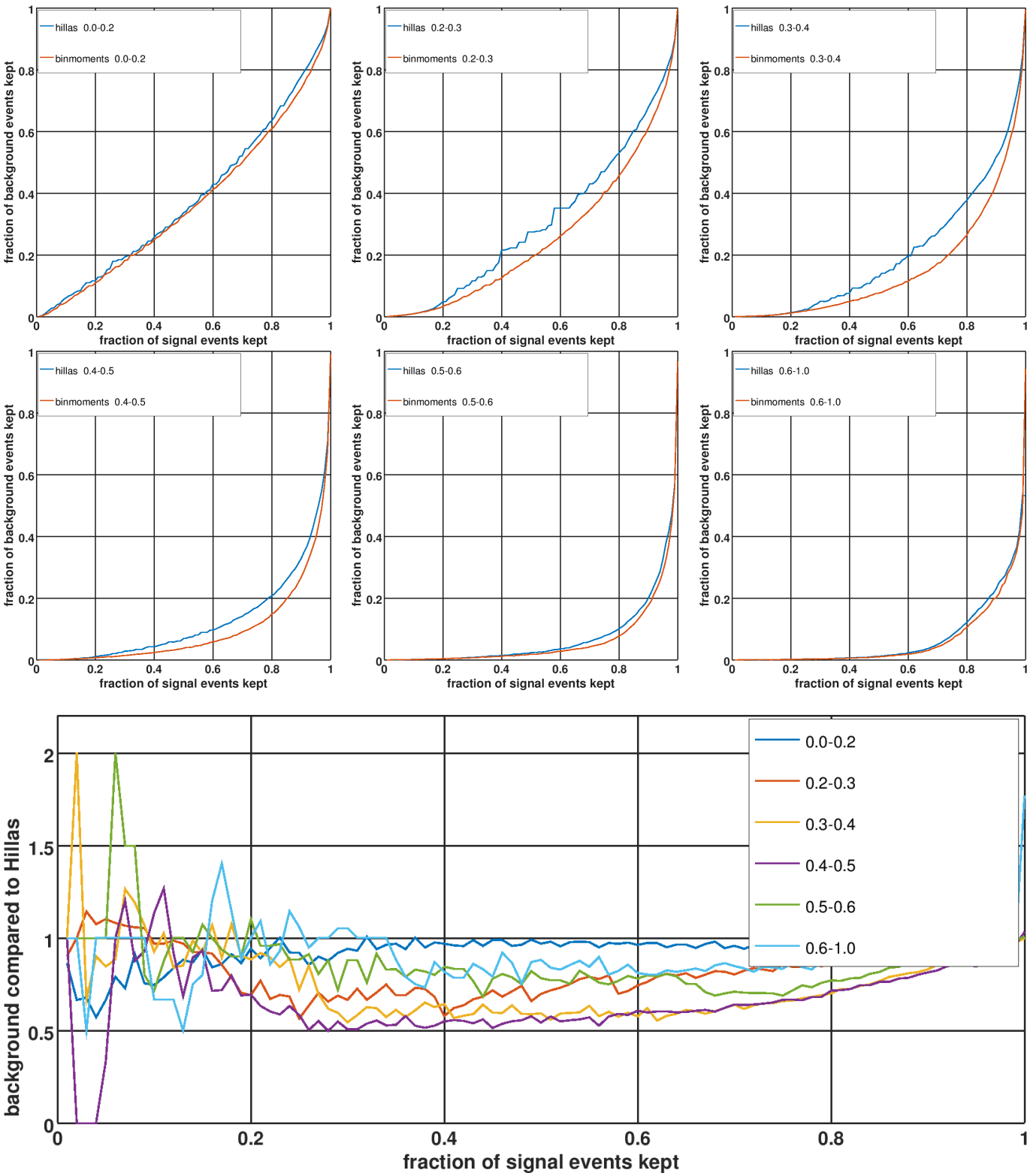


Fig. 9.40.: Top: ROC curves for different image sizes for an SVM-classification using the Hillas parameters of 4,7 cleaned showers (blue) and an BDT-classification using the bin moment analysis parameters described in this section. Bottom: ROC curve ratios (red curve divided by blue curve) for different image sizes.

9.14. Deep Neural Networks

DNNs are like normal Neural Networks, except that they are designed to work on images and usually have many more and different kinds of layers:

- The most important layer is the 'convolutional layer' (that is why DNNs are sometimes also called 'Convolutional Neural Networks'). This layer consists of a set of filters, each of which can learn a different feature of the image. Since the filter coefficients are the same for all positions of the filter in the image, the filter is translation invariant. The filter size must be defined in advance and is usually set to 3x3, 5x5 or 7x7.
- The convolutional layer is followed by a 'pooling layer', which takes the maximum of a small region (in most networks 2x2). This layer reduces overfitting and reduces the number of calculations in the next convolutional layer significantly (factor 4). Other functions than the maximum can be used, but have not proven to be as effective.
- The pooling layer is followed by a ReLU (Rectified Linear Units) layer, which applies the activation function $f(x) = max(0, x)$.

These layers are then connected in series so that combinations of features can be detected at several scales. At the end of the networks there are "normal" neural network layers, which weight the features and produce an output, in this case a binary output (γ or proton).
Since the convolutional layer produces a vector of features of the input image, it is possible to feed its output into a different machine learning algorithm, which is more powerful than the normal neural network layers, e.g. an SVM or a BDT. However, learning is not as straightforward then: First, a complete DNN (with convolutional and normal layers) would have to be trained, then the normal layers would have to be cut off, then another machine learning algorithm (like SVM or BDT) would have to be trained with the outputs of the convolutional layers. Here, only a classical DNN is used.

9.14.1. Training procedure

Since most DNN libraries cannot handle images with hexagonal pixels, the shower images are upsampled and interpolated onto a square grid. Training and testing is done separately for different image sizes (regions of interest size: 0.1-0.2, 0.2-0.3, 0.3-0.4, 0.4-0.5). Training for the largest size bin (0.6-1.0) was not possible, because there were too few images.
While the procedure is similar as for the other classification algorithms, there is one important difference: When the performance of the DNN is compared to the Hillas SVM and Hillas-multiclean SVM classification algorithms from the previous sections, these two methods perform slightly worse than in the tests before, because here, all parameters are fixed. In the previous sections, the ROC curves and q-factor plots were created by choosing the best combination of parameters (best image cleaning, best SVM parameter C, best kernel function) for each bin (see section 9.5). This means that for a fixed point on the signal axis, the lowest background value for that amount of signal was chosen in all ROC curves.
Here, a fixed image cleaning (4,7), a fixed $C = 1$ and a fixed kernel function (radial basis) was used for the SVM-Hillas classification, and that is why it performs slightly worse. For SVM-Hillas-Multiclean, the difference is not as large, because it uses different image cleaning anyway.

9.14.2. Network architecture

From the more than 100 different network architectures that have been tested, the following is one of the most powerful for γ/hadron separation:

- input layer
- convolutional layer with 32 filters of size 7 x 7 moving with step size 1 in both directions
- relu layer
- max-pooling layer of size 2x2, non-overlapping
- convolutional layer with 8 filters of size 5 x 5 moving with step size 1 in both directions
- relu layer
- convolutional layer with 8 filters of size 3 x 3 moving with step size 1 in both directions
- relu layer

- max-pooling layer of size 2x2, non-overlapping
- fully connected layer (normal NN layer) with 128 nodes
- relu layer
- fully connected layer with 32 nodes
- relu layer
- fully connected layer with 1 output node

9.14.3. Results

The performance of the described DNN for γ/hadron separation for HESS CT5 single-telescope shower images can be seen in figure 9.41. It outperforms the Hillas-SVM and the Hillas-Multiclean-SVM methods by a significant margin. With more training data, the results would probably be even better.

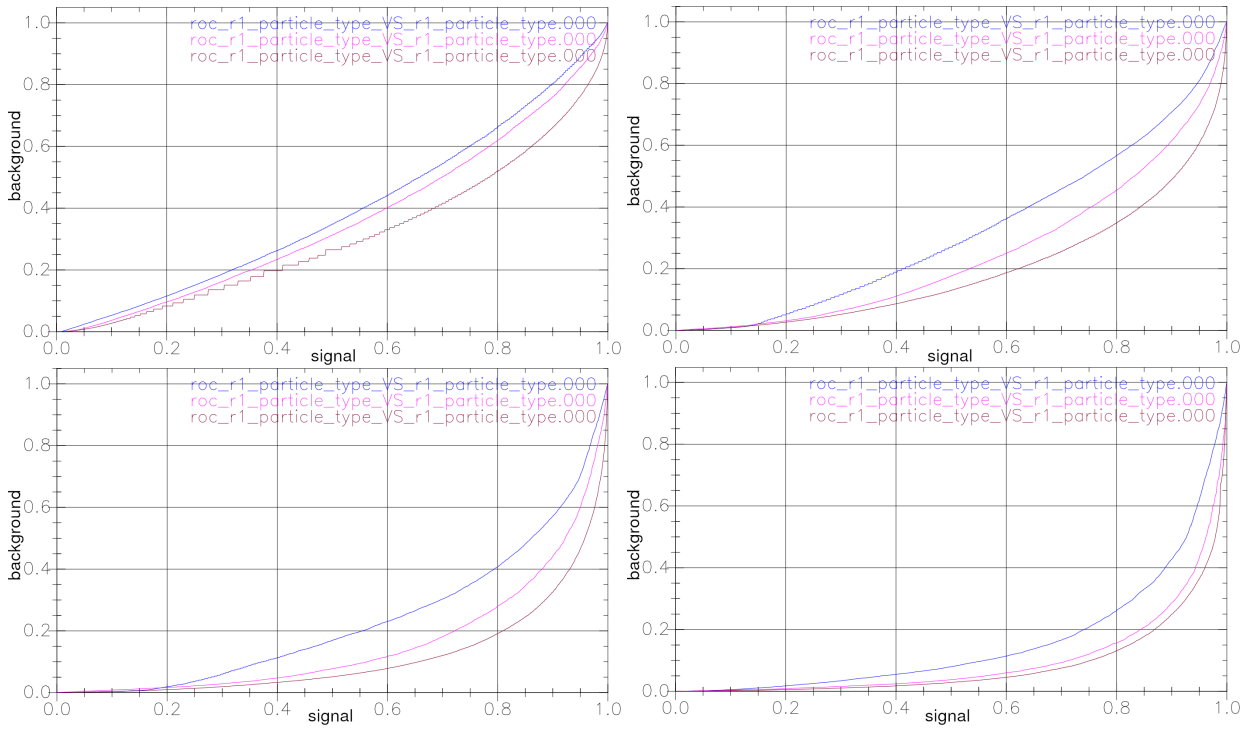


Fig. 9.41.: Top: ROC curves for ROI sizes 0.1-0.2 m (left) and 0.2-0.3 m (right). Bottom: ROC curves for ROI sizes 0.3-0.4 m (left) and 0.4-0.5 m (right).

9.15. Summary

Several methods for γ/hadron separation have been tested.

- Hillas analysis
- Hillas analysis with multiple image cleanings
- projection onto Hillas axes
- Lorentzian fit
- geometry and histogram analysis
- moment analysis in bins along the major Hillas axis
- deep neural networks

For all methods except for the DNN, the shower image was cleaned, parameterized and all parameters were concatenated and fed into an SVM or a BDT.
SVM-classification based on the Hillas parameters is very powerful and robust. Only three parameters - Hillas width, length and log(sum) - capture many of the distinguishing features of γ-rays and protons. They are easy to calculate and thus well suited for a normal analysis and an online analysis.
From the new methods, the Hillas analysis with multiple image cleanings, the moment analysis in bins along the major Hillas axis, and deep neural networks showed the best performance. For the standard analysis in MESS, the Hillas analysis with multiple image cleanings is used, because it is robust, easy to implement and it does not need as many training samples and training time as the deep neural networks.
Deep neural networks are very interesting and powerful, but they need strong strong hardware and a lot of time for training. Since most implementations of DNNs are not suited for hexagonal grids, the camera images must be interpolated onto a square grid. When these obstacles are out of the way, DNNs are very useful for classifying IACT images. In this study, they outperformed all other classification methods.

Multi-telescope events In this chapter, only single-telescope events were classified, but the extension to multi-telescope events is simple. The parameters that are calculated from the shower images by the various methods must be concatenated to a long vector and can then be fed to an SVM or a BDT. Since SVMs and BDTs can deal with empty inputs, it is not a problem if only some of the telescopes are triggered by an event and their parameterizations appear in the vector, while the rest of the elements in the vector are empty/undefined. This is done in all MESS analyses of real data and works extremely well.
For DNNs the classification of multi-telescope events is not as straightforward, but it is possible to classify each telescope event separately with the DNN and later feed all responses to an SVM or a BDT.

9.16. Appendix

9.16.1. Hillas parameters

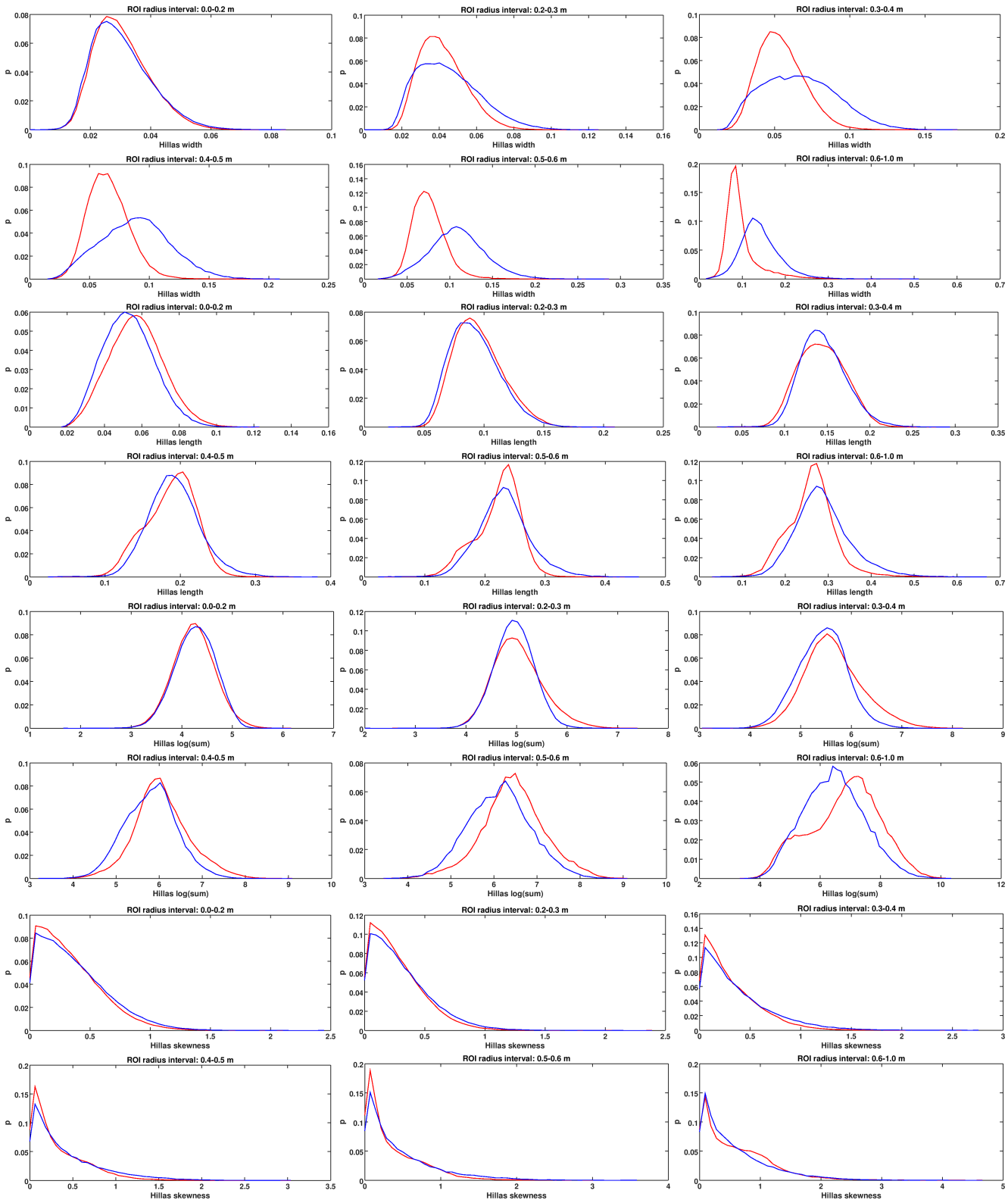


Fig. 9.42.: From top to bottom: Hillas width, length, log(sum) and skewness of gamma and proton showers for different ROI sizes (red: gamma, blue: proton).

9.16.2. Lorentzian fit errors

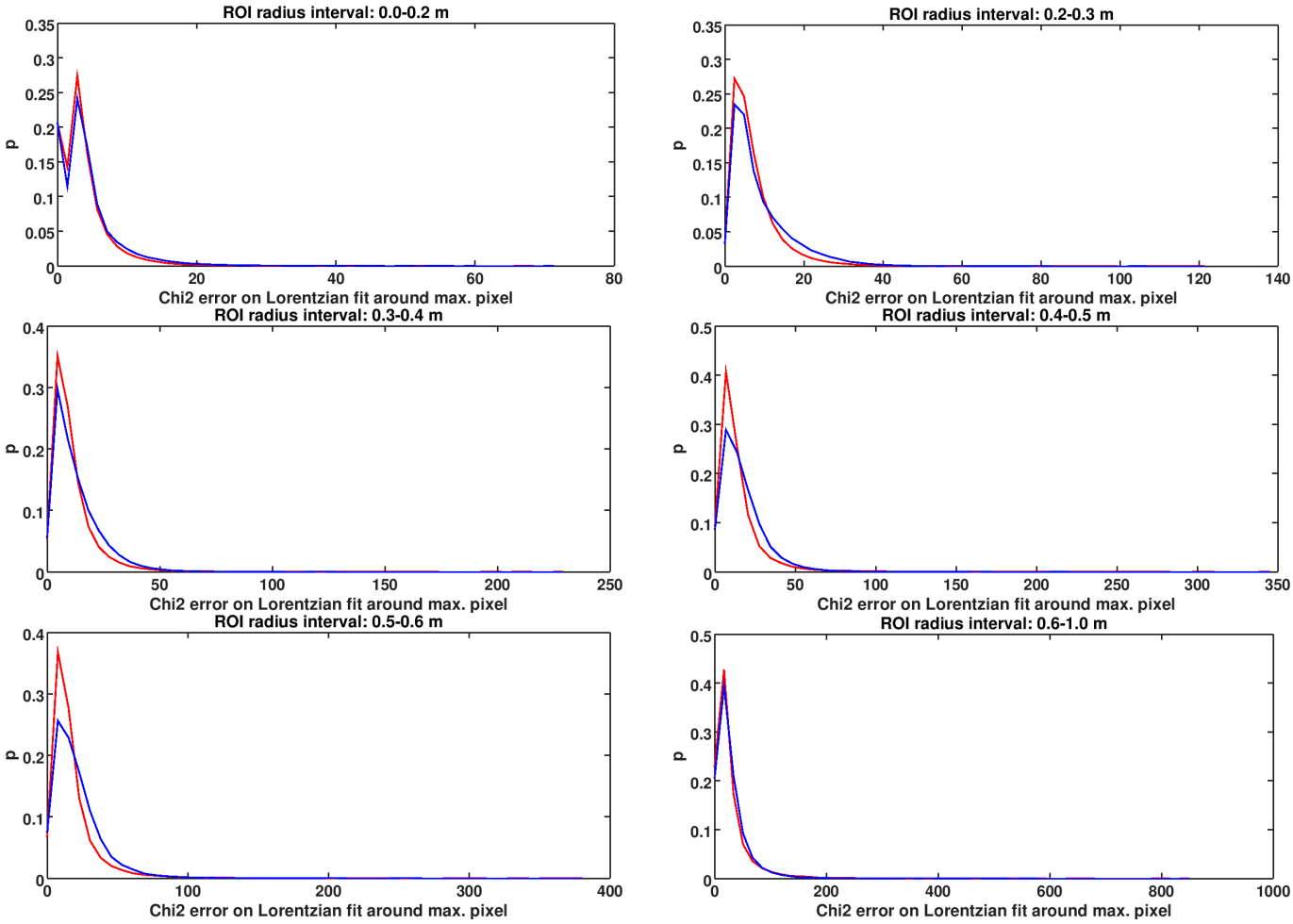


Fig. 9.43.: From top to bottom: Errors of Lorentzian fit of gamma and proton showers for different ROI sizes (red: gamma, blue: proton).

9.16.3. Distances to Hillas axes

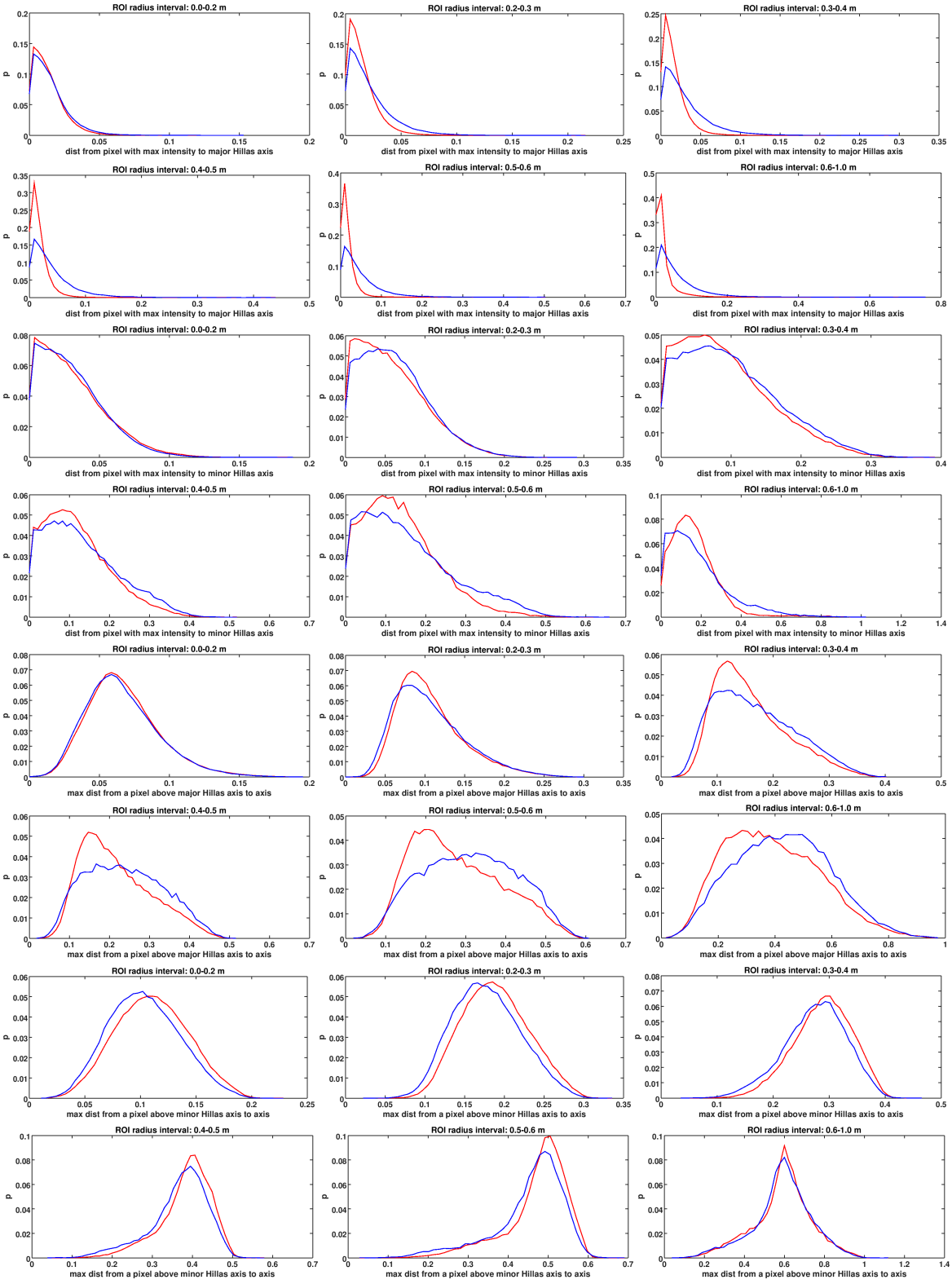


Fig. 9.44.: From top to bottom: Distance from max. pixel to major Hillas axis for different ROI sizes (red: gamma, blue: proton). Distance from max. pixel to minor Hillas axis for different ROI sizes. Distance from pixel with max. distance above the major Hillas axis to that axis for different ROI sizes. The distribution of the distance from pixel with max. distance below the major Hillas axis looks similar and is not shown here. Distance from pixel with max. distance above the minor Hillas axis to that axis for different ROI sizes. (red: gamma, blue: proton) The distribution of the distance from pixel with max. distance below the minor Hillas axis looks similar and is not shown here.

9.16.4. Distances to Hillas axes

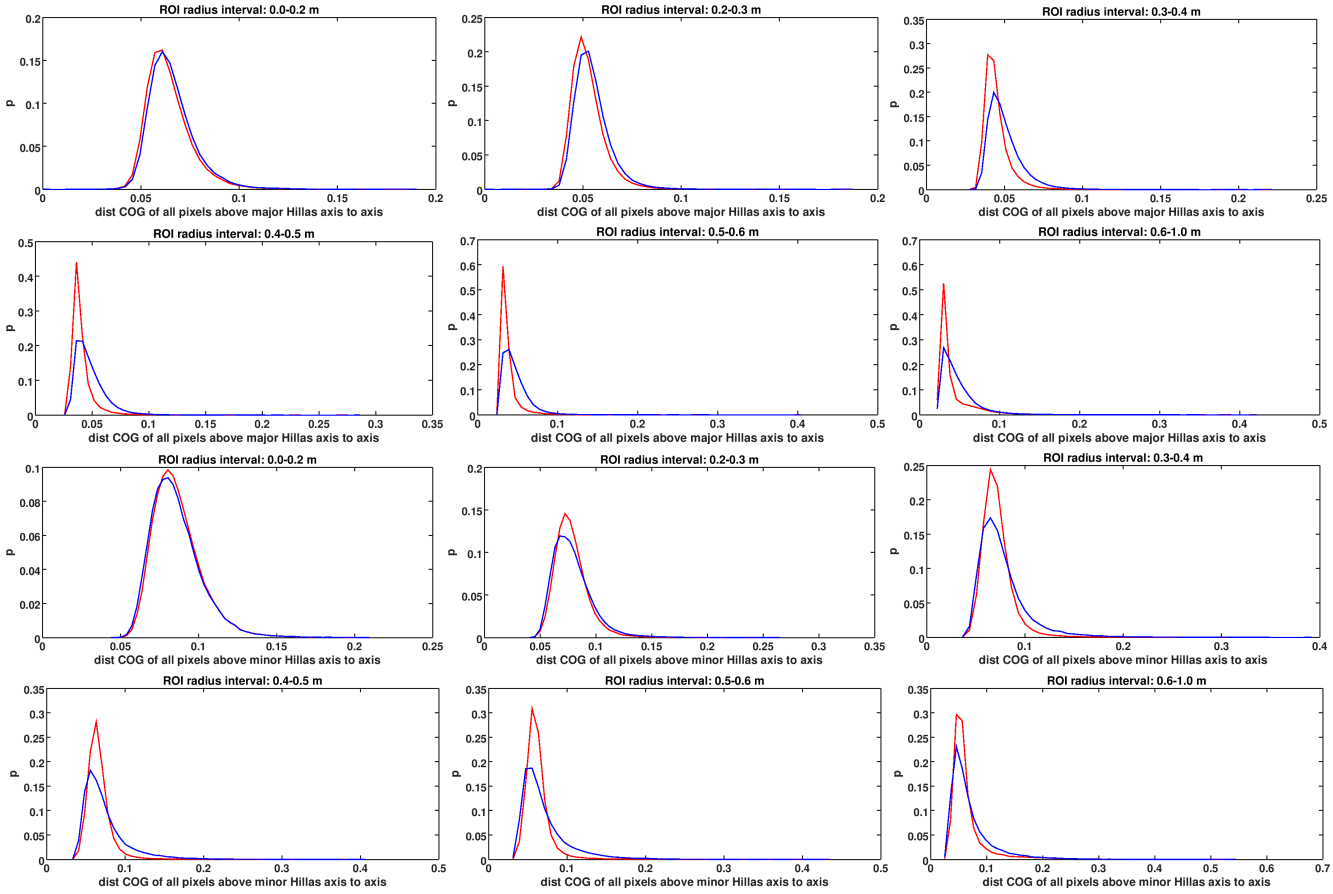


Fig. 9.45.: Top: Distance from COG of all pixels above the major Hillas axis to that axis for different ROI sizes (red: gamma, blue: proton). The distribution of the distance from COG of all pixels below the major Hillas axis looks similar and is not shown here. Bottom: Distance from COG of all pixels above the minor Hillas axis to that axis for different ROI sizes (red: gamma, blue: proton). The distribution of the distance from COG of all pixels below the minor Hillas axis looks similar and is not shown here.

9.16.5. Other distances

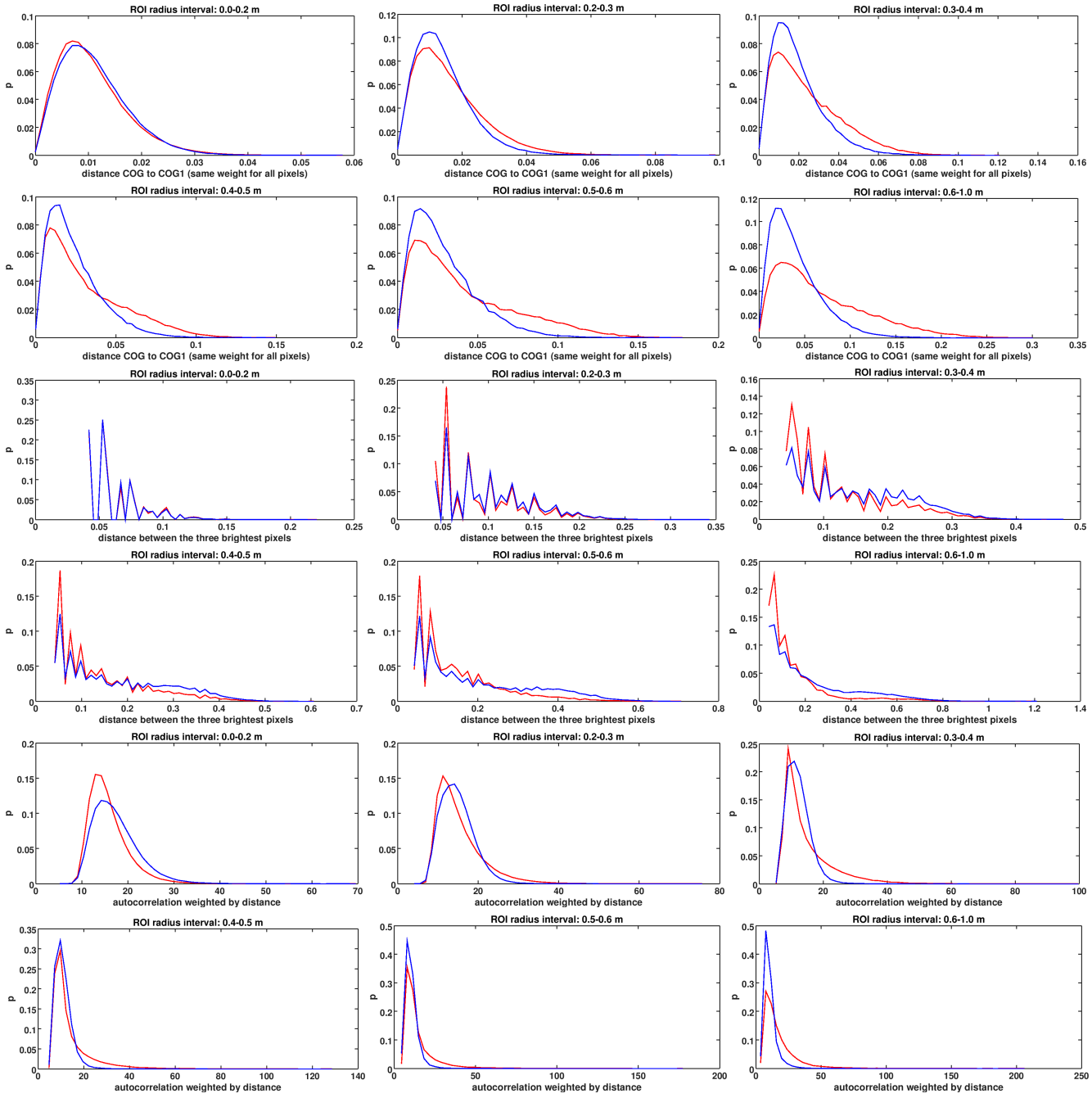


Fig. 9.46.: From top to bottom: Distance from center of gravity (1-norm) to center of gravity for different ROI sizes (red: gamma, blue: proton). Distance between the two brightest pixels for different ROI sizes. Sum of all products of all intensity pairs weighted by inverse distance for different ROI sizes.

9.16.6. Neighborhood

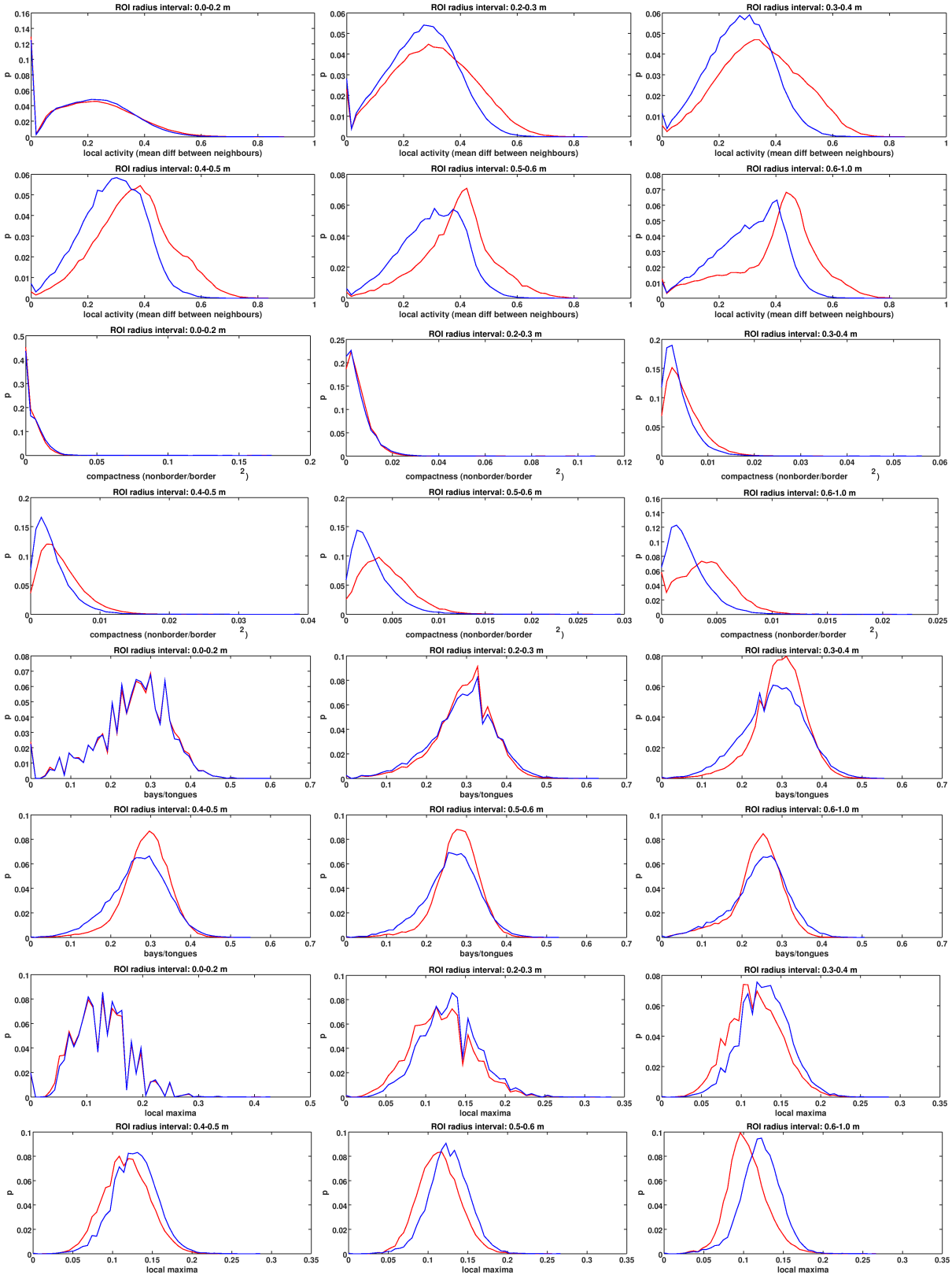


Fig. 9.47.: Local activity (mean difference between pixel and its neighbors) for different ROI sizes (red: gamma, blue: proton). Compactness of shower image (n pixels/border2) for different ROI sizes. Bays (pixels with 4 or 5 neighbors) over tongues (pixels with 1, 2 or 3 neighbors) for different ROI sizes. Local maxima for different ROI sizes.

9.16.7. Convex hull

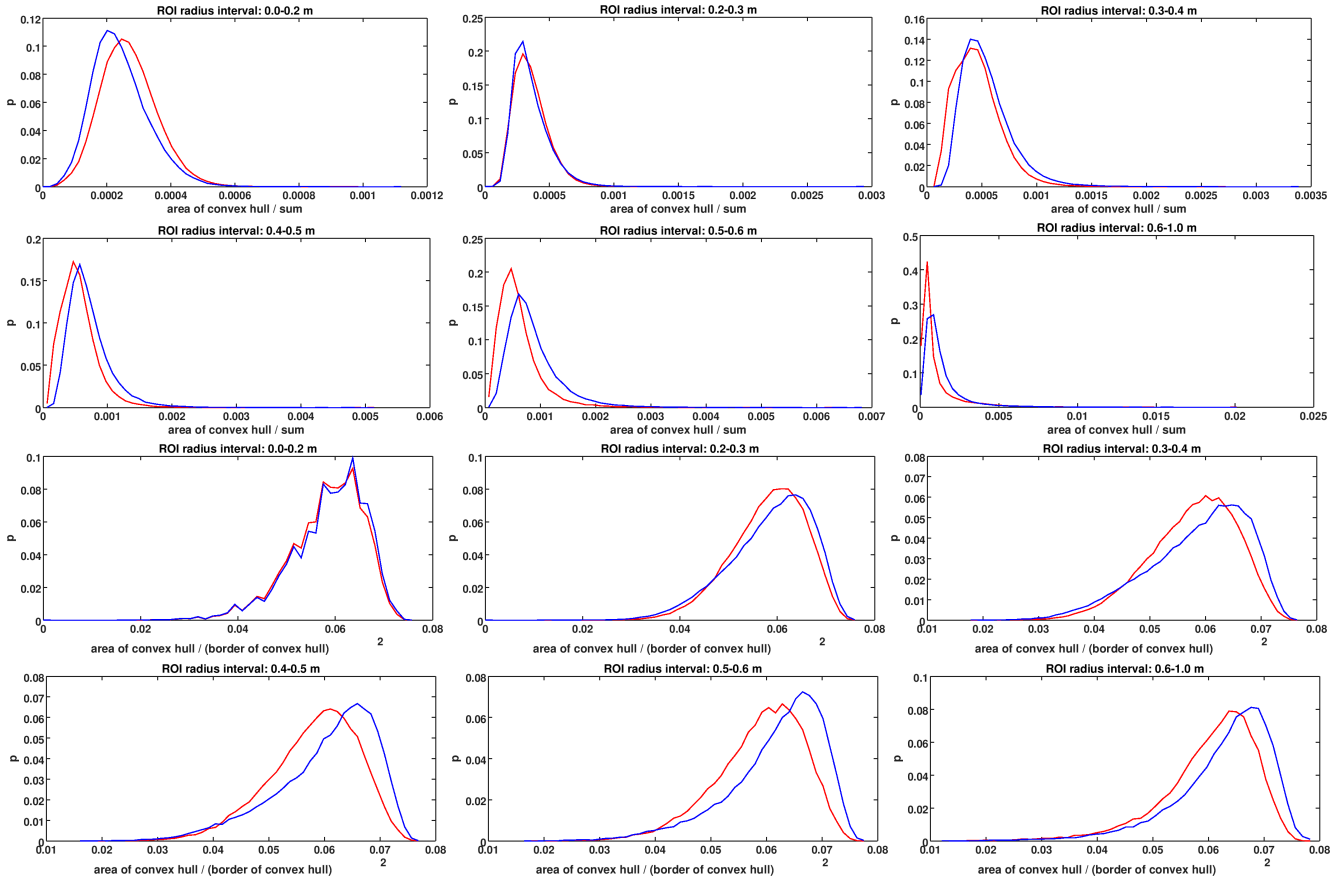


Fig. 9.48.: Top: Density of convex hull (area/amplitude) for different ROI sizes (red: gamma, blue: proton). Bottom: Compactness of convex hull (area/border2) for different ROI sizes.

9.16.8. Segmentation related parameters

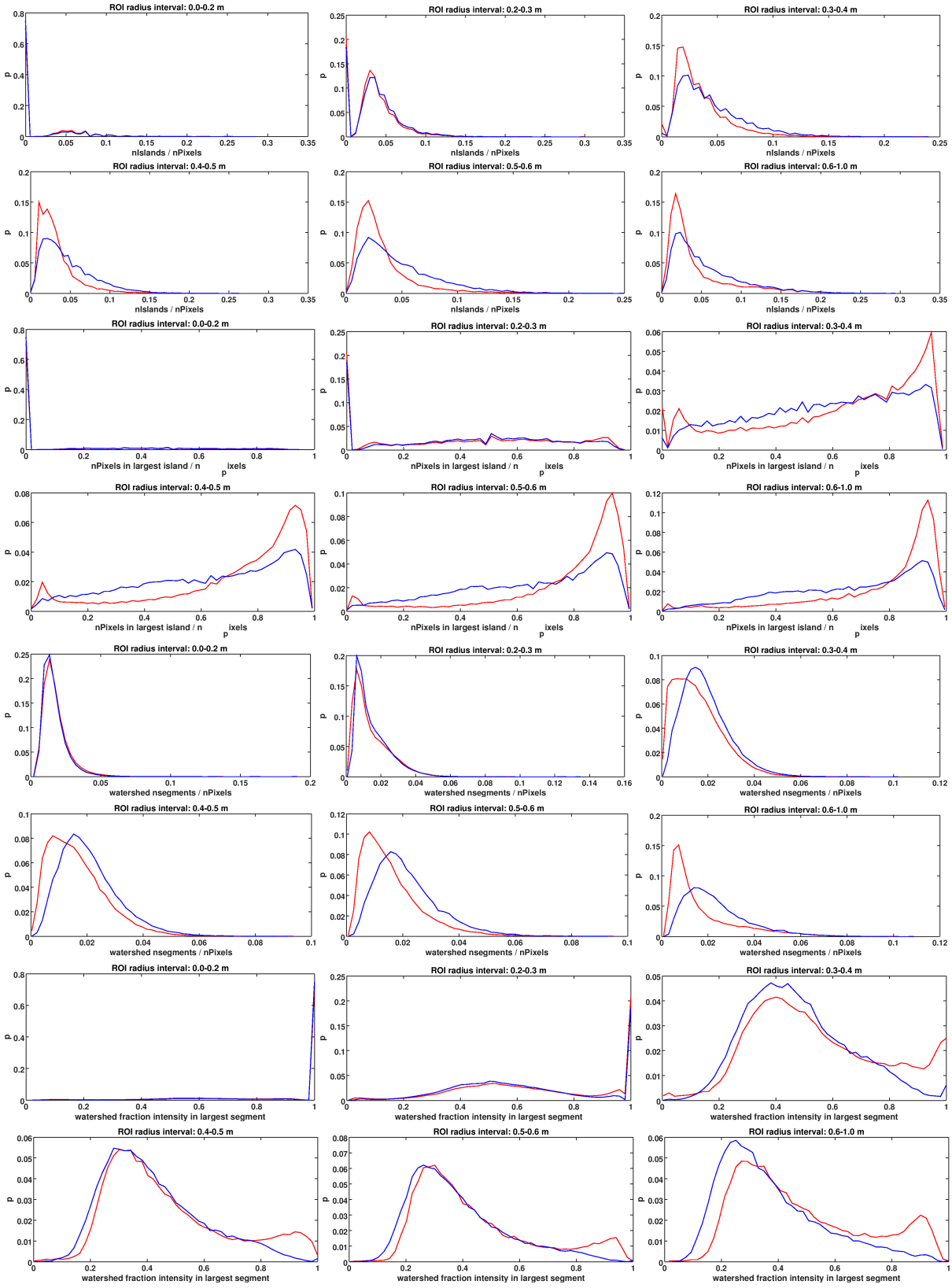


Fig. 9.49.: From top to bottom: Number of islands (connected components) in image divided by number of pixels for different ROI sizes (red: gamma, blue: proton). Number of pixels in largest island (connected component) divided by number of pixels for different ROI sizes. Watershed transform (n segments/amplitude) for different ROI sizes. Watershed transform (fraction of amplitude in largest segment) for different ROI sizes.

9.16.9. Histogram related parameters

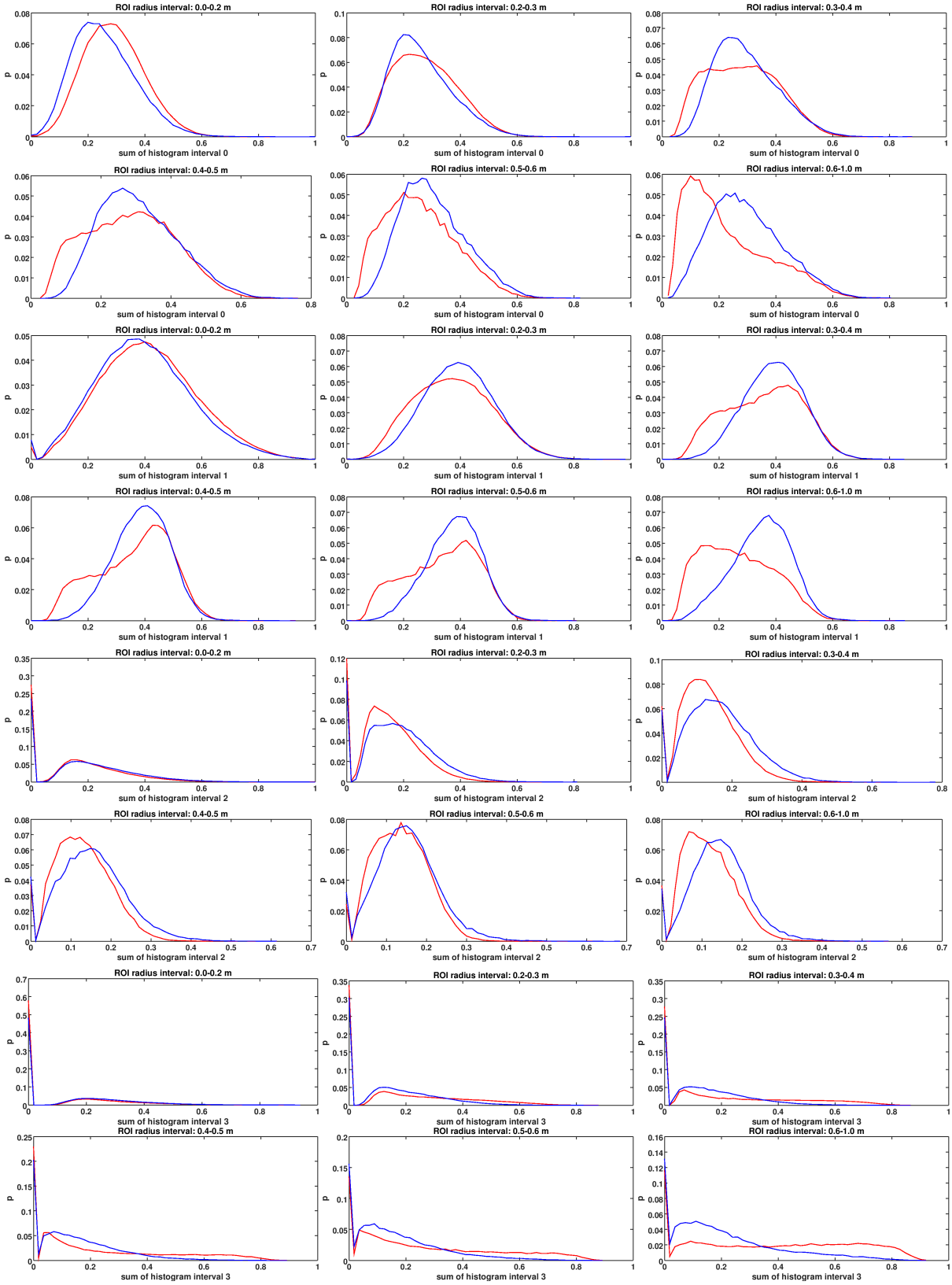


Fig. 9.50.: From top to bottom: Sum of all pixels between 1 pe and 3 pe divided by amplitude for different ROI sizes (red: gamma, blue: proton). Sum of all pixels between 3 pe and 8 pe divided by amplitude for different ROI sizes. Sum of all pixels between 8 pe and 12 pe divided by amplitude for different ROI sizes. Sum of all pixels above 12 pe divided by amplitude for different ROI sizes.

9.16.10. Density and entropy

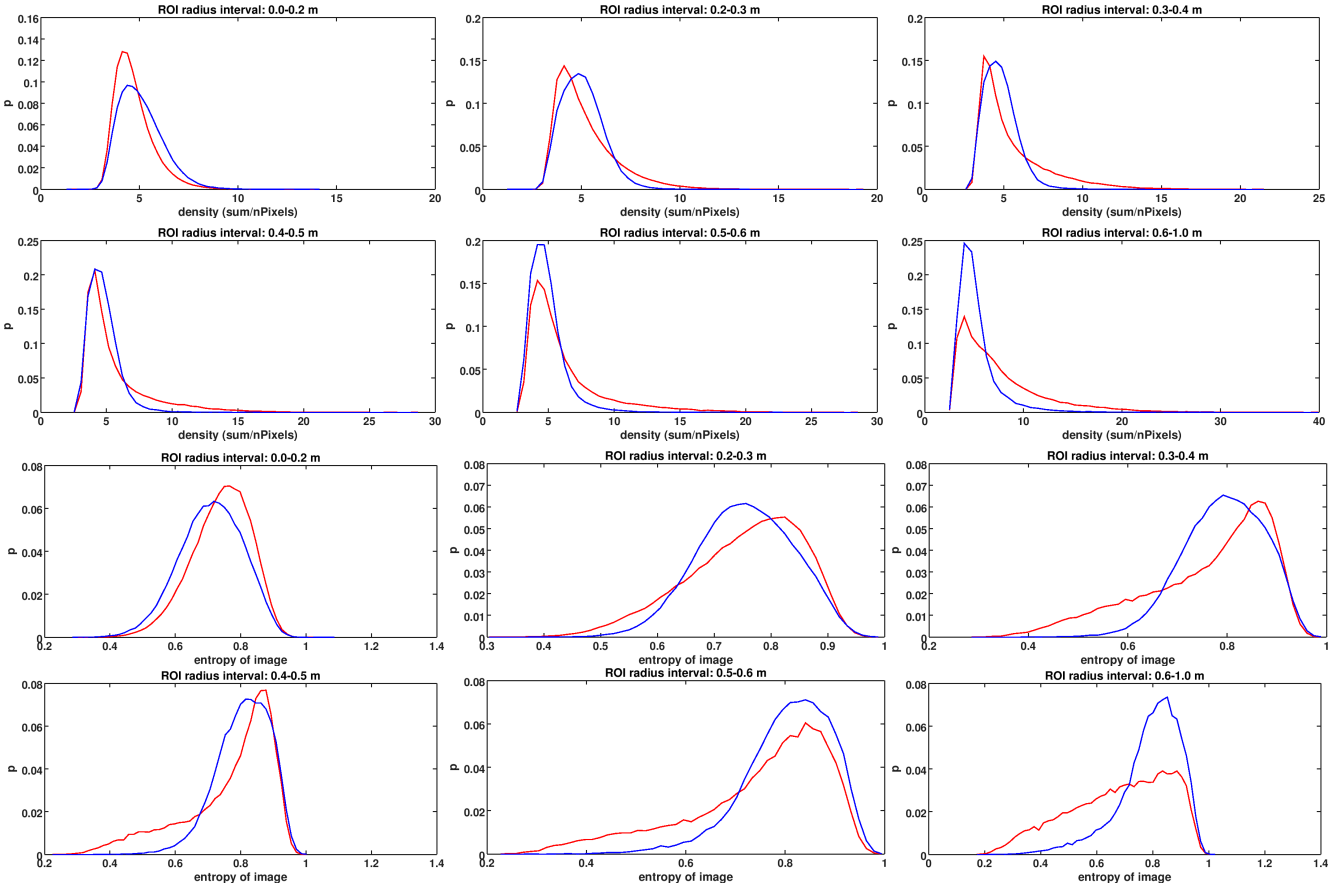


Fig. 9.51.: Top: Density (amplitude/number of pixels) for different ROI sizes (red: gamma, blue: proton). Bottom: Image entropy for different ROI sizes.

Part II.

Algorithms for the Compression of Scientific Data

10. Raw Data of Scientific Experiments

Many scientific experiments measure data and convert the analog signals to 12 bit digital data (numbers from $0 \ldots 4095$), which are then stored as 16 bit unsigned integers, because this is the smallest data type ≥ 12 bit that is supported by most computers. Usually, the measured data is noisy with only occasional signal, and the samples are strongly correlated, so the differences between samples are small.
Here, some experiments the MPIK is involved in and the data they record are described. The data shown here are not really the detector signals, but what is currently written to file after some processing.

10.0.1. GERDA

GERDA (Germanium Detector Array), located in Italy, at the Laboratori Nazionali del Gran Sasso, is an experiment for the search of neutrinoless double β-decay in Ge-76, which cannot be explained with the Standard Model. Ge-76 has a half-life of 2×10^{21} years, and when it decays, it emits two electrons and two neutrinos. The half-life of the neutrinoless decay is at least 8.0×10^{25} years[20]. In order to increase the probability of a detection, large amounts of Germanium are needed. GERDA uses 40 kg of highly purified Ge-76, and its successor, LEGEND (Large Enriched Germanium Experiment for Neutrinoless $\beta\beta$ Decay), will use 1 ton. In the neutrinoless decay, the energy of the electrons equal the decay energy, so the detector requires a very good energy resolution. The data is recorded with similar electronics as the CTA FlashCam, but is has a lower noise level and the signals look different.

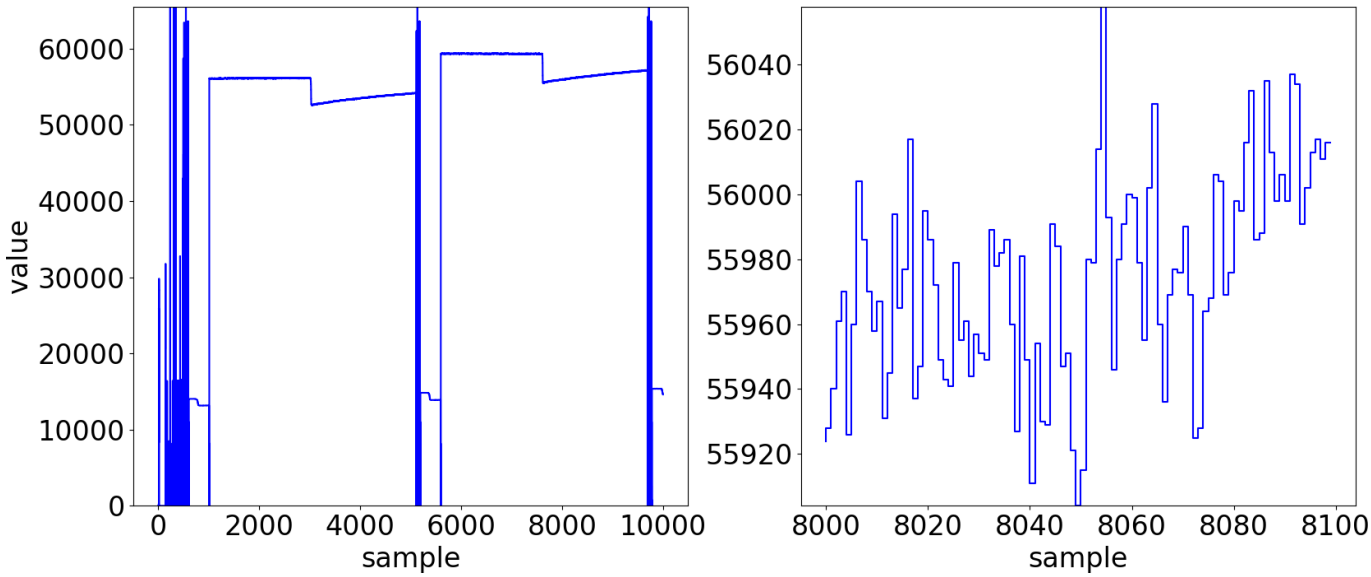


Fig. 10.1.: GERDA raw data. The left shows 200 consecutive ADC values, the right side shows an enlarged view.

10.0.2. FlashCam

FlashCam has been developed for recording γ-rays with energies from 50 GeV to 1 TeV and will be used in MSTs (Medium Size Telescopes). It has 1764 PMTs that are placed on a hexagonal grid. The photoelectron counts in a pixel are sampled. The number of samples per pixel can be chosen, but typically, all pixels provide 22 samples. It uses FlashADCs to convert analog signal to digital values. The data rate is expected to be 3 GB/s.
Real FlashCam data that was measured in the laboratory and assembled to events in the camera server is used in the compression benchmarks. Figure 10.2 shows two samples of the data, with different noise levels. The peaks (left) are the integrated signal of the low-gain channel. Such periodic outliers significantly increase the dynamic range required to store the data, and are challenging to many compression algorithms.

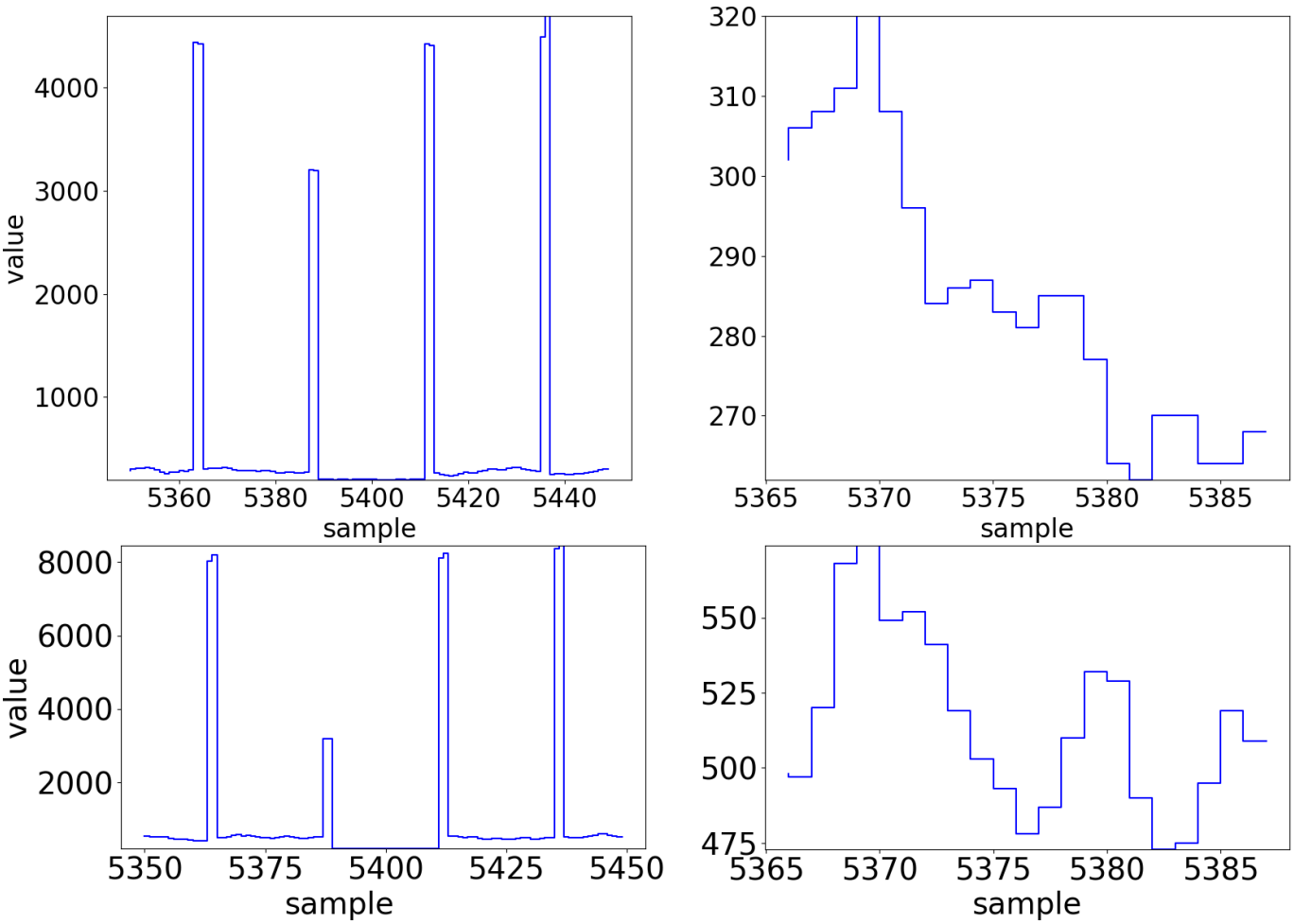


Fig. 10.2.: FlashCam data with an NSB of 300 MHz (top) and a higher NSB of 1200 MHz (bottom). The left side shows the 4×22 samples of four consecutive pixels interleaved with the integrated pixel signal. The right side shows the trace of one single pixel, without the first sample.

10.0.3. CHEC

CHEC (Compact High-Energy Camera) has been developed for recording photons with energies from 1 TeV to 300 TeV and will be mounted on SSTs (Small Size Telescopes) for the CTA experiment. It has 2048 pixels and records 128 samples per pixel with a rate of several hundred Hz. Figure 10.3 shows that CHEC raw data is very noisy, with large differences between consecutive samples, so no high compression rate should be expected.

10.0.4. HAWC

HAWC has already been described briefly in part one. It records data with 1200 PMTs. When triggered, each PMT records a long trace of almost 2000 samples. The header of a trace is an extreme outlier, and after a short signal the rest of the trace is relatively noise-free, with a low dynamic range.

10.0.5. Monte Carlo (MC) Simulations

MC simulations are essential for IACT experiments like CTA, and the data provided by simulations are very similar to real data. However, there is a peculiarity about this data set: The CTA MC data are stored in *simtel* format, which uses its own compression scheme, so compressing those files with *fc16* and other integer compression algorithms would not result in successful compression. Instead, the files were read into memory, uncompressed, and then dumped to disk without any compression. Thus, this data set contains headers, a small amount of different data structures, and mostly traces. Because it is a memory dump, the traces can be unaligned, which means that the address of an unsigned 16 bit integer is an odd number. This poses an extra challenge to the compression algorithms.

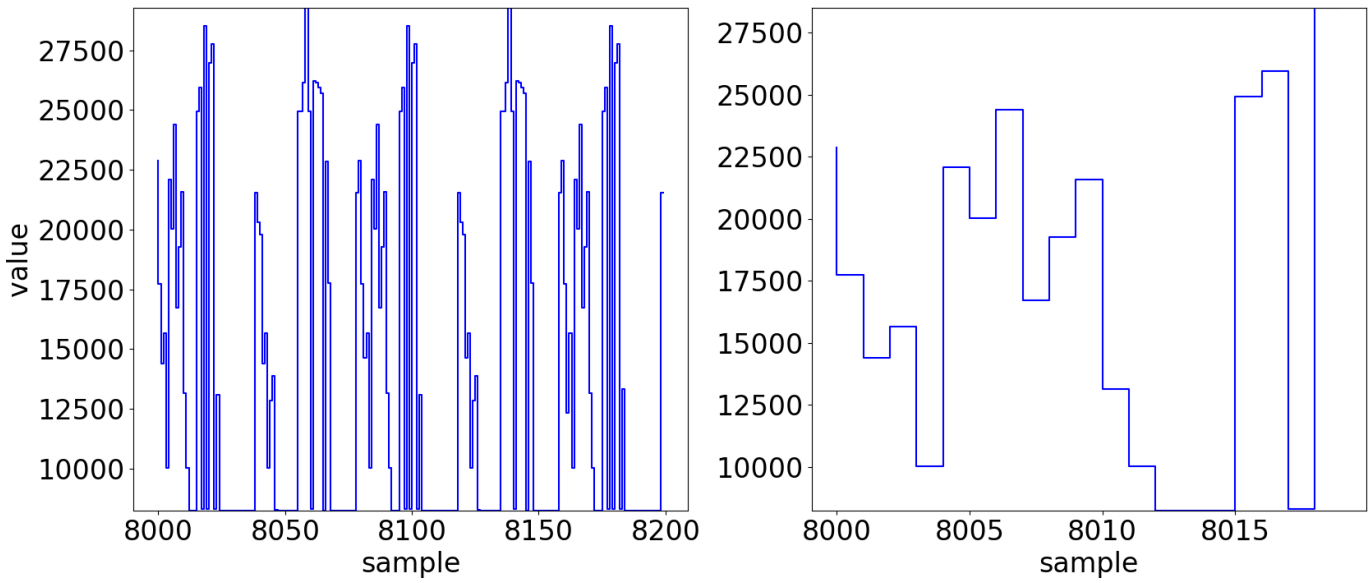


Fig. 10.3.: CHEC raw data. The left shows 200 consecutive ADC values, the right side shows an enlarged view.

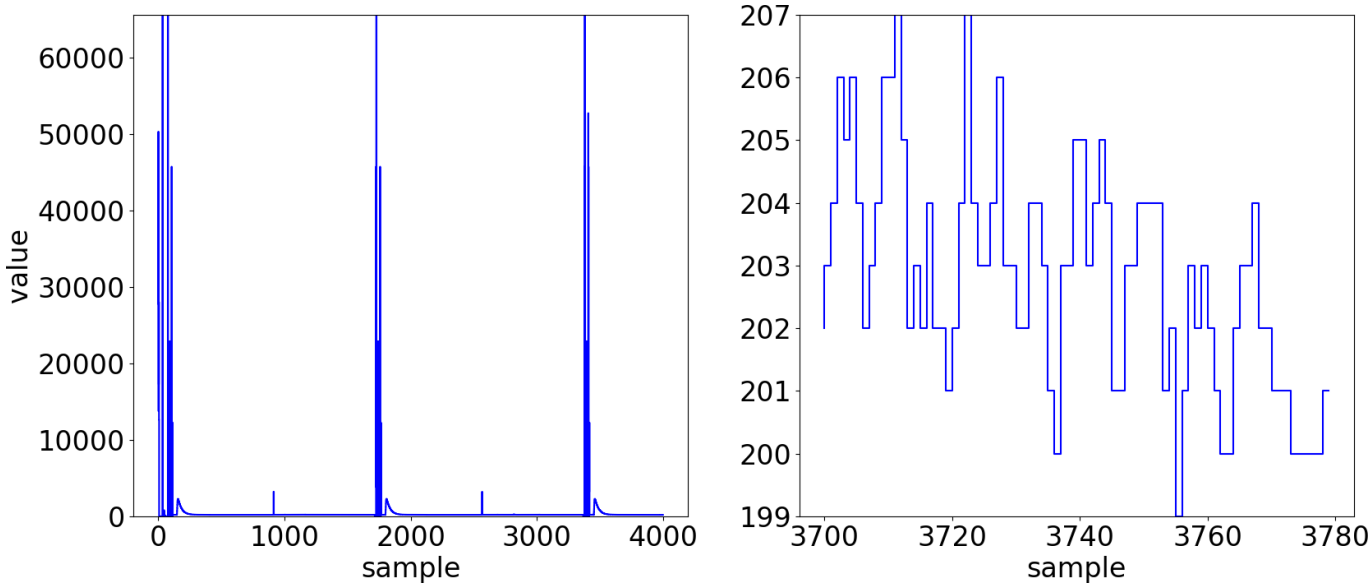


Fig. 10.4.: HAWC raw data. The left shows 200 consecutive ADC values, the right side shows an enlarged view.

However, it can be expected that most of the real data in CTA will be aligned, so the integer compression algorithms will perform better. The results on the FlashCam data are representative.

10.1. Advantages of compression

The huge data rates of today's experiments require compression, not only for saving disk space, but also for increasing I/O speed. Often, signal extraction and noise reduction cannot be done immediately, because the complete data set is needed for the calibration afterwards. And even after calibration, it is often desired to keep as much data as possible in order to re-calibrate and reanalyze the data, when errors were found in the original algorithms.
While disk space has become cheap, the time for reading and writing the data (I/O) is often the bottleneck. HDD (Hard Disks Drives) offer I/O speeds of ≈ 0.1 GB/s, and SSD (Solid State Disks) offer ≈ 1 GB/s. This is by far not enough to handle the data streams of large experiments. The solution is to install disk arrays which can achieve aggregate speeds of some GB/s. However, the larger these arrays, the more expensive they are, not to mention the network interfaces. And since in particular astroparticle physics experiments need to be far away from cities - and thus also far away from power plants, computing centers and fast telecommunication lines - the problem of efficient

data handling becomes even more important.

Compression for higher I/O speed Compression of the data can help improve the I/O speed, because the amount of data read from or written to disk decreases by factor r, which is the compression ratio. However, when optimizing a compression algorithm for a low r, the compression speed v_c and the decompression speed v_d must not be disregarded. If $r \times v$ is not well above the disk I/O speed, there might be no advantage in compression.
It is important that the compression algorithm is *lossless*, because since it is not always possible to distinguish between signal and noise during data acquisition, it would be too dangerous to use a lossy compression algorithm and perhaps loose important parts of the data.
Most general-purpose lossless compression algorithms, like gzip, achieve good compression ratios and decompression speeds, but with insufficient compression speed. During data acquisition, however, there can be time constraints due to high data rates, so high compression speeds would be very helpful.

Example 1 The data rate of input stream is 2 GB/s, the compression ratio $r = 0.3$, the compression speed 1 GB/s and the write speed of the disk is 1 GB/s. If the compression speed was fast enough, the input stream could be compressed to 0.6 GB/s and the disk could easily store it. However, since the compression speed is only 1 GB/s, it is not possible.

Example 2 The input stream and the disk are the same, but this time a different compression algorithm is used, which compresses worse ($r = 0.4$), but faster (3 GB/s). Here, the input stream is compressed to 0.8 GB/s, and since the compression speed is above the input data rate, it is possible to store it to disk. Furthermore, if the (de)compression is done quickly, there is more time for data analysis.

An algorithm that compresses 16 bit unsigned integers as well as the established compression algorithms, but orders of magnitude faster, should be of interest for the scientific community and everybody else, who is dealing with similar data. While not particularly intelligent or sophisticated, the algorithms in this chapter are very fast and achieve comparable compression ratios. Especially the last presented algorithm, `fc16`, is a significant improvement, because it surpasses all known compression algorithms in speed, while still achieving competitive compression ratios.
The four algorithms described here are a trade-off between speed and compression ratio. They do not provide the maximum possible compression, but are very fast, resulting in significantly increased I/O speed. They are based on the assumption that the input data consists of consecutive unsigned 16 bit integers, where the differences of the integers are small. And even if some of the input values are not subject to this constraint, effective compression is still possible.
After a brief description of the data used for the tests and an introduction to AVX2 follows the list of algorithms that were implemented and tested, in order of increasing sophistication. The first two algorithms are part of the prototype file format *MES*. They have been superseded by the third algorithm, which is provided as standalone program and as library. Several changes in algorithm 3 lead to algorithm 4, which is currently the best algorithm of all four.
The disadvantages of each algorithm motivated the creation of a better algorithm, Since some ideas of obsolete algorithms might become interesting again in a different context, they are all explained in some detail here.

11. Fast Lossless Compression of 16 bit Time-Sampled Waveforms

.
During the development of the MES file format, several compression algorithms for the camera raw were developed and implemented, and four of them are presented in this chapter.
The first algorithm is used in the MES file format, when event raw data is stored with option `fast`. The second algorithm is also used in the MES file format. It is used when event raw data is stored with option `small`. Later, it was tried to use these two algorithms also for data of other experiments, but it was found that the algorithms lack generality.
The third algorithm was the first attempt to compress arbitrary 16 bit data. It is not used anywhere, but it introduces new concepts and is the basis for the fourth algorithm, `fc16`.

11.1. Storing Blocks with Reduced Range of 4, 8 or 16 bits

An algorithm for the compression of unsigned 16 bit integers using AVX2 has been developed (see listings 11.1 and 11.2) and is presented here. It first computes the minimum of a block of 16 values, stores it and subtracts it from all values. Then, it checks how the 16 residual values can be encoded:

- If they can be encoded using only 4 bits for each value, which means that they all have to be ≤ 15, it is marked in the header and the 16×4 bits are written.
- If some values are > 15, it does the same range check for 8 bit, and if all values are ≤ 255, it is marked in the header and 16×8 bits are written.
- If some values are > 255, it is marked in the header and 16×16 bits are written.

Encoding the correct range takes up 2 header bits, and since the minimum is also needed, additional 16 bits need to be stored. However, quantizing the minimum to 2 bit steps (which means that the 2 lowest significant bits are set to zero) reduces the number of bits needed to be stored to 14 bit. Combined with the 2 range bits, this results in a 16 bit header. The residual numbers then use 64, 128 or 256 bits, so the possible compression ratios are $\frac{16+64}{256} = 0.31$, $\frac{16+128}{256} = 0.56$ and $\frac{16+256}{256} = 1.06$.

Example

$$v = (92, 90, 103, 210, 99, 101, 103, 101, 100, 89, 92, 91, 95, 97, 101, 100) \tag{11.1}$$
$$m = 89 \Rightarrow q = 88 \tag{11.2}$$
$$v - q = (4, 2, 15, 122, 11, 13, 15, 13, 12, 1, 4, 3, 7, 9, 13, 12) \tag{11.3}$$
$$r = (2, 0, 4, 7, 4, 4, 4, 4, 4, 4, 2, 1, 3, 3, 4, 4) \tag{11.4}$$

Here, v is the input vector, m is the minimum, q the minimum quanitized to 2 bits, and r is the required range in bits. Since after subtraction of the minimum, all values are ≥ 0 and none exceeds the 16 bit range, they can be safely stored in an unsigned 16 bit integer.

11.1.1. Benchmark

The file format and the I/O library were benchmarked with CTA MC prod 3 proton simulations:
`proton_20deg_180deg_run13*___cta-prod3-demo_desert-2150m-Paranal-demo2sect.simtel`

The full image information in the `.simtel` files was converted to the `.mes` format, including all events, all pixels and all samples, but omitting all auxiliary information (photon bunches, event parameters etc.). A single file was then created by concatenating all converted `.mes` files. Without compression, the file has a size of 16.74 GB, and with the built-in `fast` compression applied, the file size is reduced to 7.55 GB. The contents of the file are:

events	2.044×10^5
telescope events	6.612×10^5
pixels	1.212×10^8
samples	8.376×10^9

In CTA prod 3, most telescopes have two gain channels and 22, 60 or 128 samples per pixel, and the telescope with many pixels are gone. The test system is an `Intel(R) Core(TM) i5-6600T CPU @ 2.70GHz` (35W low power) with 32 GB RAM and a `512 GB Samsung 950 Pro` SSD that reads with 2.5 GB/s and writes with 1.5 GB/s. First, the caches were dropped:

```
echo 3 > /proc/sys/vm/drop_caches
```

to be sure that the files were really read from disk, and a part of the benchmark done. Then, the same part was repeated immediately afterwards, with the file still in memory. The two numbers are given in table 11.1, the result of the test with the data in the cache is written in brackets. For the read test, events were just read and decoded into the memory structures. Pure writing/encoding speed is difficult to measure, because it is a streamed file format and only one event at a time is kept in memory. So for the write test, in order to disentangle read and write performance, the `event_write` routine was put inside a loop and executed 20 times for each event, then the elapsed time was divided by 20. To obtain the I/O speed in units of uncompressed raw bytes per second, the number of samples/s needs to be multiplied by 2. This benchmark shows that with a single core, the I/O per televent can probably be handled fast enough for any stage in the pipeline. But it is unlikely that much additional I/O can be done with the remaining cores. Note: The file format performs equally well on HESS raw data, HESS calibrated data and HESS MC. I/O speeds are more than 10 times higher than with the HESS software and files are 2 times smaller. HESS raw data is not sampled, but the compression algorithm also works on the scalar ADC values.

Compression ratio	0.45	
Read speed	3522 (4926)	Events/s
Read speed	11391 (15910)	Televents/s
Read speed	1443 (2015)	Megasamples/s
Write speed	2708	Events/s
Write speed	8758	Televents/s
Write speed	1110	Megasamples/s
Read+Write speed	1303 (1879)	Events/s
Read+Write speed	4210 (6070)	Televents/s
Read+Write speed	534 (769)	Megasamples/s
Memory usage	50	MegaBytes
Cores used	1	

Table 11.1.: Compression ratio, compression and decompression speeds of MES file format and CTA MC prod 3 simulations. The values in brackets are obtained when the data is cached by the OS.

Summary This algorithm is used in the MES file format (see 4) when event raw data is compressed with option `fast`. It is fast (> 4 GB/s), but very primitive and sensitive to outliers, because even one single large number can force the entire block of 16 numbers to be encoded with more bits. Also, even if the residuals are very small, at least 4 bits are always used for each value. Lastly, since channels that have the minimum value are stored twice (first as minimum in the header and later as 0 in the residuals), space is wasted.

```
max4 = _mm256_set1_epi16(16);
max8 = _mm256_set1_epi16(256);
mod4 = _mm256_set1_epi16(65532);
x = _mm256_loadu_si256((__m256i const *) &v[p]); // load v
min = _mm256_min_epu16(x, _mm256_alignr_epi8(x, x, 2)); // find minimum
min = _mm256_min_epu16(min, _mm256_alignr_epi8(min, min, 4));
min = _mm256_min_epu16(min, _mm256_alignr_epi8(min, min, 6));
min = _mm256_min_epu16(min, _mm256_alignr_epi8(min, min, 8));
min = _mm256_min_epu16(min, _mm256_permute2x128_si256(min, min, 1));
min = _mm256_and_si256(min, mod4); // remove the two lowest significant bits.
x = _mm256_subs_epu16(x, min); // subtract minimum
unsigned short minval = _mm256_extract_epi16(min, 0);
if (_mm256_movemask_epi8(_mm256_cmpgt_epi16(max4, x)) == -1) { // each fits into 4 bits
    *(unsigned short *) &BUF[BP] = minval | 1;
    BP += 2;
    x = _mm256_packus_epi16(x, x);              // x = 00 0a.00 0b.00 0c.00 0d
    x = _mm256_permute4x64_epi64(x, _MM_SHUFFLE(3,1,2,0)); // x = 0a.0b.0c.0d
    y = _mm256_slli_epi16(x, 4);                           // x = 0a 0b.0c 0d
    y = _mm256_srli_epi16(y, 8);                           // y = a0 b0.c0 d0
    x = _mm256_and_si256(_mm256_set1_epi16(15), x);        // y = 00 a0.00 c0
    x = _mm256_or_si256(x, y);                             // x = 00 0b.00 0d
    x = _mm256_packus_epi16(x, x);                         // x = 00 ab.00 cd
    x = _mm256_permute4x64_epi64(x, _MM_SHUFFLE(3,1,2,0)); // x = ab.cd.
    _mm256_storeu_si256((__m256i *) &BUF[BP], x); // we stored nv*0.5 bytes
    BP += (l + (l%2)) / 2; // we don't want to loose the half-filled byte
} else if (_mm256_movemask_epi8(_mm256_cmpgt_epi16(max8, x)) == -1) {// each 8 bits
    *(unsigned short *) &BUF[BP] = minval | 2; BP += 2;
    x = _mm256_packus_epi16(x, x);
    x = _mm256_permute4x64_epi64(x, _MM_SHUFFLE(3,1,2,0));
    _mm256_storeu_si256((__m256i *) &BUF[BP], x); // we stored nv*1 byte BP += l;
} else { // each fits into 16 bits
    *(unsigned short *) &BUF[BP] = minval | 3; BP += 2;
    _mm256_storeu_si256((__m256i *) &BUF[BP], x);  // we stored 16*2 bytes BP += 2*l;
}
```

Listing 11.1: Compression algorithm that is used in the MES file format.

```
min = _mm256_set1_epi16(gmin);
switch (b) {
    case 1:
        x = _mm256_loadu_si256((__m256i const *) &BUF[BP]);
        x = _mm256_cvtepu8_epi16(_mm256_extracti128_si256(x, 0));
        y = _mm256_andnot_si256(_mm256_set1_epi16(15), x);
        x = _mm256_and_si256(_mm256_set1_epi16(15), x);
        y = _mm256_slli_epi16(y, 4);
        x = _mm256_or_si256(x, y);
        x = _mm256_cvtepu8_epi16(_mm256_extracti128_si256(x, 0));
        BP += (l + (l%2)) / 2; // we don't want to loose the half-filled byte
        break;
    case 2:
        x = _mm256_loadu_si256((__m256i const *) &BUF[BP]);
        x = _mm256_cvtepu8_epi16(_mm256_extracti128_si256(x, 0)); BP += l;
        break;
    case 3:
        x = _mm256_loadu_si256((__m256i const *) &BUF[BP]); BP += 2*l;
        break;
}
x = _mm256_adds_epu16(x, min);
_mm256_storeu_si256((__m256i *) &v[p], x);
```

Listing 11.2: Decompression algorithm that is used in the MES file format.

11.2. Integer Wavelet Transform with Fixed Huffman Residual Coding

If the distribution of the input data is known, an optimized dictionary can be created using Huffman coding, which assigns a distinct bit string to each number, with length inversely proportional to the frequency of the number[31]. However, most of the time it is better to first predict and approximate the next sample based on the surrounding samples and only then use the Huffman encoder on the difference between predicted and true value.

11.2.1. Differential coding

Differential coding, which is the simplest prediction, assumes that the next sample is equal to the current sample, so the residuals between predicted and true values are the differences between successive samples.

Example Consider a vector v and its differences $d_i = v_{i+1} - v_i$:

$$v = (0, 0, 2, 2, 4, 6, 8, 10, 9, 8, 7, 6, 8, 8, 7, 9, 7, 9, 7, 8, 10, 8, 7, 5, 7, 9, 10, 11, 13, 15, 15, 17)$$
$$d = (0, 2, 0, 2, 2, 2, 2, -1, -1, -1, -1, 2, 0, -1, 2, -2, 2, -2, 1, 2, -2, -1, -2, 2, 2, 1, 1, 2, 2, 0, 2)$$

While v has a large spread, d has not (see figure 11.1).
Furthermore, the number 2 is so frequent that a possible Huffman encoding could be to assign the bit string `0` to the number 2, and the bit strings `100`, `101`, `110` and `111` to the four remaining numbers -2, -1, 0 and 1 (see figure 11.1).

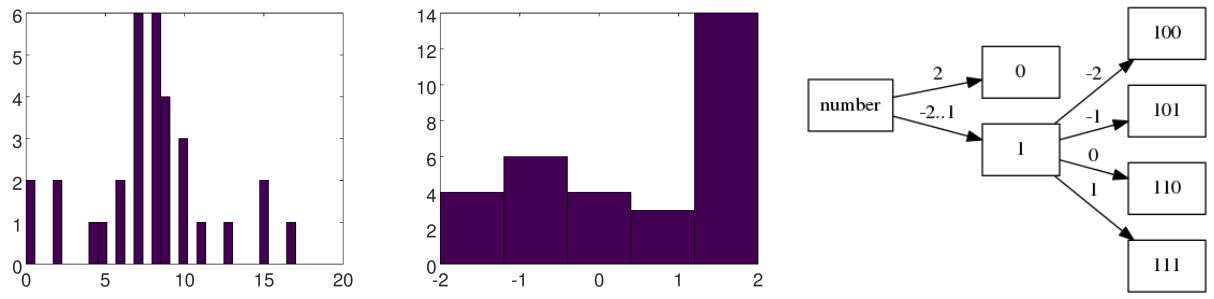


Fig. 11.1.: Left: histogram of values v. Center: histogram of differences d. Right: Huffman tree for encoding the differences.

The vector v can then be encoded by storing the first number as offset, followed by the differences d, and the resulting bit string will be:

$$00000000.110.0.110.0.0.0.0.101.101.101.101.0.110.101.0.100.0.100.111.0.100.101.$$
$$\hookrightarrow 100.0.0.111.111.0.0.110.0$$

The dots simply show the separation between numbers, but they do not have any meaning and do not exist in the bit stream. However, if the data is noisier or more complex, the residuals produced by differential coding become large, and it is better to increase the window size to more neighboring samples to obtain a better prediction for the next sample.

11.2.2. Integer Wavelet Transform

The Fourier transform can represent any signal as a linear combination of frequencies, so for harmonic signals this is the ideal representation. However, Fourier coefficients are not limited to a certain time window, so when applied to data that exhibits sharp edges or aperiodic signals, many Fourier coefficients are needed to represent the data correctly. Furthermore, the runtime is of order $\mathcal{O}(n \log n)$ and computing the sines and cosines is expensive. The Discrete Wavelet Transform on the other hand captures both frequency and location in time and can be implemented in $\mathcal{O}(n)$. It is given by:

$$W_{ji} = \int_{-\infty}^{\infty} \psi_{ji} v(t) dt \tag{11.5}$$

Here, ψ_{ji} is the mother-wavelet with scaling parameter j and positional parameter i:

$$\psi_{ji} = \frac{1}{\sqrt{2^j}} \psi \left(\frac{t - i2^j}{2^j} \right) \tag{11.6}$$

This transform decomposes the signal into sub-sampled and detail information and can be implemented as a filter bank[32], as shown in figure 11.2: The input v is split into even and odd samples v_{2i} and v_{2i+1}. Some of the even samples around v_{2i} are used to predict v_{2i+1}, and the difference between prediction and actual value becomes the detail information d_i. Some of these differences around v_{2i+1} are then used to update the even samples, and this value becomes the sub-sampled information s_i. This process is called *lifting*.

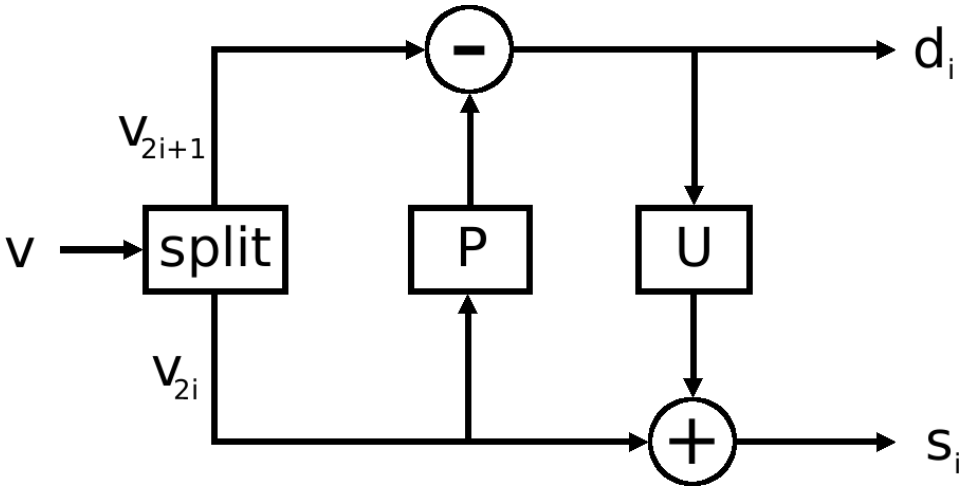


Fig. 11.2.: Prediction-update lifting scheme.

Implementation

Unfortunately, it is not easily possible to parallelize, because of the dependency on the previously transformed values. Border problems must also be handled for this algorithm, and implementing them in AVX2 is difficult and does not pay off, because of the small window size of 16 samples. So instead of AVX2 code conventional C code is used, which is still fast here, because the wavelet transform that is used in this algorithm is very simple (see figure 11.3) and involves only few instructions:

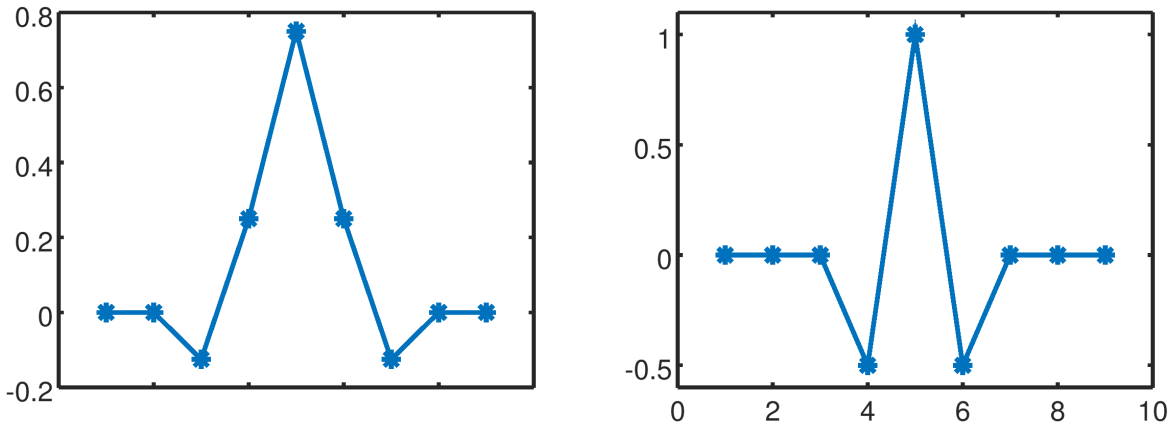


Fig. 11.3.: The wavelet used in algorithm 2. Left: Scaling function. Right: Wavelet function.

$$d_i = v_{2i+1} - \frac{v_{2i} + v_{2i+2}}{2} \tag{11.7}$$

$$s_i = -\frac{v_{2i-2}}{8} + \frac{v_{2i-1}}{4} + \frac{3}{4}v + \frac{v_{2i+1}}{4} - \frac{v_{2i+2}}{8} \tag{11.8}$$

Using the lifting scheme for this wavelet transform [33], the transform can be simplified to:

$$d_i = v_{2i+1} - \frac{v_{2*i} + v_{2i+2}}{2} \tag{11.9}$$

$$s_i = v_{2i} + \frac{d_{i-1} + d_i}{4} \tag{11.10}$$

Since the data are integers, the division must be done with care:

$$d_i = v_{2i+1} - \lfloor \frac{v_{2i} + v_{2i+2}}{2} + \frac{1}{2} \rfloor \tag{11.11}$$

$$s_i = v_{2i} + \lfloor \frac{d_{i-1} + d_i}{4} + \frac{1}{2} \rfloor \tag{11.12}$$

Fortunately, bit shifts behave exactly as required, so they are used. Ignoring the border problems for now, this leads to the following C code for the transform:

```
for (i = 2; i < nv; i+=2) {
    d0 = d;
    s0 = s;
    d = v[i-1] - ((v[i-2] + v[i]) >> 1);
    s = v[i] + ((d0 + d) >> 2);
    encode(mem, d);
    encode(mem, s-s0);
}
```

The detail part d of the transformed signal already is a small number, but the sub-sampled signal s is not. By applying the algorithm again, half of s could be turned into details and the other half into a sub-sampled version of s. This could be done n times, until $1 - 2^{-n}$ of the input samples are details. However, this would be too time consuming, so for encoding s, simply the differences are stored. Except at the borders, all values are small now.
These small values are now fed into a Huffman encoder, but since generating the optimal dictionary would take too long, a hardcoded Huffman tree is used that is optimal for the kind of data that is expected from CTA. Since there was no real camera data at the time the algorithm was created, Monte Carlo data was used. The distribution of residuals after the wavelet transform is shown in figure 11.4.

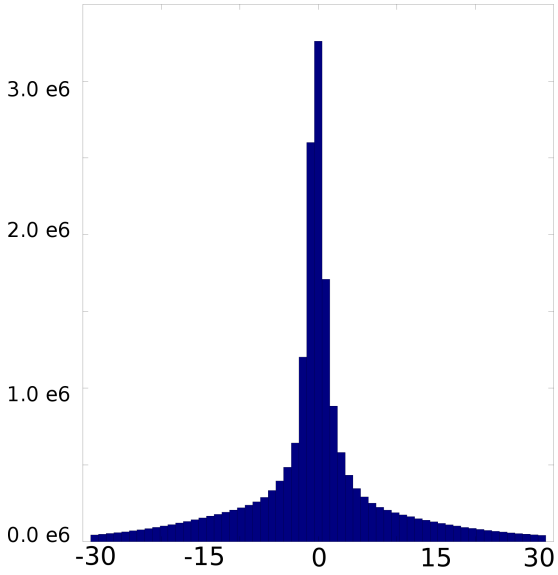


Fig. 11.4.: Residuals distribution after wavelet transform of typical Monte Carlo CTA data.

For this distribution, the Huffman tree in figure 11.5 was manually created. It is a trade-off between compression ratio and speed. The function `encode` takes a number to encode, finds the corresponding bit pattern in the Huffman tree and writes it to the bit stream.

Finally, the inverse integer transform is:

$$v_{2i} = s_i - \frac{d_{i-1} + d_i}{4} \tag{11.13}$$

$$v_{2i+1} = d_{i-1} + \frac{v_{2i-2} + v_{2i}}{2} \tag{11.14}$$

And the code for the inverse transform looks like this:

```
for (i = 2; i < nv; i+=2) {
    d0 = d;
    d = decode(mem);
    s += decode(mem);
    v[i] = s - ((d0 + d) >> 2);
    v[i-1] = d + ((v[i-2] + v[i]) >> 1);
}
```

11.2.3. Summary

The final algorithm reads a block of 256 unsigned short integers, performs a one-level integer wavelet transform and encodes the residuals with a hardcoded Huffman tree. The algorithm is used in the MES file format (see 4) when event raw data is compressed with option `small`. It yields better compression than the first algorithm, but it is almost 10 times slower. On an average computer in 2019, this algorithm compresses with only $\approx$ 400 MB/s, compared to 3-4 GB/s of algorithm 1. It is that slow because of the frequent lookups in the Huffman tree and because the bit stream cannot be easily created with AVX2 instructions, so usual scalar CPU operations are used.

Furthermore, the above wavelet transform can produce values that exceed the range of an unsigned short, so additional overflow checks are necessary, which cost time. Indeed, this transform often produces unreasonably large residuals, which makes it probably better suited for smoother data, like natural images, instead of noisy ADC data.

The last problem is the *hardcoded* Huffman tree: if the noise level changes significantly, the tree is not optimal anymore. And obviously, compressing data from other experiments would work even worse with this algorithm. In fact, when the first real camera data arrived, it turned out that the measured compression ratio was much worse than for the simulations, because the data is much noisier than the Monte Carlos and has periodic spikes.

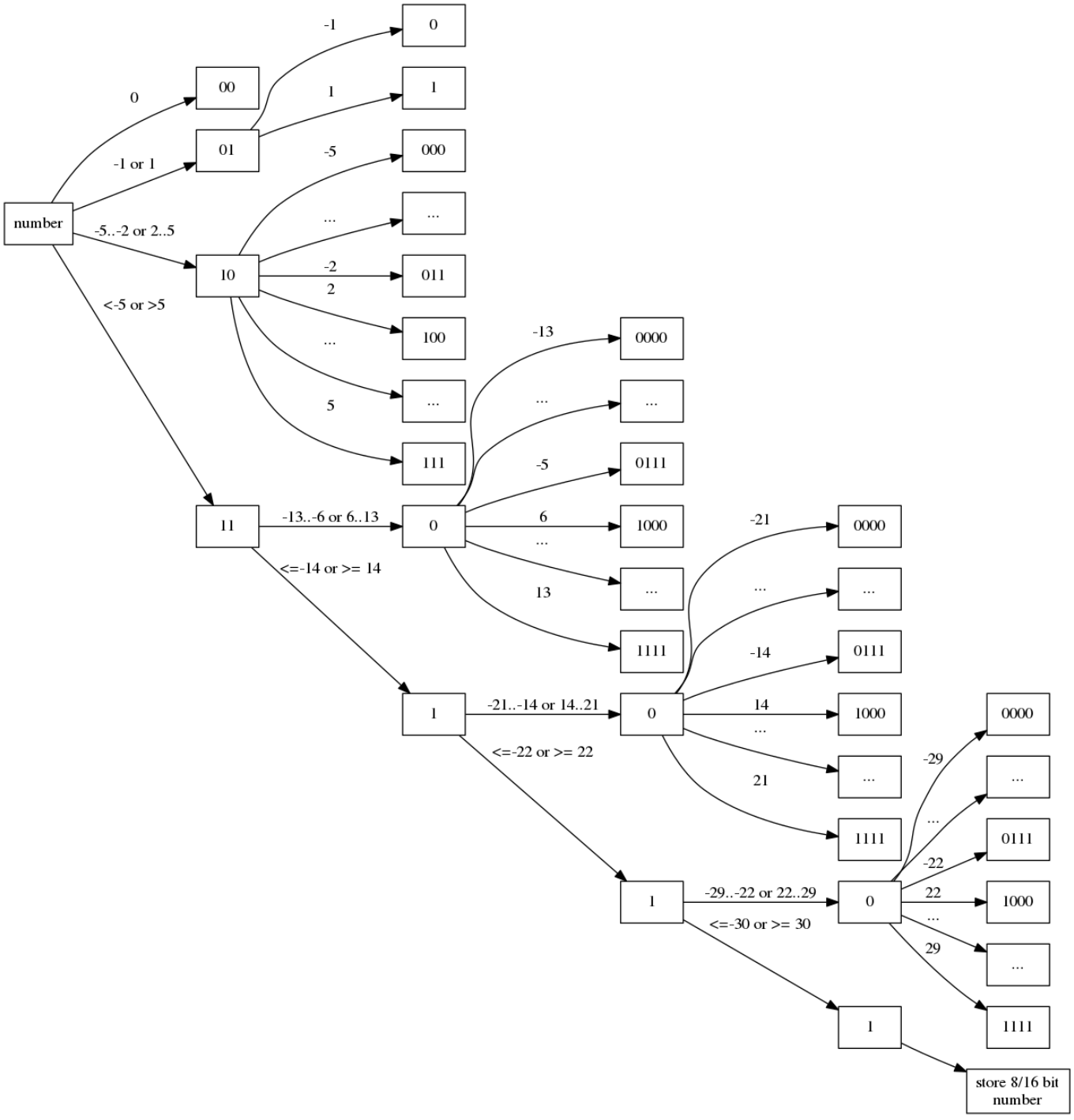


Fig. 11.5.: Huffman tree that maps frequent residuals of the wavelet transform of typical CTA Monte Carlo data to short bit strings.

11.3. Morphological Wavelet Transform and Residual Range Encoding

The goal for this new algorithm is to increase the compression ratio and to be adaptive to different noise levels and even data from other experiments. The experience gained from the previous algorithms suggests that for fast compression of scientific data, a wavelet transform of low computational complexity and some kind of range coding for the residuals probably yields the most efficient compression in terms of ratio and speed. If there are outliers, it would be desirable if they did not force the entire block of 16 values to be encoded with a large range.

11.3.1. Algorithm

The algorithm reads a block of 16 unsigned 16 bit integers and does some heuristic checks in order to catch trivial cases, like constants or incompressible data. It calculates the minimum of these values, subtracts it from them and if:

- more than eight of them are $>= 2^8$: mark this in the header (0) and copy the 16 values to the output array (this copies probably incompressible data)
- they are all 0: mark this in the header byte (1) and store the minimum (this encodes constants)
- they are all $< 2^4$: mark this in the header byte (2), store the minimum and store the 16 four-bit values into 8 bytes
- less than eight of them are $< 2^4$ and the rest is $< 2^8$: mark this in the header byte (3) and store the 16 eight-bit values into 16 bytes
- more than eight of them are $< 2^4$ and the rest is $>= 2^4$: mark this in the header byte (4), read 240 values (16 have already been read), perform a four-level morphological wavelet transform and encode the residuals with 4-6-8-16 bit mask encoding, which will be explained later. This branch of the compression algorithm is called over 90% of the time, if the input is typical 16 bit data from scientific experiments: occasional signals on top of variable noise around a baseline that might change over time.

Then, if there are more than 256 values left to compress, go to the beginning, otherwise copy the remaining values and return. Parallelization of this algorithm is straightforward, because larger blocks can be processed independently.

11.3.2. Mathematical morphology

Mathematical morphology is a theory for the analysis of geometric structures[36]. The basic concept is to traverse the structure X with a structuring element B and modify each point according to the desired operation:

dilation: $\delta_B(X) = X \oplus B = \cup_{x \in X} B_x$
erosion: $\epsilon_B(X) = X \ominus B = \{x | B_x \in B\}$

with $B_x = \{b + x \mid b \in B\}$. So the dilation is the union set of all points in X, each extended by the neighborhood defined by B, and the erosion is the set of all points in X for which B, translated to that point, lies completely in X. A dilation inflates the object and closes small holes inside the object and bays at the border of the object, whereas an erosion deflates the object and removes small islands outside of the object and land tongues at the border of the object.

Example Let the input be a 1 bit image with white background and black foreground (see figure 11.6 left). After dilation with a 3×3 pixel square as structuring element, every background pixel that is a neighbor of a foreground pixel has been turned into a foreground pixel (center), whereas after erosion of the original image, every foreground pixel that is a neighbor of a background pixel has been turned into a background pixel (right).
This concept can be extended from bit masks to gray-scale images and other signals. Let f be the input signal and b the structuring function that has support B. Dilation and erosion are then defined as:

$$(f \oplus b)(x) = max_{z \in B}(f(x + z)) \tag{11.15}$$
$$(f \ominus b)(x) = min_{z \in B}(f(x + z)) \tag{11.16}$$

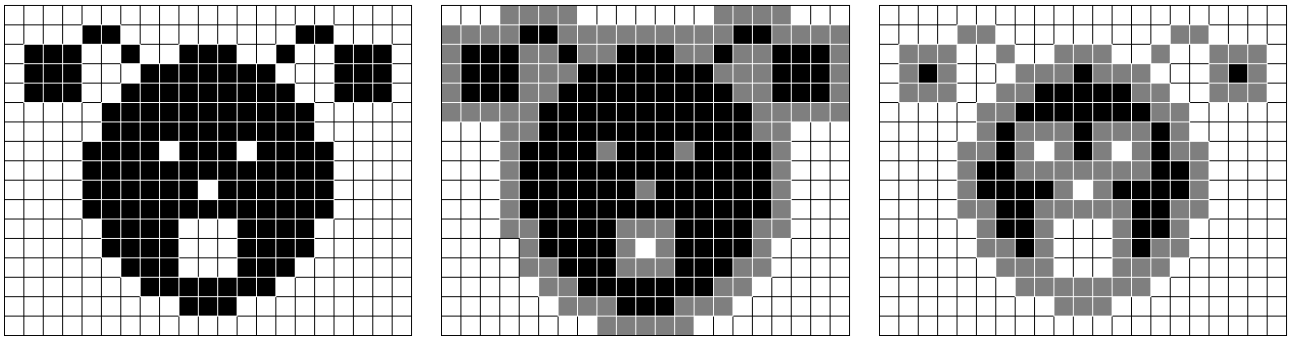


Fig. 11.6.: Left: Original image, white pixels are 0 and black pixels are 1. Center: Dilation. Right: Erosion. This example has been taken from [37].

11.3.3. Morphological Wavelet Transform

In case of the 1-dimensional 16 bit unsigned integer input stream that has to be compressed, the structuring element is chosen to be only 2 pixels wide. If erosion is then used as low-pass filter and the difference of eroded signal and the original signal as high-pass filter, a morphological wavelet transform can be defined[38]:

$$s_i = \min(v_{2i}, v_{2i+1}) \tag{11.17}$$

$$d_i = v_{2i} - v_{2i+1} \tag{11.18}$$

The inverse transformation is:

$$v_{2i} = s_i + \max(d_i, 0) \tag{11.19}$$

$$v_{2i+1} = s_i - \min(d_i, 0) \tag{11.20}$$

The minima are guaranteed to stay in the range of an unsigned short $(0 \dots 65535)$, the differences, however, can exceed that range - they are in the interval $[-65535, \dots, 65535]$. Extracting and storing the signs of the differences prevents such overflows. Also, since the input data is noisy, the signs of the differences are mostly random, so there is no negative impact on the compression ratio. These calculations can be efficiently executed with AVX2 instructions, using only 1 cycle for calculating 16 differences or 16 minima. However, much more time is spent on rearranging and preparing the data inside the registers. In the following code, 32 unsigned shorts are read from memory and transformed using the morphological wavelet just described:

```
mask0 = _mm256_setr_epi8(2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13,
                         2,3,0,1,6,7,4,5,10,11,8,9,14,15,12,13);
mask1 = _mm256_set1_epi32(0x0000FFFF);
x0 = _mm256_loadu_si256((__m256i const *) v); v+=16;
x1 = _mm256_loadu_si256((__m256i const *) v); v+=16;
y0 = _mm256_shuffle_epi8(x0, mask0); y1 = _mm256_shuffle_epi8(x1, mask0);
x0 = _mm256_and_si256(x0, mask1); x1 = _mm256_and_si256(x1, mask1);
y0 = _mm256_and_si256(y0, mask1); y1 = _mm256_and_si256(y1, mask1);
x0 = _mm256_packus_epi32(x0, x1); y0 = _mm256_packus_epi32(y0, y1);
x0 = _mm256_permute4x64_epi64(x0, _MM_SHUFFLE(3, 1, 2, 0));
y0 = _mm256_permute4x64_epi64(y0, _MM_SHUFFLE(3, 1, 2, 0));
min = _mm256_min_epu16(x0, y0); sgn = _mm256_cmpeq_epi16(min, x0);
d = _mm256_subs_epu16(_mm256_max_epu16(x0, y0), min);
```

In the compression algorithm, blocks of 256 values are transformed to 128 minima and 128 differences. The differences are usually small, while the minima need to be transformed again, in order to yield 64 minima and 64 differences. This recursive process could continue until there is only one minimum and 255 differences left, however, here it stops at the fourth level, yielding 240 differences and 16 minima (see figure 11.7). Further decompositions would not exploit the parallelism offered by AVX2 fully, because the last 16 minima at the fourth level would have to be split up to two half-filled AVX2 registers:
In contrast to other wavelets that usually use the mean as low-pass filter, morphological wavelets do not merge spikes into the sub-sampled signal, resulting in smaller differences at the next decomposition level and thus, in a better compression ratio for such data.

Example Let the input signal v be a noisy baseline with occasional spikes on top, $M(v)$ the pairwise mean of v, $m(v)$ the pairwise minimum of v, and $d(v)$ the pairwise absolute differences of v

$$\begin{aligned} v &= (1,0,3,1,100,4,1,4) & d(v) &= (1,2,96,3) \\ M(v) &= (0,2,52,2) & d(M(v)) &= (2,50) \\ M(M(v)) &= (1,27) & d(M(M(v))) &= (26) \\ M(M(M(v))) &= (14) \end{aligned}$$

$$\begin{aligned} v &= (1,0,3,1,100,4,1,4) & d(v) &= (1,2,96,3) \\ m(v) &= (0,1,4,1) & d(m(v)) &= (1,3) \\ m(m(v)) &= (0,1) & d(d(m(v))) &= (1) \\ m(m(m(v))) &= (0) \end{aligned}$$

The residuals to encode when using the mean are $1,2,96,3,2,50,26,14$, but when using the minimum the residual are $1,2,96,3,1,3,0,1$. This example shows how the mean carries spikes down to lower wavelet levels, resulting in more large differences overall, whereas the minimum rids itself of the spikes early, resulting in more smaller differences. The worst case for this wavelet transform is a high baseline with occasional downward spikes, in which case the *maximum* of pairwise samples would be the better low-pass filter in the wavelet decomposition. However, in most data sets the baseline is below the signal, and marking whether the minimum or the maximum was used would consume extra header space.
Finally, in order to encode the 240 differences and 16 minima efficiently, the following bitmask coding is used.

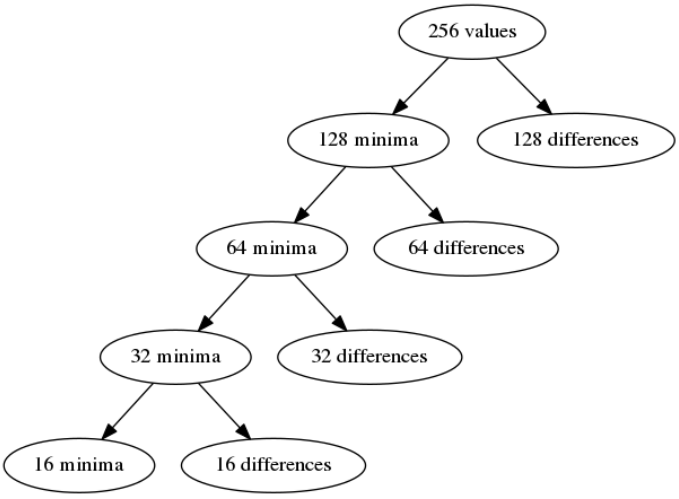


Fig. 11.7.: 4-level morphological wavelet decomposition

11.3.4. 4-6-8-16 Bitmask Encoding

The residuals produced by the wavelet transform are all in the interval $[0\ldots65535]$, because the signs are stored separately. Usually, they are very small, so they can be stored using much less than the 16 bit the original numbers used. However, using range coding alone for a block of 16 residuals (like in algorithm 1), even one single large value forces the other 15 values of the block to be encoded with more bits.
One solution for dealing with such outliers is to use a bit mask for distinguishing different ranges and then store the values with the necessary number of bits. For example, if the two ranges are 4 bit and 16 bit and 11 numbers are 4 bit and 5 are 16 bit, there is the overhead of 16 bits for the mask plus 11×4 bits plus 5×16 bits for the values. Additional 4 bits are lost, because the 44 bits of 4 bit values are not aligned to a byte border. This adds up to $16+44+80+4=144$

bits compared to the original 256 bits. If there were 11×4 bits and 5×12 bits to store, 2×4 bits are lost because of byte alignment. Although it is possible to concatenate the two bit streams before storing them to memory, such bit handling is expensive and complicated with AVX2.
The problem with a single bit mask is that it gives opportunity for only two ranges and it is not clear which ranges to choose. 4 bit and 16 bit might be good values for some data, but for other data, 2 bit and 8 bit might be better. In order to be flexible towards different noise levels and outliers, the ranges 4,6,8 and 16 bit are chosen. Marking the required range requires an overhead of 2 bits per number, for the four possibilities 4, 6, 8 or 16 bit.
The 4 least significant bits (bits 0-3) of all 16 numbers are always stored. Then, for all numbers that do not fit into the 4 bit range, the next 2 bits (bits 4 and 5) are stored. Then, for all numbers that do not fit into the 6 bit range, the next 2 bits (bits 6 and 7) are stored. Finally, for all numbers that do not fit into the 8 bit range, the last 8 bits (bits 8-15) are stored.
So for a block v of 16 unsigned shorts, the three bit masks are created as follows:

$$b4_i = \begin{cases} 0, & \text{if } v_i < 16 \\ 1, & \text{otherwise} \end{cases} \qquad b6_i = \begin{cases} 0, & \text{if } v_i < 64 \\ 1, & \text{otherwise} \end{cases} \qquad b8_i = \begin{cases} 0, & \text{if } v_i < 256 \\ 1, & \text{otherwise} \end{cases} \tag{11.21}$$

Note: A small gain in compression ratio can be achieved by exploiting the knowledge that the bitmasks already exclude some ranges. For example, if the bitmasks indicate that the number needs 6 bits to be stored, it cannot be a number from $0 \ldots 15$. So the range for 6 bit numbers can be shifted from $0 \ldots 15$ to $16 \ldots 79$. The same goes for the 8 bit range, which can be shifted from $0 \ldots 255$ to $80 \ldots 335$. These optimizations are done in the algorithm, but in order to keep the following examples clear, they are not done here.
The three masks are convenient to have for the algorithm, but storing them as they are would not be good, because they are redundant, which is easy to see because since each number is in exactly one of the four ranges, 2 bit must suffice to store the number. The following transformation combines the three masks into two masks:

$$x = (b4 \textbf{ xor } b6) \textbf{ xor } b8 \qquad y = b6 \textbf{ and } b8 \tag{11.22}$$

The inverse transformation is:

$$b4 = x \textbf{ or } y \qquad b6 = y \qquad b8 = x \textbf{ and } y \tag{11.23}$$

Example For the following block v of 16 unsigned shorts, B_i is the number of bits needed to store v_i, $b4$, $b6$ and $b8$ are the bitmasks that define which of the ranges $(4, 6, 8, 16)$ needs to be used for storing a number, and x and y are the transformed bit masks.

```
 v = (0,20,2,1,19023,550,128,127,255,256,60,70,14,102,22,62)
 B = (0, 5,2,1,   15, 10,  8,  7,  8,  9, 6, 7, 4,  7, 5, 6)
b4 = (0, 1,0,0,    1,  1,  1,  1,  1,  1, 1, 1, 0,  1, 1, 1)
b6 = (0, 0,0,0,    1,  1,  1,  1,  1,  1, 0, 1, 0,  1, 0, 0)
b8 = (0, 0,0,0,    1,  1,  0,  0,  0,  1, 0, 0, 0,  0, 0, 0)
 x = (0, 1,0,0,    1,  1,  0,  0,  0,  1, 1, 0, 0,  0, 1, 1)
 y = (0, 0,0,0,    1,  1,  0,  0,  0,  1, 0, 0, 0,  0, 0, 0)
```

The output of this bitmask encoding is: the bit masks x and y, the bits 0-3 of all values in v, bits 4-5 of all v_i for which $b4_i = 1$, bits 6-7 of all v_i for which $b6_i = 1$, and bits 8-15 of all v_i for which $b8_i = 1$.

11.3.5. Summary

This algorithm is robust to outliers and noisy data, achieves good compression ratio and speeds of 1.8 GB/s per core for typical data. However, after some testing with different data sets, it turned out that the compression ratio was not always as good as expected and that it could be significantly increased, if the ranges (4,6,8 and 16 bit) were not hardcoded and instead dynamic ranges were used. Therefore, this algorithm was not benchmarked, but instead it was modified and turned into the final and currently best compression algorithm `fc16`, which is described and thoroughly benchmarked in the next chapter.

11.4. fc16 - Morphological Wavelet Transform with Dynamic Bit Range Residual Encoding

This algorithm reads a block of 256 values, performs a four-level morphological wavelet transform, stores the sign bits of the residuals, and finally encodes the residuals using either a fixed bit range or bitmask encoding. So it is similar to algorithm 3, but the final encoding uses a different check for trivial cases and it chooses from 16 different ranges and range combinations the one that results in the best compression ratio, instead of using one fixed 2-bit mask, like in algorithm 3. On the one hand storing which of the 16 encodings was the optimal one consumes additional 4 header bits, but on the other hand, the overhead of 32 bits that are necessary for the 4-6-8-16 bit encoding of algorithm 3 is gone. However, as will be show now, some of the 16 encodings use a bit mask as well, but those bit masks only distinguish between two ranges and thus only consume 1 bit per value, so for the whole block of 16 values an overhead of only 16 bits is lost.

11.4.1. Dynamic encoding of residuals of the Wavelet transform

For a block of 16 unsigned shorts, it is first checked how many of them need how many bits to encode. Based on the distribution of the required bit ranges, it is decided if all of them are stored as n-bit values or if bitmask encoding is used and some of them are stored as n-bit values and the rest as m-bit values:

0 bit range: all values are 0: do not write anything
1 bit range: each value can be encoded using 1 bit: the 16×1 bit are stored
2 bit range: each value can be encoded using 2 bits: the 16×2 bits are stored
3 bit range: each value can be encoded using 3 bits: the 16×3 bits are stored
4 bit range: each value can be encoded using 4 bits: the 16×4 bits are stored
2/5 bit mask: no value exceeds 5 bit and there are enough 2 bit values to make bit mask encoding pay off, compared to storing all values with 5 bit: store the bit mask, the lowest 2 bit of all values and the higher 3 bit of all 5 bit values
5 bit range: each value can be encoded using 5 bits: the 16×5 bits are stored
3/6 bit mask: no value exceeds 6 bit and there are enough 3 bit values to make bit mask encoding pay off, compared to storing all values with 6 bit: store the bit mask, the lowest 3 bit of all values and the higher 3 bit of all 6 bit values
6 bit range: each value can be encoded using 6 bits: the 16×6 bits are stored
4/8 bit mask: no value exceeds 8 bit and there are enough 4 bit values to make bit mask encoding pay off, compared to storing all values with 8 bit: store the bit mask, the lowest 4 bit of all values and the higher 4 bit of all 8 bit values
8 bit range: each value can be encoded using 8 bits: the 16×8 bits are stored
6/12 bit mask: no value exceeds 12 bit and there are enough 6 bit values to make bit mask encoding pay off, compared to storing all values with 12 bit: store the bit mask, the lowest 6 bit of all values and the higher 6 bit of all 12 bit values
12 bit range: each value can be encoded using 12 bits: the 16×12 bits are stored
6/16 bit mask: there are enough 6 bit values to make bit mask encoding pay off, compared to storing all values with 16 bit: store the bit mask, the lowest 6 bit of all values and the higher 10 bit of all 16 bit values
8/16 bit mask: there are enough 8 bit values to make bit mask encoding pay off, compared to storing all values with 16 bit: store the bit mask, the lowest 8 bit of all values and the higher 8 bit of all 16 bit values
16 bit range: the 16×16 bits are stored

For each block of 16 unsigned shorts, the encoding that uses the least number of bits is used. Since there are 16 different encodings, 4 header bits are needed to tell which encoding was chosen. Being able to choose from more than these 16 possibilities would certainly help encoding the numbers even better, but more than 4 header bits would be needed, and furthermore, more encodings would require more range tests, which would slow down the algorithm. The above combinations are a compromise between compression ratio and speed, because they cover many noise levels and signal strengths and still do not require too much checking. Most importantly, they are simple enough to be programmed with AVX2 instructions, which is usually not true for more sophisticated algorithms, especially if they use Huffman trees and bit streams.

Compressed format

The layout of a compressed block of 16×16 values is:

Size [bytes]	Content
8	16 times the choice between $16 = 2^4$ bit ranges (16×4 bits)
3	the position of the two variable-length blocks in the encoded block
2	the smallest of the 16 minima
30	15 sign masks (15×16 bits = 30 bytes)
variable	16 variable length blocks with differences encoded at bit level

Since the 16 4-bit range specifiers are consecutively stored as a 64 bit number, it is possible to check with one single `if`, if the data was incompressible: As can be seen in the list of encodings above, if a block of 16 values is not compressible and can only be stored by copying them, its 4-bit header is 15, which is **1111** in binary. If all 16 blocks are stored in that way, the 64 bit number has all bits set, which can be checked with `if (x == -1)`, where `x` is the 64 bit long integer that contains the 16 4-bit range specifiers.
Incompressible data almost always has all header bits set to 1. However, if not correctly aligned to a 16 bit boundary in memory, compressible 16 bit data also has all header bits set to 1, which happens if the file with the data to be compressed has an odd number of header bytes. Here are two examples that show the importance of correct alignment for 16 bit unsigned integer compression.

Example 1

Byte	0	1	2	3	4	5
Bits	10101010	00000000	11111111	00000000	00100100	00000000
16 bit int	170		255		36	

In example 1, the data is aligned to 16 bit, so the algorithm would be able to store the numbers as at least 8 bit values instead of 16 bit values. In the next example, the numbers are the same, but a header byte is added, which causes the numbers to be unaligned.

Example 2

Byte	0	1	2	3	4	5
Bits	00000000	10101010	00000000	11111111	00000000	00100100
16 bit int	43520		65280		9216	

In example 2, the data is not aligned to 16 bit anymore, so the lower byte becomes the upper byte and thus, the data becomes incompressible for the range encoder.
In such a case, all 64 header bits are set to one, signaling that the data is misaligned (or really not compressible). The block of 256 values are not re-read, but instead simply be written without compression. Afterwards, however, the next byte from the input stream is simply copied (to the output stream), causing the input stream to be aligned to 16 bit. If the data is really incompressible, this does not improve anything, but neither does is make anything worse.

11.4.2. Benchmark

Here, algorithm 4 (in the following referred to as `fc16`) is compared to state-of-the-art compression programs (see table 11.2):

Program	Version	Command line in benchmark
lzma[41]	5.2.2	`lzma FILE`
gzip[42]	1.5	`gzip -1/-9 FILE` (fast/best compression)
zstd[43]	1.3.3	`zstd -b1/-b9 FILE` (fast/best compression)
snappy[44]	1.1.7	`snappy_unittest FILE`
density[45]	0.14.2	`density_benchmark FILE` (Chameleon, Cheetah, Lion)
lz4[46]	1.8.2	`lz4 -1/-9 FILE` (fast/best compression)
TurboPFor[47]	03.2018	`icapp -s2 FILE` (p4nzenc16, p4nzenc128v16, vbzenc16)
fc16	0	`fc16 < IN > OUT`

Table 11.2.: Compression programs that participate in this benchmark. From the various compression algorithms the libraries *density* and *TurboPFor* offer, the ones mentioned here in the table performed best on the data set of this benchmark.

Some programs allow control over the compression ratio in exchange for compression speed. Usually, this is done over the command line switches `-1` (maximum speed) and `-9` (maximum compression).
The benchmarks were run on a 20-core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz with 512 GB RAM, 32 kB L1-cache per core, 256 kB L2-cache per core, and ≈ 1 GB/s disk I/O speed. All programs in this benchmark were run with a single thread, without any other demanding programs running on the machine. Since the (de)compression speeds can be close to or above the disk I/O speed, all benchmarks were done in-memory using the programs supplied by the authors, so the disk I/O does not throttle the algorithm. For gzip and lzma, there is no in-memory benchmark mode, but since their bottleneck is not the I/O, they simply read the test files from disk and write the compressed output to `/dev/null`, which means that the output is discarded and no time spent on writing anything to disk.
Algorithms 1-3 are not part of this benchmark, because they are in all aspects inferior to algorithm 4 (`fc16`). There are many other compressors, but the above list covers everything from best compression ratio to fastest compression. Comparing to lzma, gzip, zstd, snappy and density is unfair, because those are general-purpose compressors, which means that they can compress any type of data, not just integers, like TurboPFor and fc16. However, since many scientists use them on their data, it might be interesting for them to know how the compression ratio and speed would change, if they used TurboPFor or fc16. Since the compression suites TurboPFor and density provide several compressors, the best three are included in this benchmark.
The test data set comprises the following files (ordered from low noise levels to high noise levels):

hawc.dat HAWC data
gerda.dat GERDA data
ctamc.dat CTA prod3 Monte Carlo simulations
fc_300MHz.dat FlashCam artificially triggered data with a photon rate of 300 MHz/pixel.
fc_1200MHz.dat FlashCam artificially triggered data with a photon rate of 1200 MHz/pixel.
chec.dat CHEC-S data (this data is not aligned to 16 bit)
chec_aligned.dat the same as `chec.dat`, but with the first byte of the file removed, so the data is aligned to 16 bit

11.4.3. Results

As can be seen in figures 11.8 and 11.9, `fc16` compresses almost as good as the strongest compression algorithm (lzma), but 3 orders of magnitude faster. The closest competitor in compression speed (vbzenc16) has a much worse compression ratio and decompresses significantly slower. Also, its compression speed drops down to half the compression speed of fc16, when the data is not that simple to compress (CHEC and ctamc). The closest competitors in decompression speed and compression ratio (p4nzenc16 and p4nzenc128v16) compress 80% slower than fc16. They have a slightly better compression ratio than fc16, but decompress slower in almost all cases. Furthermore, they cannot handle unaligned data.
When compared to the other fast integer compression algorithms p4nzenc16 and p4nzenc128v16, and is almost always the fastest compressor and decompressor. The general-purpose compressors snappy, chameleon, cheetah, lion and lz4 lose against fc16, because they are slower and their compression ratio is worse. The other general-purpose compressors lzma, gzip and zstd rival fc16 in compression ratio, but are orders of magnitude slower.
The two top plots in figure 11.10 show the average ratio of (de)compression speed and compression ratio for all data sets. These numbers are important, because they tell the user how much uncompressed data can be processed ("written and compressed" or "read and decompressed"). The bottom plot combines compression speed, decompression speed and compression ratio. It shows the mean of compression and decompression speed, divided by compression ratio, which is the average speed in which uncompressed data can be processed.
Figure 11.11 shows the compression algorithms in a 2D map, so it is easier to see how they compare to each other both in compression ratio and speed. The optimum algorithm would be in the upper left corner (lowest compression ratio and highest speed). fc16 is on top for both compression and decompression speed, and the third best in terms of compression ratio.

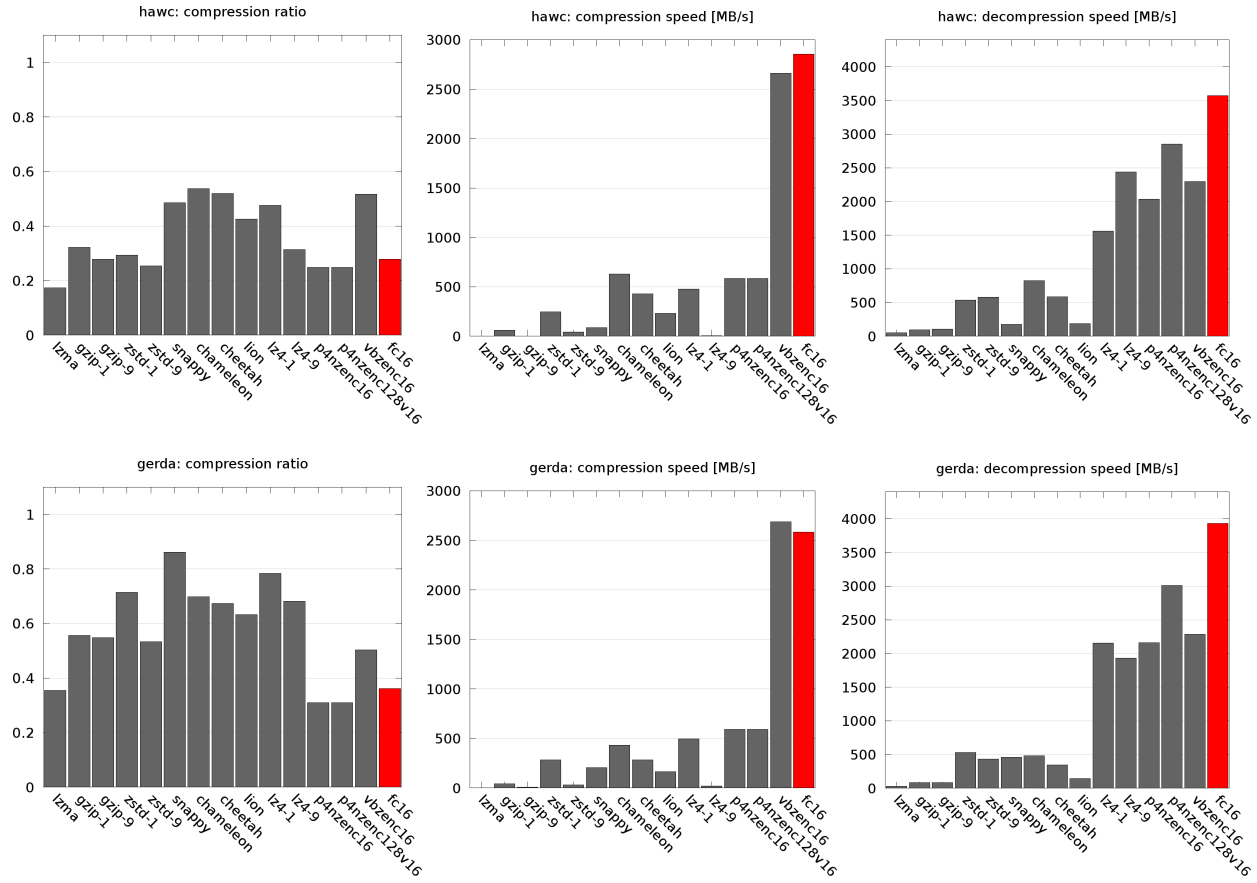


Fig. 11.8.: Compression ratio (left), compression speed (middle) and decompression speed (right), for typical data from HAWC (top) and GERDA (botto).

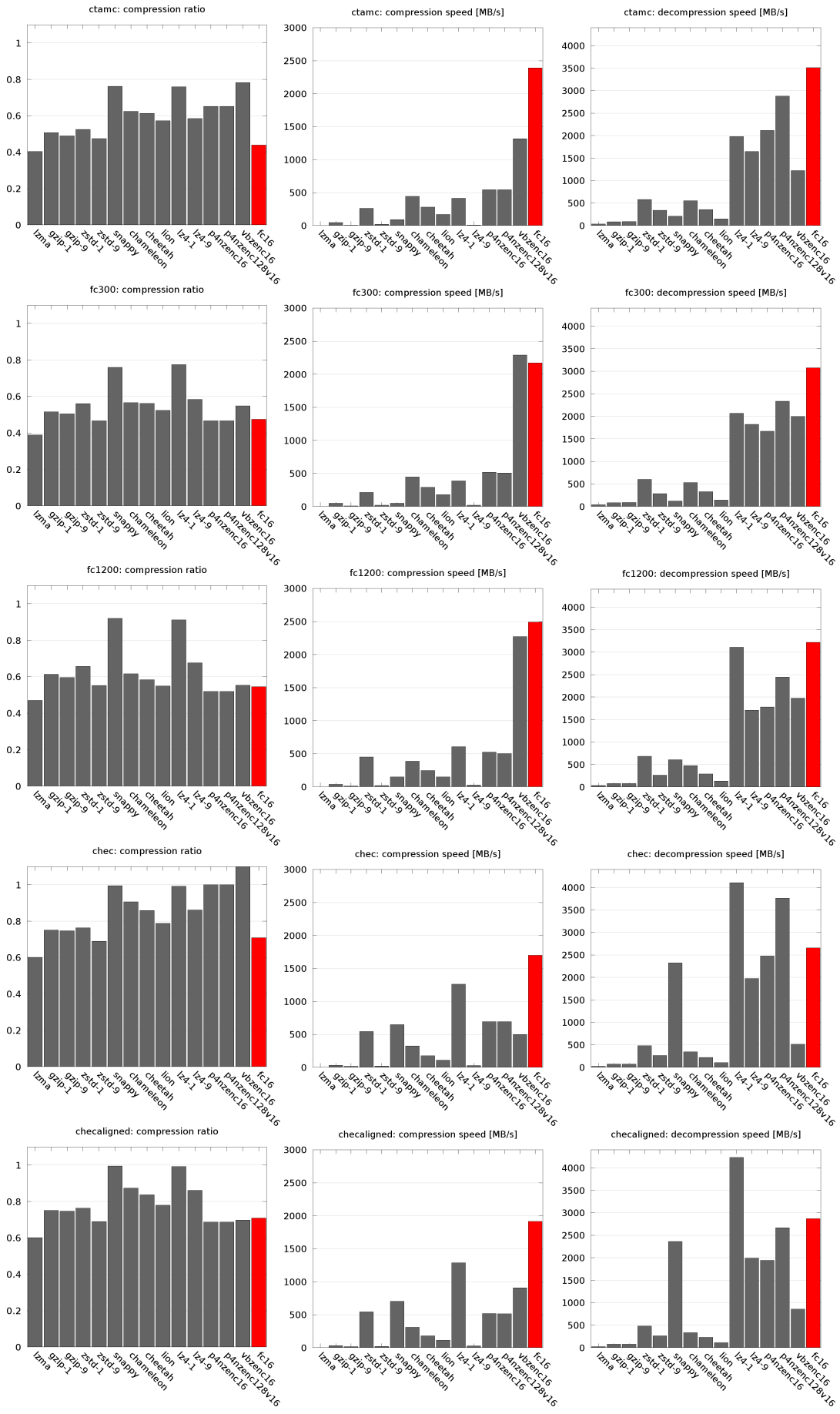


Fig. 11.9.: Compression ratio (left), compression speed (middle) and decompression speed (right), for CTA MC, FlashCam data (300 and 1200 MHz), CHEC-S data (un)aligned to 16 bit (from top to bottom)

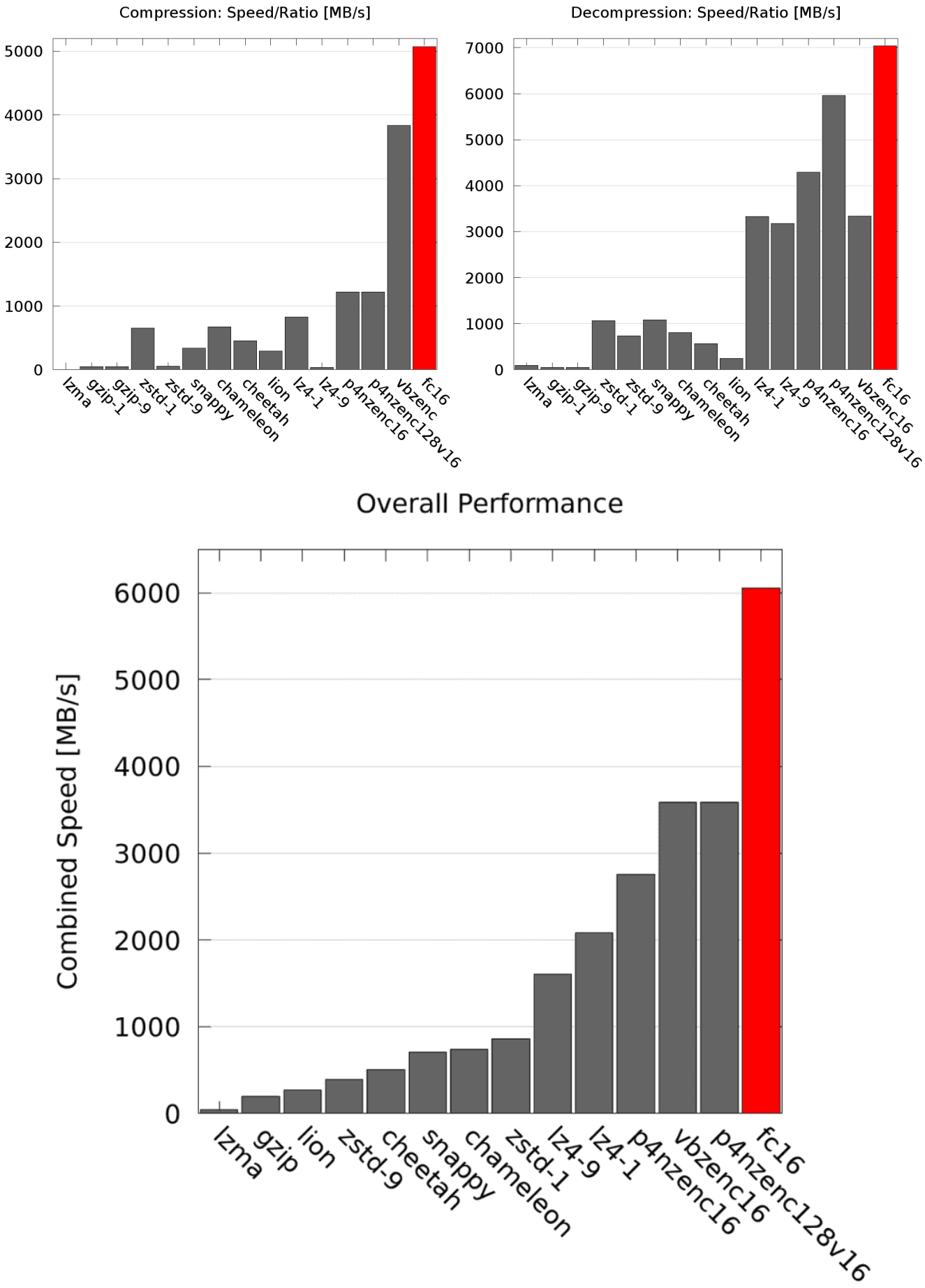


Fig. 11.10.: Top left: Compression speed over compression ratio. Top right: Decompression speed over compression ratio. The same data set has been used as in figure 11.11. Bottom: Mean of compression and decompression speed, divided by compression ratio. This is the average speed in which uncompressed data can be processed by an algorithm.

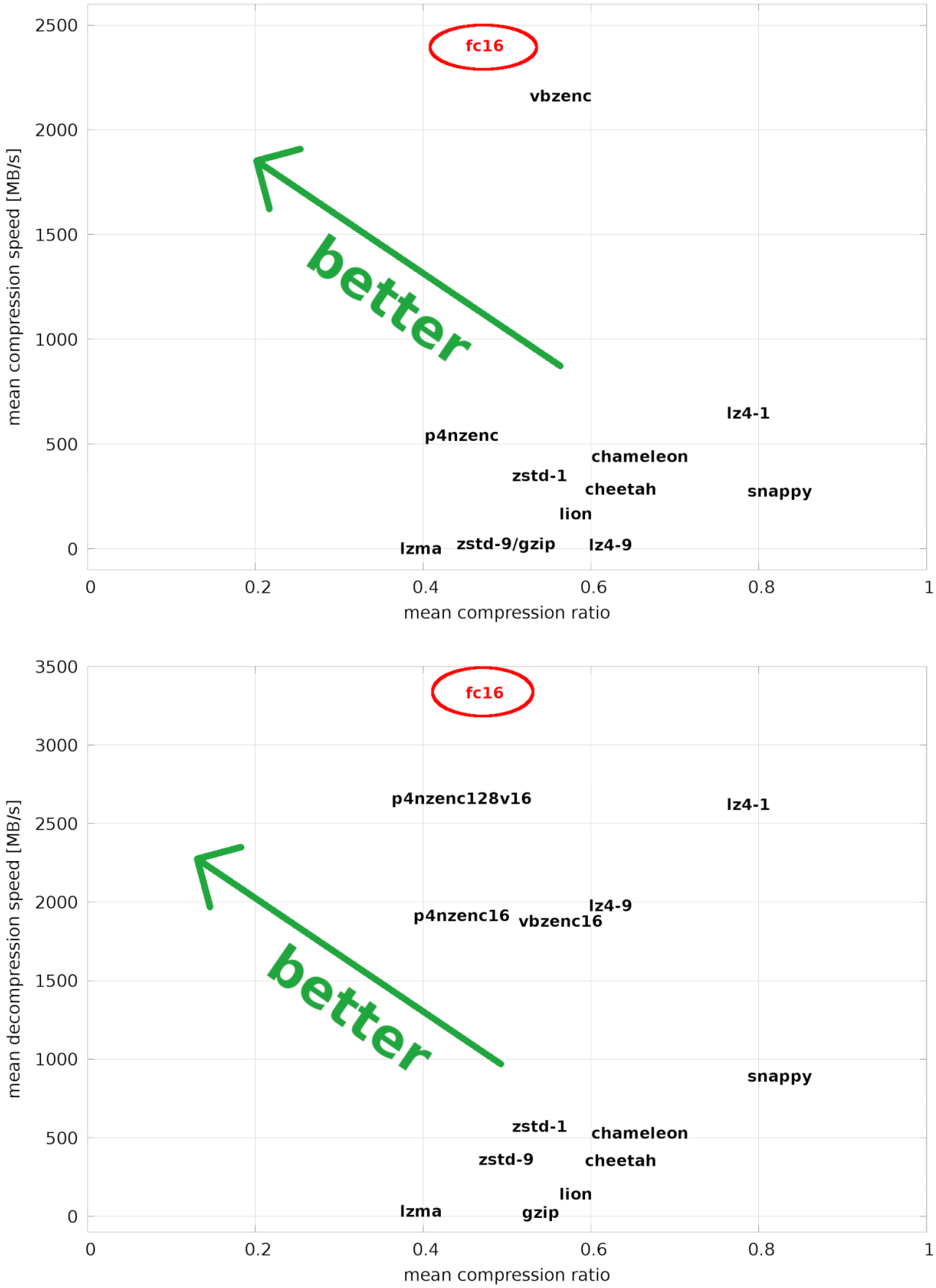


Fig. 11.11.: Mean compression ratio vs. mean (de)compression speed. The unaligned CHEC-S data and the CTA Monte Carlo data are not part of the data set used to generate these plots, so the TurboPFor algorithms that cannot handle such data are not penalized.
Since p4nzenc16 and p4nzenc128v16 have the same compression ratio and compression speed, they have been merged. The same has been done for gzip-1, gzip-9 and zstd-9.

11.4.4. Usage

Application The compression algorithm is provided as executable program `fc16`. The executable does its I/O via Unix pipes (*stdin* and *stdout*):

```
fc16 < in > out
```

Optionally, the number of threads and the block size can be specified on the command line. That way it can be adapted to machines with different cache sizes and different numbers of cores.

Library The algorithm is also provided as the library `libfc.o`.for developers who compress data in memory and maybe store it in their own file format. Additionally, it provides functions for buffered file I/O with on-the-fly compression. The routines `fcopen`, `fcclose`, `fcflush`, `fread` and `fcwrite` are modeled after the `stdio.h` routines of the C standard library, so people can easily modify their existing programs or write new ones.

External pipe Finally, there is an `fcpopen` call, which sets up a pipe to `fc16` and routes all input data through it. This is the preferred option for people who want to take advantage of on-the-fly compression, but do not want to change more than one line of code in their program. This approach can slow down the I/O speed, but unless the I/O speed of the hardware is close to the I/O speed of Unix pipes (5-10 GB/s), this is not an issue.

11.4.5. Summary

The *fc16* algorithm is superior to the previous three algorithms, both in terms of speed (at least 25% faster than algorithm 3) and compression ratio (at least 10-20% better than algorithm 3). Although it can be used as a library, it is straightforward to use the executable. Because the algorithm also works for unaligned data, users do not have to worry about file formats, in which the 16 bit data is interleaved with other data. Such file formats are common in scientific experiments, and the only reasons for not using a fast external compressor are:

- there are large amounts of data that are not 16 bit unsigned integers
- the files are large and random access to the file (seeking) is needed through a table of contents
- the compressed data is needed only in-memory

Compared to the compression algorithms commonly used in physics (e.g. gzip), this new algorithm is orders of magnitude faster and offers better compression. The TurboPFor algorithm `p4nzenc128v16` compresses slighly better, but is four times slower and does not work on unaligned data. Only `lzma` compresses significantly better (10-20%) than `fc16`, but given that it is 1000 times slower, there is only a limited number of use cases where it should be preferred over `fc16`.

11.4.6. Outlook

The compression ratio could be improved by making the algorithm stateful. A larger set of possible encodings could be defined, from which a subset may be chosen for the encoding of a block of 256 values. Depending on how often each encoding was used, a different subset might be chosen from the large set. So, for example, if the data is very noisy, there is no need to include the first 8 encodings of the list above, but instead they could be replaced with other encodings, which are better suited for larger numbers, like 10 bit range or 10-14 bitmask encoding. It is also possible to reduce the header size from 4 bits to 2 or 3 bits and only include those encodings that are useful for that kind of data.

11.4.7. Update

The algorithm has since been extended and can be used to compress images, in particular raw images of Bayer sensors (CRCC and others). Different companies have tested `fc16` on their data and reported it to achieve the same or better compression ratios as commercial algorithms, while being multiple times faster.
The `fc16` algorithm is being patented, so if it is going to be implemented and used other than for private use, the author must be contacted.

12. Fast Reading and Writing of Text Files Containing Numbers

12.1. Introduction

It is a common problem for scientists to store large amounts of numbers to disk, and there are three ways of doing it:

1. using **common data types** (provided by the hardware and most programming languages),
2. applying a **custom encoding** that is tailored to the problem, and
3. converting all numbers to **ASCII text**.

Each way has advantages and disadvantages, depending on the use case. For example, if only few numbers need to be stored, it is not necessary to optimize for a small file size and the data can be simply dumped from memory to disk. If, on the other hand, huge amounts of numbers need to be stored, a space-efficient format is desirable.
Sometimes, like in the case of CTA, the effort of inventing a custom encoding is justified. In other cases, however, it might be more important to have the data in text format, so it can be easily accessed by text-based tools like `grep`, `sed`, `awk` etc.
Here are some examples of custom encodings, hardware supported data types and numbers converted to text.

Example 1 A time series of binary values (0/1 or *yes*/*no* or ...) can be encoded by storing 8 of these values in a byte. If the values are random this encoding is optimal.

Example 2 A matrix of floating point values with 3 significant digits and values ranging from -1 to 1 can be encoded by using 11 bits per number: The possible values are $v = (-1.000, -0.999, -0.998, ..., 1)$, and there are 2001 of them. Using 11 bits offers 2048 mappings from v to $w = (0..2047)$ This can be easily done by removing the offset of v, stretching it to the range of w and rounding it to the next integer: $w = \text{round}((v + 1) * 1000)$ Although the values $2001..2047$ are never used, this encoding is still close to optimal, assuming that the values have a flat random distribution.

Example 3 A memory array of 64 bit floating point values, also called *double* precision values, is written to disk. Each value has 12 significant digits and ranges from -10^{100} to 10^{100}, so the maximum available precision (16 decimal digits) and range ($\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$) of a *double* is not used. Still, it is a good idea to store the array to disk as it is, because it is fast and the format is supported by most programming languages. In C, the array can be written to file with a single call to `write` or `fwrite`, and reading can be done with `read` or `fread`. No conversion needs to be done and not much space is wasted. The I/O will be simple and fast.

Example 4 An array of 32 bit floating point numbers, also called *single* precision values, is written to disk. A *single* ranges from ($\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$) and offers a precision of 7 decimal digits. In this example, we assume that the precision and the range of a *single* are fully exploited. In order to be able to process the resulting file with text-based command line utilities and read it with a common text editor, the values are written to file in text format. In C, this can be done calling `fprintf(file, "%g ", x[i])`, with different formats and number of digits possible. In the text file, each digit consumes 1 byte, so the number -123.45689 will consume 11 bytes, plus 1 byte for the character that separates numbers. This is 3 times as much as a 32 bit float would consume. On the other hand, the number 0 will consume only 1+1 bytes.

Example 5 A random matrix of size 5000×5000 is created in Matlab/Octave:

```
x = exp(randi(1400,5000,5000,1)-700) .* randn(5000,5000);
```

The function `randi` creates a matrix of size 5000×5000, filled with random numbers from 1 to 1000. After subtracting 700 and applying the exponential function, the values in the matrix are equally distributed between $\approx \pm 10^{-304}$ and $\approx \pm 10^{304}$, which is almost the range of a *double*. The element-wise multiplication with the 5000×5000 matrix of normally distributed random numbers changes every value slightly, so the probability of duplicates in the final matrix is significantly reduced.
The size of the matrix in memory is 200 MB, and when it is saved to disk in binary format, it occupies the same space. Since the numbers are random doubles, it is not possible to achieve a smaller file size than 200 MB. When saved as a text file, its size is 548 MB, but it can be immediately processed by almost any program.
Table 12.1 gives an overview of the advantages and disadvantages of different methods of storing numerical values.

Storage method	**common data types**	**custom encoding**	**text**
fast	✓	✓	
simple I/O routines	✓		✓
space-efficient		✓	
data inspection on text-level			✓
flexible ranges		✓	✓

Table 12.1.: Overview of different methods of storing numerical data.

As text files with numerical data are used by many people, it would be helpful to make storing text as numbers smaller and faster. The size of a text file can be reduced by compressing it with a program like *gzip* or *lzma*. Whenever a text file is written (or read), it simply has to be piped through the compressor (or decompressor). Unfortunately, most compression algorithms are very slow ($\approx$ 1-10 MB/s compression and 10-100 MB/s decompression) and might slowdown the I/O.
Methods for fast reading and writing of text files containing numbers are presented in the following.

12.2. Reduced Character Set

When numbers are stored in text files, they look like this:

0.12 1 -1.99e50 4.56834257868647E-2 299792458 2e+64

For the numbers, the 10 digits, a dot, the letter 'e' or 'E' and the plus and minus signs are needed. As delimiter, usually space, tab or comma is used. Newlines are also needed, because they are often used to mark the end of an array. So only the 16 characters shown in table 12.2 are needed to encode any array or matrix of numbers.

Character	**ASCII number**	**`fc4` value**
0..9	48..57	0-9
.	46	10
+	43	11
-	45	12
e or E	101 or 69	13
<space>	32	14
\n	10	15

Table 12.2.: Characters needed for representing numbers in text files and their ASCII/fc4 mappings.

Elements of this reduced character set R can be encoded with 4 bits, which is half the size of an ASCII character. In other words: by simply reducing the set of allowed characters, the size of a file with numbers stored as text can be reduced to half the size.

12.3. Fast Compression of Numbers Stored in ASCII Format

The `fc4` algorithm takes a data stream as input and reads it in blocks of 64 bytes. If the block contains characters that do not belong to the set of 16 characters defined above, the header byte is set to 1 and the block is simply copied. Otherwise, the header byte is set to 0 and the 64 bytes are squeezed into 32 bytes.
The compression ratio is thus:

- $(1 + 32)/64 \approx 0.52$ in the best case scenario (every blocks only contain characters $\in R$),
- $(1 + 64)/64 \approx 1.02$ in the worst case scenario (every block contains a character $\notin R$), or
- between for mixed cases.

Since the delimiter between numbers can vary between files ('`<space>`', '`<tab>`' and '`,`' are the most common), it can be specified on the command line and is used for the whole file, with `<space>` being the default. The delimiter is written into the header of the compressed file, so the decompressor will be able to recreate the original file. The algorithm is extremely simple and en/decoders can be written with few lines of code.

12.3.1. `fc4`

Since the operations for compressing the data can be easily parallelized, the algorithm is implemented using AVX2 (Advanced Vector Extensions 2), which is supported by all modern desktop processors. The resulting C code has less than 600 lines of code, including multi-threading and debug functions. The compression and decompression functions only have 100 and 60 lines of code.
In benchmark mode, when the data is compressed in memory, 3.7 GB/s are achieved on a ThinkPad T480s with an Intel i5 CPU. Data are decompressed with 7.7 GB/s. These numbers are well above the I/O speed of current disks. A quick comparison to other algorithms shows that it is comparable in terms of compression ratio and superior in terms of compression and decompression speed. The overall performance is defined as $p = (v_c + v_d)/(2r)$, with v_c being the compression speed, v_d the decompression speed, and r the compression ratio. Thus, p represents the mean I/O speed of uncompressed data.

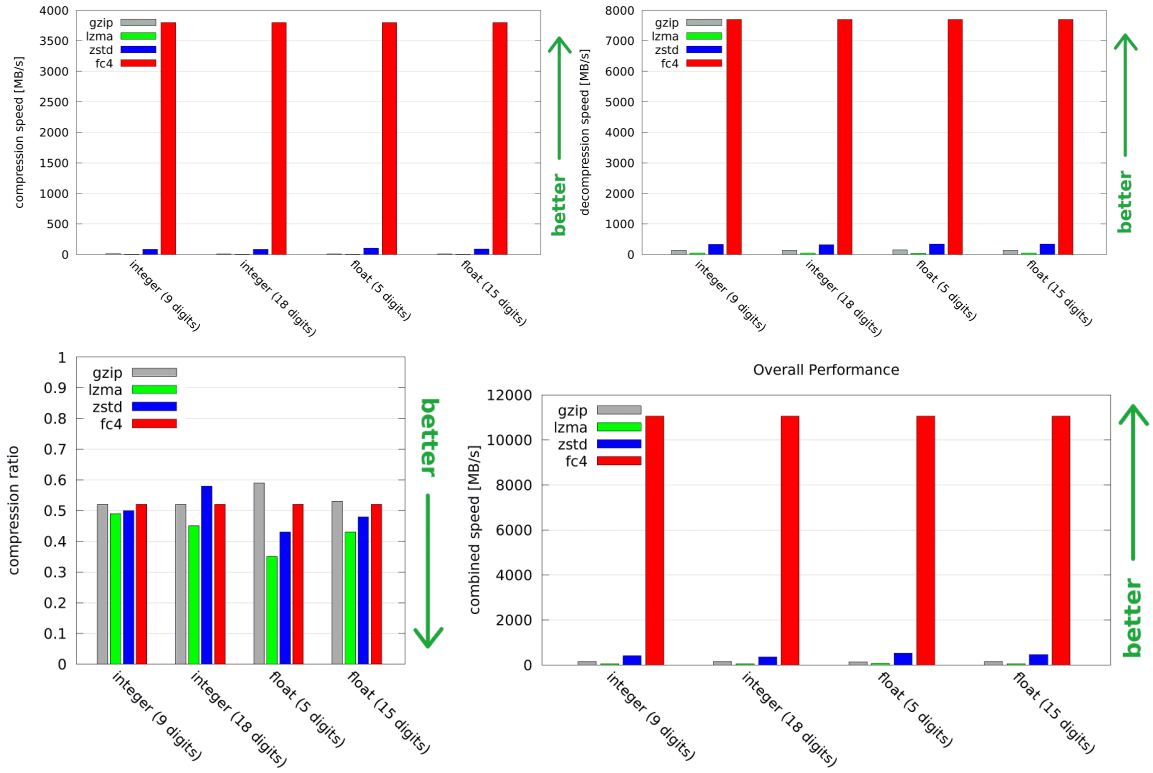


Fig. 12.1.: Top: Compression (left) and decompression (right) speeds of gzip, lzma, zstd and fc16 for integers and floats of different lengths that are stored as ASCII text. Bottom: Compression ratio (left) of these programs and their overall performance.

12.4. Converting Integer Numbers from Text to Binary

The last section showed that `fc4` is able to compress files that contain mostly numbers with 3.7 GB/s (decompression: 7.7 GB/s). In order to benefit from such a fast algorithm, fast conversion functions from text to floating point and integer values are needed.
`atol/atoi`: On a ThinkPad T480s notebook with an Intel i5 processor, using the GNU C standard library function `atol` for the conversion from text to 64 bit integer values, only 27 million numbers per second can be converted. This is equivalent to 525 MB/s of ASCII text, because the numbers that are converted use ($\approx$ 20 characters/number, including sign and string termination symbol).
If the numbers in the text file were stored with less digits (32 bit integer, $\approx$ 11 characters/number, including sign and separator), 35 Million numbers (370 MB of text) can be converted per second.
It might be confusing that more numbers result in less text. This happens because those numbers are shorter.

12.4.1. `fatoi`

Since `atoi` and `atol` are the bottleneck when integers are to be read from text files, the algorithm `fatoi` was developed for fast ASCII to integer conversion. The conversion algorithm is straightforward, but the implementation with AVX2 instructions is tricky:

- if the first character is a '-', it is marked that the number is negative
- the string without '-' is loaded into an AVX2 register v, which is large enough (32 bytes, so there is space for 32 digits, and a 64 signed int can only have 19 digits)
- the end of the string is a '\0', so v is checked against 0, the mask is extracted and the number of leading zero bits is counted in order to get the number n of digits, a mask is created with the first n bits set to 1
- it is necessary to flip and shift the number so that the lowest significant digit is at position 0, because later, it the digits will be multiplied with 10^0, 10^1, 10^2, ...
- since arbitrary bit shifts at runtime are not possible with AVX2, a shuffle mask is created that maps each elements to the same position minus the desired shift; special care has to be taken because of lane crossing
- the number 48 is subtracted from v, so the ASCII values become numbers from 0 to 9
- v is converted to 16 bit numbers, and each four consecutive elements in v are multiplied with 1, 10, 100, 1000
- they are horizontally added, leading to four numbers in v that range from 0 to 9999
- the same is done with higher powers of 10, first with 10000, then with 10000000
- the resulting 64 bit numbers are extracted from the AVX2 register and the final multiplications and additions are done to obtain the result, and the sign is applied

With this algorithm, 97 million conversions from text to 64 bit integer can be done per second, which amounts to 1.9 GB of ASCII text per second (speedup: 3.6). For integers with less digits, 115 million conversions can be done per second, which amounts to 1.2 GB/s of text (speedup: 3.3). The `fatoi` function has 38 lines of code.

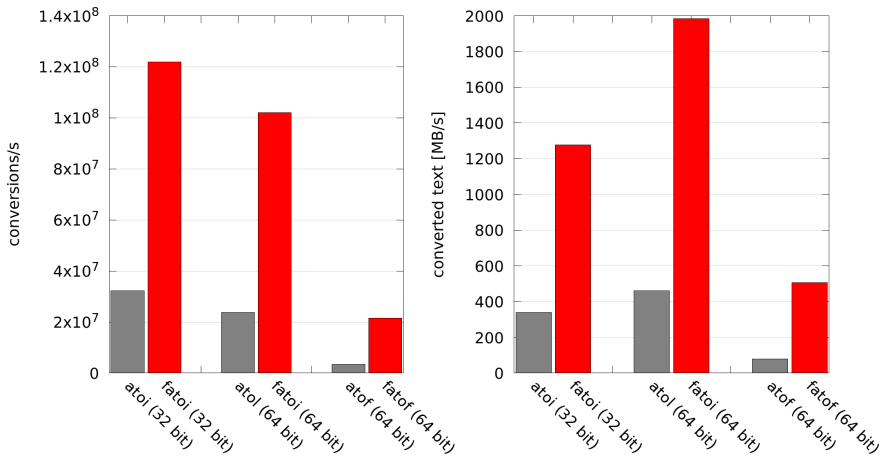


Fig. 12.2.: Fast routines for converting numbers stored as text to integer and floating point binary format.

12.5. Converting Floating Point Numbers from Text to Binary

`IEEE-754 format` A 32 bit floating point number in IEEE-754 format consists of a sign bit s, an 8 bit exponent e as power of 2, and a 23 bit mantissa m, and its value is:

$$(-1)^s \; 2^{e-127} \left(1 + \sum_{i=1}^{23} m_{23-i} \; 2^{-i}\right) \tag{12.1}$$

For a 64 bit float, the length of e is 11 bits and the length of m is 52 bits, and its value is:

$$(-1)^s \; 2^{e-1023} \left(1 + \sum_{i=1}^{52} m_{52-i} \; 2^{-i}\right) \tag{12.2}$$

For better readability, this can also be written as:

$$s \; 2^e \; m \tag{12.3}$$

The input string is given in decimal scientific format (e.g.: -1.234e5), which can also be written as:

$$s \; 10^E \; M, \tag{12.4}$$

with E and M being the decimal exponent and mantissa.
`atof` The GNU C standard library function `atof` can convert 3.4 million numbers (80 MB of text) per second from text format ($\approx$ 24 characters/number: digits, 'e', exponent, signs and separator) to 64 bit floating point values. If the numbers in the text file were stored with less digits ($\approx$ characters/number), 6.7 million numbers (82 MB of text) can be converted per second.

12.5.1. `fatof`

Since `atof` is the bottleneck in text-based floating point number I/O, the algorithm `fatof` for fast ASCII to floating point conversion was developed and is presented here. In order to convert the decimal floating point number to its binary equivalent, the mantissa must be parsed and converted to binary format. This can be done with the integer conversion algorithm from above. The resulting 52 bit integer fits well into a 64 bit long integer.

Handling the exponent

Afterwards, the mantissa must be multiplied by 10^E, which can be done in different ways. The decimal dot of the mantissa is ignored and the mantissa is treated like an integer; for compensation, the decimal exponent is reduced. Example:

$$1.23456789 \times 10^{100} = 123456789 \times 10^{92} \tag{12.5}$$

Floating point operations Calculating 10^E in double precision floating point (e.g. using `pow(10, E)` or `exp(log(10)` and multiplying it with the mantissa gives good results, but fails in some cases. The reason was not investigated further, because those operations are very slow (11 million conv/s, 258.5 MB/s text).

Reducing the decimal exponent while increasing the binary exponent

This approach uses the identity

$$10^E \; M = 2^E \; 5^E \; M = 2^e m \tag{12.6}$$

In principle, M must simply be divided by 5^E, but without running out of precision. This can be done by first multiplying by a small power of 2 and increasing the exponent of 2, and then dividing by small powers of 5, so that the mantissa has approximately the same length as before.

Because multiple multiplications and divisions introduce rounding errors, a larger data type should be chosen. On the x86-64 platform, the compilers `gcc` and `clang` support signed and unsigned 128 bit data types, by combining two 64 bit registers. Multiplications and divisions can then be performed on these data types.
The following example shows how the decimal exponent can be reduced:

$$3 \cdot 10^{13} = 3 \cdot 10^7 \cdot (5^6 \cdot 2^6) \tag{12.7}$$

$$= \left(30000000 \cdot 5^3 \cdot \frac{2^7}{2^7}\right) \cdot 5^3 \cdot 2^6 = \frac{3750000000}{2^7} \cdot 5^3 \cdot 2^{13} \tag{12.8}$$

$$= 29296875 \cdot 5^3 \cdot \frac{2^7}{2^7} \cdot 2^{13} = 28610229 \cdot 2^{20} \tag{12.9}$$

When searching for suitable exponents for 5 and 2, a good compromise between speed and accuracy must be found. The smaller the exponent, the more mantissa digits can be kept without multiplication overflow, and the larger the exponent, the faster the decimal exponent is reduced.

Lookup table With a lookup table, the loop of the above method can be avoided. For each decimal exponent E in range $-350 \ldots 350$, a binary exponent $e(E)$ is calculated for which the quotient $q(E) = 10^E/2^e \approx 1$. e is an integer from -1163 to 1163 and stored in the lookup table. q is a rational number close to zero, and its first 21 digits are stored in the lookup as an 128 bit unsigned integer. Although more digits would be even better to reduce rounding errors, it is not possible, because when it is multiplied by the mantissa, it must not overflow the 128 bit unsigned integer. The numbers were calculated in Mathematica, because of its capability of dealing with arbitrary precision numbers, and then filled into a structure in the C program. Since there is no direct way of filling a 128 bit integer at compile time, the first digits of q are written as unsigned 64 bit integer, which is casted to a 128 bit unsigned integer, then it is multiplied by 100 and finally the last two digits are added:

```
__uint128_t quot[701] = {
    ((__uint128_t) 125280926705350798 2UL)*100+79,
    ((__uint128_t)  78300579190844248 9UL)*100+24,
    ((__uint128_t)  97875723988555311 1UL)*100+55,
    ((__uint128_t) 12234465498569413 89UL)*100+44,
    ...
};
```

When a mantissa M and an decimal exponent E are given, M is multiplied with $q(E)$ and the binary exponent is set to $e(E)$. With 25 million conversions per second from text to 64 bit double (576 MB/s of text) and 26 million conv/s from text to 32 bit float (312 MB/s of text), this is the fastest among all tested approaches. The `fatof` function has 37 lines of code.

12.6. Converting Integer Numbers from Binary to Text

When binary numbers in memory need to be written to disc in ASCII format quickly, a fast conversion routine is needed.
The GNU C standard library provides `printf` and its siblings `sprintf`, `fprintf` etc. They are flexible and can print several values at the same time. 10.9 million conversions per second (resulting in 211 MB/s of text) are performed when 64 bit integers with ≈ 19 digits are converted to text using `sprintf`. When 32 bit integers with ≈ 9 digits are converted to text, 12.7 million conversions per second can be done, which amounts to 133 MB/s of text.

12.6.1. `printint`

Since it would be desirable to have a faster conversion routine than `printf`, the algorithm `printint` was developed for fast conversion from 64 bit integers to text.
In order to calculate the decimal digits of the binary number in memory, it has to be divided by powers of ten, and the remainder has to be processed the same way. With AVX2, some of the operations can be vectorized, leading to a significant speedup.
42.8 million conversions per second (resulting in 832 MB/s of text) can be done when 64 bit integers are converted to text using `printint`, and 42.9 million conv/s (449 MB/s text) when converting 32 bit integers.

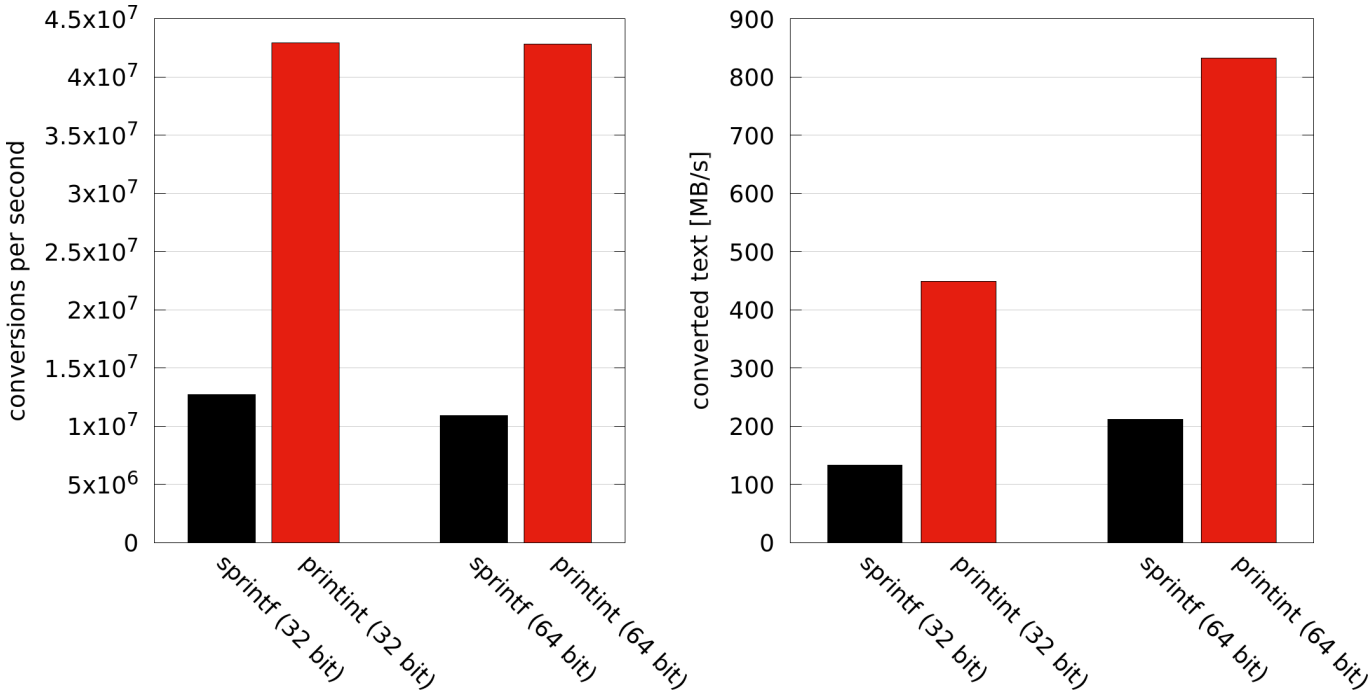


Fig. 12.3.: Fast routines for converting integers from binary to text format.

12.7. Summary

Both the floating point and the integer conversion functions are multiple times faster than the functions of the C standard library. Combined with the `fc4` compressor, which achieves similar compression as the classical compression algorithms with orders of magnitude higher compression and decompression speed, they make an interesting case for text I/O when dealing with numbers.
The `printfloat` routine for converting floating point numbers to text has not been written yet. It will be done using the principles presented above.

13. Conclusion

Several algorithms for dealing with scientific data have been developed and presented.

13.1. Impact on Data Acquisition and Storage

Scientific data that is stored for a longer time is almost always compressed, because although disk space has become cheap, a compression factor of 0.5 in space reduction still means significant savings. Regarding the compression and decompression speed, it depends on what is going to be done with the data. If computing intensive analyses are performed, the disk I/O and decompression times are not the bottleneck. But if, for example, the data is re-calibrated or if simple histograms are created, the execution time depends solely on the disk I/O and (de)compression speed. An important insight is that fast compression can increase the I/O speed. For typical scientific raw data, factors between 2 and 4 were measured.
An algorithm that compresses 16 bit unsigned integers as well as the established compression algorithms, but orders of magnitude faster, should be of interest for the scientific community and everybody else dealing with similar data.

13.1.1. Relevance for CTA

CTA will be the next large ground-based gamma-ray observatory, with one telescope array located in Chile and the other on La Palma. Both sites are far away from the data centers, where all data will be archived and where people will be able to access it. Since the raw data rates are expected to exceed 200 PB/year and only 4 PB/year can be transferred with the Gigabit line, a compression factor of at least 0.02 would be needed. However, tests with several compression algorithms show that no compression factor smaller than 0.3 should be expected. Thus, it is inevitable to calibrate the data on site and remove most of the background already there. Because the data reduction might remove some of the signal, which cannot be seen with current analysis and calibration methods, all pixels in a certain neighborhood around pixels with strong signal must also be kept.

MES File format This requires a file format that allows to store traces, as well as derived values (intensities, times of maximum, ...), both in form of full camera images, pixel lists or regions of interest. Despite the data reduction, the data rates will still be high, and the importance of a flexible data format that efficiently compresses the recorded events and the subsystem data remains. In contrast to external compressors like `fc16` or `gzip`, the MES file format has built-in compression only for the data blocks. Since the headers are not compressed, it is possible to quickly search through the stream, only looking at the headers, until a header value meets a given condition. For example, this is useful for fast synchronization of different streams, where only the `time_id`s in the headers are compared.
The MES file format is a prototype for the CTA file format. It fulfills all requirements and has been thoroughly tested.

`fc16` compression algorithm During data acquisition and lowest-level calibration, the data is dumped to disk and then reread several times in order to calculate the calibration parameters. The I/O is usually limited by the disk and the network, and not by the CPU, so a fast compression/decompression algorithm can improve the I/O significantly. With `fc16`, typical CTA raw data can be compressed in real-time to approximately half the size. This means that the I/O speed of the disk is doubled, and that only half the disk space and half the network speed (telescope to on-site data center) are needed. Since it is robust to outliers, adapts to different noise levels, and does not require the incoming data to be aligned, almost any data that consist mainly of 16 bit unsigned integers can be efficiently compressed.

Recommendation In the camera server, where the camera-specific raw-data is taken and processed, camera-specific software is used to assemble the events and perform the lowest-level calibration. For faster I/O and for saving disk space, the use of `fc16` is recommended.

From the point on where the data of the different cameras is merged (i.e. the rest of the experiment), the MES file format is recommended. It is fast and flexible and supports all data structures (e.g. parameter sets, histograms) that are needed in the experiment.
Since the built-in compression of the MES file format is significantly inferior to `fc16`, it should be updated to offer `fc16` compression for camera raw data. `fc16`'s block size of 256 is not a problem, because the traces are stored consecutively in a large memory block of length $n_{\text{pix}} \times n_{\text{samples}}$ (convenience pointers for per-pixel access are set later).

A. Appendix

A.1. Abbreviations and special terms

ADC Analog-to-Digital Converter
ARS Analog Ring Sampler
AVX2 Advanced Vector Extensions 2
Amplitude sum of all pixels in a shower image
CHEC (Compact High-Energy Camera)
CPU Central Processing Unit
CT Cherenkov Telescope, the five HESS telescopes are named CT1-CT5
DNN Deep Neural Network
HESS II Phase two of the experiment, also telescope CT5 exists
HESS I Phase one of the experiment, only telescopes CT1-CT4 exist
HESS High Energy Stereoscopic System
Histset Histogram Set
I/O Input/Output: reading and writing of data
LST Large-Sized Telescope
MC Monte Carlo
MESS Modular Efficient Simple System
MES Modular Efficient Storage
ML Machine Learning
MPIK Max-Planck-Institut für Kernphysik (MPI for Nuclear Physics)
MST Medium-Sized Telescope
Mono Analysis in connection with the HESS telescopes, it means that only the events of a single telescope are used in the analysis
NSB Night Sky Background
OS Operating System
PE Photo Electron - unit for the intensity of shower pixels, because the incoming Cherenkov light is proportional to the number of freed electrons in the photomultiplier tube
PMT Photomultiplier Tube
Parset Parameter Set
ROI Region Of Interest
SNR Supernova Remnant
SST Small-Sized Telescope
SVM Support Vector Machine
Televent Telescope Event
VHE very-high energy
WIMP Weakly Interacting Massive Particle
Zero Suppression removing the background from the measured data; this term usually means cleaning of airhower images

Bibliography

[1] *Viktor Franz Hess: Über Beobachtungen der durchdringenden Strahlung bei sieben Freiballonfahrten*, Physikalische Zeitschrift 13, 1912, S. 1084–1091.

[2] *T. C. Weekes et al.: Observation of TeV gamma rays from the Crab nebula using the atmospheric Cerenkov imaging technique*, The Astrophysical Journal 342(1):379-395, June 1989

[3] *S. P. Wakely and D. Horan: TeVCat: An online catalog for Very High Energy Gamma-Ray Astronomy*, http://tevcat.uchicago.edu

[4] *R. C. Lamb et.al.: Observations of Markarian 421*, 24th International Cosmic Ray Conference, Vol. 2, held August 28-September 8, 1995

[5] *Pierre Auger et. al.: Extensive Cosmic-Ray Showers*, Rev. Mod. Phys. 11, 288 – Published 1 July 1939

[6] *Li, T.-P. and Ma, Y.-Q.: Analysis methods for results in gamma-ray astronomy*, Astrophysical Journal, Part 1 (ISSN 0004-637X), vol. 272, Sept. 1, 1983, p. 317-324.

[7] *Ignacio de la Calle Pérez: A study of the polarization of the Cherenkov Radiation in Extensive Air Showers of Energy around 1 TeV*, Diploma thesis.

[8] *HAWC*, https://www.hawc-observatory.org

[9] *Werner Hofmann et al.: Comparison of techniques to reconstruct VHE gamma-ray showers from multiple stereoscopic Cherenkov images*, Astropart.Phys.122:135-143,1999

[10] *HESS Collaboration: The H.E.S.S. Galactic plane survey*, https://doi.org/10.1051/0004-6361/201732098

[11] *H.E.S.S. Collaboration: Observations of the Crab Nebula with H.E.S.S*, Astron.Astrophys.457:899-915,2006

[12] *Gerd Pühlhofer: H.E.S.S. Highlights*, 7th International Fermi Symposium 2017

[13] *P. Vincent et al.: Performance of the HESS cameras*, Proceedings of the 28th International Cosmic Ray Conference. July 31-August 7, 2003. Tsukuba, Japan

[14] *G. Giavitto et al.: The upgrade of the H.E.S.S. cameras*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 876, 21 December 2017, Pages 35-38.

[15] *F. Werner et al.: Performance Verification of the FlashCam Prototype Camera for the Cherenkov Telescope Array*, Proceedings of the 9th International Workshop on Ring Imaging Cherenkov Detectors (RICH 2016), Lake Bled, Slovenia.

[16] *T. Hassan et al.: Monte Carlo Performance Studies for the Site Selection of the Cherenkov Telescope Array*, Astroparticle Physics 93, May 2017.

[17] *K. Bernlöhr et al.: Monte Carlo design studies for the Cherenkov Telescope Array*, Astroparticle Physics Volume 43, March 2013, Pages 171-188.

[18] *CTA Collaboration*, Design concepts for the Cherenkov Telescope Array CTA: an advanced facility for ground-based high-energy gamma-ray astronomy

[19] *A. M. Hillas: Cerenkov light images of EAS produced by primary gamma*, In NASA. Goddard Space Flight Center 19th Intern. Cosmic Ray Conf., Vol. 3 p 445-448.

[20] *GERDA collaboration, M.Agostini et al.: Improved Limit on Neutrinoless Double-β Decay of ^{76}Ge from GERDA Phase II*, Phys. Rev. Letters 120 (2018), 132503

[21] *J. Hahn et. al.: Impact of aerosols and adverse atmospheric conditions on the data quality for spectral analysis of the H.E.S.S. telescopes*, DOI: 10.1016/j.astropartphys.2013.10.003.

[22] *Andreas Hillert: Improving H.E.S.S. cosmic-ray background rejection by means of a new Gamma-Ray Air Shower Parameterisation (GRASP)*, Ph.D. Thesis.

[23] *Hassan, T. et al: Second large-scale Monte Carlo study for the Cherenkov Telescope Array*, ICRC 2015.

[24] *Stegmann, C. for the H.E.S.S. collaboration, 2012, 5th International Meeting on High Energy Gamma-Ray Astronomy*, AIP Conference Proceedings, Volume 1505, pp. 194-201.

[25] *Parsons R.D. and Hinton, J.A.: A Monte Carlo Template based analysis for Air-Cherenkov Arrays*, 2014, APh, 56, 26-34.

[26] *J. L. Contreras et al: Data model issues in the Cherenkov Telescope Array project* , ICRC 2015.

[27] *R. Marx, R. de los Reyes: A Prototype Data Format for the Cherenkov Telescope Array: Regions Of Interest (ROI)*, ICRC 2015.

[28] *K. Bernlöhr: Simulation of Imaging Atmospheric Cherenkov Telescopes with CORSIKA and sim_telarray*, Astropart.Phys 30, pages 149-158, 2008.

[29] *Intel Intrinsics Guide*, https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[30] *Agner Fog: Instruction Tables. List of Instruction latencies, throughputs and micro-operation breakdowns for Intel, AMA and VIA CPUs*, http://www.agner.org/optimize/instruction_tables.pdf

[31] *D. A. Huffman: A method for the construction of minimum-redundancy codes*, Proceedings of the I.R.E.. September 1952, S. 1098-1101.

[32] *Wim Sweldens: The Lifting Scheme: A Construction of Second Generation Wavelets*, Journal on Mathematical Analysis. SIAM. 29 (2): 511–546, 1997.

[33] *Calderbank, A. R., et al.: Wavelet transforms that map integers to integers.*, Applied and computational harmonic analysis 5.3 (1998): 332-369.

[34] *Hopcroft, J.; Tarjan, R. (1973): Algorithm 447: efficient algorithms for graph manipulation*, Communications of the ACM, 16 (6): 372–378, doi:10.1145/362248.362272

[35] *Fernand Meyer: Un algorithme optimal pour la ligne de partage des eaux.*, Dans 8me congrès de reconnaissance des formes et intelligence artificielle, Vol. 2 (1991), pages 847–857, Lyon, France.

[36] *Jean Serra: Image Analysis and Mathematical Morphology*, ISBN 0-12-637240-3

[37] *https://fr.wikipedia.org/wiki/Morphologie_math%C3%A9matique*, 2018.

[38] *Henk J. A. M. Heijmans: Nonlinear Multiresolution Signal Decomposition Schemes—Part II: Morphological Wavelets*, IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 9, NO. 11, NOVEMBER 2000

[39] *gsl*, https://www.gnu.org/software/gsl 2.5

[40] *blosc*, https://blosc.org

[41] *lzma*, https://tukaani.org/xz/

[42] *gzip*, https://www.gnu.org/software/gzip/ Version: 1.5

[43] *zstd*, https://github.com/facebook/zstd

[44] *snappy*, https://github.com/google/snappy

[45] *density*, https://github.com/centaurean/density

[46] *lz4*, https://github.com/lz4/lz4

[47] *TurboPFor*, https://github.com/powturbo/TurboPFor

[48] *Thomas Keck: FastBDT: A speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification*, arXiv:1609.06119

A.2. Acknowlegdements

I would like to thank:

Werner Hofmann for accepting me as a PhD student and for allowing me to work in such an excellent environment. He always helps me with elegant solutions to all kinds of problems, and I am extremely grateful for his exceptional and kind support and for all I have learnt from him.

Peter Fischer for accepting me as a PhD student, for always being very friendly, encouraging and supportive, and for making it possible for me to enroll at the Heidelberg University.

Mizzi for solving all my computer problems instantly, for significantly improving my programming skills, for helping me all the time with everything and for all the nice anecdotes.

the smart and friendly (former) members of the Hofmann/Hinton group at the MPIK, who were always ready to help or chat: Christoph, Dan, Dirk, Faical, Felix, German, Henning, Jim, Joachim, Konrad, Michael, Peter, Raquel, Ruth, Simon, Svenja, Vincent.

the MPIK computer administrators Frank Koeck and Mizzi for providing an extremely well managed, robust and highly performant computing environment.