IACT* DATA PROCESSING WITH PYTHON?

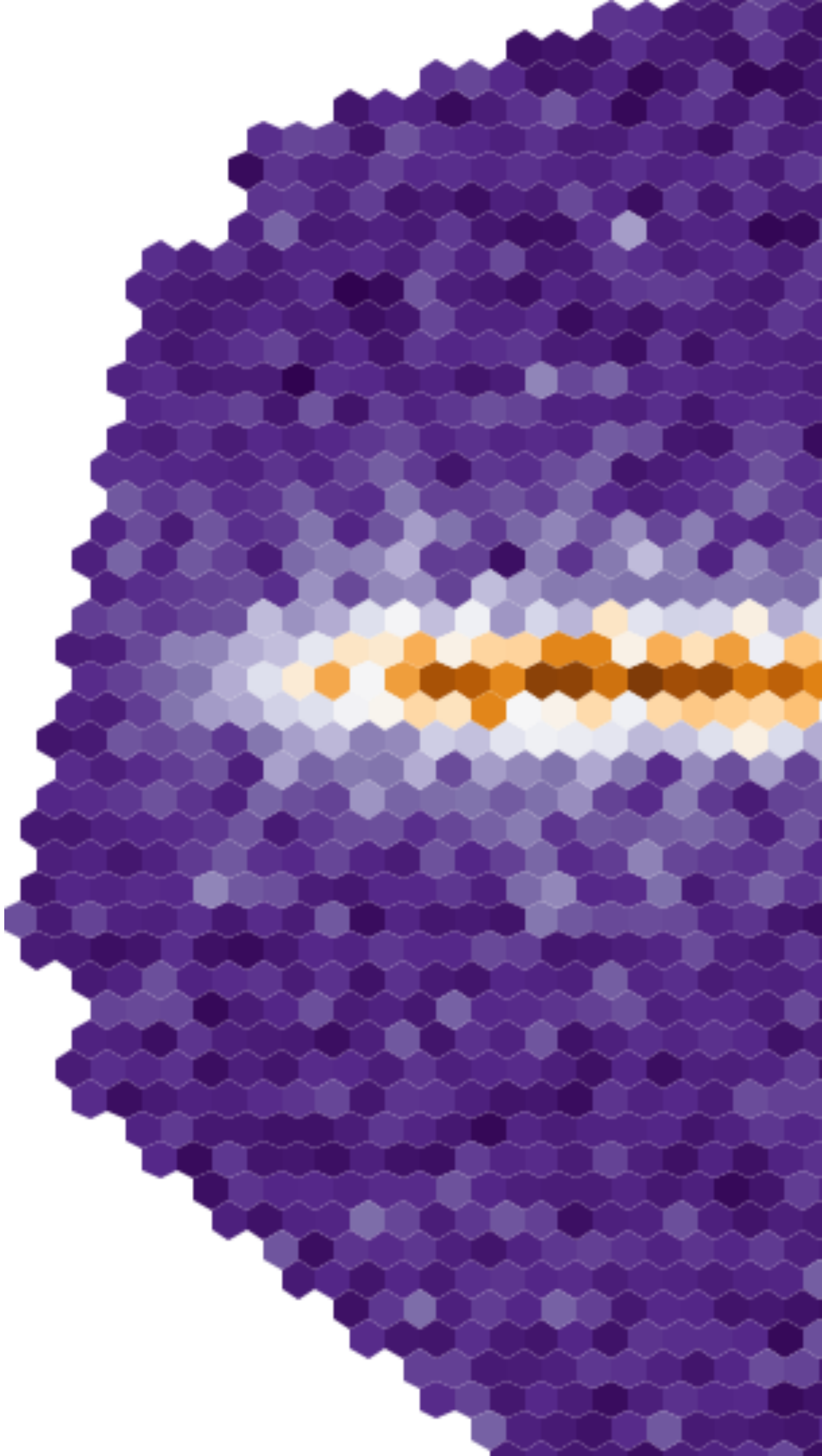
Exploring "big" science data processing using a python framework

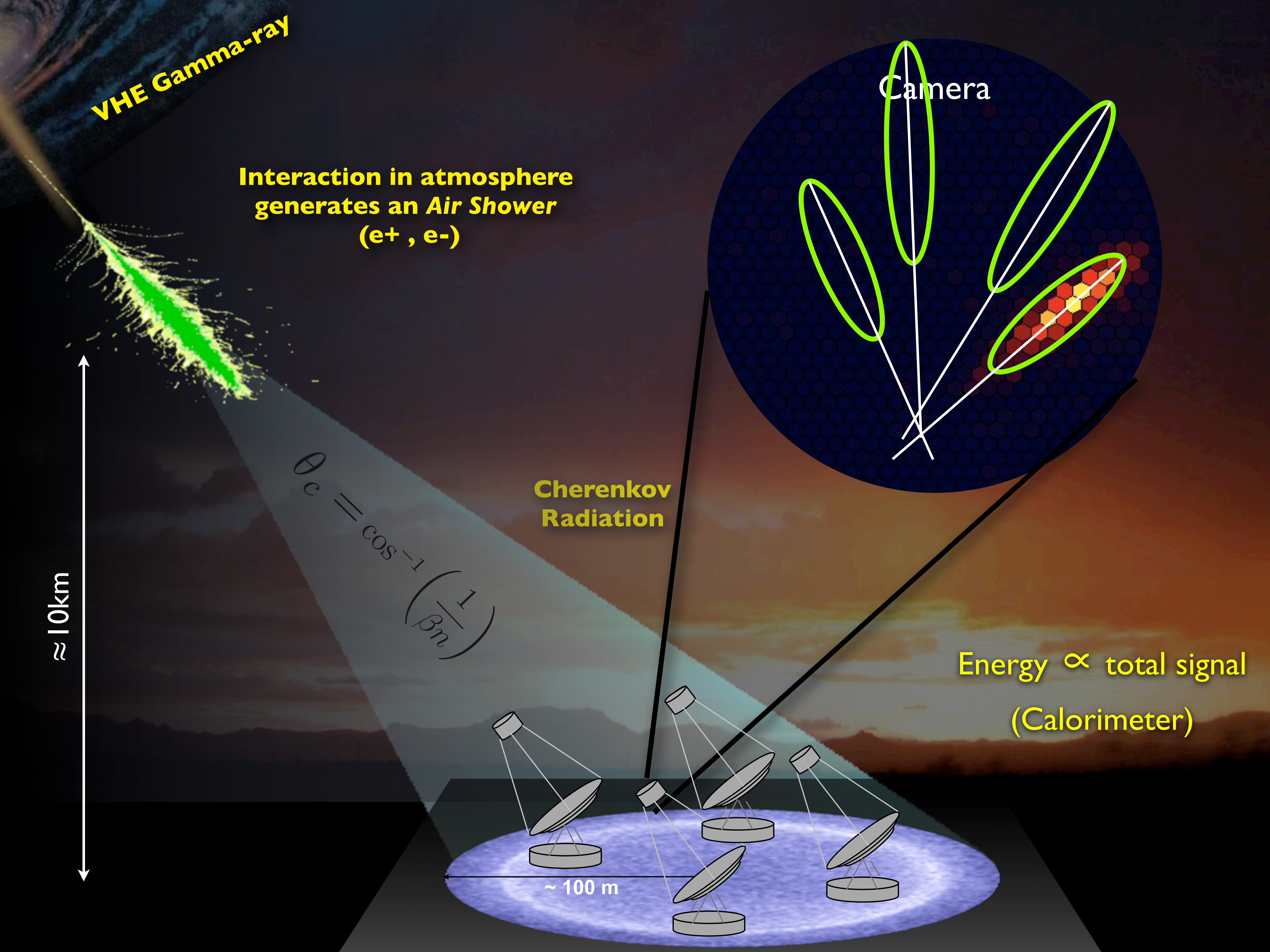
Karl Kosack (CEA Saclay, France)

*Imaging Atmospheric Cherenkov Telescope

THE PROBLEM

What are the inputs and goal?

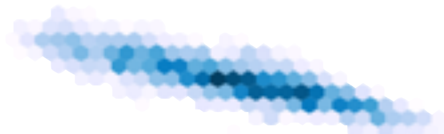




VHE Gamma-ray
Interaction in atmosphere generates an *Air Shower* (e+ , e-)
Camera
$\theta_c = \cos^{-1}\left(\frac{1}{\beta n}\right)$
Cherenkov Radiation
≈ 10km
Energy ∝ total signal
(Calorimeter)
~ 100 m

GOAL

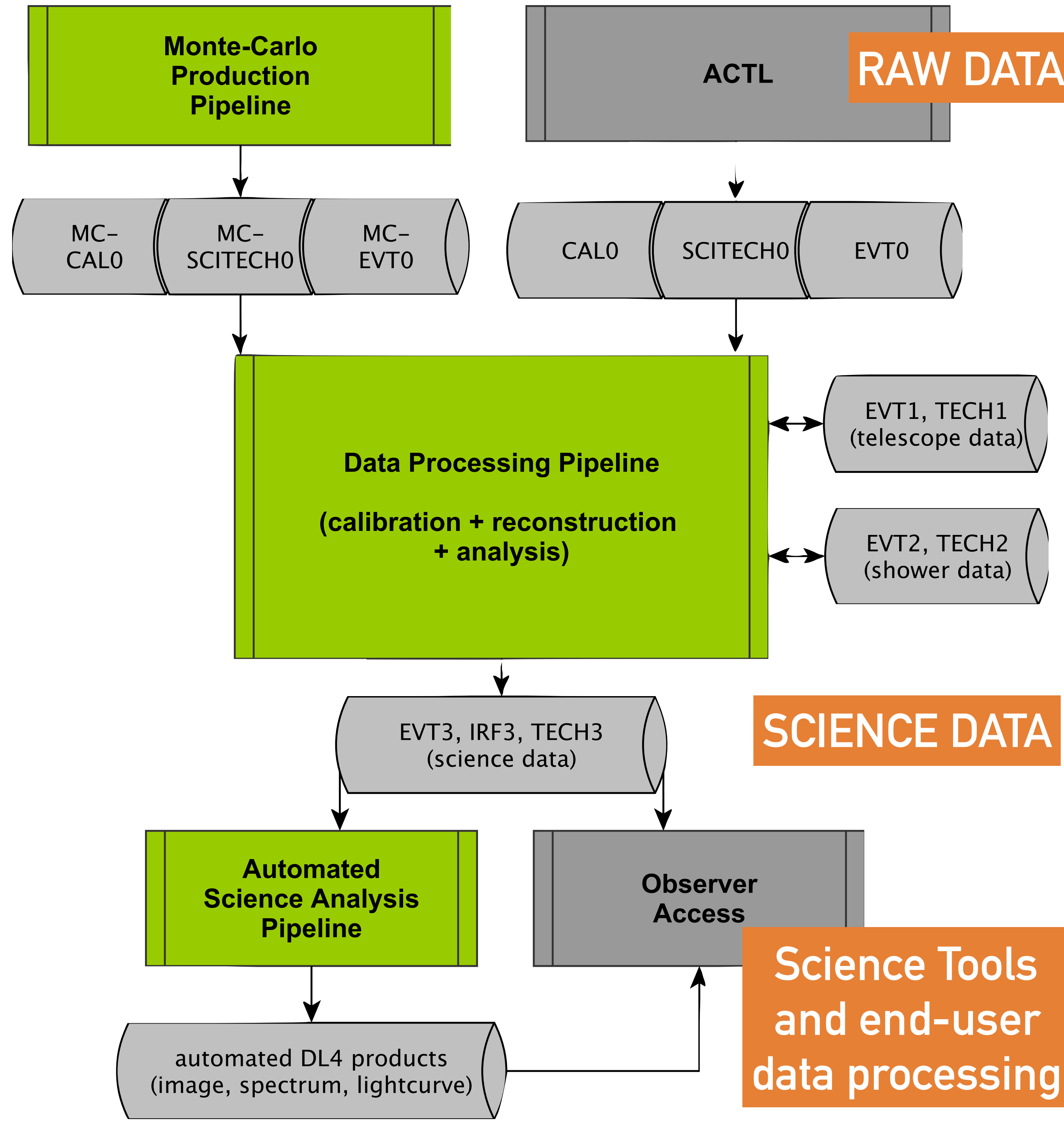
Process raw data :



- images of air-showers produced by gamma-rays or cosmic rays, seen by an array of telescopes

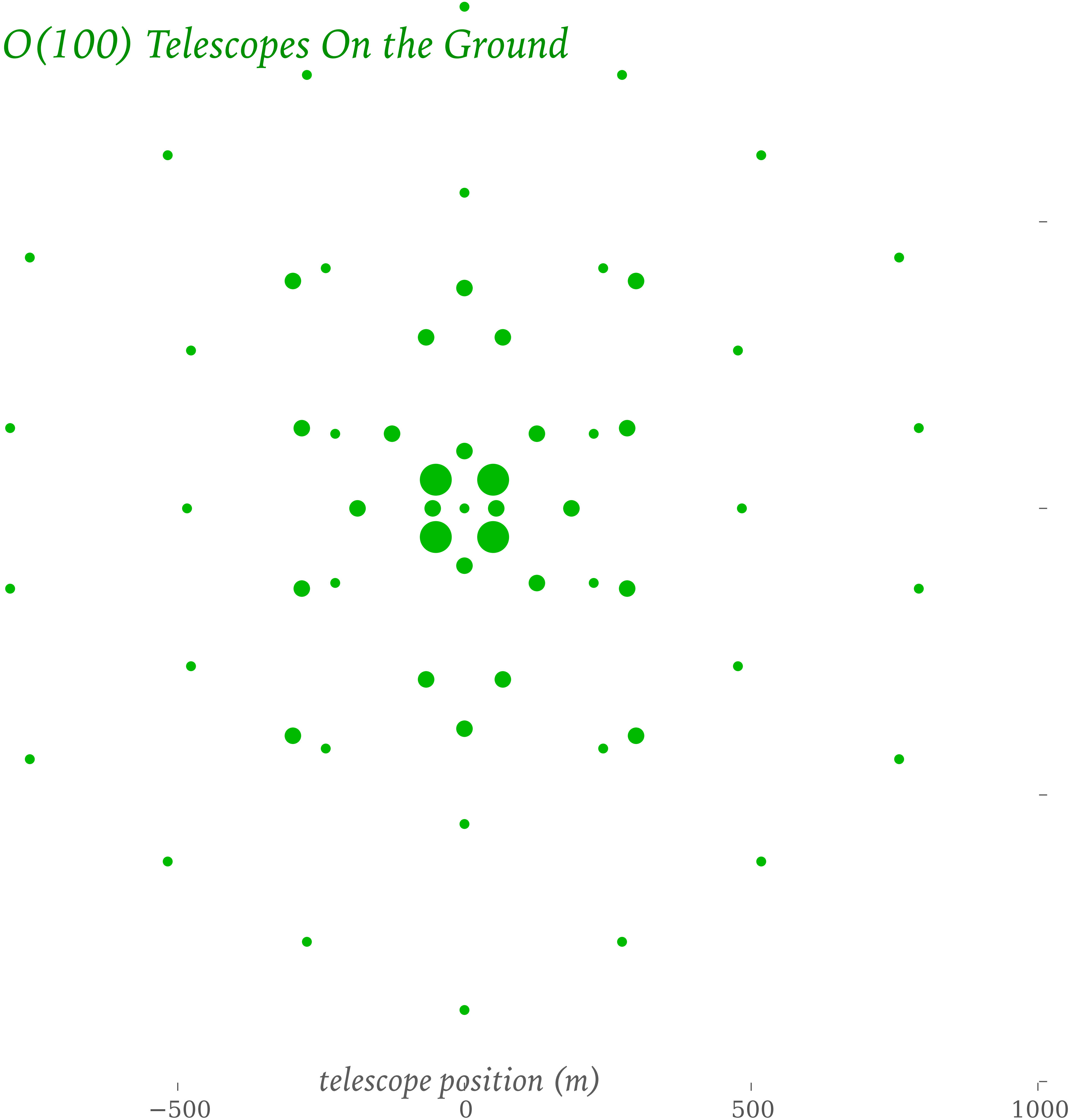
Produce science data products:

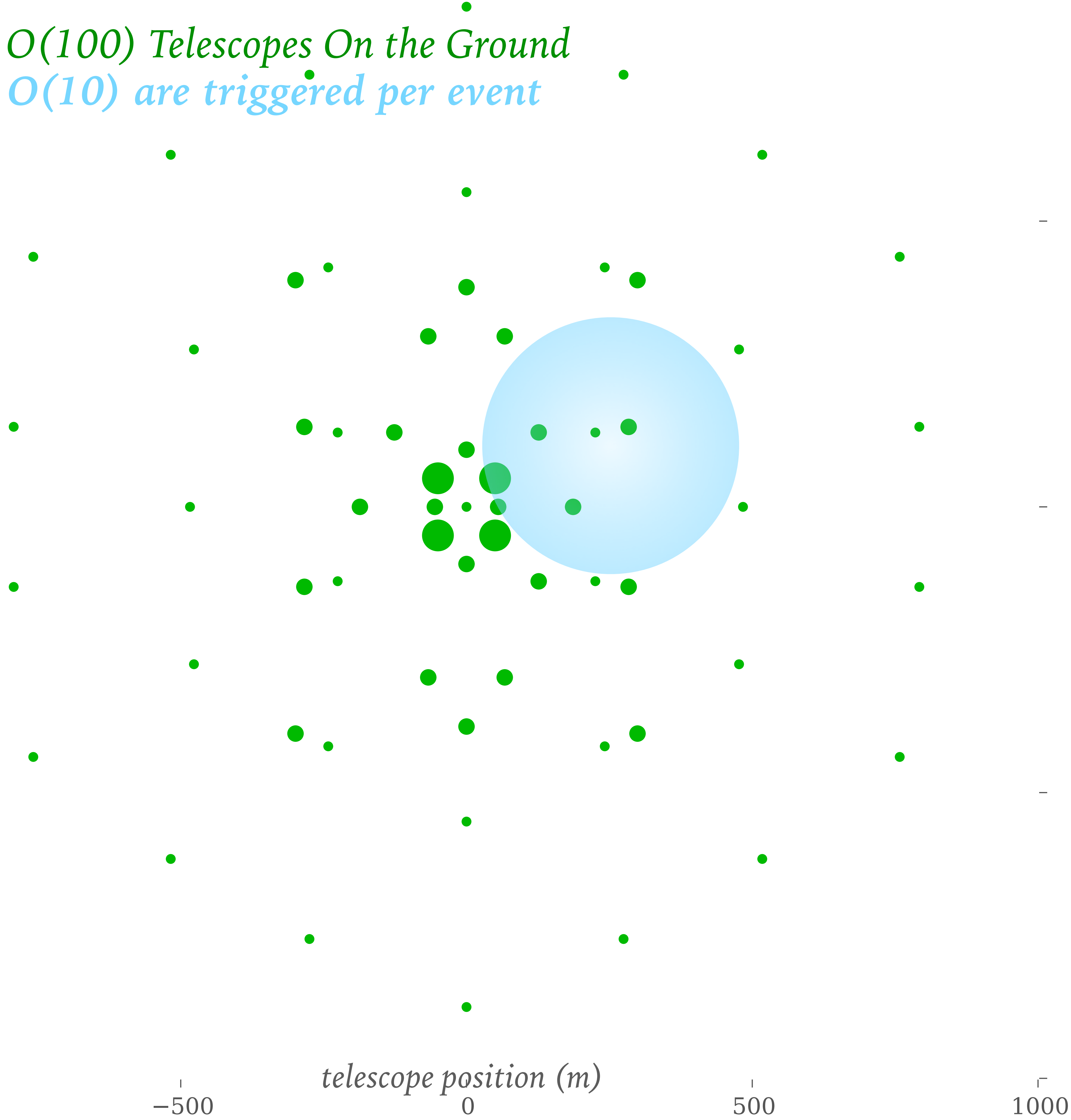
- event lists (photons + bg)
- instrument response tables
- These are then given to end-users to do science
  - GammaLib/CTools
  - similar to Fermi science tools
  - See also GammaPy
  - produce gamma-ray sky images, spectra, light-curves…



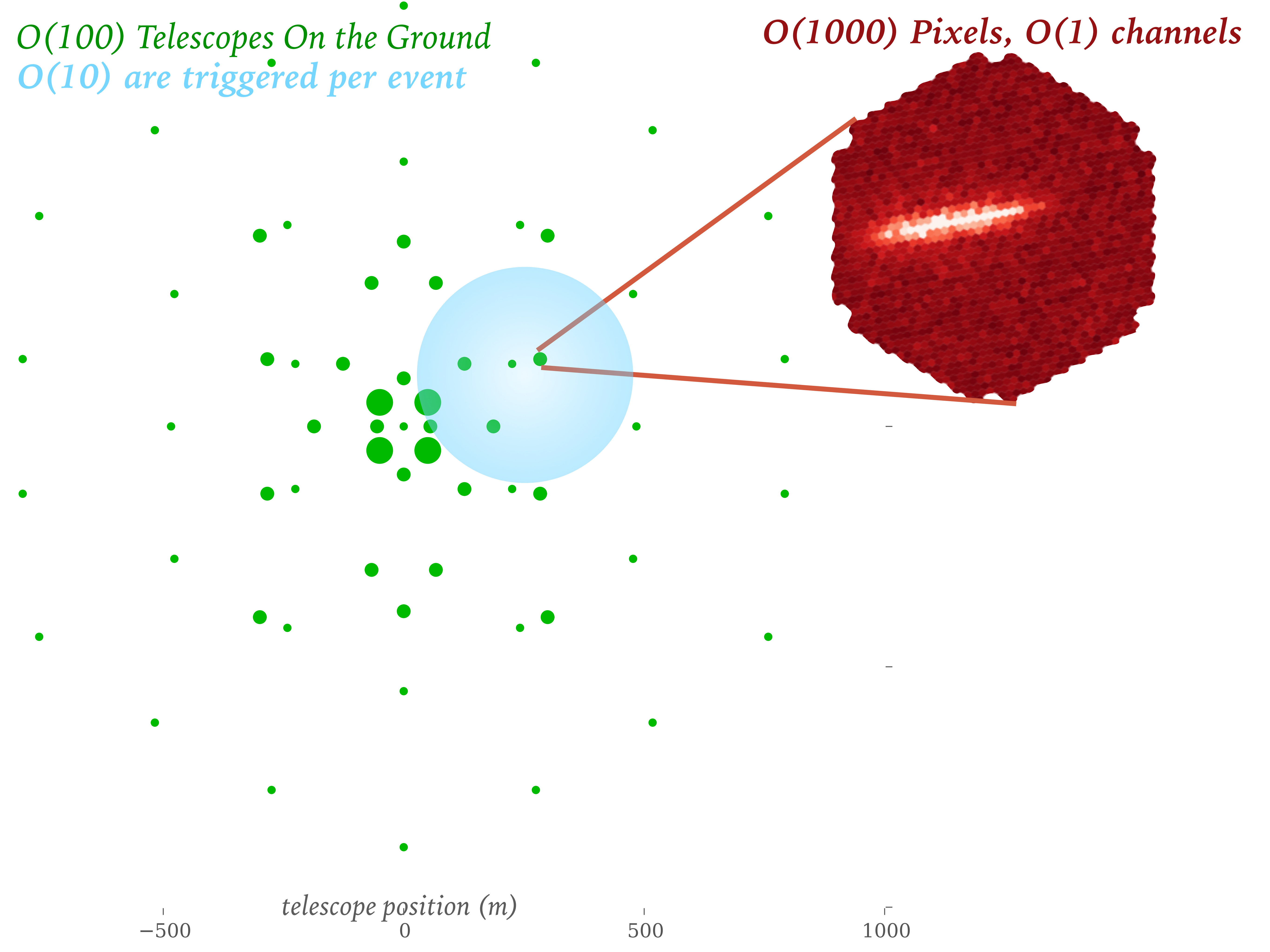
[CTA Technical Design Report]

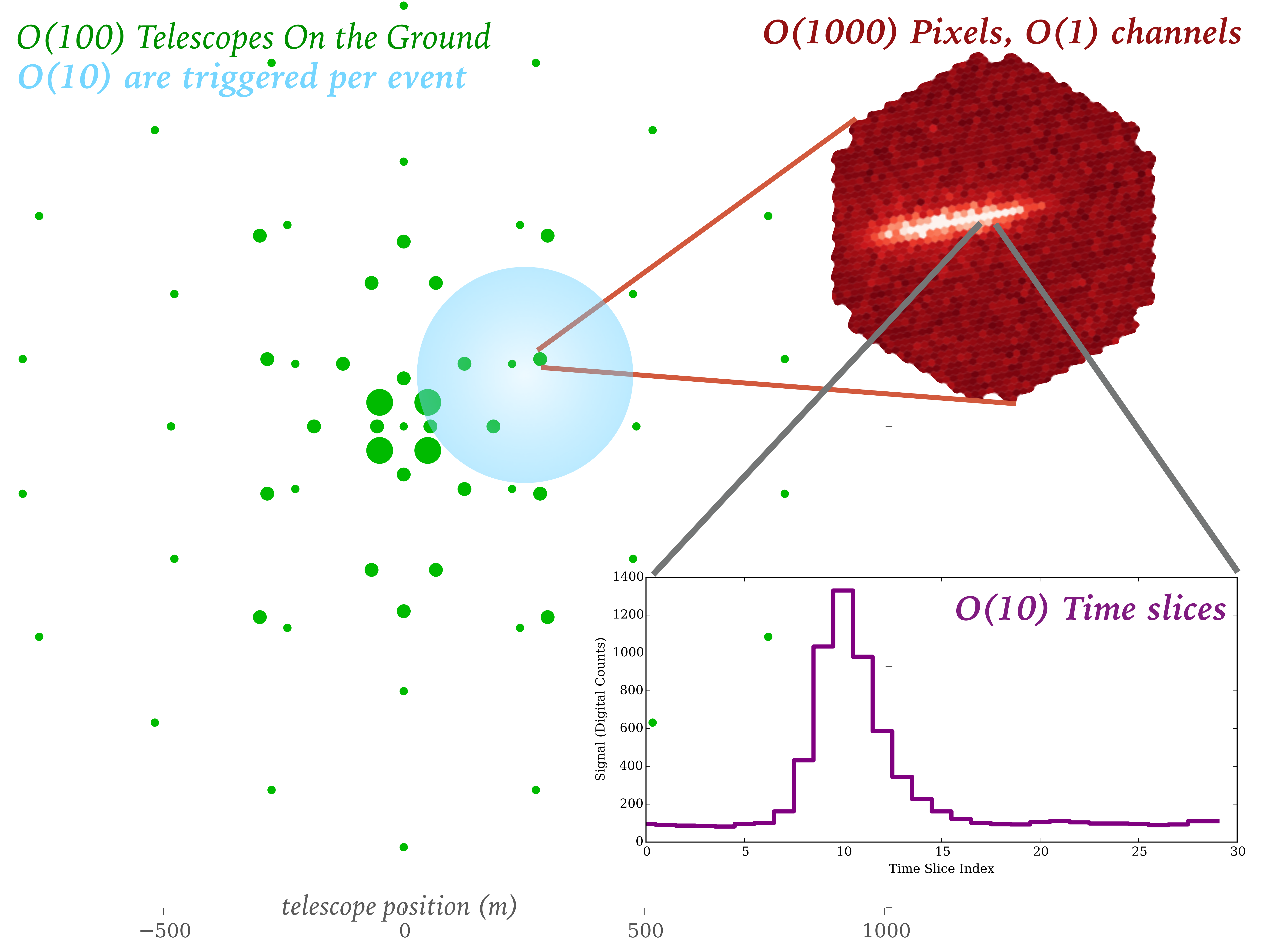
O(100) Telescopes On the Ground

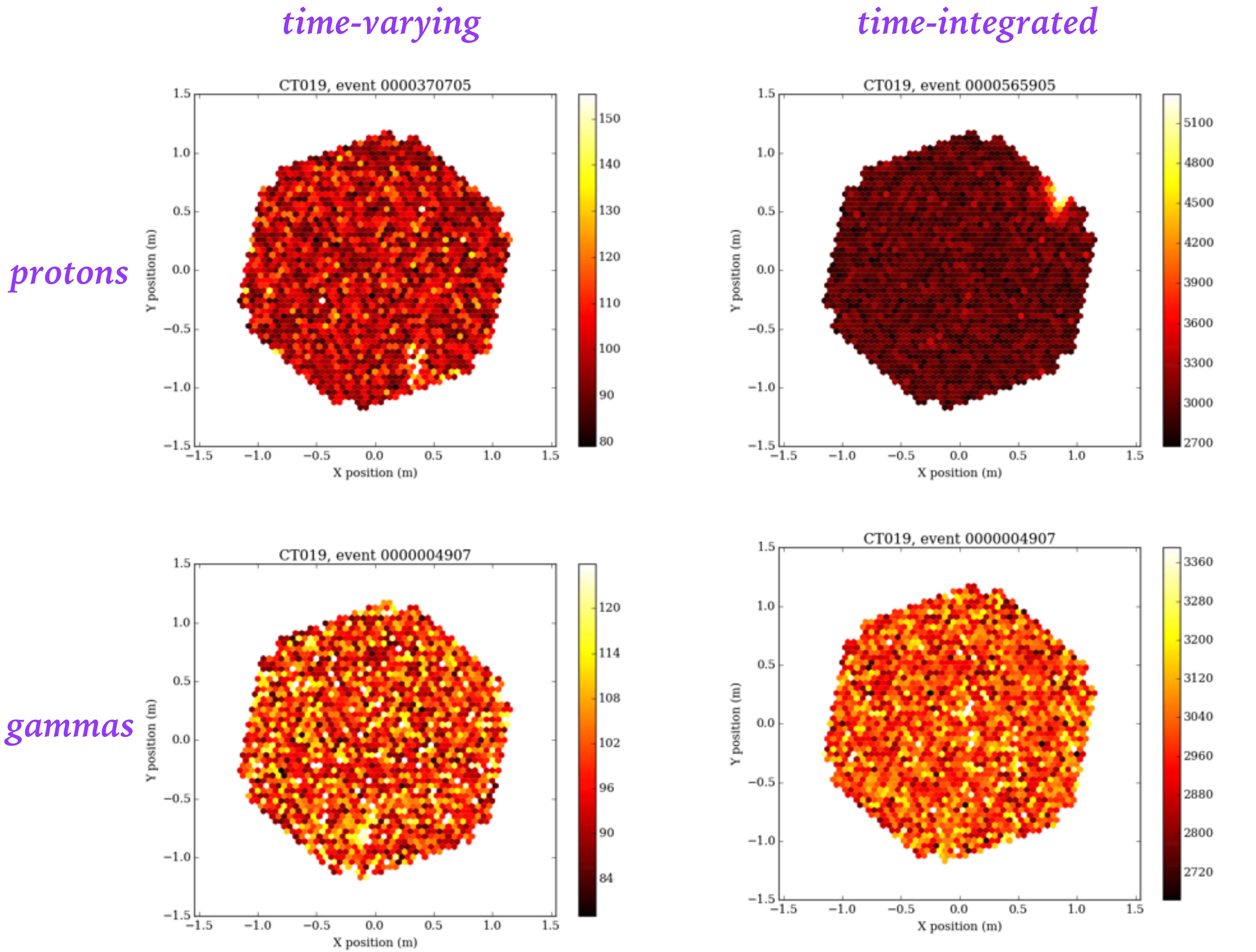




O(100) Telescopes On the Ground
O(10) are triggered per event
telescope position (m)
−500
0
500
1000


O(100) Telescopes On the Ground
O(10) are triggered per event
O(1000) Pixels, O(1) channels
telescope position (m)
−500
0
500
1000

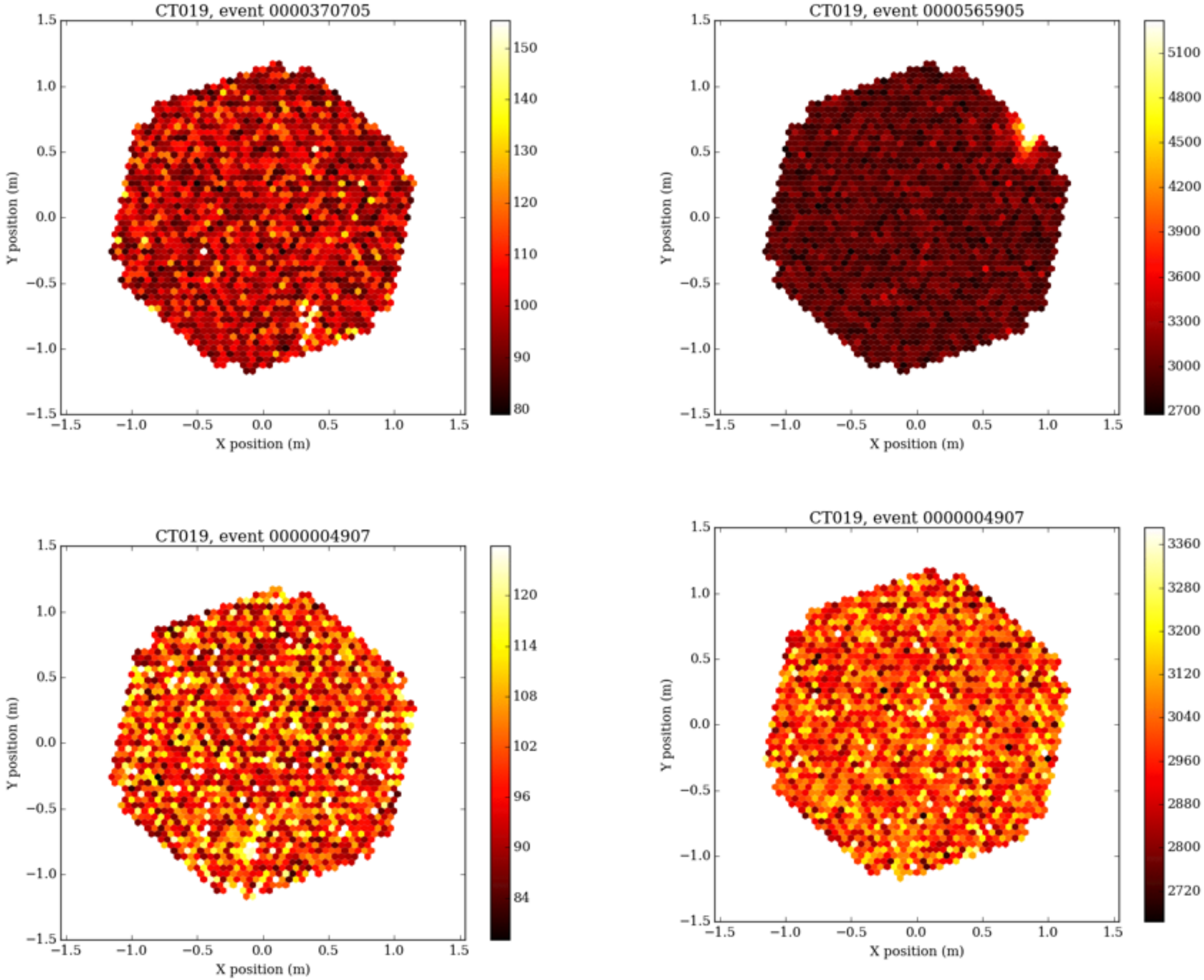

O(100) Telescopes On the Ground
O(10) are triggered per event
O(1000) Pixels, O(1) channels
O(10) Time slices
Signal (Digital Counts)
Time Slice Index
telescope position (m)
−500
0
500
1000


time-varying
time-integrated
protons
gammas
CT019, event 0000370705
CT019, event 0000565905
CT019, event 0000004907
CT019, event 0000004907
X position (m)
Y position (m)
150
140
130
120
110
100
90
80
5100
4800
4500
4200
3900
3600
3300
3000
2700
120
114
108
102
96
90
84
3360
3280
3200
3120
3040
2960
2880
2800
2720

time-varying *time-integrated*

protons

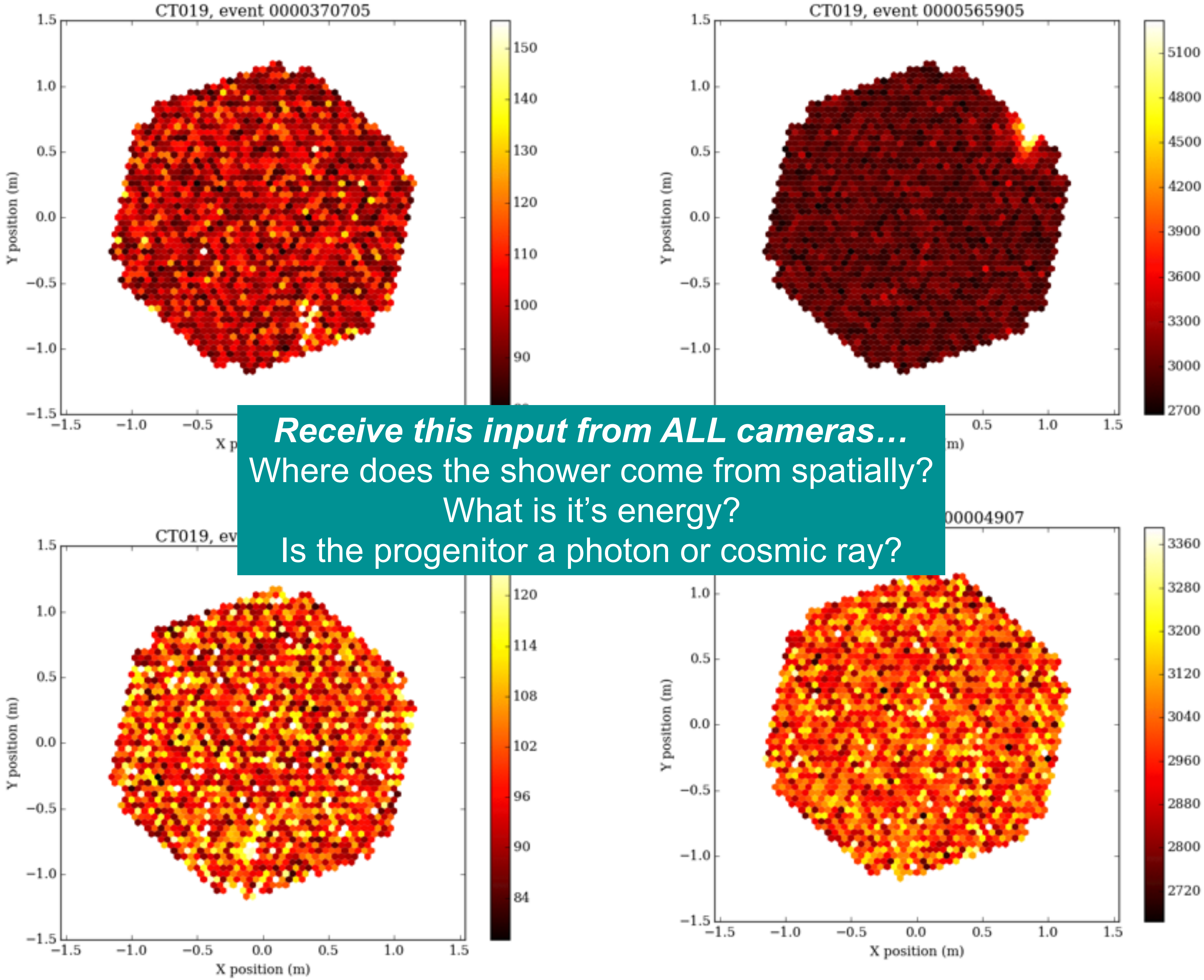
gammas



time-varying *time-integrated*

protons

gammas



Receive this input from ALL cameras…

Where does the shower come from spatially?

What is it’s energy?

Is the progenitor a photon or cosmic ray?

DATA VOLUME

DATA VOLUME

Trigger rate is O(10,000) Hz
(really more like 30kHz)

DATA VOLUME

Trigger rate is O(10,000) Hz
(really more like 30kHz)

Data volume is therefore
O(10 tels • 1000 pix • 10 times • 10000 Hz)

- = O(10) GB/s
- = 10 CERNs !

DATA VOLUME

Trigger rate is O(10,000) Hz
(really more like 30kHz)

Data volume is therefore
O(10 tels • 1000 pix • 10 times • 10000 Hz)

- = O(10) GB/s
- = 10 CERNs !

Non-trivial data volume!

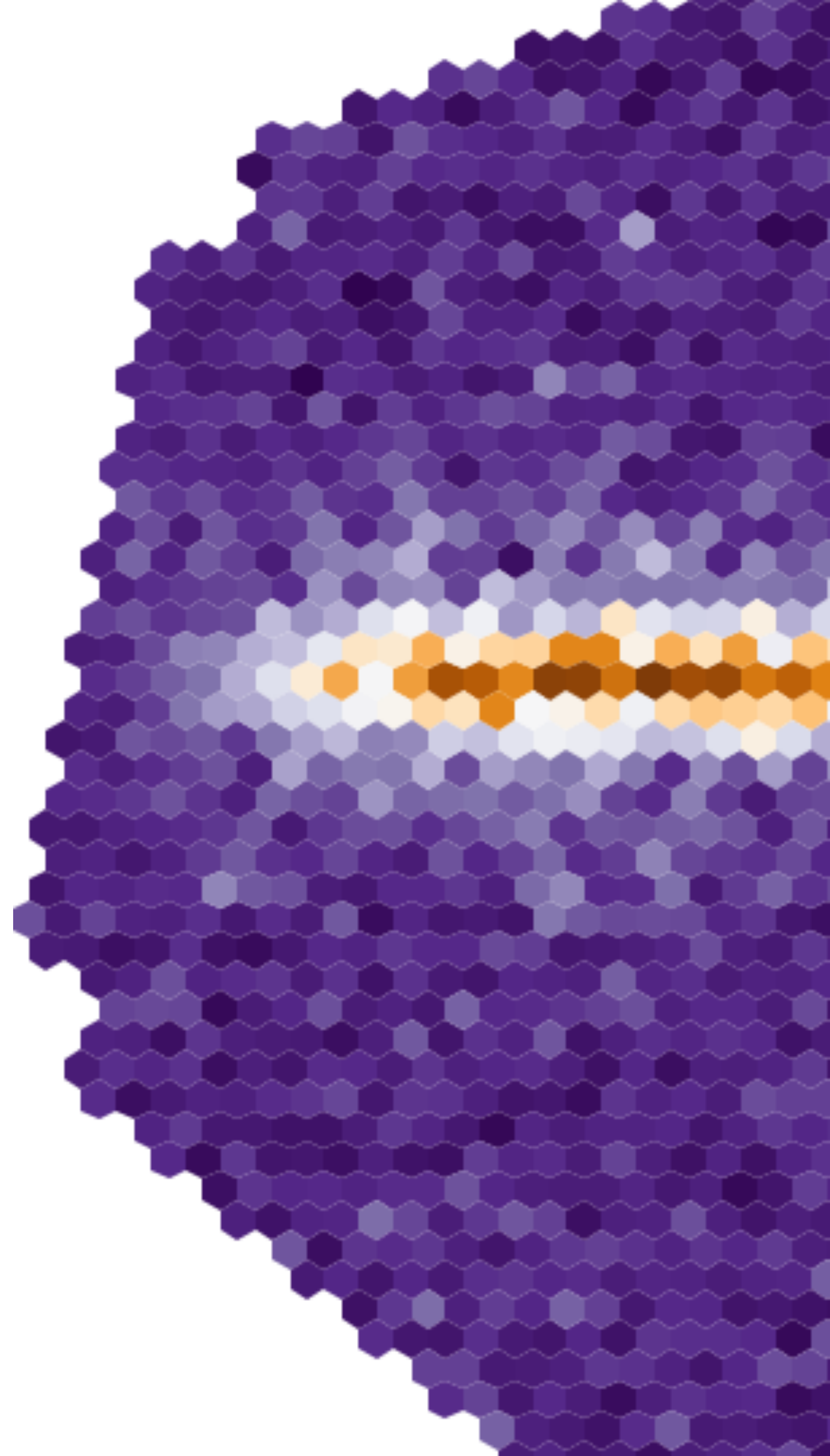
- need to reduce by a factor of >20-100 on-site (compression and suppression)
    - implies robust software,
    - streaming, possibly real-time
- even afterward estimate **4 PB/yr!**
    - will want to re-process it all at least annually! (grows in time)
    - push I/O (and CPU) limits
    - parallelism is *strictly required*

Big Data.

(well, at least for high-energy astrophysics)

PROCESSING THE DATA

what algorithms need to be applied?



TOOLS/ALGORITHMS 1

Generalization of "Images"

- non-square pixels, **triangular** or **cartesian basis**, **gaps** (6 different camera types/geometries)
- data cubes in time, or images of *time parameters*

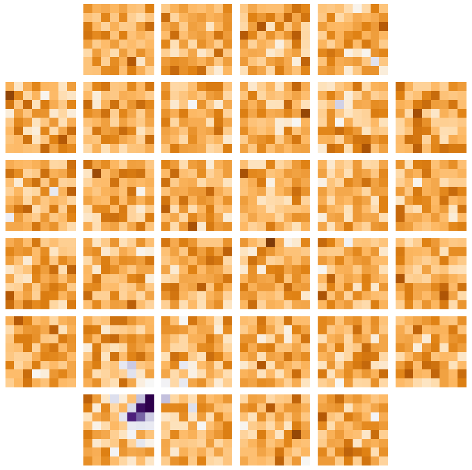
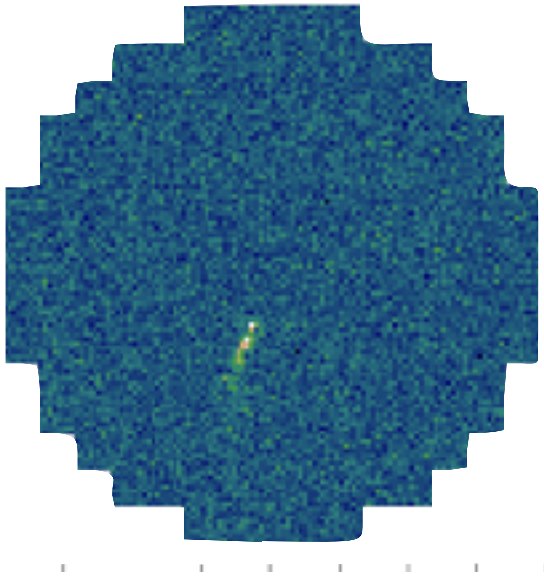
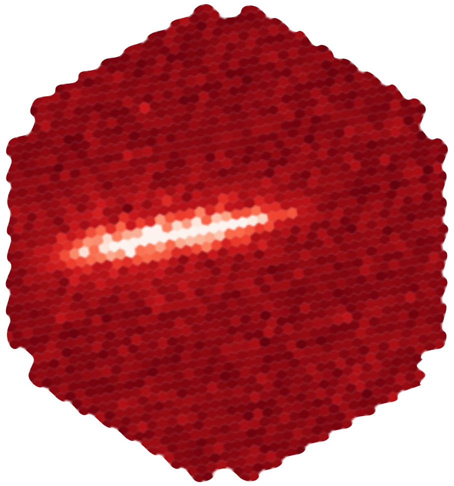


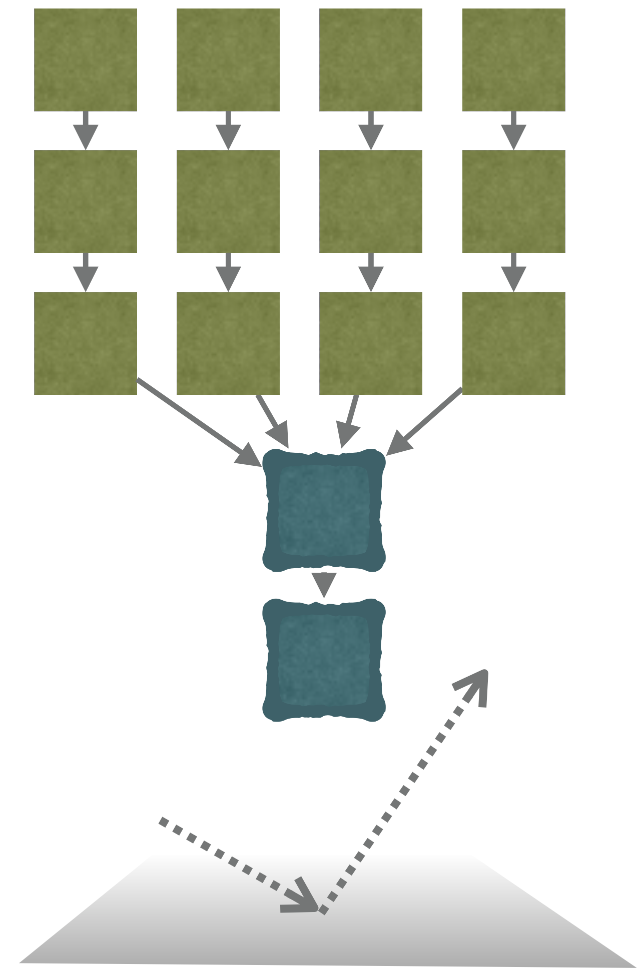
Image and Signal processing

- signal processing: peak finding, integration
- calibration (background, flatfield, time, optics…)
- image de-noising and inpainting (identification of signal region and missing information)
- image feature extraction (characterization of image)
- advanced techniques (wavelets, compressed sensing, and beyond)



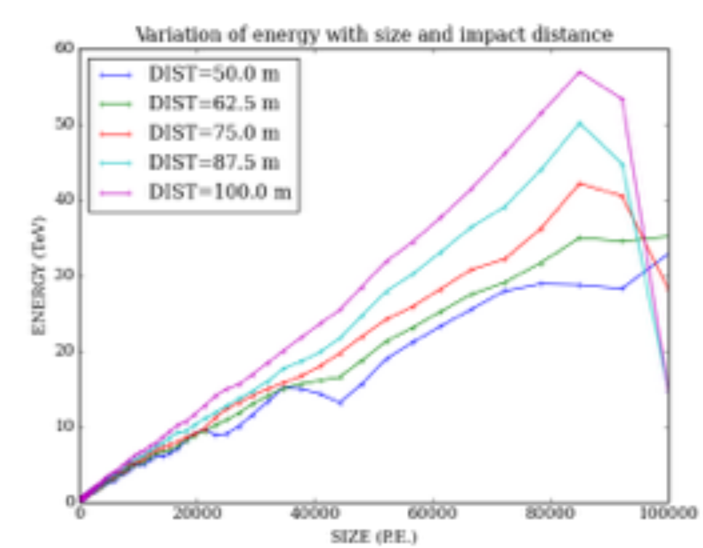


TOOLS/ALGORITHMS 2



Event Reconstruction:

- data synchronization
  ("join" operation on multiple telescope streams)
- likelihood minimization (with large dimensionality)
- ND interpolation (where N>3)
- 3D geometry and linear algebra
- coordinate transformations
  (in addition to standard astronomical ones)
  - ➤ detector plane for each telescope, including pointing corrections
  - ➤ nominal plane (common view of shower from all telescopes)
  - ➤ impact or ground plane (where the shower hits)
  - ➤ need for speed optimizations here (could contribute)
- machine learning (regression)
  - ➤ energy or shower determination from many input parameters
  - ➤ may explore even deep learning (convolutional neural networks, etc) for image processing



TOOLS/ALGORITHMS 3

Event Classification:

- is an event a gamma, proton, electron, muon?
- machine learning (classification)
  - decision trees (BDTs, Random Forests) or Support Vector Machines, etc.

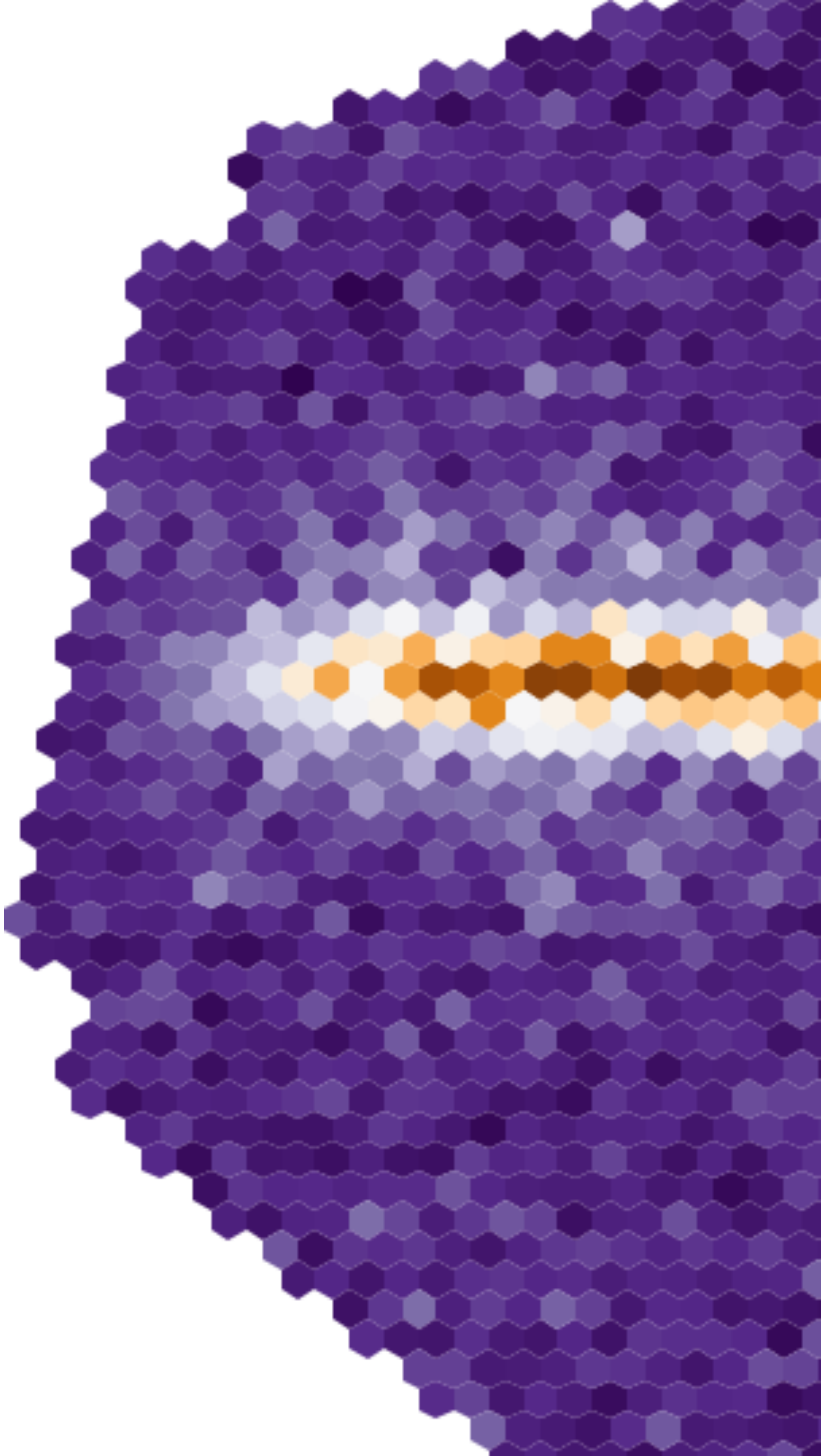
scikit-learn, nolearn, theano, TensorFlow …

Data Quality Monitoring and response matrices:

- diagnostic and summary plots
- visualizations (debugging of methods, see images, 3D recon…)
- histogramming and interpolation (multi-dimensional)
- data fitting of simple or complex models
- outlier detection

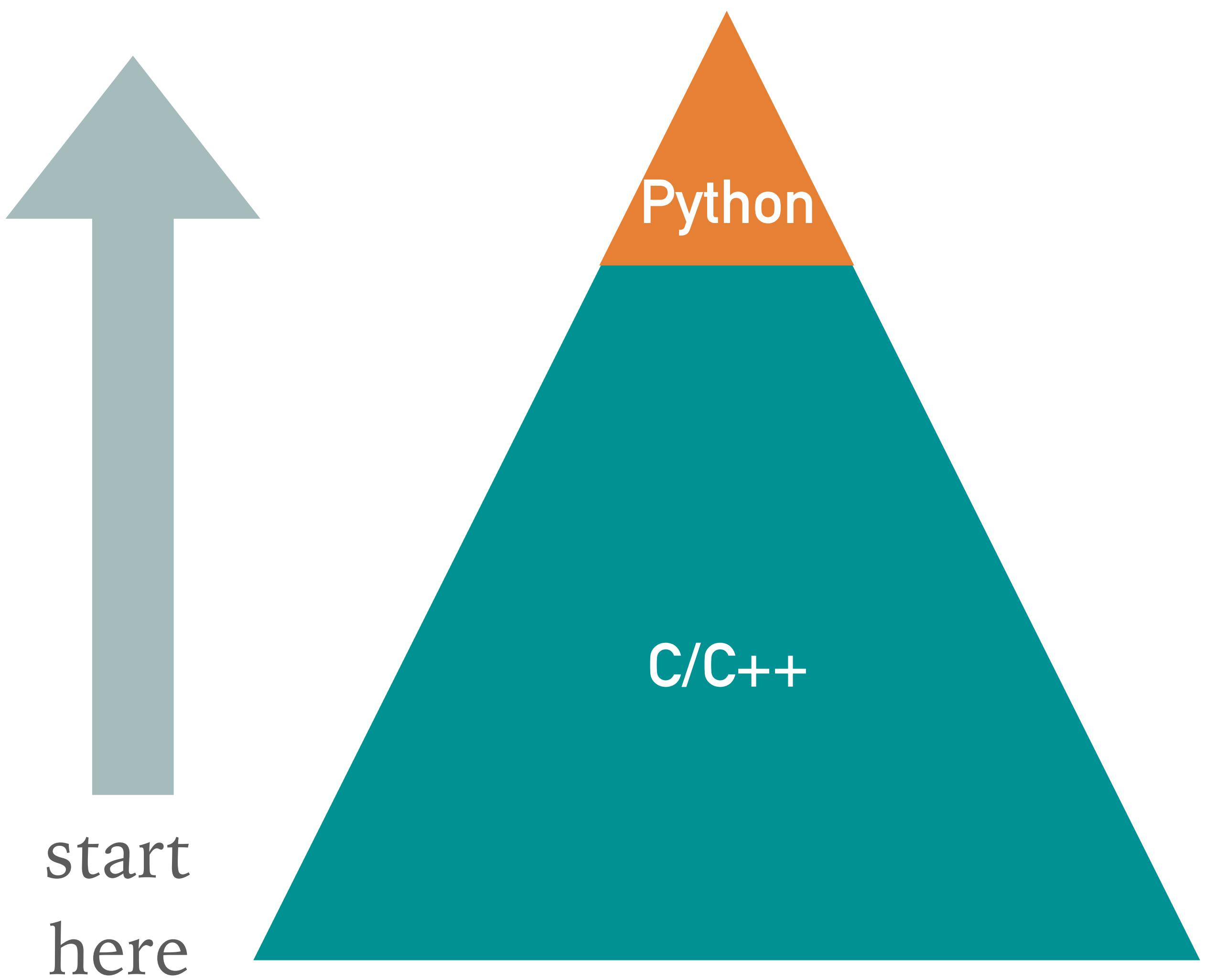
FRAMEWORK

how do we use python to help implement these algorithms?



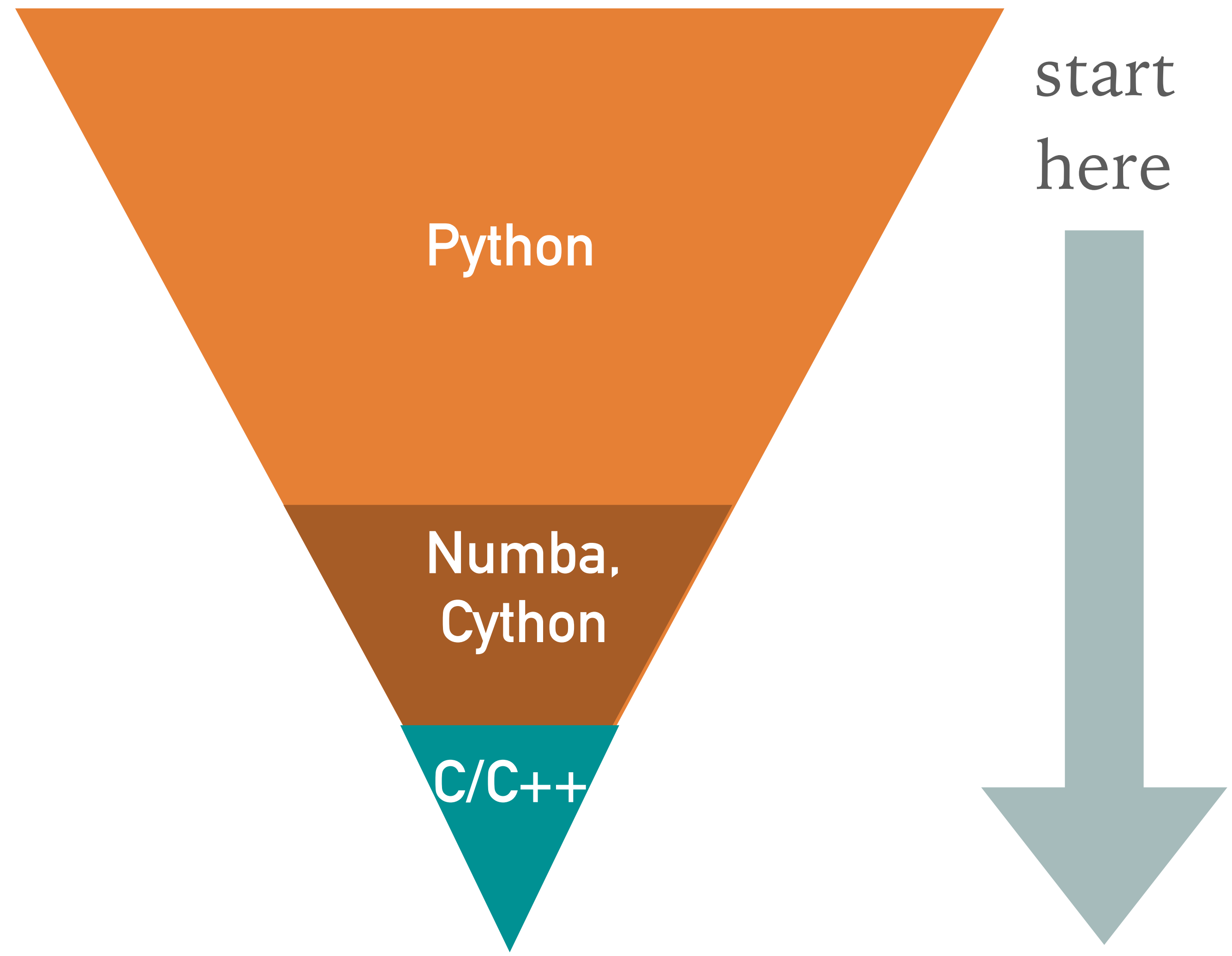
BUILDING A FRAMEWORK

Bottom-Up approach



Most current frameworks did it this way (if they use python at all)

Top-Down approach



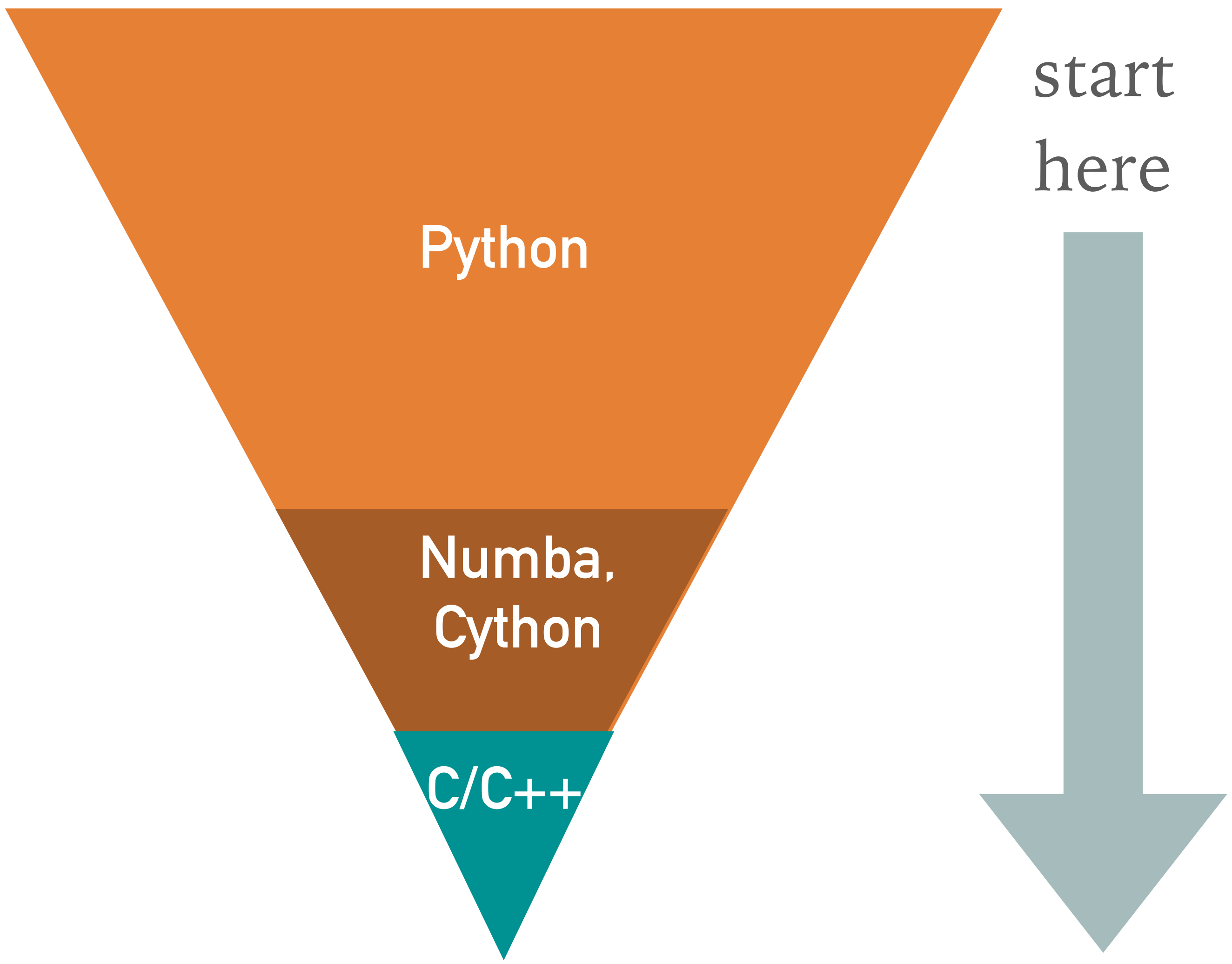
Our approach: start early with python and high-level API

BUILDING A FRAMEWORK

Bottom-Up approach

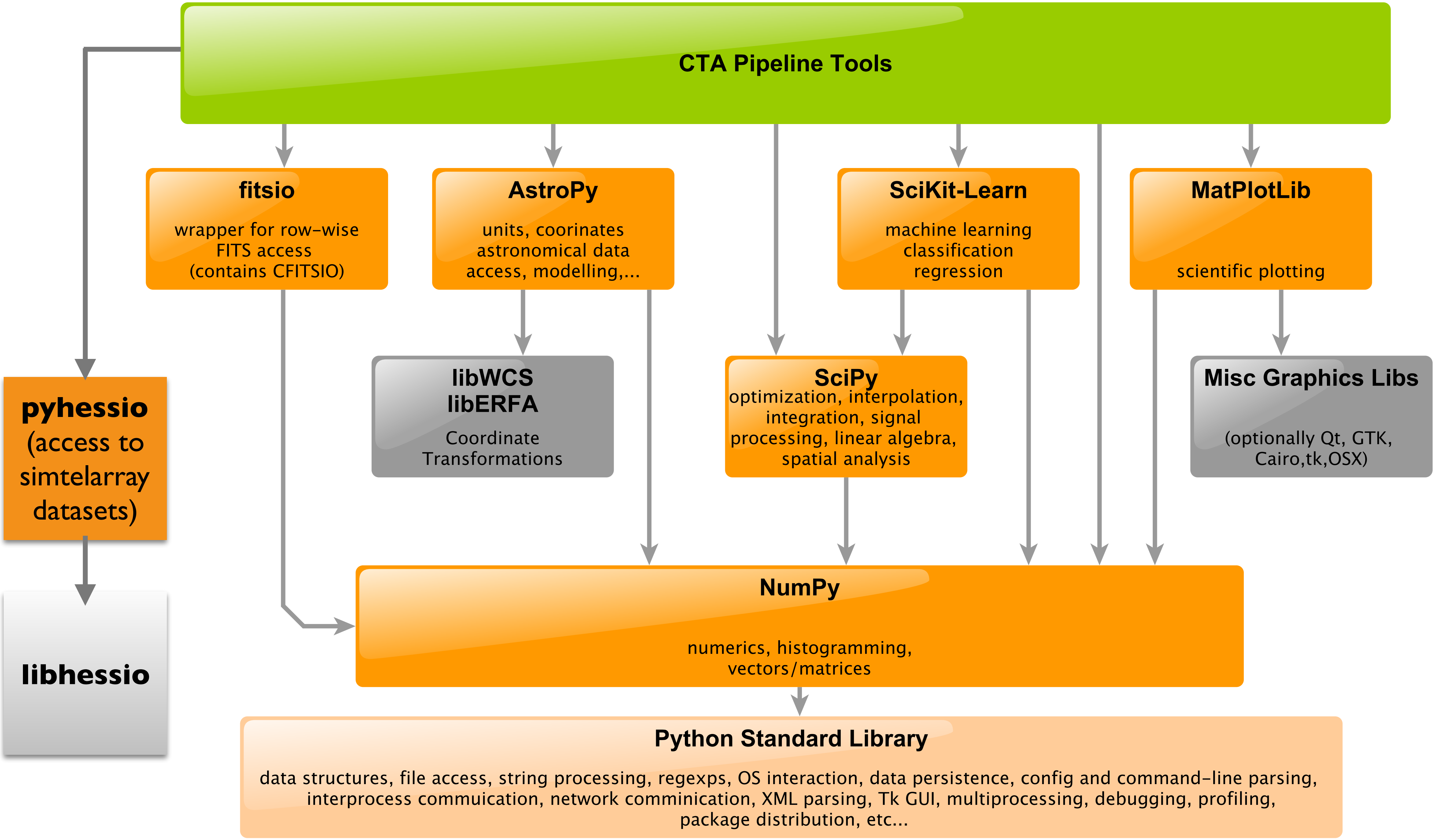
Most current frameworks did it this way (if they use python at all)

Top-Down approach



Our approach: start early with python and high-level API

CORE DEPENDENCIES



leverage code developed by wide scientific and industrial community!

HOW DOES PYTHON HELP?

Clean Design is very important, particularly for dependencies:

- python community has a relatively common design direction
- language stresses simplicity and good practices

Support libraries aren't just from a small team:

- large developer bases
- wider experience (not just astro or HEP, biology, stats, economics, …)

Packaging and distribution is quite advanced:

- pypi, conda, …

Lots of good work to build on from developers of AstroPy and derivative packages like GammaPy!

- The excellent design work provides a model for other packages
- solved a lot of common problems that we can now ignore!

REPLACE THIS KIND OF CODE:

```
int Intensity::TailcutsCleaner::Tailcut(Sash::Telescope& tel)
{

  DEBUG_OUT << "Telescope: " << tel.GetId() <<"
"<<fInputIntensityName.c_str()<< std::endl;

  const Sash::IntensityData* intcal
    = tel.Get<Sash::IntensityData>(fInputIntensityName.c_str());

  if (intcal == 0){
    Error("Tailcut","Cannot get Intensity Object");
    return 0;
  }
  Sash::IntensityData* intclean =
tel.Handle<Sash::IntensityData>(GetName());

  Sash::PointerSet<Sash::Pixel>& nzclean = intclean->GetNonZero();
  intclean->SetDataMode(Sash::IntensityData::ZeroSup);

  // Assume Zero Suppression Mode!!
  const Sash::PointerSet<Sash::Pixel> &nzcal
    = (intcal->GetDataMode() == Sash::IntensityData::ZeroSup
       ? intcal->GetNonZero()
       : tel.GetConfig()->GetPixelsInTel());

  Sash::PointerSet<Sash::Pixel>::const_iterator pixcal = nzcal.begin();

  for(; pixcal!= nzcal.end(); ++pixcal){

    // Check if Pixel passes high Tailcut
    Float_t intensity = intcal->GetIntensity(*pixcal)->fIntensity;
    UChar_t chan =  intcal->GetIntensity(*pixcal)->fUsedChannel;

    if (intensity < fTailCutThresholds[1]) continue;

    Sash::ListIterator<Sash::Pixel> neighbour
      = (*pixcal)->GetNeighbourList().begin();
    Sash::ListIterator<Sash::Pixel> endNeighbour
      = (*pixcal)->GetNeighbourList().end();

    for(;neighbour != endNeighbour; ++neighbour) {

      Float_t intensitySecond
          = intcal->GetIntensity(*neighbour)->fIntensity;
      UChar_t chanSecond
          =  intcal->GetIntensity(*neighbour)->fUsedChannel;

      Int_t accept;

      // Check if Neighbour Pixel is in between High and Low Cuts, if
      // so add it to image

      if ((accept = (intensitySecond > fTailCutThresholds[0]))
        && (intensitySecond < fTailCutThresholds[1])){
        if (nzclean.insert(*neighbour)){
          intclean->GetIntensity(*neighbour)->fIntensity
            = intensitySecond;
          intclean->GetIntensity(*neighbour)->fUsedChannel
            = chanSecond;
        }
      }

      // Add original Pixel if it has a neighbour above LOW limit

      if (accept && (nzclean.insert(*pixcal))){
        intclean->GetIntensity(*pixcal)->fIntensity = intensity;
        intclean->GetIntensity(*pixcal)->fUsedChannel = chan;
      }
    }

  }


  return (nzclean.size() > 0);

}
```

WITH THIS CODE:

Much more maintainable:

```
def tailcuts_clean(geom, image, pedvars, picture_thresh=4.25,
                   boundary_thresh=2.25):

    clean_mask = image >= picture_thresh * pedvars
    boundary_mask = image >= boundary_thresh * pedvars
    boundary_ids = [pix_id for pix_id in geom.pix_id[boundary_mask]
                    if clean_mask[geom.neighbors[pix_id]].any()]

    clean_mask[boundary_ids] = True
    return clean_mask
```

note: this can probably be optimized even further

STREAM PROCESSING:

Can't load all events at once into memory! process event-by-event (or chunk by chunk)…

Natural in Python due to generators/iterators!

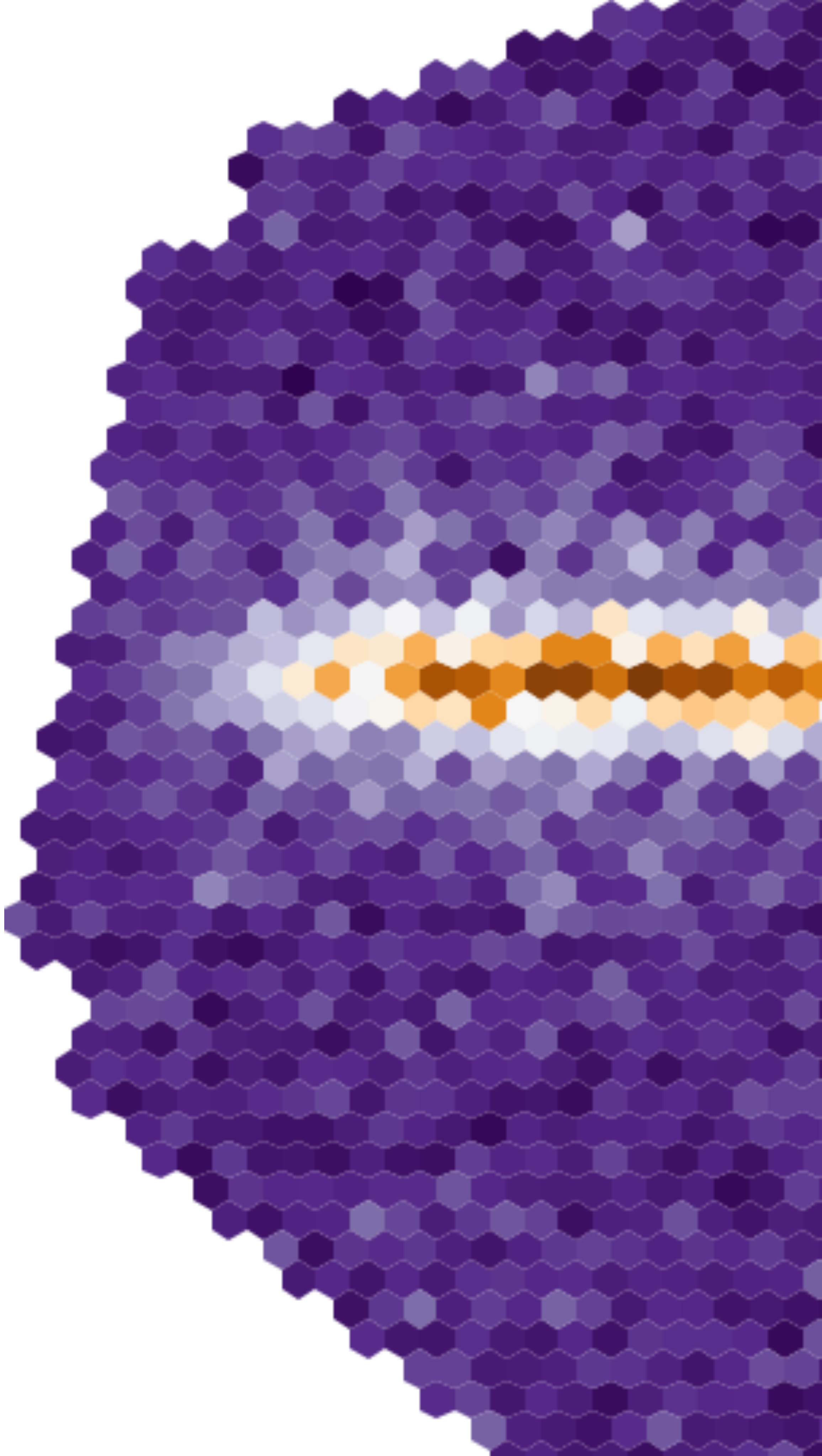
```
from ctapipe.io.hessio import hessio_event_stream

events = hessio_event_stream("somefile.simtel.gz")

for event in events:
   do_things_to_event(event, threshold=1.23)
   do_something_else(event)

# even this is possible! (with some generator chaining and operator overloading)
pipe = events | do_things_to_event(threshold=1.23) | do_something_else
# equivalent to:
pipe = do_something_else(do_things_to_event(events, threshold=1.23))
map(None, pipe)
```

- event is a container class or dictionary, only a single event in memory at once to save memory
- can use any construct that works on iterables:
  - **itertools** (split, chain, repeat, group_by, etc)
  - **multiprocessing** (multiprocessing.Pool.map(func, events))
  - other fancier things… (*see later…*)



SPEED?

Python is too slow, isn't it?

"premature optimization is the root of all evil" — *Knuth*

yes, but even so…

NUMPY

Simplest way to achieve high speeds:

- avoid for-loops with
  - ➤ vector operations on NDArrays
  - ➤ advanced slicing notation, fancy-indexing
  - ➤ broadcasting, ufuncs
- often to cleaner, shorter, more maintainable code
- but <u>not always easy to implement or comprehend!</u>

```
def np_jacobi( a ):
   """ solve heat equation for entire 2D array at once """
   a_ijp1 = a[1:-2,1:-2]
   a_ij   = a[1:-1,1:-1]
   a_im1j = a[0:-2,1:-1]
   a_ip1j = a[2:,1:-1]
   a_ijm1 = a[1:-1,0:-2]
   a_ijp1 = a[1:-1,2:]

   a[1:-1,1:-1] = 0.5*a_ij + 0.125*( a_im1j + a_ip1j + a_ijm1 + a_ijp1 )
   return a
```

NUMBA

Automatic Just-In-Time (JIT) compilation via LLVM

- pure python code, no need to rewrite
- can even produce GPU code
- achieve speed similar to Fortran (!) for simple functions
- but some complex data-structures or features are not supported

```
from numba import jit

@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
```

practically magic!

http://numba.pydata.org

CYTHON

optimizing static compiler for Python and Cython extension language

- write python (or python-like) code that natively works with C/C++ (to/from)
- fine tune optimizations
- a bit more control (and work) than numba:

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

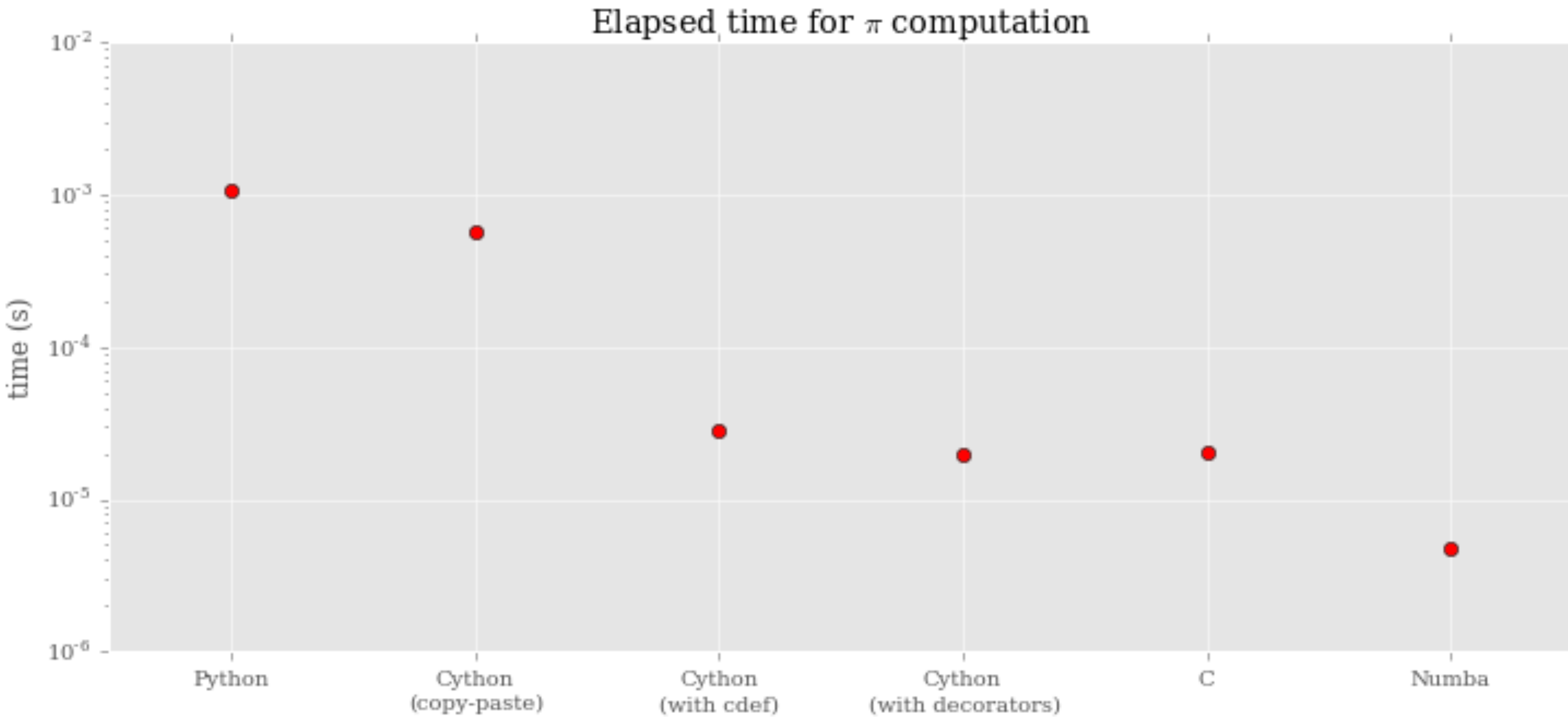
FALLBACK: C/C++!

We can always write (simple) code in C/C++ and wrap it!

- preferred method if the code needs to be shared with other non-python frameworks (DAQ, etc)
- only allow for low-level algorithms to avoid dependency creep
  - no reliance on complex data structures
  - no (or nearly no) dependencies

Many wrapper possibilities:

- **ctypes** (used to wrap our low-level data format library in a few lines)
- **swig** (auto-generate interfaces, including support for numpy, etc)
- **boost::python** (auto-generate from C++ classes)



excerpted from "Code optimization and good practices: make the best of Python", CEA Astrophysics Python Bootcamp, Marc Joos 2015

REPLACEMENT INTERPRETERS

PyPy:

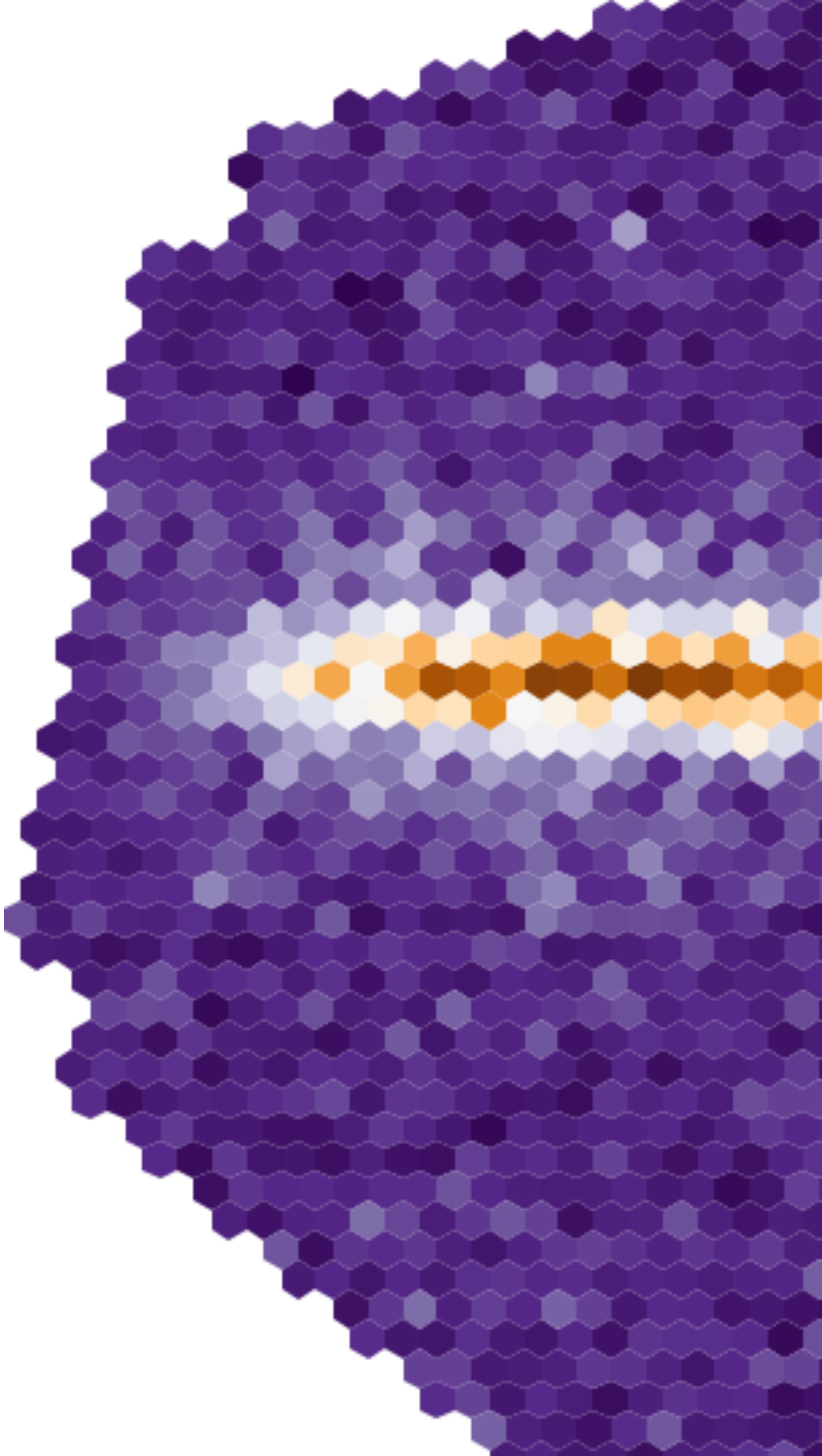
- alternative to CPython with automatic native JIT compilation
- fairly mature
- http://pypy.org

Pyston:

- fully LLVM compiled python, developed by Dropbox
- not yet ready for real usage (so far only Python 2.7 support)
- http://blog.pyston.org

PARALLEL-IZATION

Fast isn't enough: more computers!



PARALLELISM WITH PYTHON

Fortunately, lots of amazing python libraries to help!
(due to real "Big Data" and web/cloud computing)

- most leverage generator/iterator or NumPy standards
- will allow us to try out multiple solutions without being locked into one design

Many interesting solutions…

- multiprocessing (on single machine, probably not appropriate for cluster, but the same interfaces is used for other solutions)

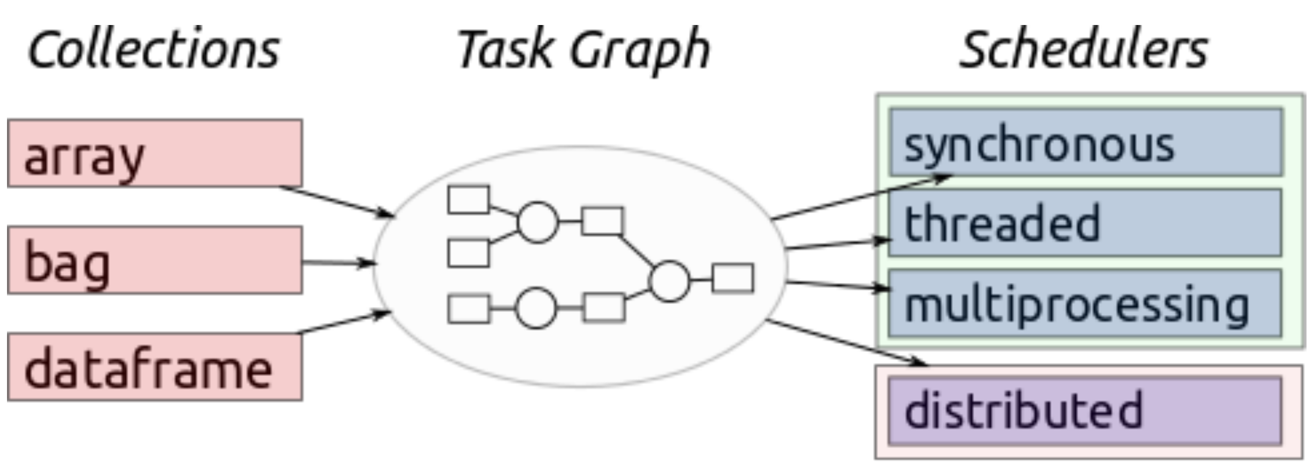
```
import multiprocessing as mp
pool = mp.Pool(processes=10)
pool.map(do_work, event_stream)
```

- **Dispy** (http://dispy.sourceforge.net) - parallel with coroutines
- **JobLib** (https://pythonhosted.org/joblib/) distributed stream-based pipelines, in the style of LabView
- **Disco**, Hadoop, etc: map-reduce style methods
- **execnet**: launch python functions as "servers" that can receive and send results
- **celery** task graphs: streaming for web

ADVANCED PARALLELIZATION

Dask

- modern framework for numeric parallelization, fully python
- fairly new, so scalability not well known
- several interfaces, not all can be applied to our problem, but easy to try.
  - *dask.do* interface and *task graphs* look promising

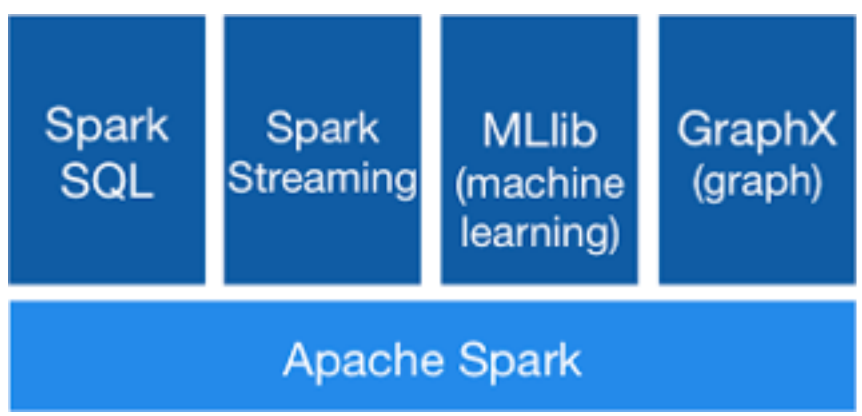


http://dask.pydata.org

ADVANCED PARALLELIZATION

PySpark (python interface to Apache Spark)

- mature technology
- Spark Streaming scales to huge sizes
- uses HDFS for I/O parallelization like Hadoop
- Supporting parallelized libraries for



- however, base system runs on Java

https://spark.apache.org/docs/latest/api/python/

FINAL COMMENTS

This is still somewhat unexplored territory (at least in HEP/astro)

Main issues are still :

- speed (both code-level and parallelization)
- Long-term maintenance (need a system that works over 30 years)
- robustness to software evolution, changes to dependencies
- ability to adapt to new technologies in the future
- However… no solution is perfect. C++ or Java do not solve these any better than python.

Final thought: our developments may lead back to the community!

- e.g. faster algorithms in astropy/scipy (if we find the existing ones aren't good enough)
- new signal processing techniques
- benefits for all!