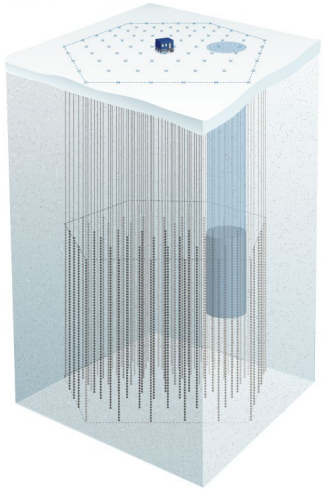
Searching High-Energy Neutrinos with IceCube and Python

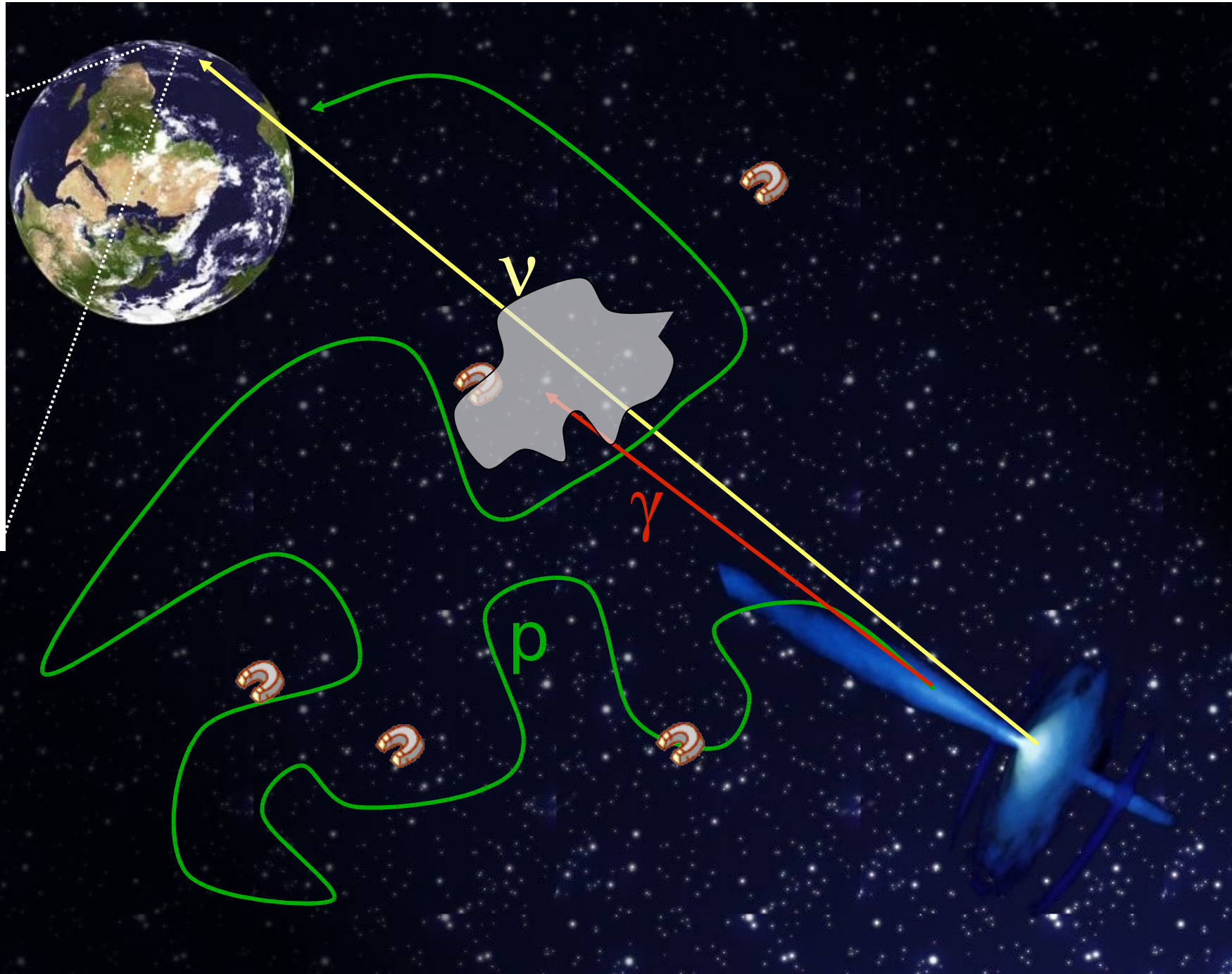
Robert Franke, Eike Middell, Sebastian Panknin, Bernhard Voigt
for the IceCube Collaboration

Outline

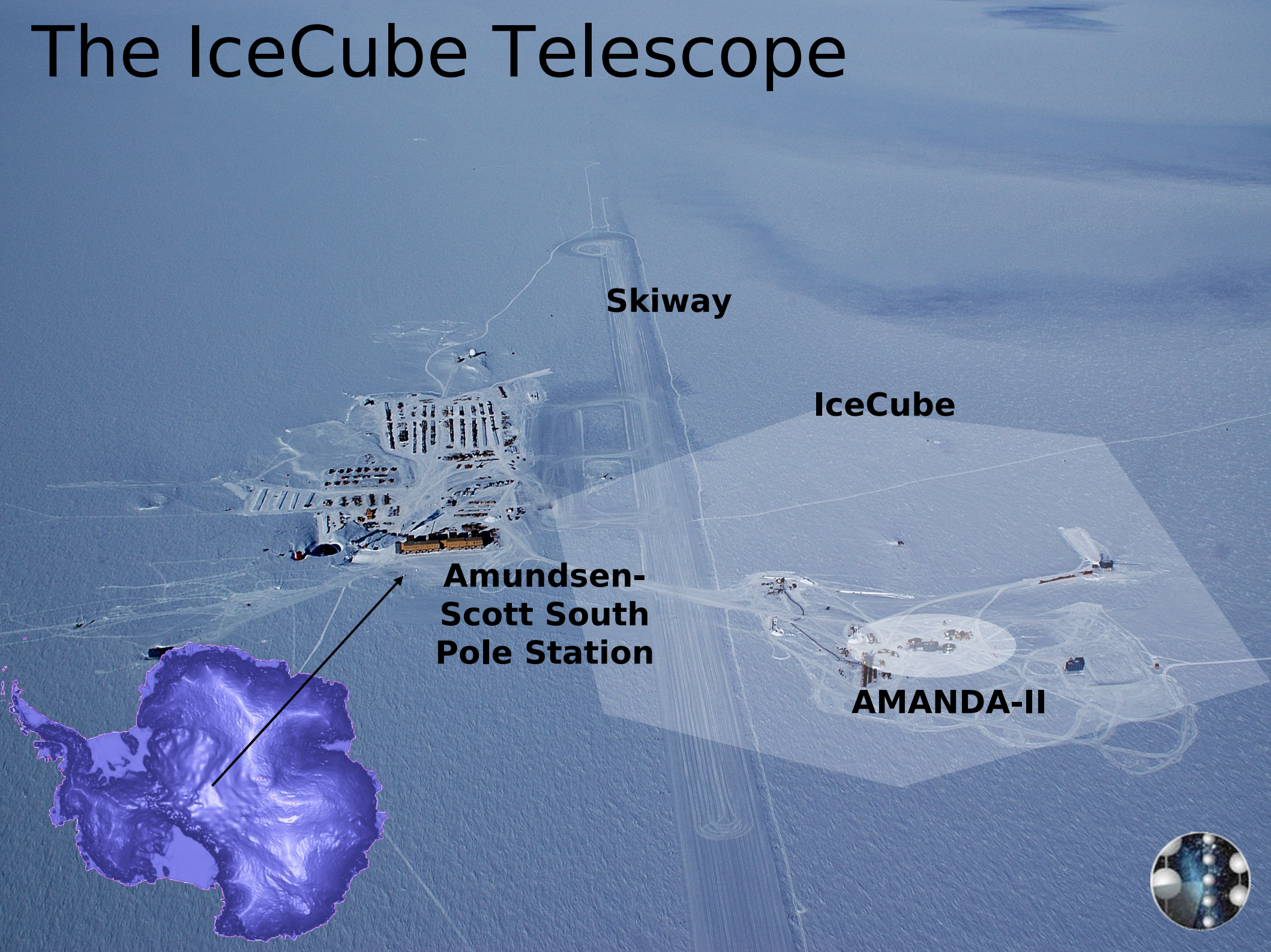
- Introduction: IceCube – Neutrino Astronomy
- Data flow (IceTray software framework)
- Controlling IceTray with Python
- Prototyping with Python
- Analysis Tools

Neutrino-Astronomy

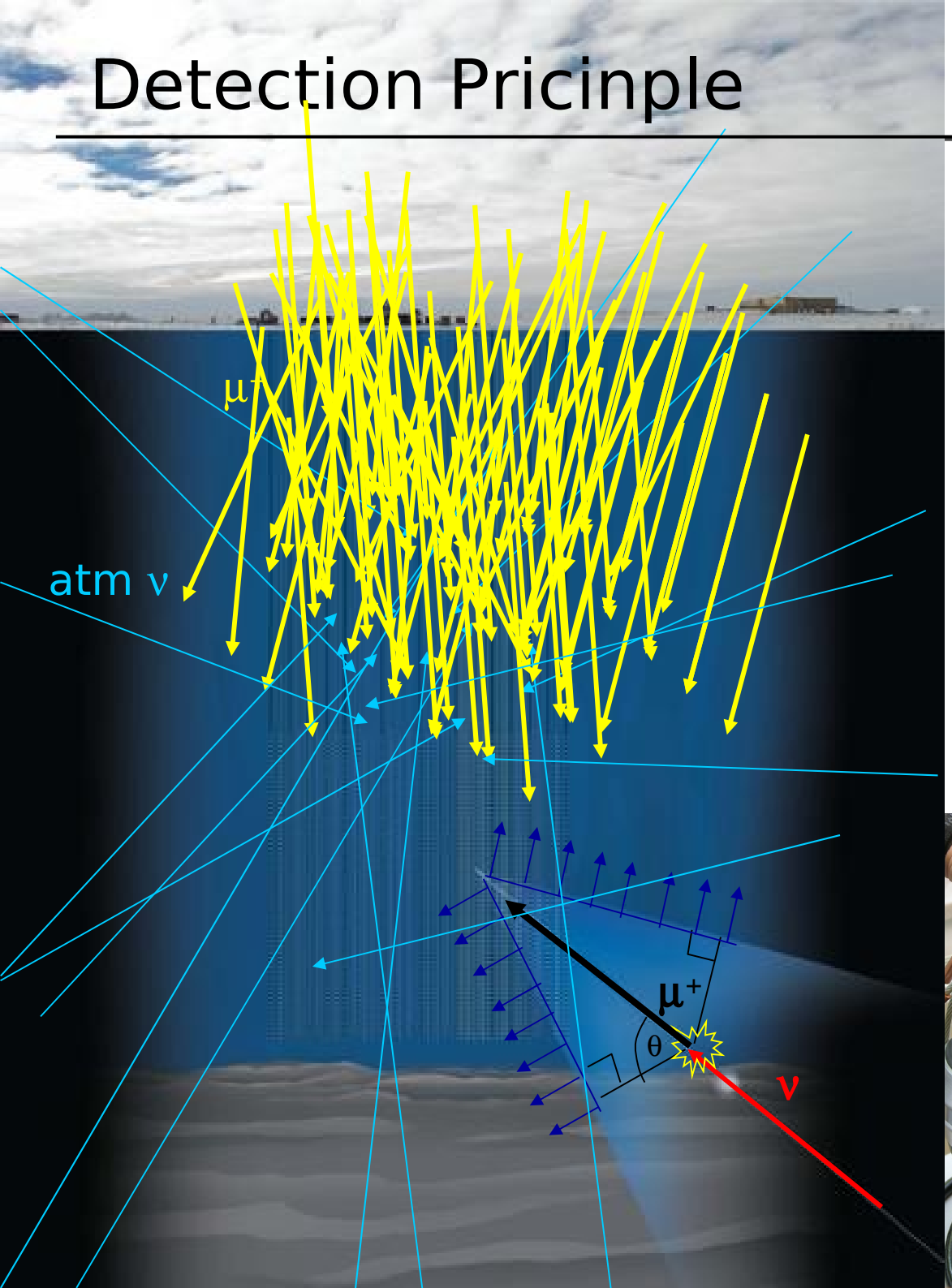


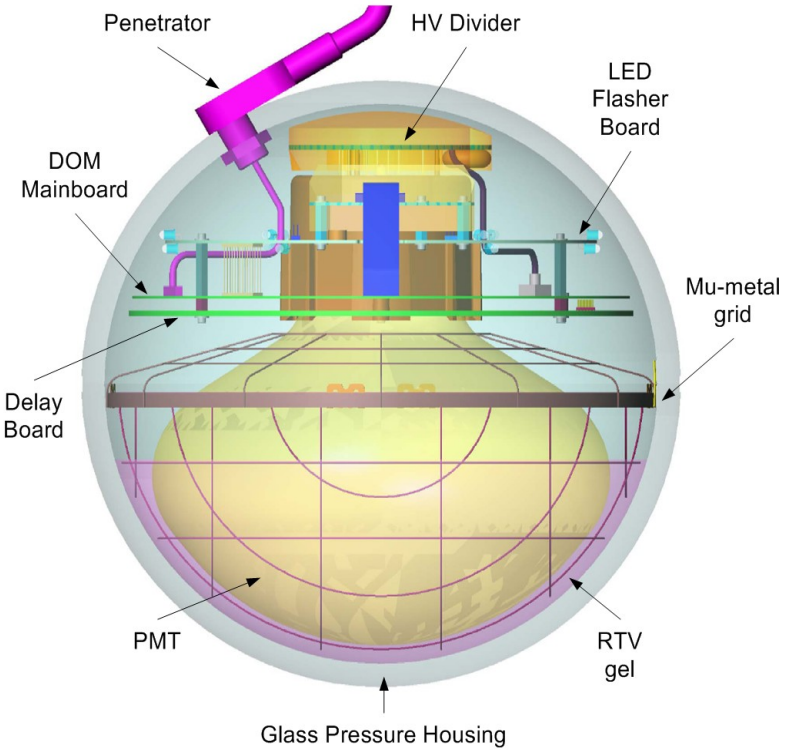


The IceCube Telescope



Detection Pricinple





IceCube Collaboration

USA:
Bartol Research Institute, Delaware
Pennsylvania State University
UC Berkeley
UC Irvine
Clark-Atlanta University
University of Maryland
University of Wisconsin-Madison
University of Wisconsin-River Falls
Lawrence Berkeley National Lab.
University of Kansas
Southern University and A&M College, Baton Rouge
University of Alaska, Anchorage

Sweden:
Uppsala Universitet
Stockholm Universitet

UK:
Oxford University

Netherlands:
Utrecht University

Belgium:
Université Libre de Bruxelles
Vrije Universiteit Brussel
Universiteit Gent
Université de Mons-Hainaut

Germany:
Universität Mainz
DESY-Zeuthen
Universität Dortmund
Universität Wuppertal
Humboldt Universität
MPI Heidelberg
RWTH Aachen

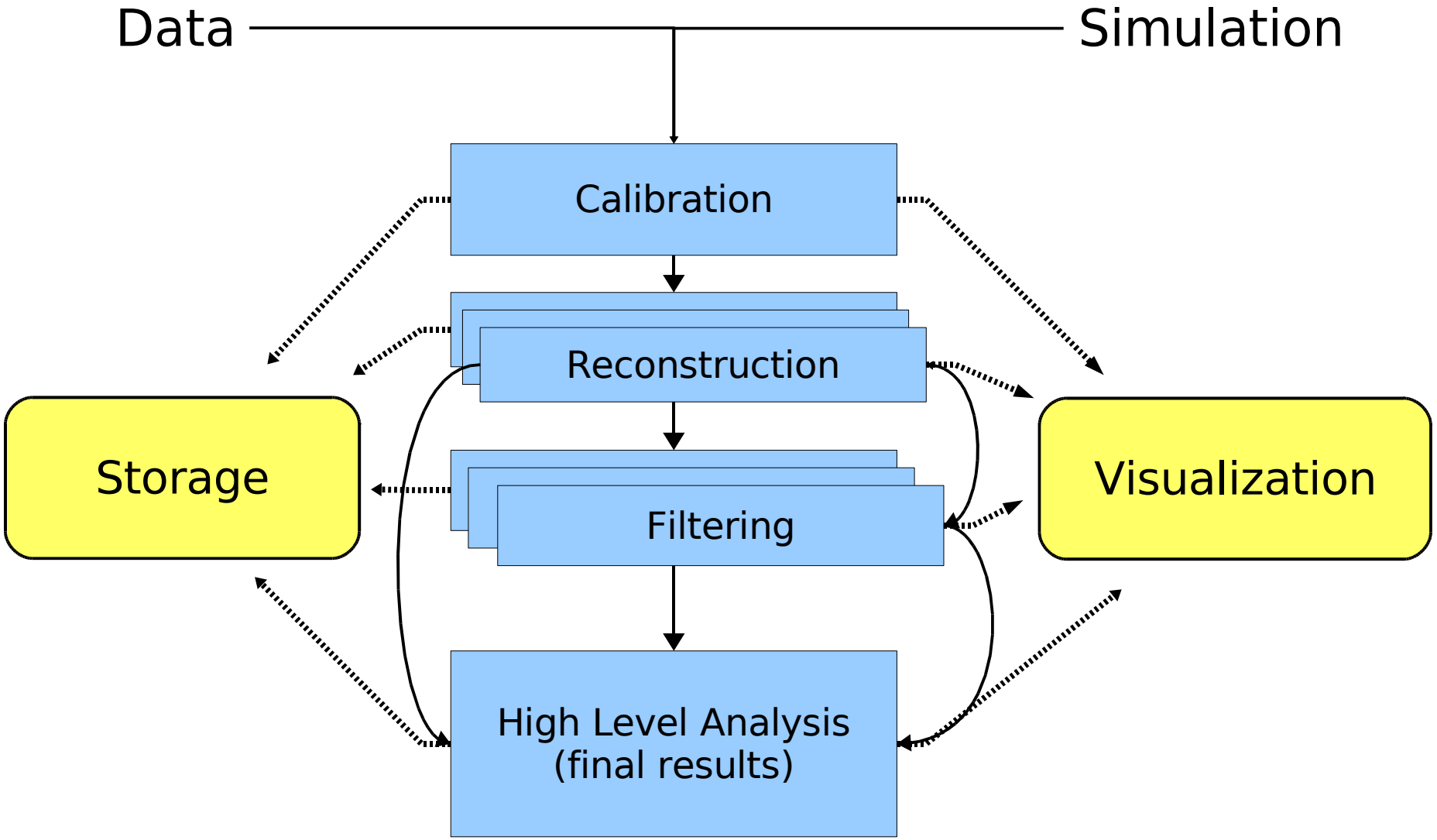
Japan:
Chiba university

New Zealand:
University of Canterbury

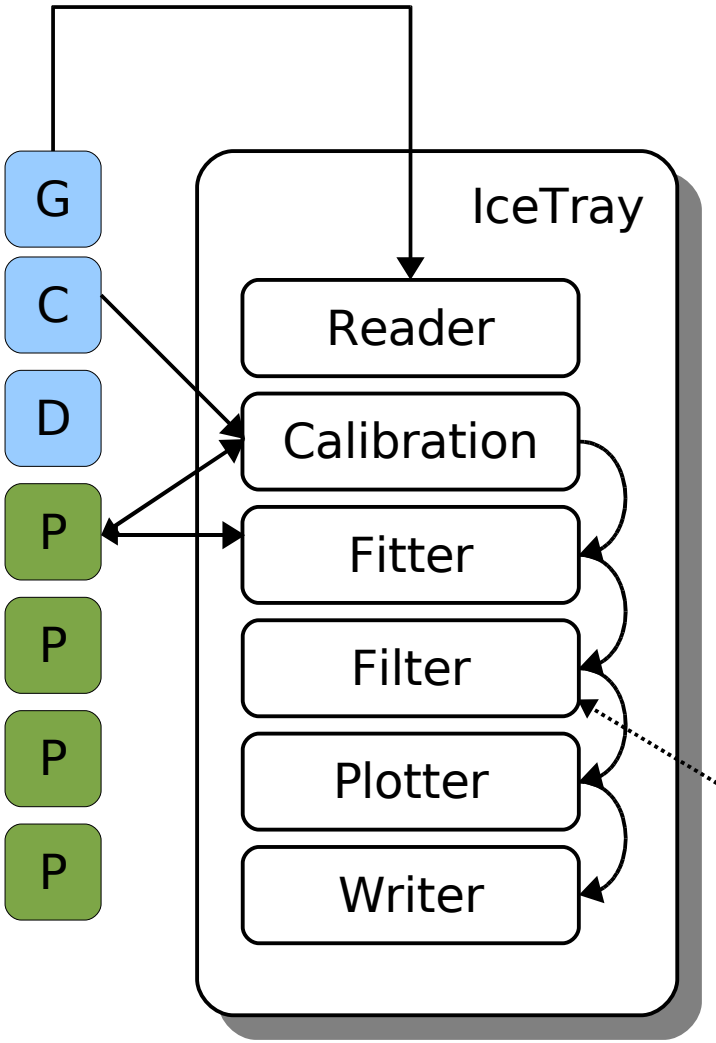
30 institutions, ~250 members
http://icecube.wisc.edu

Data Flow

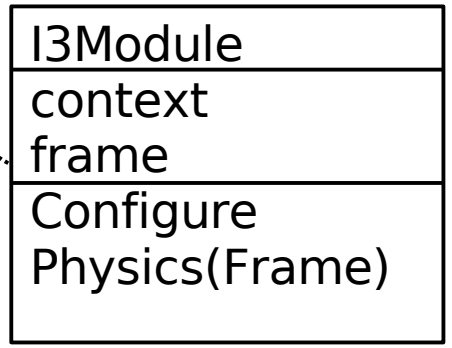
(>400 GB - 1TB/day)



IceTray



- IceTray serves data stream
  - Frame / Pop / Stop / Push
- Modules for different tasks
  - access to frame content
- Several people write modules
  - clear interface (construct, configure, physics)



Implemented in C++, lots of STL and boost stuff

Where do we use Python?

- Chain C++ module together (steering file)
  - no need to write C++ for non-module developers
  - extremely flexible, lots of OS functions, rapid development, easy to learn
  - need to interface IceTray to Python
- Analyzer's daily work – histograms, plots, calculations, selections
  - need to extract data from binary files or expose IO
- Toy simulations for fast prototyping
  - test bed for new ideas, easy to debug
  - interactive: plot wherever you want to see your data
  - need to write Modules in Python

IceTray – Steering Files

```
#!/usr/bin/env python
import I3Tray
import os
import sys
import os.path

I3Tray.load("libdataclasses")
I3Tray.load("libdataio")
I3Tray.load("libsim-services")
I3Tray.load("libsimple-generator")

def main():

    nevents = sys.argv[1]
    run = sys.argv[2]

    tray = I3Tray.I3Tray()

    tray.AddService("I3SPRNGRandomServiceFactory","random")(
        ("Seed",run))

    tray.AddModule("I3Muxer","muxer")

    tray.AddModule("I3SimpleGenerator","generator")

    tray.AddModule("I3Writer","writer")(
        ("filename", "test_run_%i" % run),
        ("streams", ["Physics", "Calibration"])

    tray.AddModule("TrashCan", "the can")
    tray.Execute(nevents)
    tray.Finish()

if __main__ == "__main__":
    main()
```

- Defines processing chain
- Parameter configuration
- Implementation:
  - wrapping I3Tray class
  - using boost.python

Wrapping with boost.python

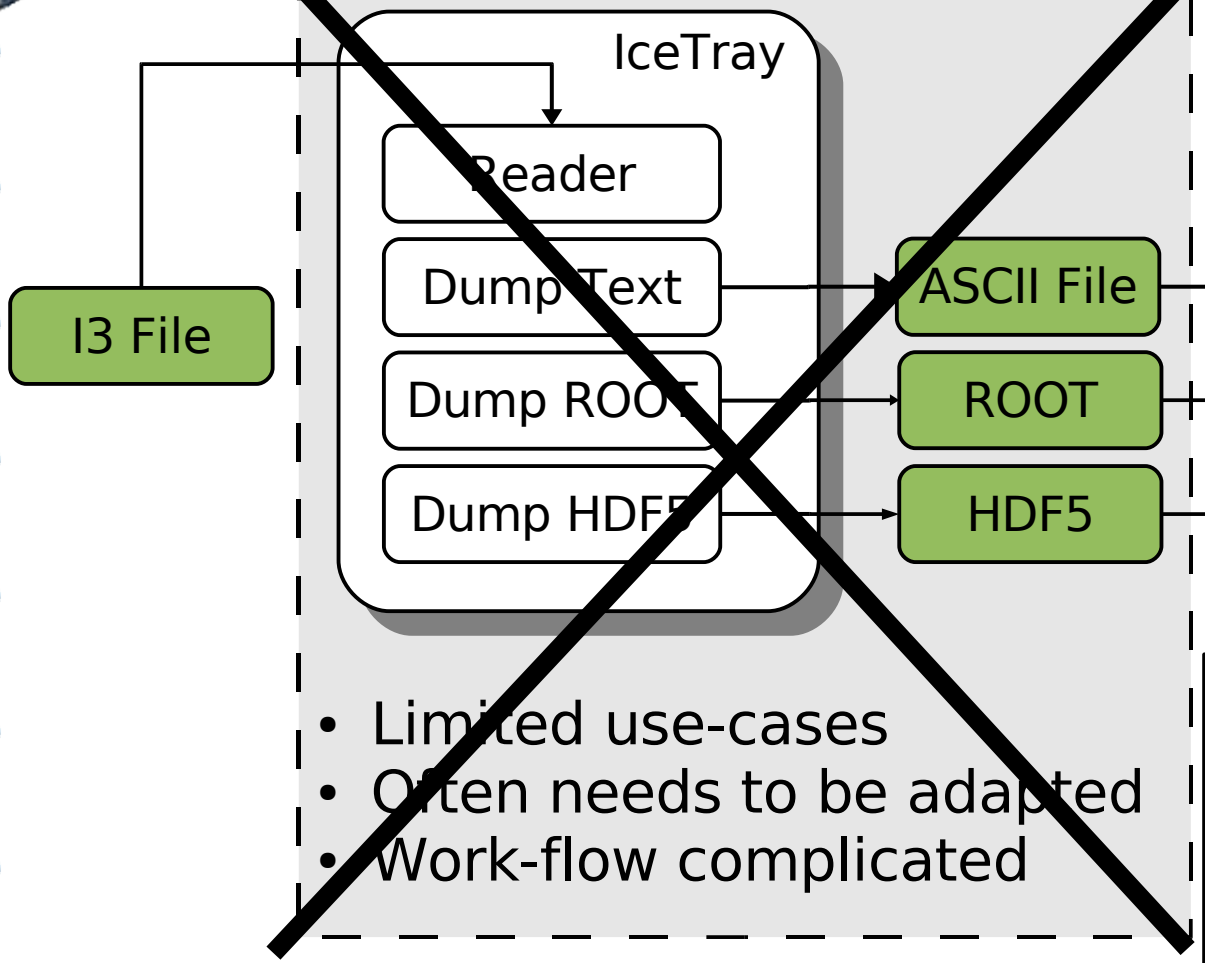
```cpp
#include <boost/python.hpp>

using namespace boost::python;
BOOST_PYTHON_MODULE(libithon)
{
  void (I3Tray::*Execute_0)(void)                    = &I3Tray::Execute;
```

```python
import libithon

class I3Tray:
    def __init__(self):
        self.last_added = None
        self.the_tray = libithon.I3Tray()

    def AddModule(self, type, name):
        self.the_tray.AddModule(type, name)
        self.last_added = name
        return self

    def AddService(self, type, name):
        self.the_tray.AddService(type, name)
        self.last_added = name
        return self

    def SetParameter(self, module, param, value):
        self.the_tray.SetParameter(module, param, value)
        return self

    def __call__(self, *args):
        for pair in args:
            print self.last_added +': ', pair[0], '=', pair[1]
            self.the_tray.SetParameter(self.last_added, pair[0], pair[1])
        return self

    def Execute(*args):
        if len(args) == 2:
            args[0].the_tray.Execute(args[1])
        else:
            args[0].the_tray.Execute()

    def Finish(self):
        self.the_tray.Finish()
```

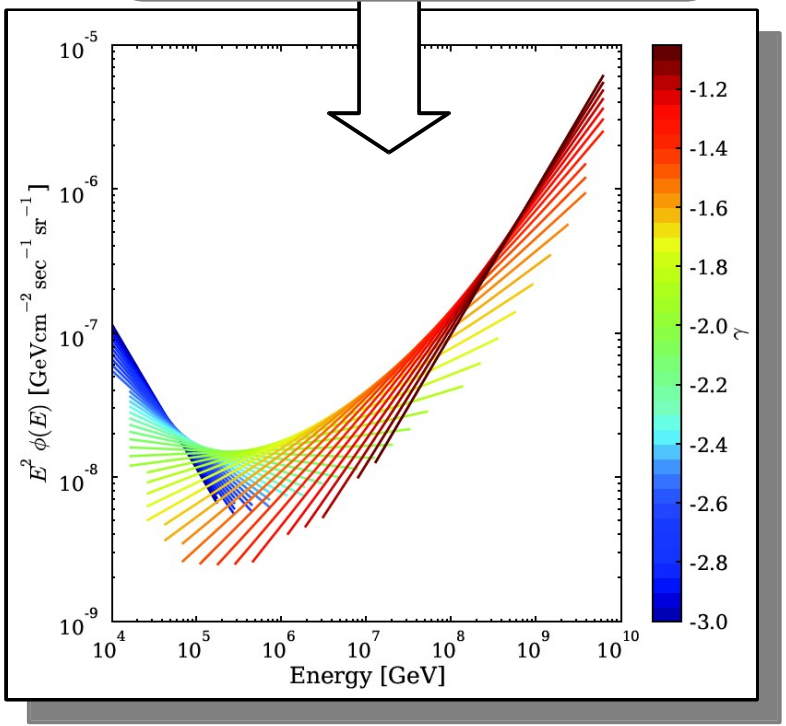
- Simple interfaces are immediately wr
- POD are mapped di Python types
- Standard container indexing suite
  - eg. vector to list, map to dict

IceCube

Analyzer's work flow



Would be nice to get rid of this!



Analyzer's daily work

- Standard tool in HEP is ROOT (root.cern.ch)
  - IceTray module to write ROOT files (almost generic)
  - PyROOT exposes ROOT classes to Python
  - cumbersome library
- Write HDF5 files
  - IceTray module to write HDF5 files (table interface)
  - dedicated converter for every class
- Python reader for IceTray binary format
  - wrap IO classes and basic dataclasses
  - flexible, full access, easy to use
    ... a little bit slow, though

Examples

- Access to data using ROOT
  - ROOT file writer module (C++)
  - PyROOT to read ROOT file
- Access to data using HDF5 files
  - HDF5 table writer module (C++)
  - pytables to read hdf5 file
- Direct access to data from Python
  - using the wrapped IO methods

Using ROOT Files

```
void I3RootWriterModule::Configure(){

  GetParameter("RootFileName",rootfilename_);

  rootfile_ = new TFile(rootfilename_.c_str(),"RECREATE");
  tree_ = new TTree("aTree","aTree");

  tree_->Branch("nModules",&nModules_,"nModules/I");
  tree_->Branch("nStrings",&nStrings_,"nStrings/I");
  tree_->Branch("meanPulses",&meanPulses_,"meanPulses/D");

}

void I3RootWriterModule::Physics(I3FramePtr frame){

  // get frame object
  I3RecoPulseSeriesMapPtr hitsmap =
     frame->Get<I3RecoPulseSeriesMapPtr>(pulseSeriesName);

  nModules_ = hitmap->size();


  // set to count number of strings
  set<int> nStrings;
  // mean number of pulses per Modu
  int nPulses = 0;

  // loop over oms in hitmap
  typename I3Map<OMKey,vector<I3Rec
  for (hits=hitsmap->begin(); hits
    const OMKey& omkey = hits->firs
    // add hits of this om
    nPulses += hits->second.size();
    // track the strings hit
    nstrings.insert(omkey.GetString
  }

  nStrings_ = nstrings.size();
  meanPulses_ = nPulses/nModules_;

  tree_->Fill();
}
```

```
import sys
from icecube.icetray import I3Tray

def main(source, sink):

    I3Tray.load('libdataio')
    I3Tray.load('libphys-services')
    I3Tray.load('librootwriter')

    tray = I3Tray.I3Tray()

    tray.AddModule('I3Reader', 'reader')(
        ('filename', source))

    tray.AddModule('I3RootWriterModule', 'writer')(
        ('RootFileName', 'out.root'))

    tray.AddModule('TrashCan', 'trash')
```

```
import ROOT

def main(path):

    f = ROOT.TFile(path)
    tree = f.Get("aTree")
    nentries = tree.GetEntries()

    nStringsLeaf = tree.FindLeaf("nStrings")
    meanPulsesLeaf = tree.FindLeaf("meanPulses")
    nModulesLeaf = tree.FindLeaf("nModules")

    for i in xrange(nentries):
        tree.GetEntry(i)
        print 'Event %i Modules: %i <Pulses>: %.3f Strings: %i' % \
              i+1, nModulesLeaf.GetValue(),
               meanPulsesLeaf.GetValue(), nStringsLeaf.GetValue())

if __name__ == '__main__':
    main(sys.argv[1])
```

IceCube

Using HDF5 Tables

```
void DummyModule::Physics(I3FramePtr frame) {
  // struct which is passed to HDF table writer method
  buffer = Buffer();

  // get frame object
  I3RecoPulseSeriesMapPtr hitsmap =
     frame->Get<I3RecoPulseSeriesMapPtr>(pul

  // number of modules
  buffer.nModules = hitsmap->size();
  // set to count number of strings
  set<int> nStrings;
  // mean number of pulses per Module
  int meanP = 0;

  // loop over oms in hitmap
  typename I3Map<OMKey,vector<I3RecoPulse> >
  for (hits=hitsmap->begin(); hits!=hitsmap-
    const OMKey& omkey = hits->first;
    // add hits of this om
    meanP += hits->second.size();
    // track the strings hit
    nstrings.insert(omkey.GetString()); // s
  }

  buffer.nstrings = nstrings.size();
  buffer.meanP = meanP/nModules;

  // now fill struct and store results in ho
  H5TBappend_records

  PushFrame(frame,"
}
```

```
import sys
from icecube.icetray import I3Tray

def main(source, sink):

    I3Tray.load('libdataio')
    I3Tray.load('libphys-services')
    I3Tray.load('libhdf-writer')

    tray = I3Tray.I3Tray()

    tray.AddService('I3HDFWriterServiceFactory', 'hdfservice') (
        ('filename', sink))

    tray.AddModule('I3Reader', 'reader')(
        ('filename', source))

    tray.AddModule('I3HDFWriter<EventInfoToTable<I3RecoPulse > >', 'info')(
        ('tablename', 'info'),
        ('key', 'RecoPulses'))

    tray.AddModule('TrashCan', 'trash')
    tray.Execute()
    tray.Finish()

if __name__ == '__main__':
```

```
import sys
import tables

def main(path):
    f = tables.openFile(path)
    for row in f.root.eventinfo:
        print 'Event %i Modules: %i <Pulses>: %.3f Strings: %i' % \
              (row['id'], row['nmodules'],
               row['mean_pulses'], row['nstrings'])

if __name__ == '__main__':
    main(sys.argv[1])
```

The Pythonic Way

```python
import sys
import numpy
import icecube.icetray as icetray
import icecube.dataclasses as dataclasses
import icecube.dataio as dataio

def main(path):
    # open file
    fh = dataio.I3File(path)

    # walk through frames
    while fh.more():
        frame = fh.pop_physics()

        # get reconstructed electronic signals
        hitmap = frame.Get('RecoPulses')

        # calculate number of contributing modules
        nModules = len(hitmap)

        # calculate mean number of pulses observed per module
        meanP =  numpy.mean([len(entry.data()) for entry in hitmap])

        # number of strings
        nStrings = len(set(entry.key().GetString() for entry in hitmap))

        # get Event ID
        eventID = frame.Get('I3EventHeader').EventID

        print 'Event %i Modules: %i <Pulses>: %.3f Strings: %i' % \
              (eventID, nModules, meanP, nStrings)

if __name__ == '__main__':
    main(sys.argv[1])
```

- Single script
- No compilation
- Developed interactively using introspection – no need to check docs

Rapid Prototyping

- Simple line-fit
  - $\vec{v} = COV(\vec{r},t)/VAR(t)$
- Compare to more than 200 rows C++ code
- Great for developing new algorithms
- Great to check details in data

```
import numpy
import icecube....

def linefit(geometry, hitmap):
    # lists of module coordinates and hit times
    coords  = []
    times = []
    for entry in hitmap:
        pos = geometry[entry.key()].position
        for pulse in entry.data():
            coords.append((pos.X, pos.Y, pos.Z))
            times.append(pulse.Time)

    # build numpy arrays
    coords = numpy.asarray(coords)
    times = numpy.asarray(times)

    # correlation between times and coordinates
    times -= times.mean()
    coords -= coords.mean(axis=0)
    speed = (coords * times[:,numpy.newaxis]).sum(axis=0) / \
            (times.var() * (len(times)-1))
    return speed

def main(path):
    fh = dataio.I3File(path)
    geometry = None
    while fh.more():
        frame = fh.pop_frame()
        if 'I3Geometry' in frame:
            geometry = frame.Get('I3Geometry').omgeo
        if 'RecoPulses' in frame:
            hitmap = frame.Get('RecoPulses')
            if geometry is not None:
                reco = linefit(geometry, hitmap)
                print numpy.linalg.norm(reco)
```

Modules written in Python

- Modules can be Python functions
  - Nice to write simple filters

```
#!/usr/bin/env python
from icecube.icetray import I3Tray

def filt(frame):
    return len(frame.Get("RecoPulses")) > 20

def main():
    tray = I3Tray.I3Tray()
    tray.AddModule("I3Reader","reader")(
        ("filename", "data.i3.gz"))

    tray.AddModule(filt, "filter")

    tray.AddModule("I3Writer","writer")(
        ("filename", "filtered" % run))

    tray.AddModule("TrashCan", "the can")
    tray.Execute()
    tray.Finish()
```

- Single script
- No compilation

- Modules in Python
  - Derive from base module class
  - Add python class as module
  - C++ instantiates Python object and calls interface

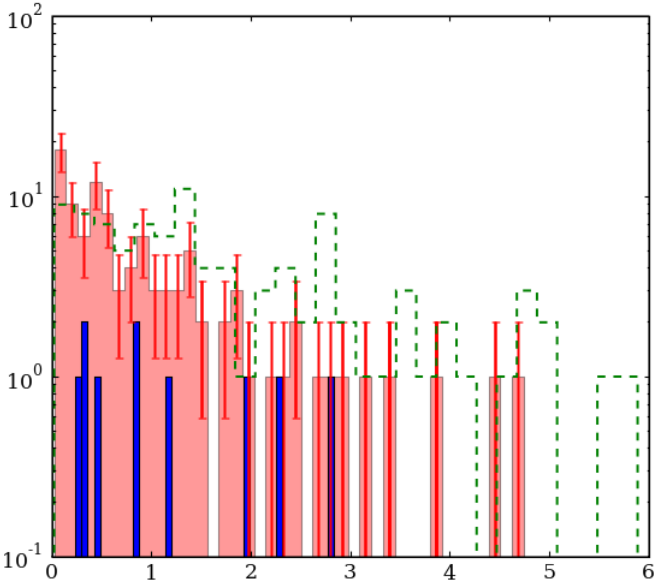
Analysis Tools

- Numpy – great data manipulation
  - Loop-less computations
  - Advanced indexing (boolean arrays) for cutting
- Scipy – fitting, statistical tests, machine learning
- Matplotlib – visualize data
  - Easy to learn
  - Nice for simple graphs, easy customization
  - Histogramming needs tweaks (next slide)
  - No GUI to manipulate plots

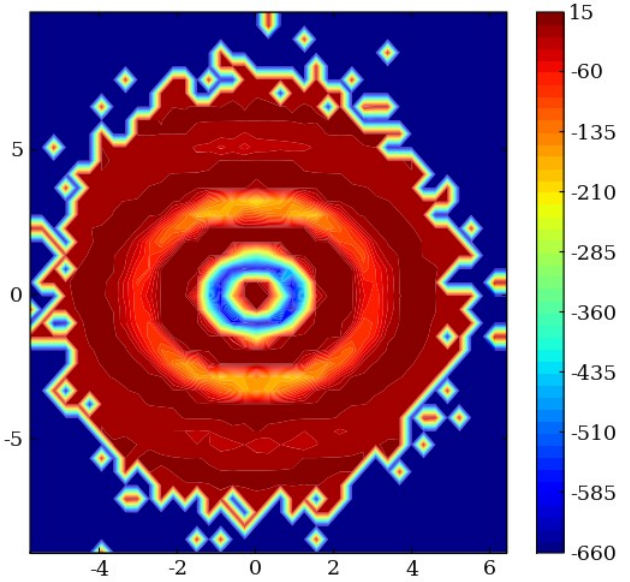
IceCube

Using Matplotlib for Histograms

- Modified histogram function
  - Based on numpy digitize & bincount
  - Weights, stat. error calculation, errorbars,logscale, filled patches, lines, bars
- 2D-histograms
  - Digitize & bincount
  - Contour or scatter plot
  - Weights, Logscale




```
foo = numpy.random.multivariate_normal([0,0],[[2,0],[0,4]],
                                       size=100000)

bar = numpy.cos(3 * n.sqrt(foo[:,0]**2 + foo[:,1]**2))
utils.hist2d(foo[:,0], foo[:,1], 40, weights=bar, style='contf')
```



Summary

- IceCube uses a costumized software framework
- Controlling done using Python scripts
  - simple interface wrapped with boost.python
- Data extraction not straightforward
  - wrapping all data IO and base classes provides easy access and interfaces for prototyping and development
- Python allows extremely fast development cycles
- matplotlib, numpy, scipy provide a great environment to develop algorithms
- Tedious implementations using C++ only for time demanding tasks