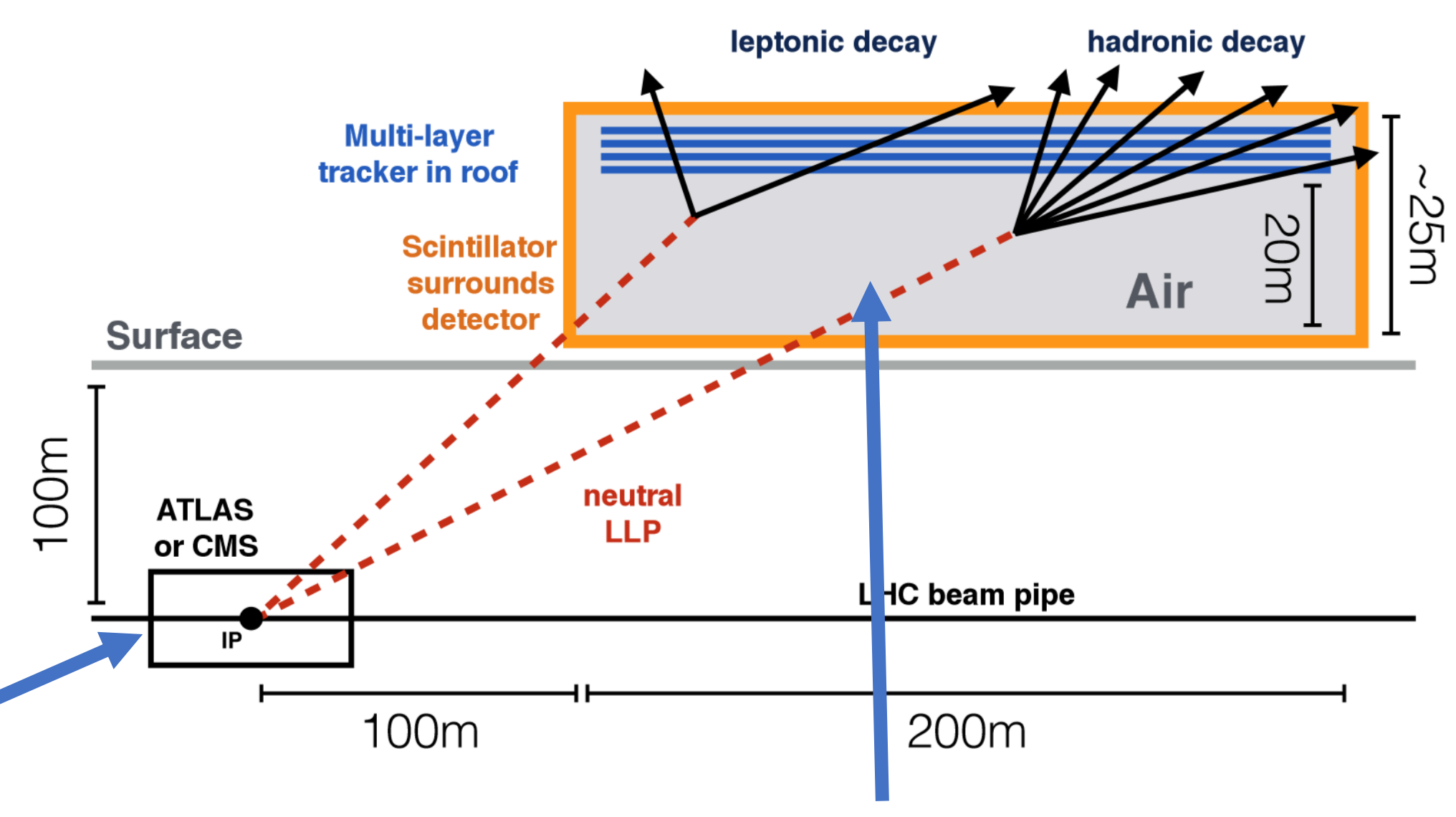
Aligning the MATHUSLA Test Stand Detector: Using TensorFlow

#470-39 Gordon Watts

What is MATHUSLA?

Long-Lived Particles (LLPs) provide a gateway to a Hidden or Dark Sector in many models. This Hidden Sector could hold answers to many of today's most pressing physics questions, in particular **Dark Matter** and the **Hierarchy Problem**.



An Ultra-Long-Lived-Particle (ULLP) produced at the LHC interaction point…

… Decays on the surface in MATHUSLA, a 5 or 6 layer tracking chamber with veto scintillator around the edges

ATLAS and CMS and LHCb are sensitive to short lifetime long lived particles – we need the long baseline and low background environment of a surface detector to be sensitive to the ULLPs.

Main backgrounds are Cosmic Rays and upward going Muons from the LHC interaction point (produced via W/Z, $t\bar{t}$, $b\bar{b}$, etc.).

The MATHUSLA Test Stand

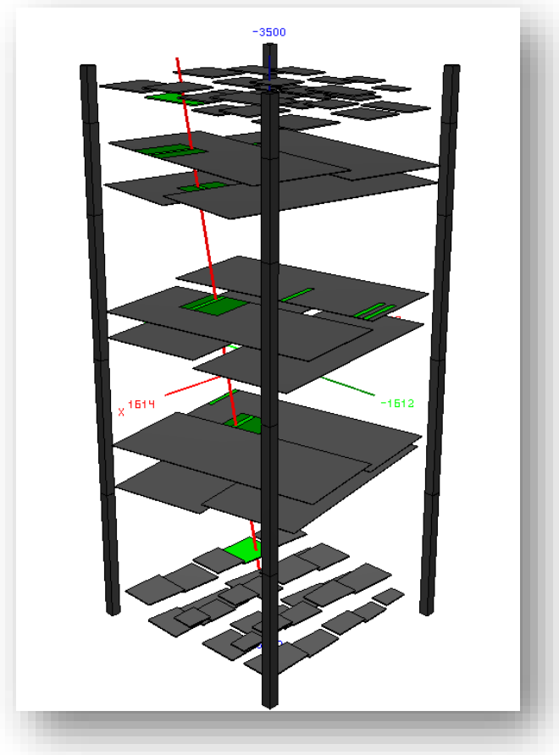
A 6 × 3 m test stand was built and run at the ATLAS IP from 2017-2018. It had 3 double layers of Resistive Plate Chambers (RPC's) for track reconstruction from the ARGO experiment, and a top and bottom layer made up of the DZERO experiment's forward muon scintillator paddles.

Its goals:

- Can we reconstruct tracks from Cosmic Rays
- Can we reconstruct upward going tracks?
- Can we tell the difference between them?

Status:

- We have millions of tracks
- GEANT4 based simulation of the test stand
- First pass at calibrations for timing finished
- Now working on acceptance calculations to determine expected upward going rate
- Geometry measured by **hand.**



Aligning the MATHUSLA Test Stand using TensorFlow

Detector Model

12 RPC's

- Each is allow to shift in their plain (x, y), and each is allowed to rotate in the xy plane.
- RPC #0 is fixed in place

Track Fits

2000 Tracks

- Hit finding is done once
- Track is re-fit each time the RPC's are moved
- Straight line fits in $x = x_0 + m_x z$ and $y = y_0 + m_y z$.

χ^2

Minimize the χ^2

- The sum of each track's χ^2
- No terms for the ad-hoc measurements made on the detector.

Tensor Flow Alignment Code

This code calculates the chi2 contribution from each hit. It does this for all 10,000 tracks (each with 6 hits) at once.

(x, y, z) locations and rotations for all 960 strips. During minimization these are allowed to float to minimize the χ^2.

Matrix of 960 hits by n_{tracks}. In each row only 6 entries are one as each track can have only 6 hits.

```
def chi2_contrib_per_hit (strip_location, strip_rz, strip_widths, x0, m, hits_used):
    strip_cos_y = np.sin(strip_rz) if type(strip_rz) is np.ndarray else tf.sin(strip_rz)
    strip_cos_x = np.cos(strip_rz) if type(strip_rz) is np.ndarray else tf.cos(strip_rz)

    Lx = strip_cos_x
    Ly = strip_cos_y
    Wx = -strip_cos_y
    Wy = strip_cos_x

    del_L = strip_widths[0] / math.sqrt(12)
    del_W = strip_widths[1] / math.sqrt(12)
    strip_ratio_Lx = Lx / del_L
    strip_ratio_Ly = Ly / del_L
    strip_ratio_Wx = Wx / del_W
    strip_ratio_Wy = Wy / del_W

    ratio_Lx = expand_to_tracks(strip_ratio_Lx, len(hits_used))
    ratio_Ly = expand_to_tracks(strip_ratio_Ly, len(hits_used))
    ratio_Wx = expand_to_tracks(strip_ratio_Wx, len(hits_used))
    ratio_Wy = expand_to_tracks(strip_ratio_Wy, len(hits_used))

    delta_x = calc_delta (strip_location[:,0], strip_location[:,2], x0[0], m[0], hits_used)
    delta_y = calc_delta (strip_location[:,1], strip_location[:,2], x0[1], m[1], hits_used)

    length_error = delta_x * ratio_Lx + delta_y * ratio_Ly
    width_error = delta_x * ratio_Wx + delta_y * ratio_Wy

    return length_error**2 + width_error**2
```

Passing in numpy arrays makes unit tests and debugging easier. This code gets around library differences.

Operator overloading means that the TensorFlow operations happen without having to write special code for the user.

Different parts of the code require the same data in different format matrices. Reshape is used throughout.

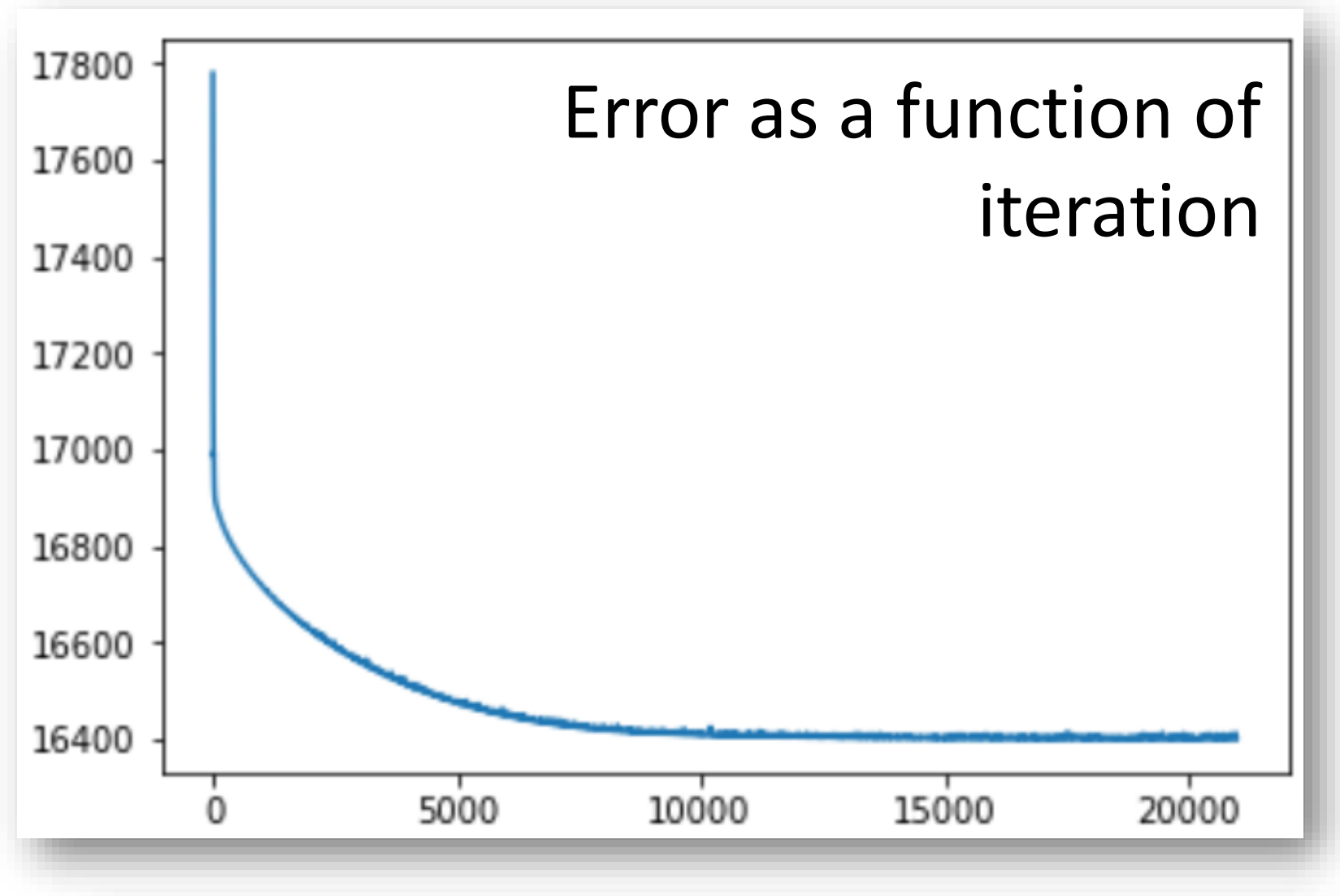
The χ^2 contribution is returned in a 960 by n_{track} matrix ready to be summed. This is a part of a TensorFlows' Directed Acyclic Graph (DAG) to compute this value.

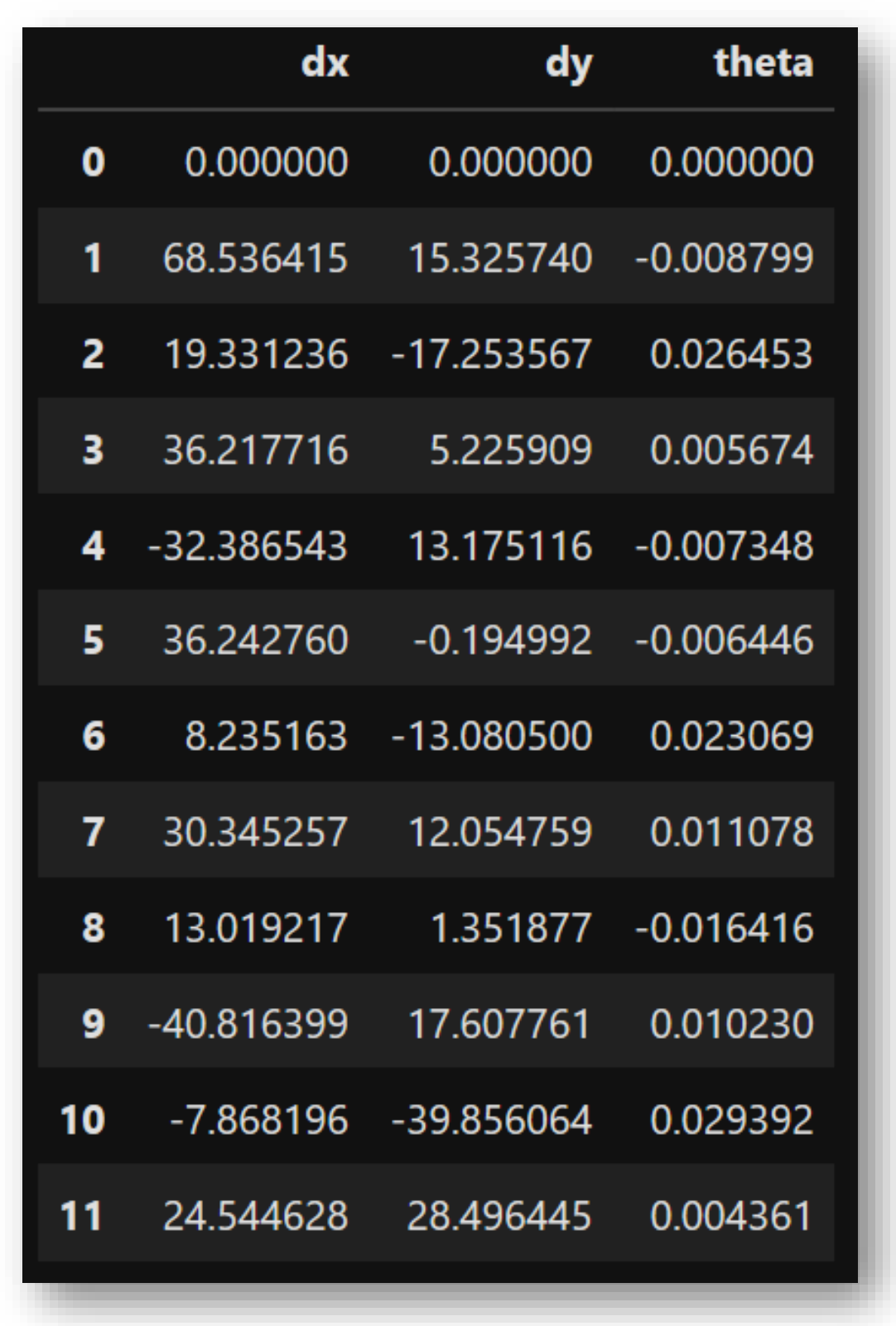
The alignment code is less than 500 lines of code (with comments). Support for geometry constants, reading in files, etc., is another 500 lines of code.

Sample Results From Running

The TensorFlow fit shown here used 1000 tracks. This is 4048 free parameters in the fit. Initial values were given for all the track fit parameters based on the ad-hoc geometry to speed convergence.

The Adam optimizer finds its minimum after 150,000 iterations. This takes about 1.5 hours on a 12 year old i7 machine. Core memory usage for 1000 tracks is about 300 MB (Windows 10 running Anaconda3 native).

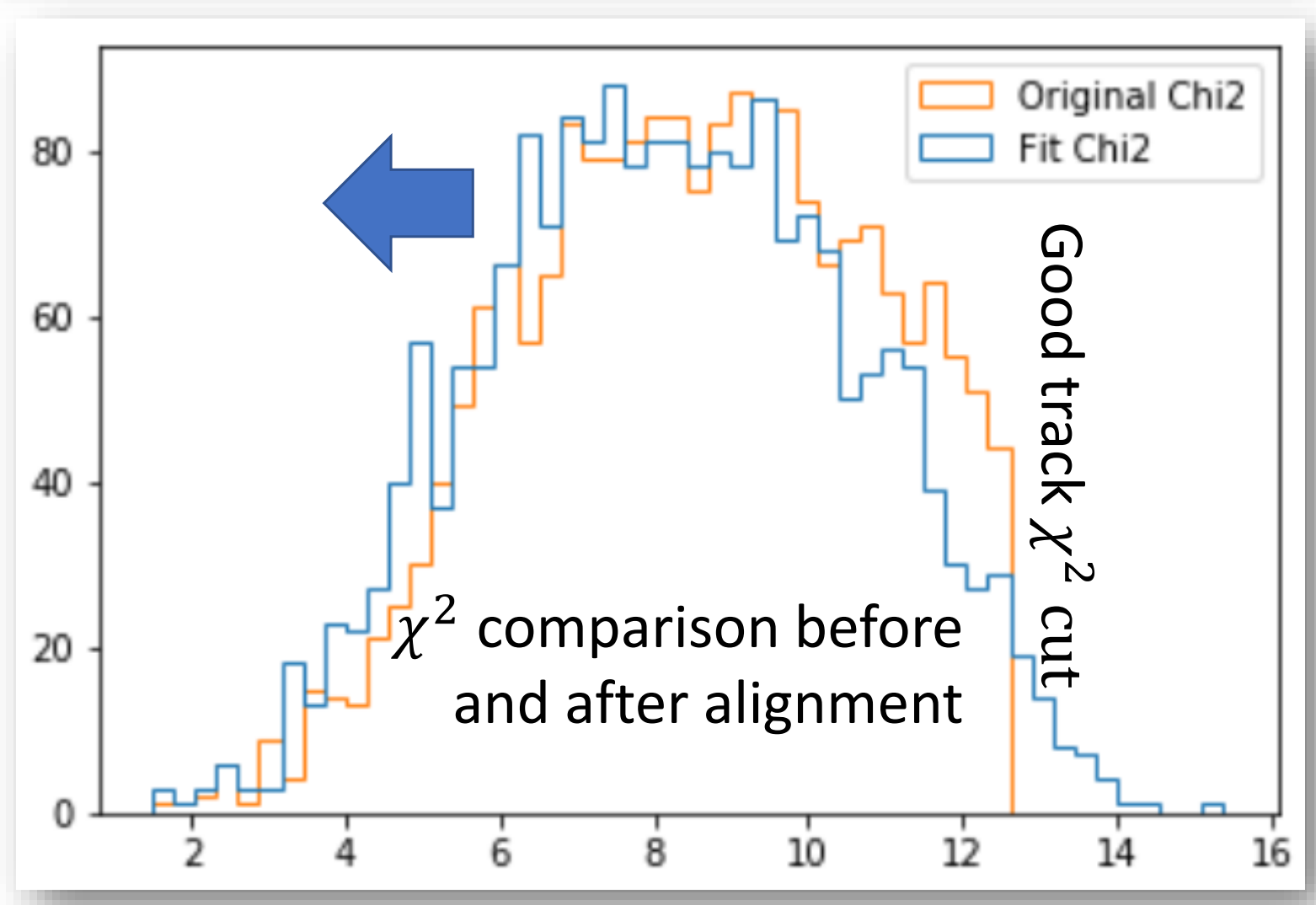




	dx	dy	theta
0	0.000000	0.000000	0.000000
1	68.536415	15.325740	-0.008799
2	19.331236	-17.253567	0.026453
3	36.217716	5.225909	0.005674
4	-32.386543	13.175116	-0.007348
5	36.242760	-0.194992	-0.006446
6	8.235163	-13.080500	0.023069
7	30.345257	12.054759	0.011078
8	13.019217	1.351877	-0.016416
9	-40.816399	17.607761	0.010230
10	-7.868196	-39.856064	0.029392
11	24.544628	28.496445	0.004361

Changes in the (x, y) and rotation of each of the 12 RPC chambers. Units are cm and radians.

Large shifts are correlated with RPC's that have few hits.



Conclusions and What Is Next

In general this is a much nicer way to write out a minimization routine that using C++ or RooFit.

- Python is a very productive language for this sort of development.
- TensorFlow and numpy make it easy to write code that is more efficient than I could probably write in C++ at the same effort level.
- Visualization from matplotlib and Jupyter Lab/Notebook made debugging using visualizations simple. The Python debugger made debugging the code easy too.

There are definitely pain points:

- Difficult to think about 1000's of tracks at once, rather than a single track at a time.
  - HEP is event oriented. Computers love vectors.
- TensorFlow builds a DAG for its compute graph. Debugging a DAG is quite difficult.
- There are times it makes sense to use numpy arrays and TensorFlow tensors interchangeably, in particular for unit testing. However, the two are not interchangeable in the code.

What is Next?

- Clean up the tracks used for alignment
- Straight line fits mean we can use a simple matrix inversion to find the track intercepts and slopes. This should speed things up further.
- Use a large GPU machine (thank you UW Astronomy) to see how the fit time changes and to include more tracks in the fit.
- Compare with more standard HEP packages