SeedTracker Results and Diagnostics

May 2, 2008

Cosmin Deaconu
Stanford/SLAC

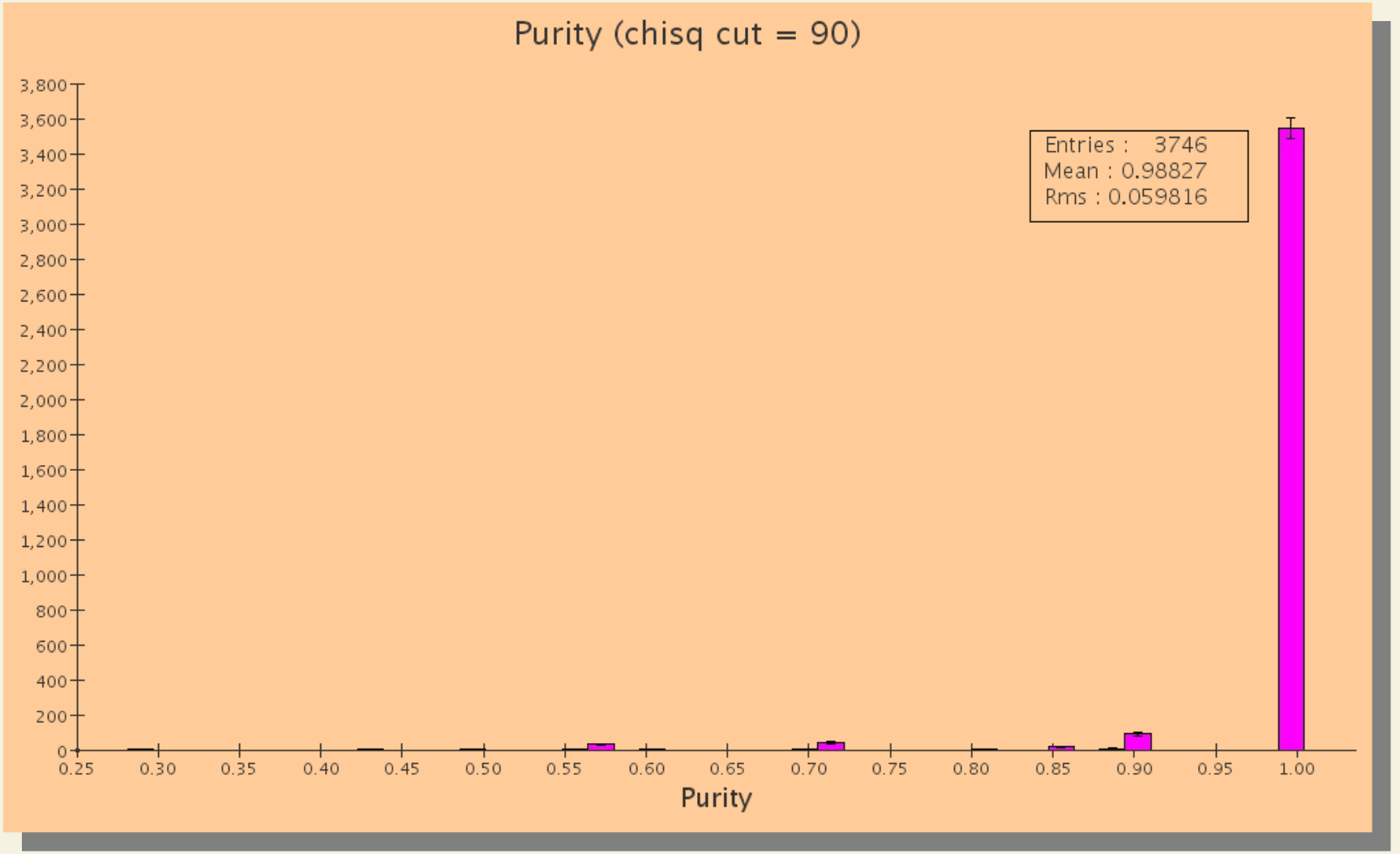
Purity:

- We define purity as the fraction of hits created by the MCParticle that has the most hits in a track.
- A purity of 1.0 implies that all hits on the track came from the same MCParticle.
- This calculation is possible because each hit knows which MCParticles contributed to it.

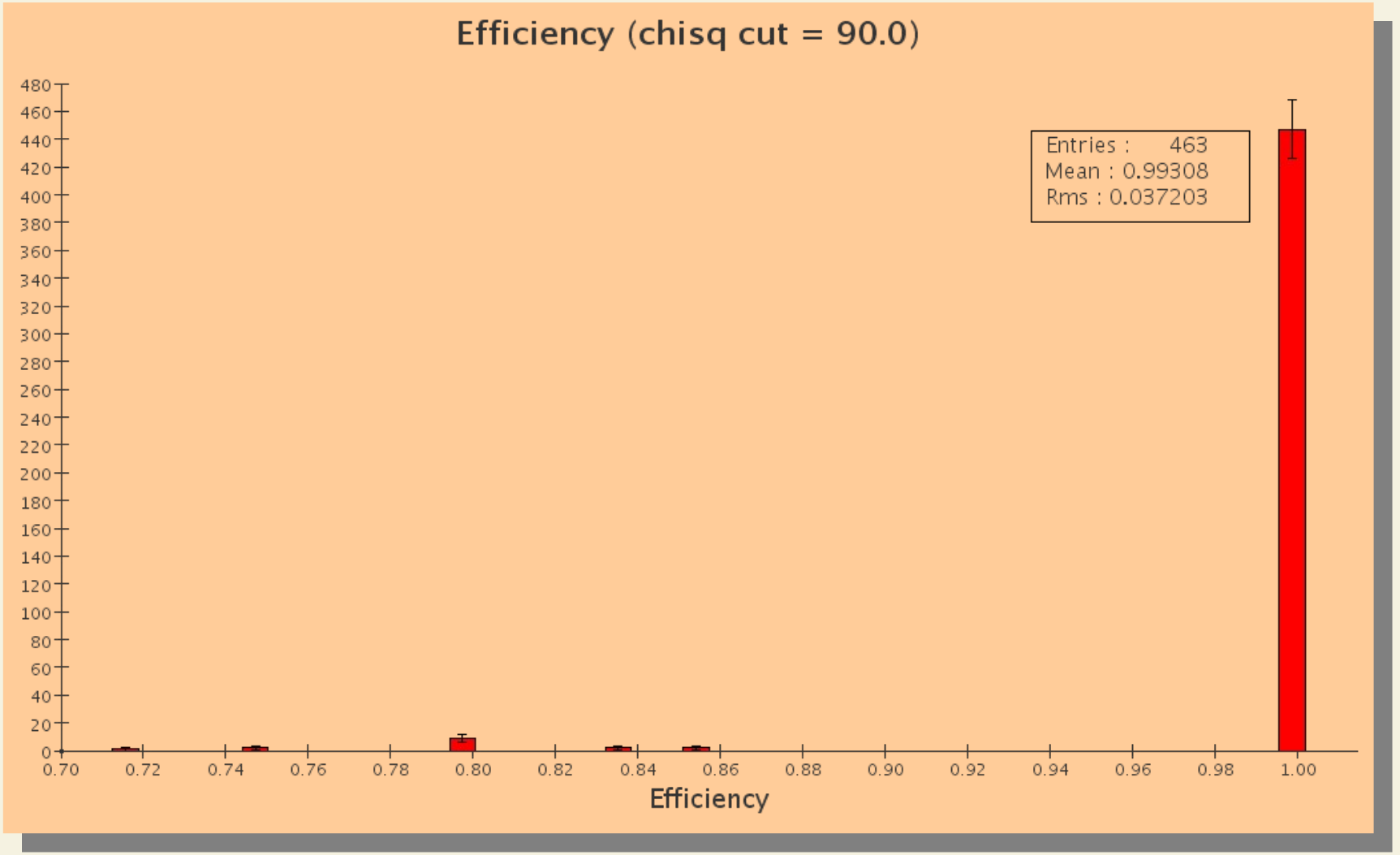
Efficiency:

- In this case, we define efficiency as the number of tracks found divided by the number of tracks findable under the given strategy.
- This definition is suitable to make sure the algorithm is working as intended.
- An MCParticle is considered a findable track if it conforms to all restrictions defined by the strategy (e.g. must have hits in the seed layer, enough hits in confirmation and extension layers, dca < max_dca, Pt > min_Pt etc.)

Purity results: ~500 events in pythiaZPoleuds-2

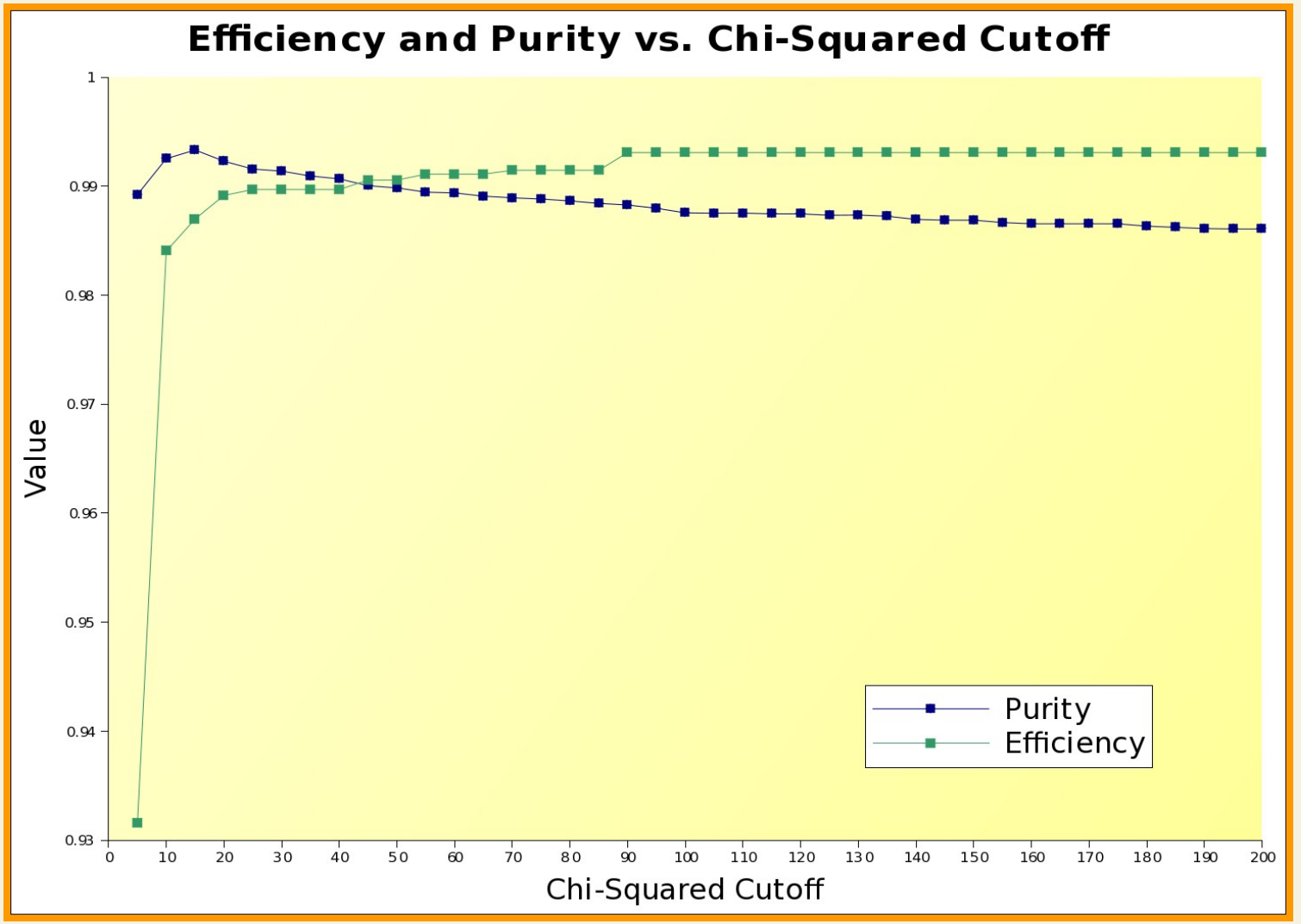


Efficiency results: ~500 events in pythiaZPoleuds-2

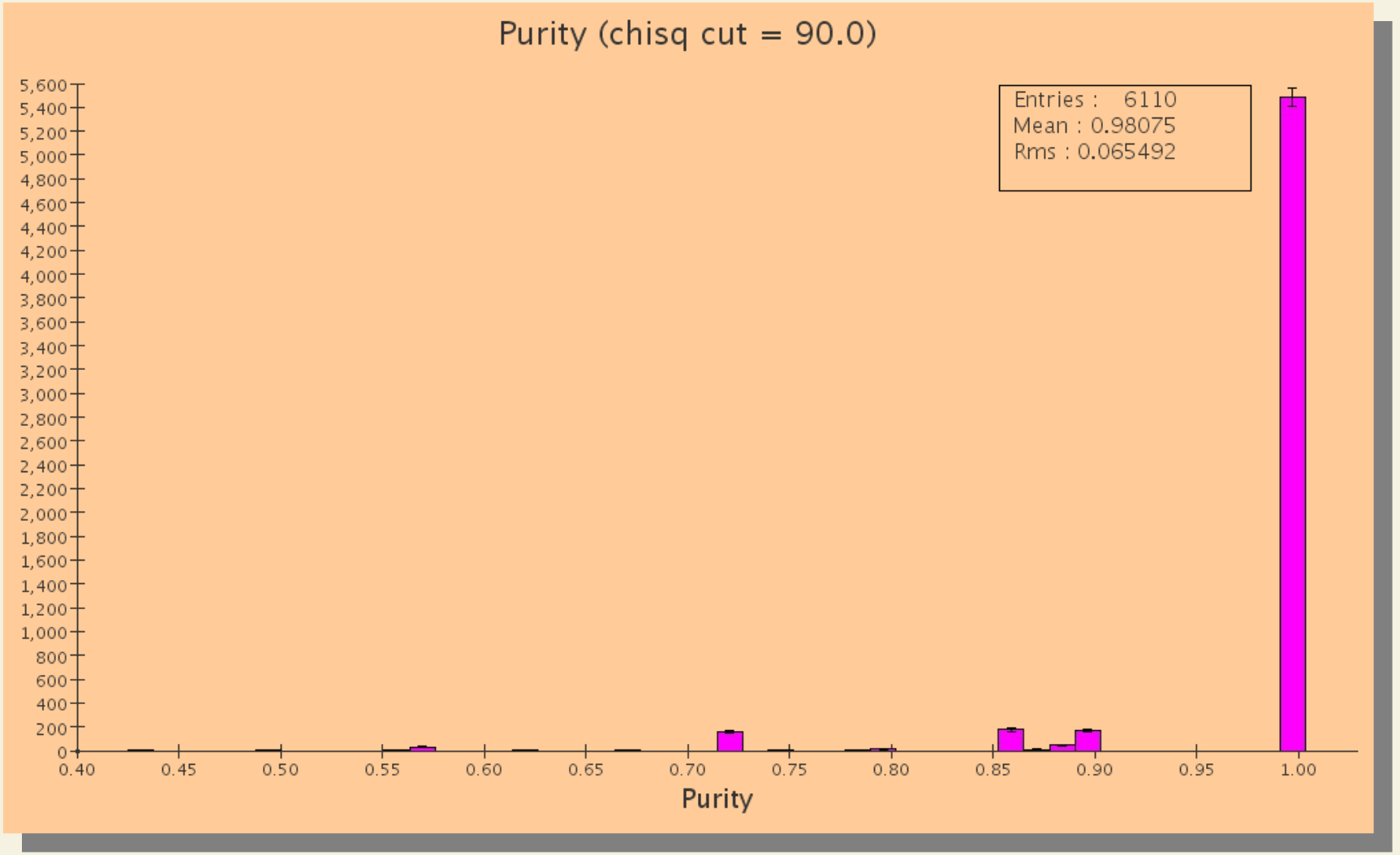


Purity and Efficiency vs. χ^2

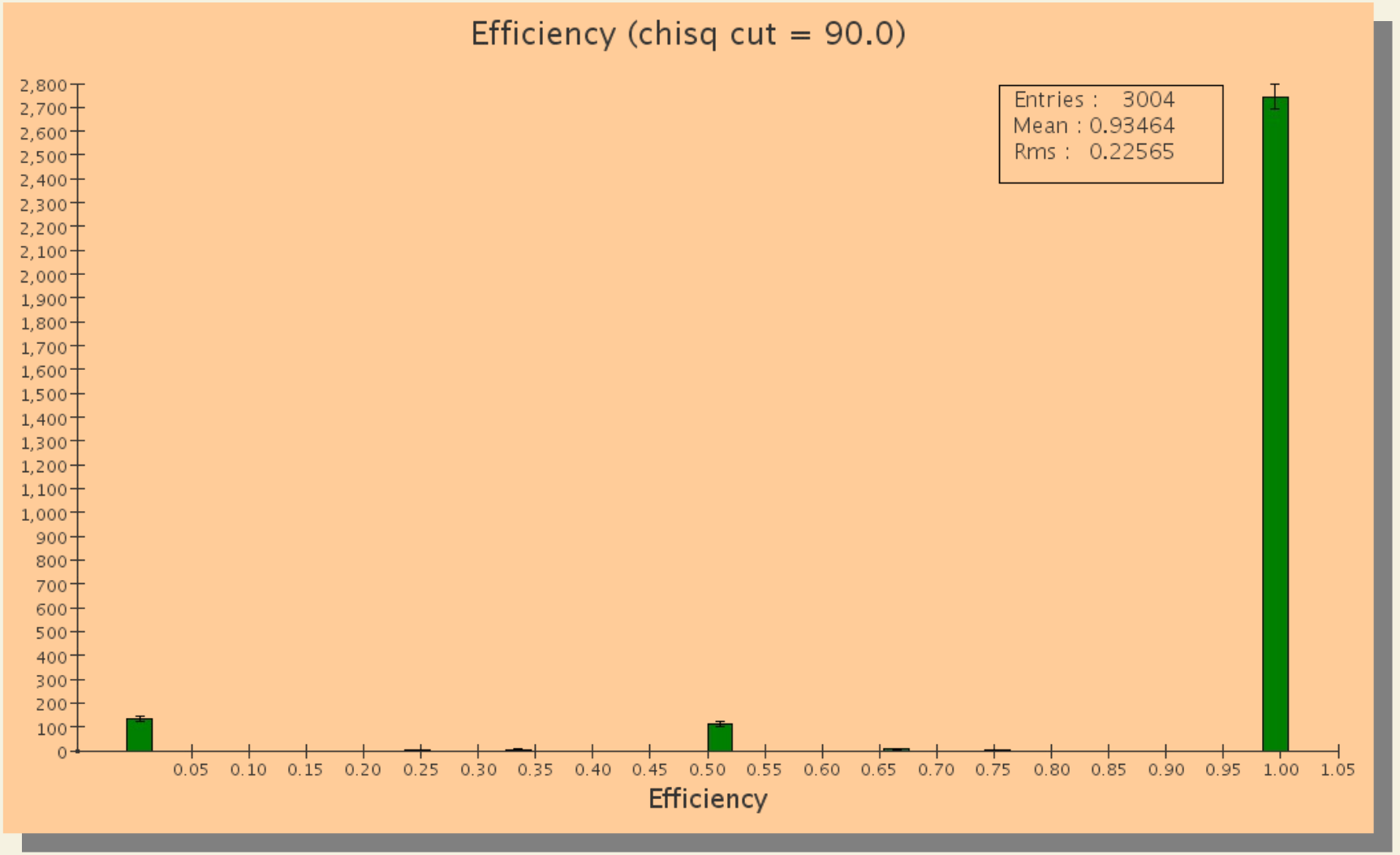
~500 events in pythiaZPoleuds-2



Purity results: ~8000 events in tau_5pi_theta20-90_10-200GeV



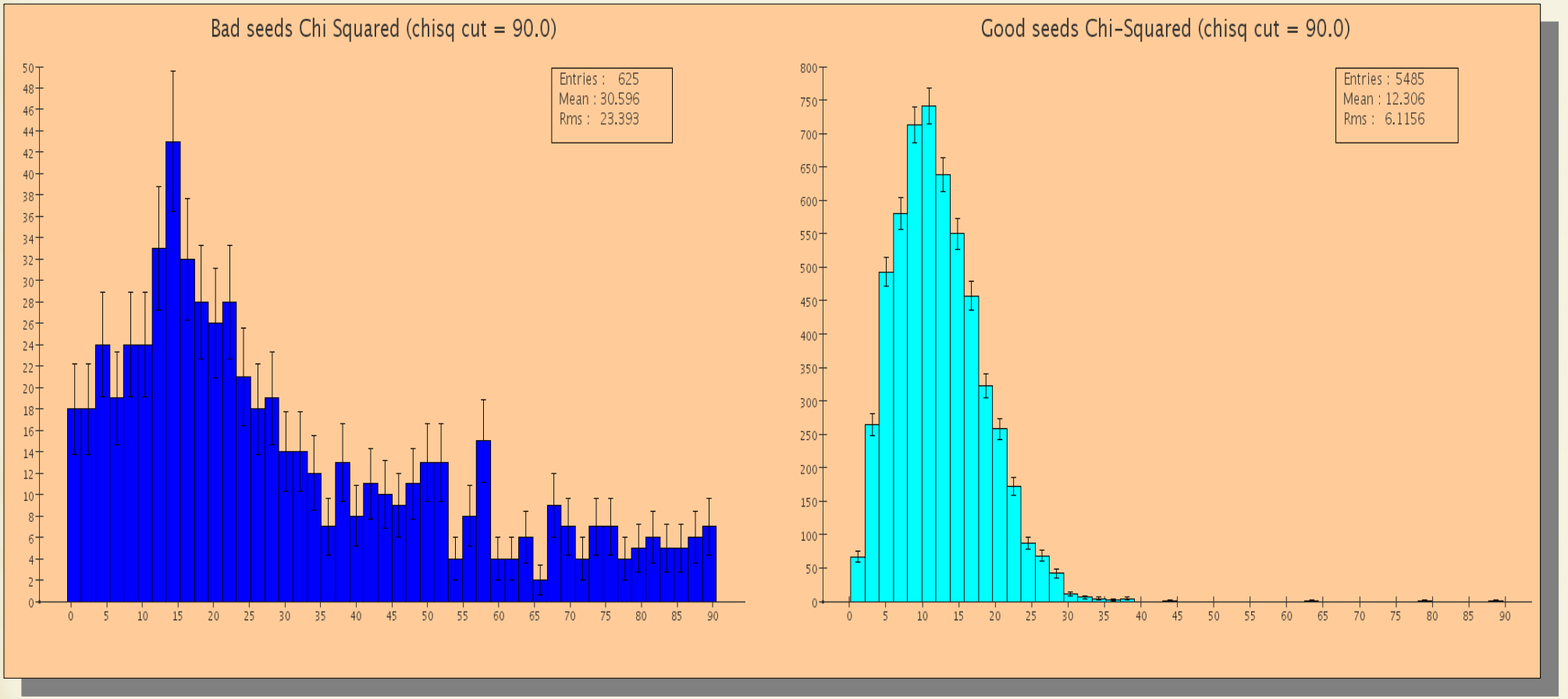
Efficiency results: ~8000 events in tau_5pi_theta20-90_10-200GeV



χ^2 distribution

~8000 events in tau_5pi_theta20-90_10-200GeV

Bad seed ≡ purity < 1



Diagnostics Framework: Motivation

- To further develop / benchmark / optimize SeedTracker, it is useful to have access to all sorts of diagnostic information.
  - Looking at event output can be fruitful, but is limited to looking at just the final product. It is difficult to determine if part of the code is making the wrong decision, for example.
  - Editing the source code is messy and makes syncing with CVS challenging.
- For these reasons, I am working on a framework to facilitate diagnostics.

Diagnostics Framework: Implementation

- A **SeedTracker** object can optionally be assigned an object implementing the **ISeedTrackerDiagnostics** interface.
- **ISeedTrackerDiagnostics** consists of a bunch of callback functions that are called by **SeedTracker** at critical points.
- **AbstractSeedTrackerDiagnostics** contains some convenient implementations. For example, it keeps updated fields for the Bfield and the currently employed strategy.
- **EmptySeedTrackerDiagnostics** extends **AbstractSeedTrackerDiagnostics** and contains stub implementations of all methods. This is probably what you want to subclass to create diagnostics.

Diagnostics Framework: Example

- Suppose we are interested in cases where **SeedTracker** compares two seeds and gets the comparison wrong.
- The applicable callback function is:

```
public void fireMergeIsBetterDiagnostics
            (SeedCandidate newSeed, SeedCandidate
            oldSeed, boolean better);
```

- We can then check (using event information) that newSeed is actually preferable to oldSeed when *better* = **true**. If it's not, we can try to figure out what went wrong.

TODO:

- Finalize list/parameters of callbacks
- Finish writing documentation
- Commit diagnostic package to CVS.
- Run tests on more events / different strategies / new detector models