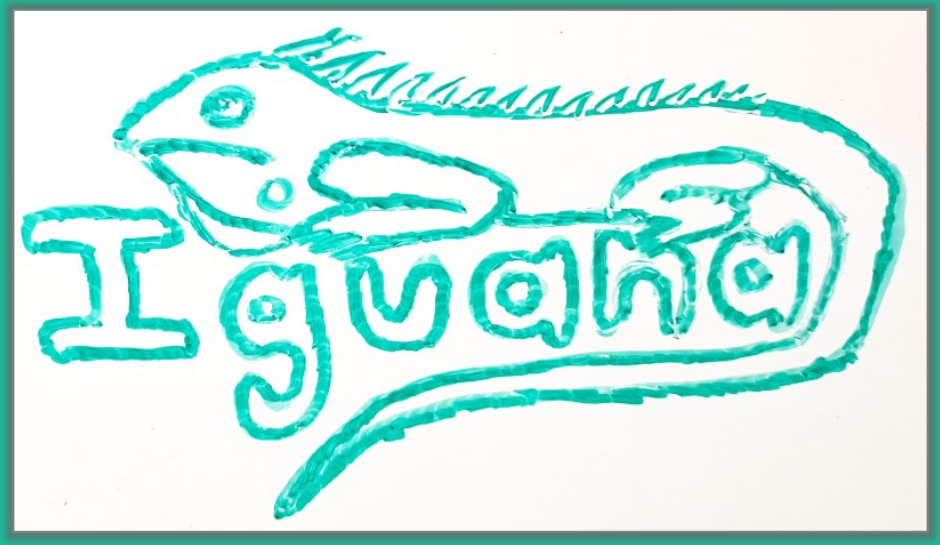


Status & Outlook

Chris Dilks

CLAS Collaboration Meeting

November 2026

Introduction

Key Issue: how do analyzers share common code?

- **Our Approach: Common Analysis Repository → named "Iguana"**
  - High-level analysis procedures / tools
  - Examples: fiducial cuts, momentum corrections, kinematics
  - Available for all analyzers and Run Groups
- **Data preservation → Algorithm preservation**

Iguana is designed to satisfy these goals

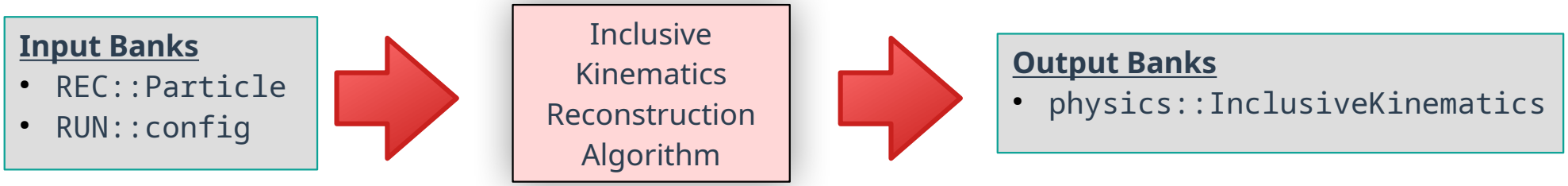
https://code.jlab.org/hallb/clas12/iguana

What's an Iguana Algorithm?

Iguana algorithm:

A function that maps a set of input HIPO banks to a set of output HIPO banks

For example:



The algorithm calculates x, Q^2, *etc.* based on scattered electron and run number (for the initial beam energy), and stores those values in a new dedicated bank

This is an example *creator* algorithm, since it creates a *new* bank

What's an Iguana Algorithm?

`REC::Particle` Bank Details

Reconstructed Particle Information

Return to main tables

Item Name	Type	Description
pid	int	particle id in LUND conventions
px	float	x component of the momentum (GeV)
py	float	y component of the momentum (GeV)
pz	float	z component of the momentum (GeV)
vx	float	x component of the vertex (cm)
vy	float	y component of the vertex (cm)
vz	float	z compo
vt	float	RF and
charge	byte	particle
beta	float	particle
chi2pid	float	Chi2 of
status	short	particle

`RUN::config` Bank Details

Run Configuration

Return to main tables

Item Name	Type	Description
run	int	RUN number from CODA or GEMC
event	int	Event number
unixtime	int	Unix time (seconds)
trigger	long	trigger bits
timestamp	long	time stamp from Trigger Interface (TI) board (4 nanoseconds)
type	byte	type of the run
mode	byte	run mode
torus	float	torus setting relative value(-1.0 to 1.0)
solenoid	float	solenoid field setting (-1.0 to 1.0)

Inclusive Kinematics Reconstruction Algorithm

`physics::InclusiveKinematics`

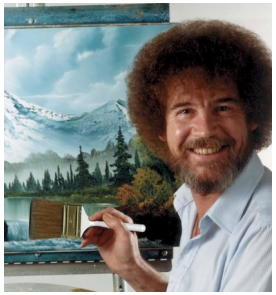
Description: inclusive kinematics

Creator Algorithm: `iguana::physics::InclusiveKinematics`

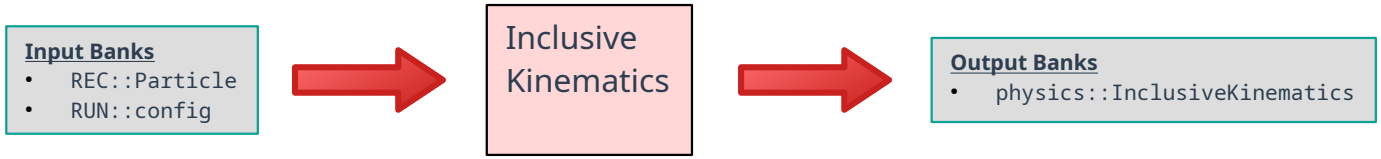
Variable	Type	Description
pindex	short	`REC::Particle` row (pindex) of the scattered electron
Q2	double	Q^2 (GeV ^{2})
x	double	x_B
y	double	y
W	double	W (GeV)
nu	double	ν
qx	double	x-component of virtual photon momentum q
qy	double	y-component of virtual photon momentum q
qz	double	z-component of virtual photon momentum q
qE	double	E-component of virtual photon momentum q
beamPz	double	beam momentum z-component (GeV)
targetM	double	target mass (GeV)

Three Iguana Algorithm Types

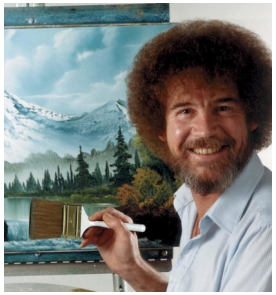
Three Iguana Algorithm Types

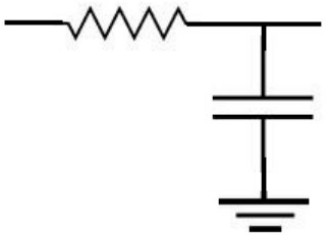


1. Creator algorithm: creates a new bank

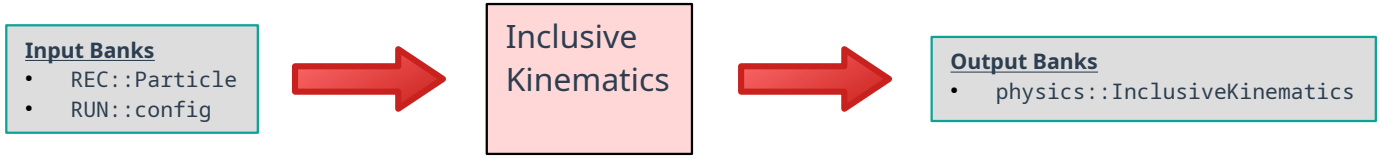


Three Iguana Algorithm Types

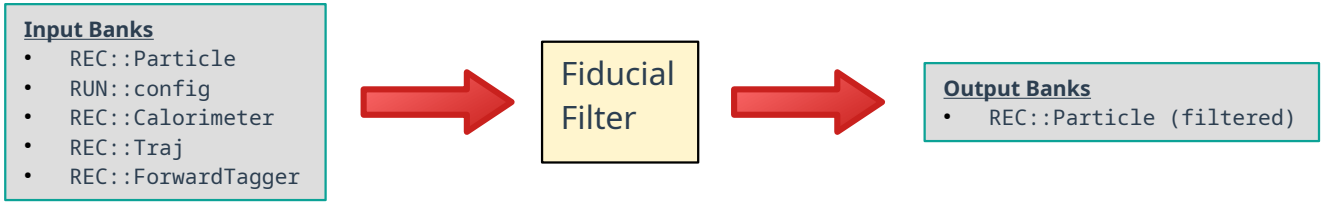




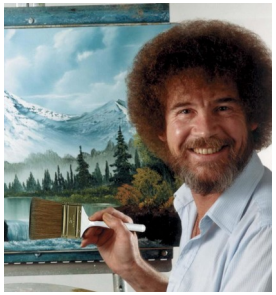
1. Creator algorithm: creates a new bank

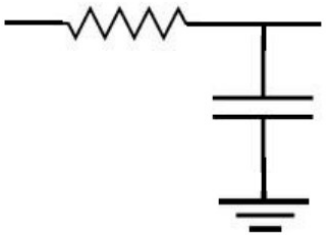


2. Filter algorithm: filters rows of a bank



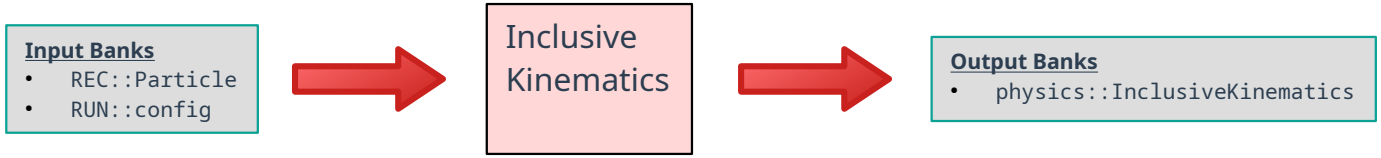
Three Iguana Algorithm Types



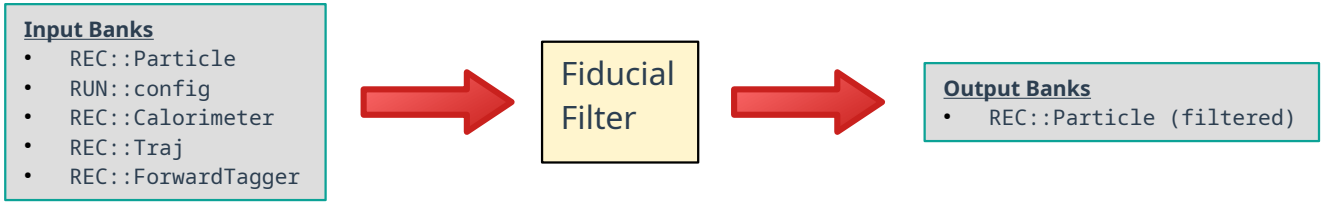




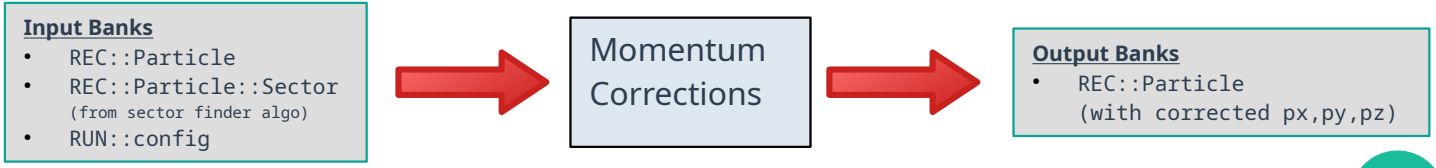
1. Creator algorithm: creates a new bank



2. Filter algorithm: filters rows of a bank

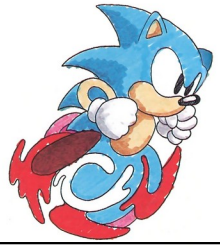


3. Transformer algorithm: mutates values within a bank



Run Function

There are a few ways to use an Iguana algorithm, but the most practical usage is via the **“Run function”***



```
bool Run(hipo::bank& bank1, hipo::bank& bank2, …) const;
```

- Parameters are lvalue references (`bank&`), so the *caller* must *own* the banks
- Some bank parameters are inputs, some outputs, some both
- Input banks which are only *read* are constant lvalue references (`bank const&`) to prevent mutation
- Boolean return value is used as an *event-level filter*: if `false` is returned, consider abandoning the current event’s processing

*Other languages use bindings or action functions – see docs for details…

Run Function – acting on `hipo::banklist`

Another type of Run function acts on `hipo::banklist ( = std::vector<hipo::bank>)`

```
bool Run(hipo::banklist& banks) const;
```

- Great for automated testing *within* Iguana
- Okay enough for simple algorithm sequences
- Terrible for a full-blown analysis → **DON'T USE THIS!**



Mutation!?



- Filter algorithms mutate the list of “active” rows
- Transformer algorithms mutate the *values*

This is “Forced Mutation”, and is the current approach

Pros

- Easy to swap in/out algorithms
- Requires practically no change to user’s downstream code
- Easy for test automation
- No need to copy input data

Cons

- A scary black box changed my numbers!
  - *User must read the algorithm and docs*
- Cannot run 2 algorithms in parallel
  - *They’re probably fast enough (~1-100 µs / ev)*
  - *Algorithm sequence ordering matters*
- Loss of original input bank values
  - *User must copy them*

Mutation!?



Proposed Alternative: "Optional Mutation", one of 2 options:

1. Mutate by Default

- Stay with the current approach, but user copies banks only if needed
- Just need a 'clone bank' method in libHipo
- No changes needed in Iguana
- No changes needed for users

2. Mutate iff Forced

- All algorithms become creators
- Mutation can happen if output bank lvalue ref points to const input bank
- Breaks the API (Iguana v1 → v2)
- Users need to update their code
- Less efficient (always copying)

The 3 Mutation Options In Practice



0. Forced Mutation

```
// run the algorithms
filter.Run(bank);
transformer.Run(bank);
creator.Run(bank, new_bank);
```

1. Mutate by Default

```
// `Clone` copies the bank & values
auto bank_unfiltered = bank.Clone();
auto bank_orig_vals  = bank.Clone();

// run the algorithms
filter.Run(bank);
transformer.Run(bank);
creator.Run(bank, newBank);
```

2. Mutate iff Forced

```
// new Run function:
bool Run(bank const& input, bank& output) const;
```

2a. no mutation

```
// existing copy constructor just copies schema
auto bank_filtered    = bank;
auto bank_transformed = bank;

// run the algorithms
filter.Run(bank, bank_filtered);
transformer.Run(bank_filtered, bank_transformed);
creator.Run(bank_transformed, new_bank);
```

2b. force mutation

```
// run the algorithms, with output = input
filter.Run(bank, bank);
transformer.Run(bank, bank);
creator.Run(bank, new_bank);
```

New Algorithms!

since v0.8

- Depolarization calculations
- Pass-2 fiducial cuts for RG-A
- Proton energy loss for RG-A
- Lepton PID
- Kaon PID
- MC Proximity Matching



Depolarization

Main Author: Chris Dilks

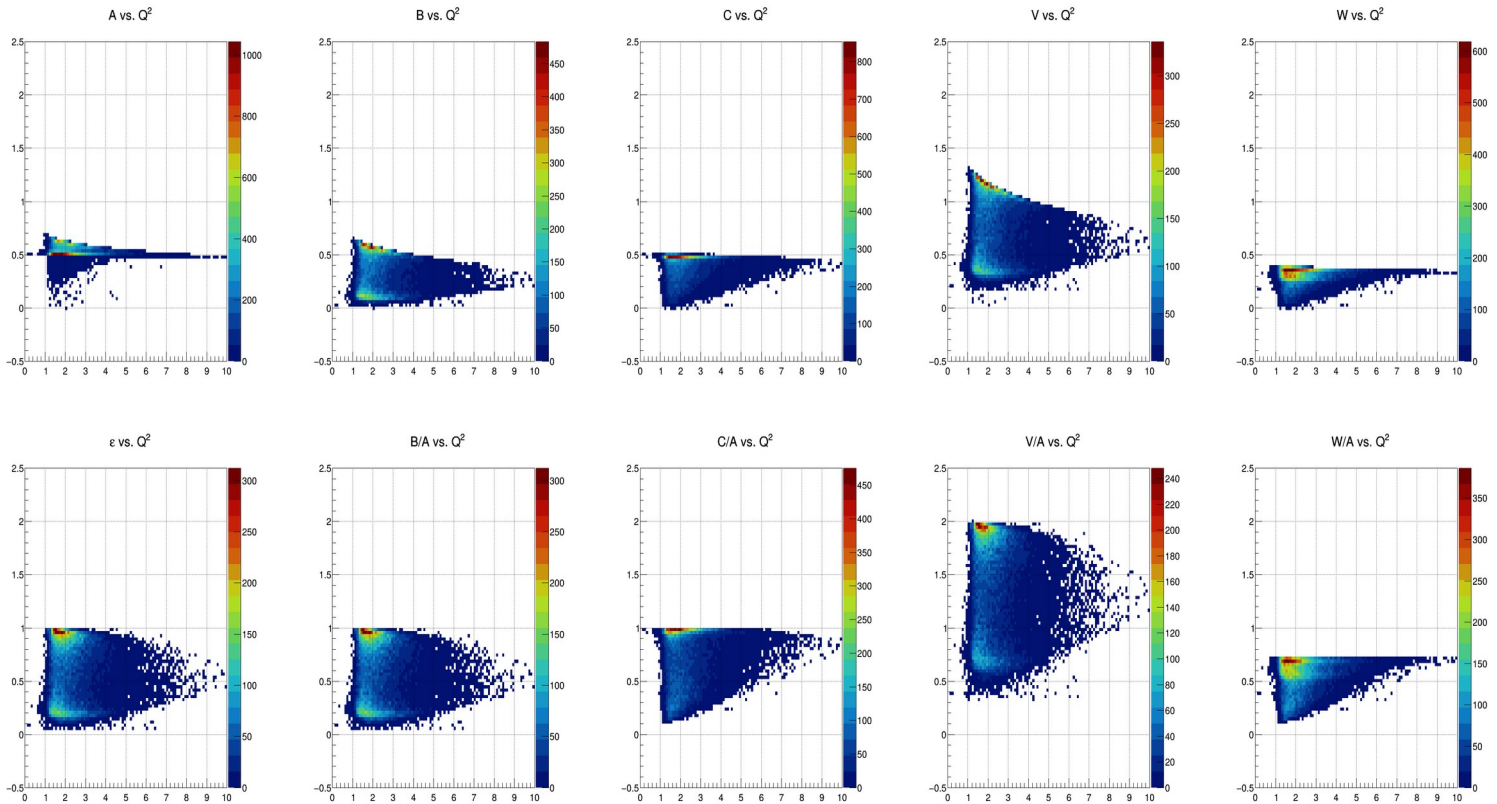
Class iguana::physics::Depolarization

Calculate depolarization factors

Algorithm Type: Creator

Created Bank:

Variable	Type	Description
epsilon	double	$\varepsilon(Q^2, x, y)$, the ratio of transverse and longitudinal photon flux
A	double	depolarization factor $A(\varepsilon, y)$
B	double	depolarization factor $B(\varepsilon, y)$
C	double	depolarization factor $C(\varepsilon, y)$
V	double	depolarization factor $V(\varepsilon, y)$
W	double	depolarization factor $W(\varepsilon, y)$



RG-A Fiducial Cuts

Main Author: Timothy Hayward

(*cf.* FiducialFilterPass1 algorithm, written by Gregory Matousek, based on Stefan Diehl's implementation)

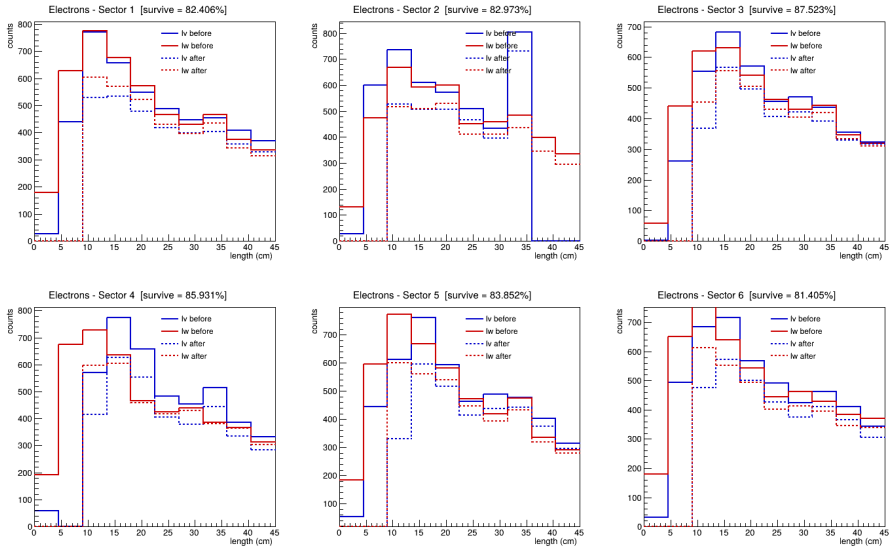
Class iguana::clas12::rga::FiducialFilterPass2

Filter the `REC::Particle` bank using subsystem-specific fiducial cuts

Algorithm Type: Filter

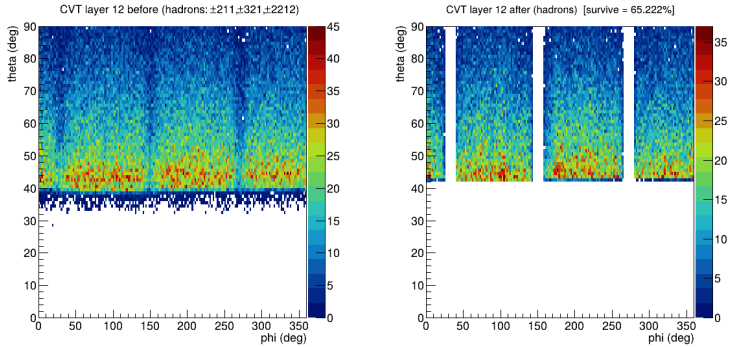
- PCal-only edge cuts on lv & lw with strictness thresholds
- Forward Tagger annulus + low efficiency hole vetoes
- Central detector (CVT) fiducial:
  - require edge > edge_min (default 0) and vetoes on gaps between CVT sectors
- Drift Chamber (DC) fiducial:
  - three region edge thresholds with separate inbending/outbending track logic

PCAL Electrons



Example validation plots

CVT Hadrons



RG-A Proton E-Loss Correction

Main Author: Timothy Hayward

Class iguana::clas12::rga::ProtonEnergyLossCorrection

Apply period-dependent momentum/angle corrections to reconstructed protons in `REC::Particle`

Algorithm Type: Transformer

For each `REC::Particle` row:

- selects protons (`pid == 2212`)
- classifies the track as FD vs CD using the `status` code
- determines the run period from `RUN::config::run`
- computes (p, theta, phi) from (px, py, pz)
- applies period- and region-dependent corrections (configured in YAML)
- writes corrected (px, py, pz) back to the bank

Validation *table*, rather than a plot ...

```
ProtonEnergyLossCorrectionValidator summary
  theta bins: 5 to 70 (deg) in steps of 5 (deg)
  total protons (all theta): 88137
  total protons in theta range: 85021

  BinRangeTheta(deg)    N        <p_before>(GeV)    <p_after>(GeV)    <delta>(GeV)
  ------------------------------------------------------------------------------
  [5.000000,10.000000)           4647             1.798896            1.804190            0.005295
  [10.000000,15.000000)          9181             1.581179            1.586485            0.005306
  [15.000000,20.000000)          11155            1.426338            1.432284            0.005945
  [20.000000,25.000000)          11267            1.333337            1.340701            0.007363
  [25.000000,30.000000)          8063             1.253362            1.263466            0.010104
  [30.000000,35.000000)          6322             1.103491            1.118207            0.014716
  [35.000000,40.000000)          6634             1.005398            1.015656            0.010257
  [40.000000,45.000000)          6716             0.922842            0.928238            0.005395
  [45.000000,50.000000)          5771             0.829814            0.832560            0.002747
  [50.000000,55.000000)          5149             0.763854            0.761973            -0.001881
  [55.000000,60.000000)          4171             0.661087            0.655264            -0.005823
  [60.000000,65.000000)          3365             0.562937            0.552127            -0.010811
  [65.000000,70.000000)          2580             0.493097            0.467930            -0.025166
```

Lepton ID Filter

Main Author: Mariana Tenorio Pita

Class iguana::clas12::LeptonIDFilter

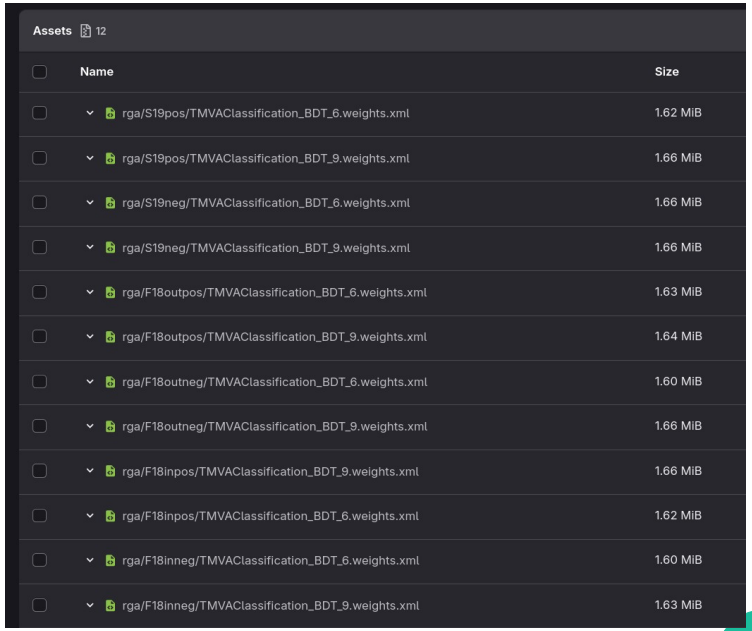
Filter the leptons from the pion contamination using TMVA models

Algorithm Type: Filter

For each lepton, either positron or electron, it takes some variables from `REC::Particle` (P, Theta and Phi) and `REC::Particle` (Sampling fraction and second moments). Using those variables, it call the TMVA method using the weight file, and it computes a score. By a pplying a cut to the score we can separate leptons from pions.

- No validation plots yet…
  these are *necessary* before this algorithm can be trusted!
  (service work needed!)

- This was the first algorithm to use the Model Registry on GitLab, a place to store (and train) models (e.g., neural networks)
  - TMVA weights files are 1.6 MB each → 20 MB total
  - Source code itself is 20 MB total
    - Dominated by Photon GBT weights within, consuming 17 MB (should be moved to model registry…)



Kaon PID

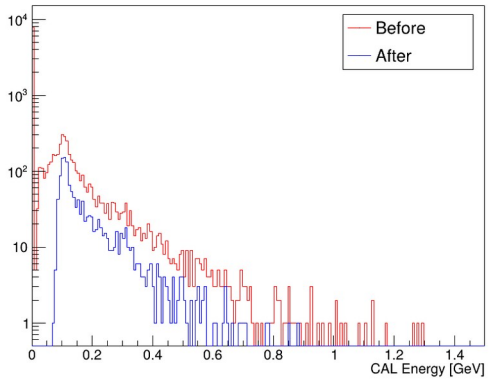
Main Author: Aron Kripko

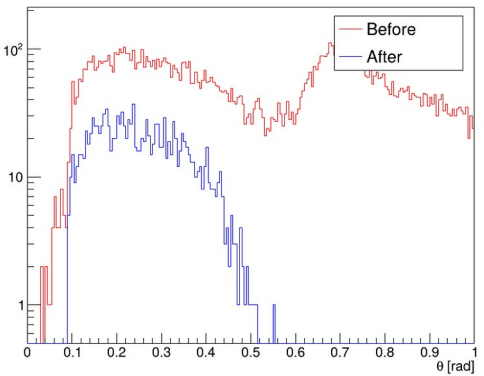
Class iguana::clas12::MesonDNNIDFilter

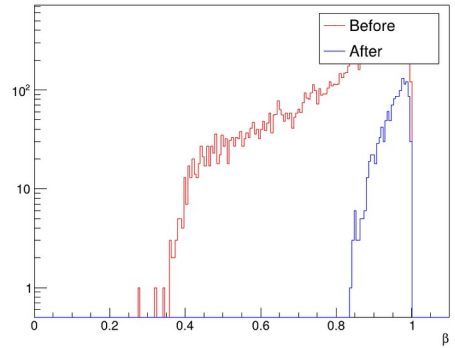
Filter charged kaons using a TMVA DNN classifier

- For each positively charged kaon candidate, a set of calorimeter, FTOF and kinematic observables is collected and passed to a TMVA DNN model
- The resulting score is compared to a configurable threshold to determine whether the candidate passes the selection

Kaons before & after filter







Monte Carlo Support

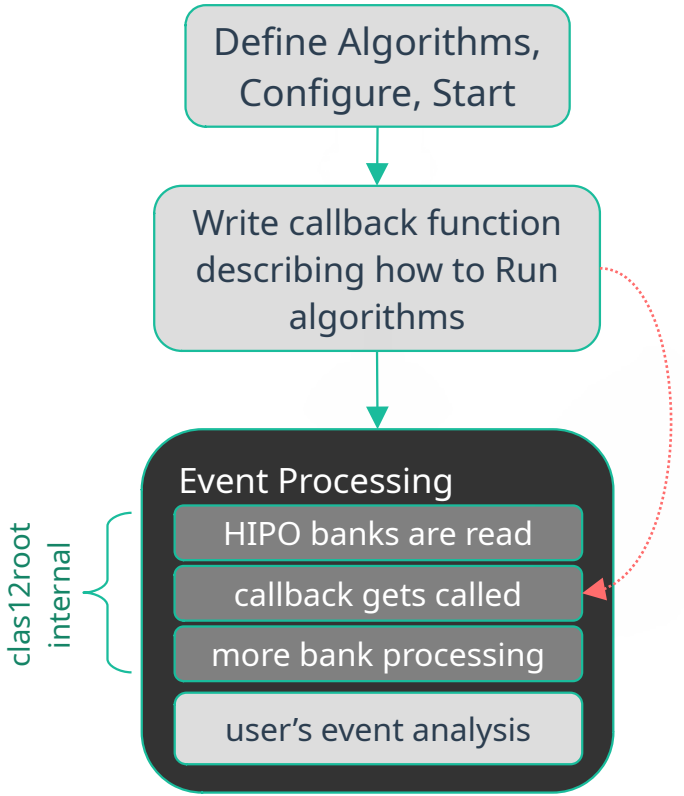
- Algorithms which process REC::Particle are generalized to handle most particle banks, e.g., MC::Particle
- InclusiveKinematics algorithm can now use MC::Lund to calculate *true* x, Q^2, etc.
- Beam energy control (if runnum==11)
- New algorithm MatchParticleProximity:
  - May be useful for files that lack truth-matching banks
  - Matches by (θ,ϕ) Euclidean distance



By Steve Hedin - File:The_Casino,_Monte_Carlo_-_panoramio.jpg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=88514874

Software Integration: Callbacks

- Some frameworks / libraries implement their own event processing, for example:
  - clas12root
  - HIPO data frame library and clas12-rad (see also Derek's slides)
- To call an algorithm's **Run Function**, use a "callback function", which describes:
  - How the **Run** functions are called
  - What order they are called in
- Give the callback function to the framework / library; examples:
  - clas12root example
  - HIPO data frame example
- The callback function will be executed at the right time during event processing
  - e.g., in clas12root, right after reading the banks but before further processing →
- This approach is for any framework / library which uses 'hipo::bank' objects
  - For those that don't, either use Action Functions, or need a new approach



Multithreading Support

- **hipo::chain** is a new multithreaded HIPO file reader, developed by Mathieu Ouillon
  - Documentation: https://pages.jlab.org/hallb/clas12/hipo-cpp/api/chain.html
  - Uses callback function approach (see previous slide)
- Thread safety of Iguana algorithms has been validated and fixed
  - Thread sanitizer
  - Bank hash comparisons between single-threaded and multi-threaded processing
  - Tested in the CI
  - Critical for data frame support
- **Iguana is thread safe!**



multi-threaded event processing, 4 threads:
each thread calls Run function on events within a HIPO record

- open HIPO file
- start iguana algorithms
- define callback function (see previous slide)

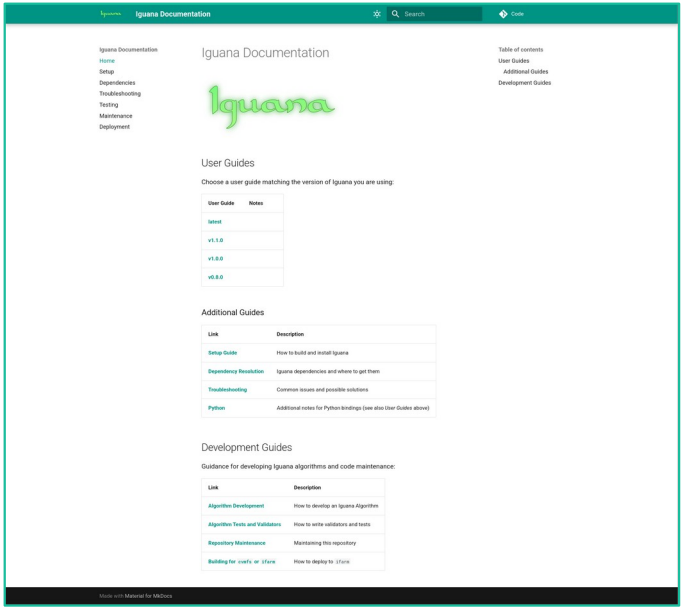
- process record 1
- process record 2
- process record 3
- process record 4
- process record 8
- process record 5
- process record 7
- process record 6
- process record 9

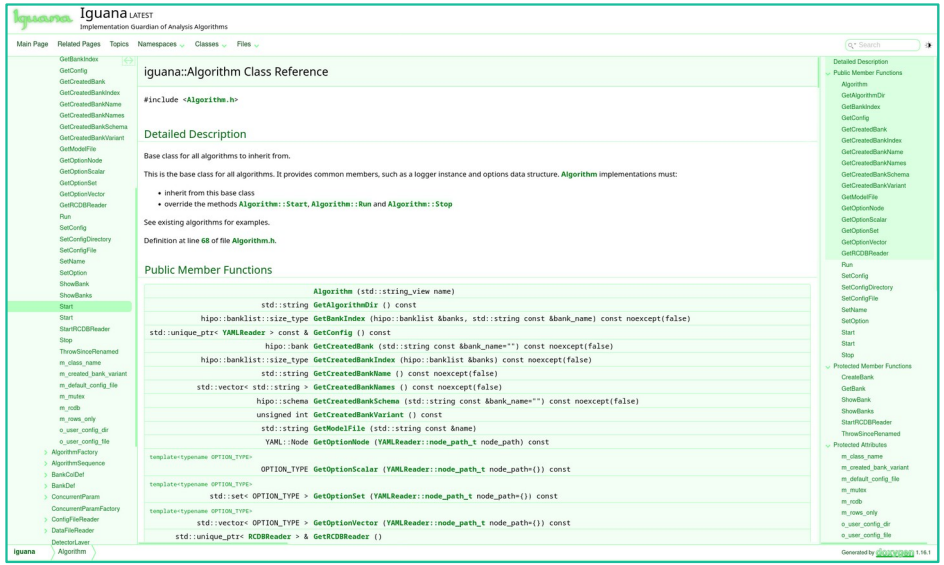
- done reading HIPO file
- continue analysis

Documentation and Resources

- API documentation for each algorithm (along with all the infrastructure)
- Documentation for each version of Iguana (on can say: “I used fiducial cuts from Iguana v1.1.0…”)

https://pages.jlab.org/hallb/clas12/iguana/





Discussion: User and Writer Engagement

- **Who uses Iguana and how do you use it?**
- **Any Issues or feedback for Iguana?**
- **We want more Algorithms!**
  - Service work?
  - Make certain parts of your analysis “accessible” to others
    - Require somehow in publication process?
    - Who writes the algorithm: the analyzer, a service worker, or Iguana developers (me)?
    - Who reviews the algorithm?
      - I review them at least for code quality and high-level integration stuff
      - Need also a reviewer who is more familiar with the details
- **Run Groups tend to have Common Procedures**
  - Reach out to Analysis Coordinators and… coordinate!

backup

Problem Statement

Dear Bob,

Here are the fiducial cuts, please see the attached code.

Cheers,

Alice



Dear Alice,

Thanks Alice! I had to change some things a bit in order to get it to integrate with my analysis code, but I think it's working well.

Cheers,

Bob

… 2 Months Later …

Dear Bob,

I have developed new fiducial cuts, please see the attached code. I also did a lot of cleanup and refactoring.

Cheers,

Alice

- Bob now has to spend time re-integrating the new version
- Alice's version may be correct and validated, but Bob may have introduced a bug while "integrating"
  - Even worse case: if Bob is using a different programming language, he may be *porting* (translating) Alice's code to that language
- What version of Alice's code is Bob actually using, and is it easy to tell?
- This is based on a true story…

Key Issue: how do analyzers share common code?

Z-vertex Filter

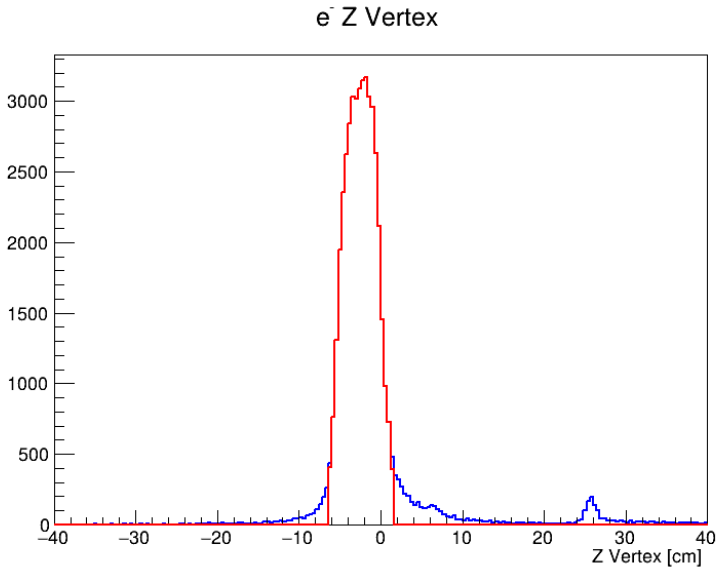
Main Author: Richard Tyson

Class iguana::clas12::ZVertexFilter

Filter a particle bank by cutting on Z Vertex

Algorithm Type: Filter

Currently just filters electron v_z




```yaml
# default cuts
- default:
  vz: [ -20.0, 20.0 ]

### RUN GROUP A ##########

# RG-A spring2018 inbending, 1st period
- runs: [ 3031, 3087 ]
  vz: [ -6.061, 1.819 ]

# RG-A spring2018 outbending, 1st period
- runs: [ 3103, 3293 ]
  vz: [ -7.272, 0.909 ]

# RG-A spring2018 inbending, 2nd period
- runs: [ 3306, 3817 ]
  vz: [ -6.061, 1.819 ]

# RG-A spring2018 outbending, 2nd period
- runs: [ 3820, 3987 ]
  vz: [ -7.272, 0.909 ]

# RG-A spring2018 inbending, 3rd period
- runs: [ 4003, 4325 ]
  vz: [ -6.061, 1.819 ]

# RG-A fall2018 inbending
- runs: [ 4763, 5419 ]
  vz: [ -6.364, 1.515 ]

# RG-A fall2018 outbending
- runs: [ 5423, 5666 ]
  vz: [ -7.879, 0.303 ]

# RG-A spring2019 inbending
- runs: [ 6616, 6783 ]
  vz: [ -6.364, 1.515 ]

### RUN GROUP B ##########
# FIXME: these are not yet the official, common RG-B vertex cuts; for now,
# they are just the cuts that C. Dilks is using for SIDIS analysis.

# RG-B spring2019 inbending # FIXME: not official (see FIXME above)
- runs: [ 6156, 6603 ]
  vz: [ -8.0, 3.0 ]

# RG-B fall2019 outbending # FIXME: not official (see FIXME above)
- runs: [ 11093, 11283 ]
  vz: [ -10.0, 2.5 ]

# RG-B spring2020 inbending # FIXME: not official (see FIXME above)
- runs: [ 11323, 11571 ]
  vz: [ -8.0, 3.0 ]
```

Momentum Correction

Main Author: Chris Dilks (for iguana impl.)

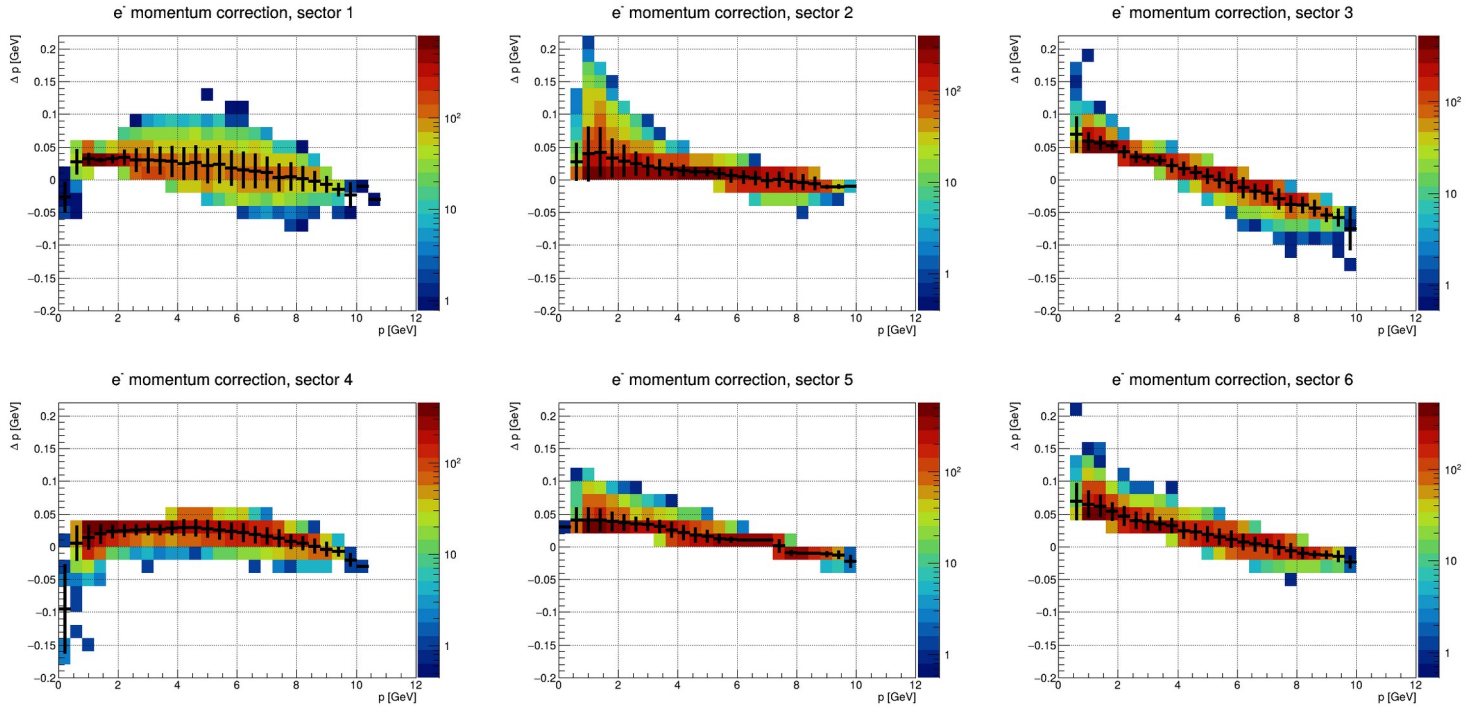
Class iguana::clas12::rga::MomentumCorrection

Momentum Corrections

Algorithm Type: Transformer

Adapted from **Richard Capobianco**'s code:
https://clasweb.jlab.org/wiki/index.php/CLAS12_Momentum_Corrections#tab=Correction_Code

NOTE: these were developed for RG-A Pass 1



FT Energy Correction

Main Author: Asli Acar

Class iguana::clas12::rga::FTEnergyCorrection

Forward Tagger energy correction

Algorithm Type: Transformer

Corrects px, py, and pz from RECFT::Particle

This has been validated only for RG-A Fall 2018 Outbending data. It *may* also be appropriate for Pass-2 data from RG-A, RG-B, and RG-K, but they should be validated.

```
auto rho   = std::hypot(px, py, pz);
auto E_new = E + 0.0208922 + 0.050158 * E - 0.0181107 * pow(E, 2) + 0.00305671 * pow(E, 3) - 0.000178235 * pow(E, 4);
return {
  E_new * (px / rho),
  E_new * (py / rho),
  E_new * (pz / rho),
  E_new
};
```

← corrected 4-momentum

No validator plots in Iguana
(but also it was a very early algorithm!)

Photon GBT Filter

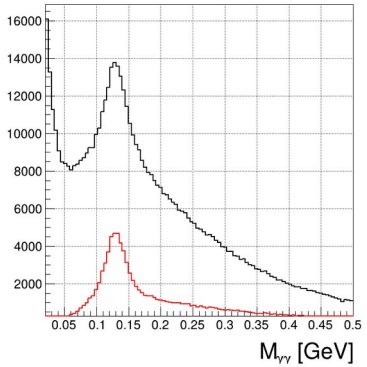
Main Author: Gregory Matousek

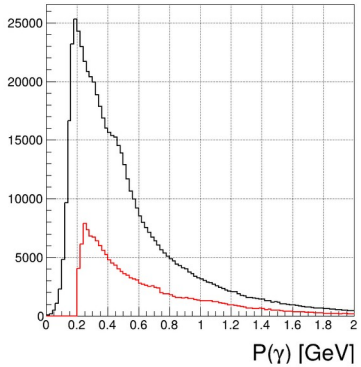
Class iguana::clas12::PhotonGBTFilter

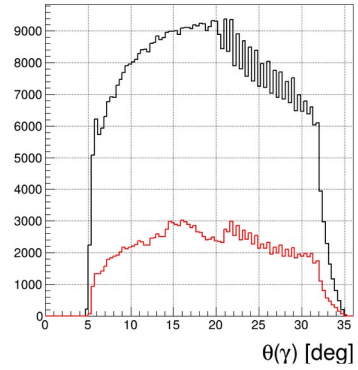
Filter the `REC::Particle` photons using pretrained GBT models

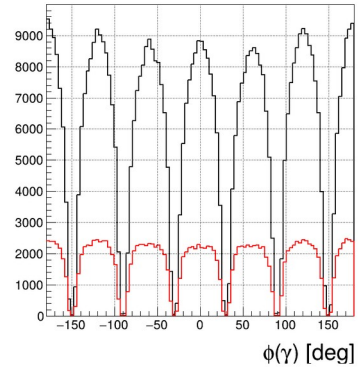
Algorithm Type: Filter

Improves photon PID using a Gradient Boosted Tree









For each photon (labeled the photon of interest or POI), we obtain its intrinsic features (energy, angle, pcal edep, etc.) and features corresponding to its nearest neighbors (angle of proximity, energy difference, etc.). This requires the reading of both the REC::Particle and REC::Calorimeter banks. An input std::vector<float> is produced and passed to the pretrained GBT models, which yield a classification score between 0 and 1. An option variable `threshold` then determines the minimum photon `p-value` to survive the cut.

Sector Finder

Main Author: Richard Tyson

Class iguana::clas12::SectorFinder

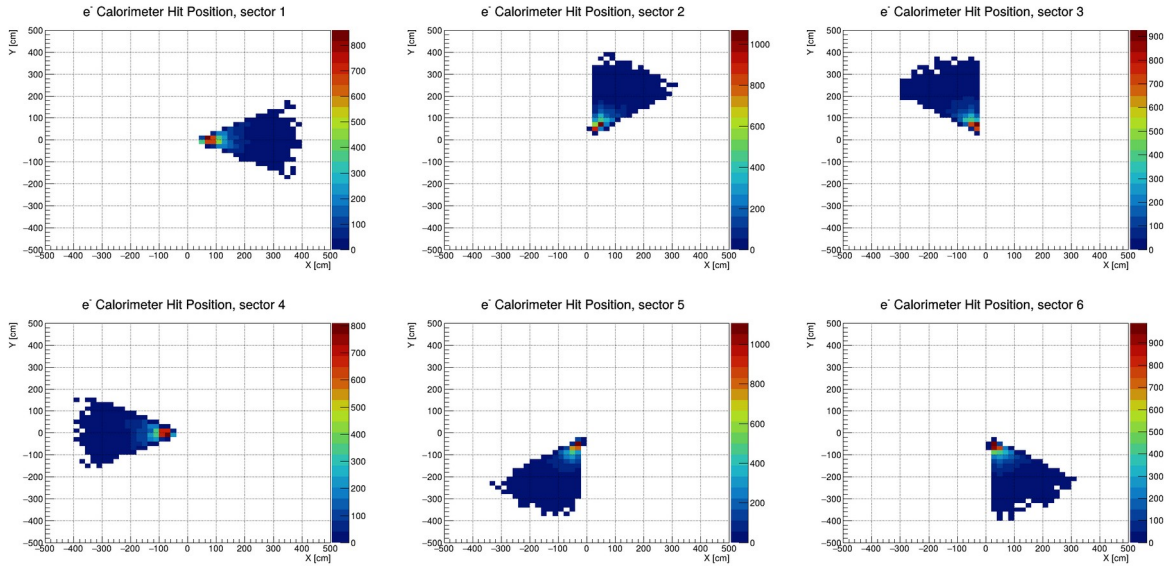
Find the sector for all rows in `REC::Particle`

Algorithm Type: Creator

- Finds a sector for each particle, following clas12root's procedure
  - First try REC::Track
  - Then REC::Scintillator
  - Then REC::Calorimeter
- Generalized to allow for custom bank choices for sector finding

Created Bank:

Variable	Type	Description
`pindex`	`short`	row number in the particle bank
`sector`	`int`	sector for this particle



Inclusive Kinematics

Main Author: Chris Dilks

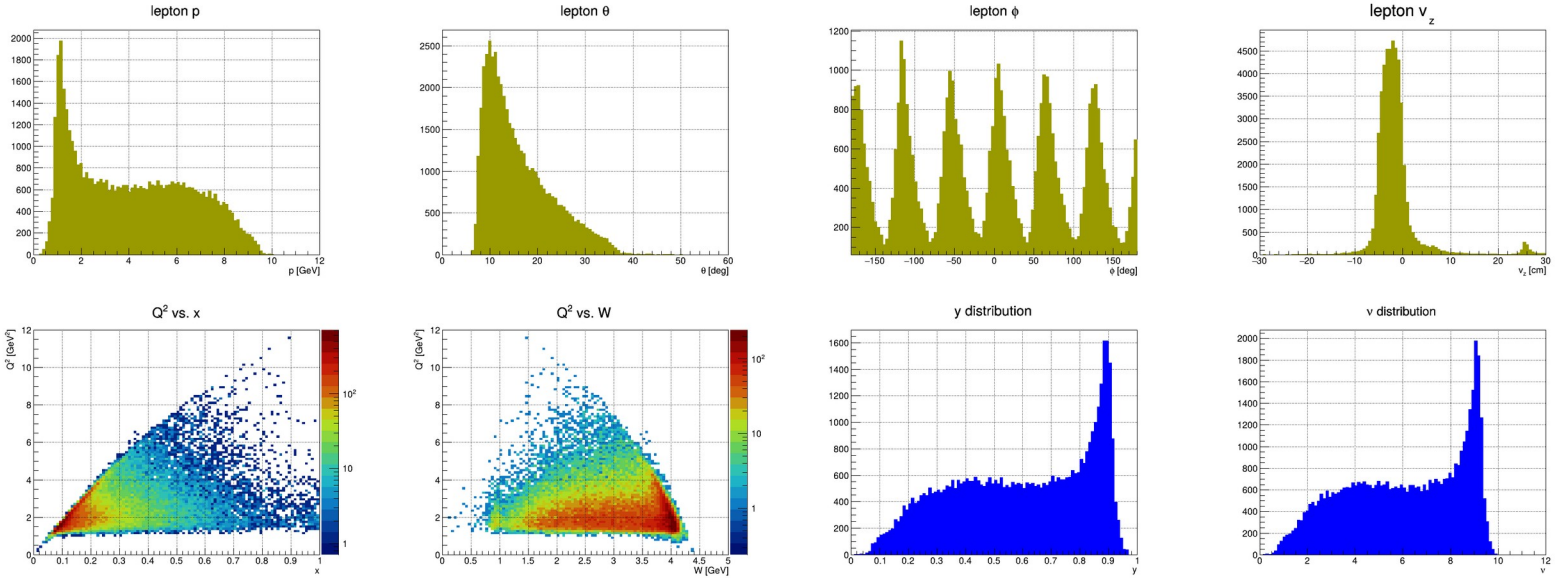
Class iguana::physics::InclusiveKinematics

Calculate inclusive kinematics quantities

Algorithm Type: Creator

Created Bank:

Variable	Type	Description
pindex	short	REC::Particle row (pindex) of the scattered electron
Q2	double	Q^2 (GeV^2)
x	double	x_B
y	double	y
W	double	W (GeV)
nu	double	ν
qx	double	x-component of virtual photon momentum q
qy	double	y-component of virtual photon momentum q
qz	double	z-component of virtual photon momentum q
qE	double	E-component of virtual photon momentum q
beamPz	double	beam momentum z-component (GeV)
targetM	double	target mass (GeV)



Single-Hadron Kinematics

Main Author: Chris Dilks

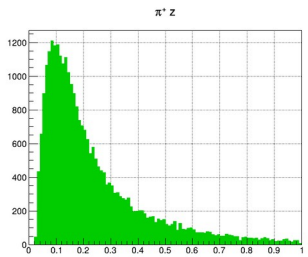
Class iguana::physics::SingleHadronKinematics

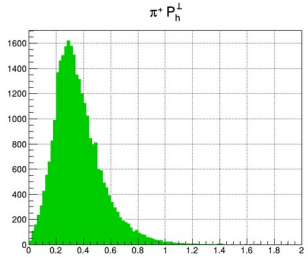
Calculate semi-inclusive hadron kinematic quantities

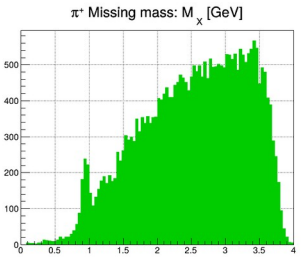
Algorithm Type: Creator

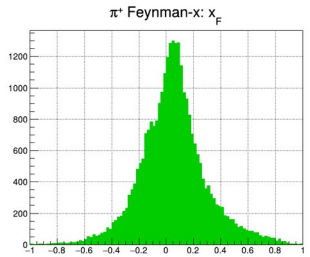
Created Bank:

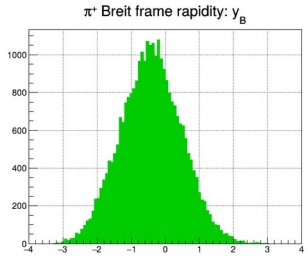
Variable	Type	Description
`pindex`	`short`	`REC::Particle` row (pindex) of the hadron
`pdg`	`int`	PDG code of the hadron
`z`	`double`	z: Momentum fraction of the fragmenting parton carried by the hadron
`PhPerp`	`double`	$P_h^\perp$: transverse momentum of the hadron in the $\perp$-frame (transverse to $\vec{q}$)
`MX2`	`double`	$M_X^2(ehX)$: Missing mass squared of the hadron
`xF`	`double`	x_F: Feynman-x of the hadron
`yB`	`double`	$y_{h,B}$: Breit frame rapidity of the hadron
`phiH`	`double`	ϕ_h: q-azimuthal angle between the lepton-scattering plane and the $\vec{q} \times \vec{P}_h$ plane; if the value is `tools::UNDEF`, the calculation failed
`xi`	`double`	ξ_h: Longitudinal momentum fraction of the nucleon carried by the hadron

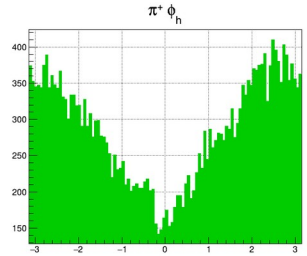


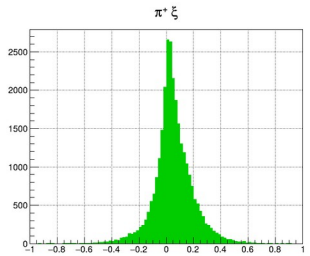












Dihadron Kinematics

Main Author: Chris Dilks

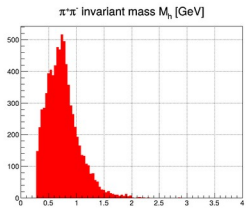
Class iguana::physics::DihadronKinematics

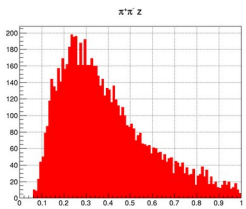
Calculate semi-inclusive dihadron kinematic quantities

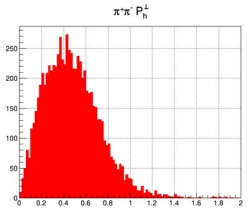
Algorithm Type: Creator

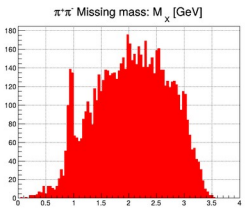
Created Bank:

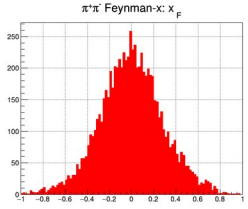
Variable	Type	Description
pindex_a	short	REC::Particle row (pindex) of hadron A
pindex_b	short	REC::Particle row (pindex) of hadron B
pdg_a	int	PDG code of hadron A
pdg_b	int	PDG code of hadron B
Mh	double	M_h: Invariant mass of the dihadron
z	double	z: Momentum fraction of the fragmenting parton carried by the dihadron
PhPerp	double	$P_h^\perp$: transverse momentum of the dihadron in the $\perp$-frame (transverse to $\vec{q}$)
MX2	double	$M_X^2(ehhX)$: Missing mass squared of the dihadron
xF	double	x_F: Feynman-x of the dihadron
yB	double	$y_{h,B}$: Breit frame rapidity of the dihadron
phiH	double	ϕ_h: q-azimuthal angle between the lepton-scattering plane and the $\vec{q} \times \vec{P}_h$ plane; if the value is tools::UNDEF, the calculation failed
phiR	double	ϕ_R: q-azimuthal angle between the lepton-scattering plane and dihadron plane; if the value is tools::UNDEF, the calculation failed
theta	double	θ: The 'decay' angle of hadron A in the dihadron rest frame, with respect; to the dihadron momentum direction

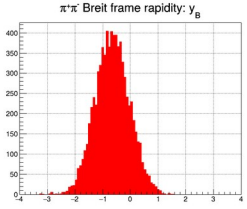


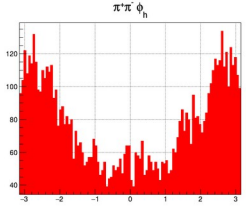


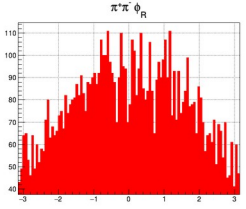


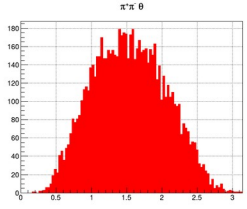












CalorimeterLinker and TrajLinker

Main Author: Chris Dilks

Class iguana::clas12::CalorimeterLinker

Link particle bank to bank `REC::Calorimeter`

Algorithm Type: Creator

This algorithm reads `REC::Calorimeter` and produces a new bank, `REC::Particle::Calorimeter`, to make it easier to access commonly used `REC::Calorimeter` information for each particle.

Class iguana::clas12::TrajLinker

Link particle bank to bank `REC::Traj`

Algorithm Type: Creator

This algorithm reads `REC::Traj` and produces a new bank, `REC::Particle::Traj`, to make it easier to access commonly used `REC::Traj` information for each particle.

- Creates new banks, for example this one from CalorimeterLinker →
- Currently only used for the (deprecated) Pass-1 Fidudial Filter
- May be applicable for other algorithms...

`REC::Particle::Calorimeter`

Description: calorimeter information linked to particles

Creator Algorithm: `iguana::clas12::CalorimeterLinker`

Variable	Type	Description
`pindex`	`short`	row number in the particle bank
`pcal_found`	`byte`	1 if PCAL info found for this particle, 0 otherwise
`pcal_sector`	`int`	PCAL sector
`pcal_lu`	`float`	PCAL distance on U-side
`pcal_lv`	`float`	PCAL distance on V-side
`pcal_lw`	`float`	PCAL distance on W-side
`pcal_energy`	`float`	Energy associated with the PCAL hit (GeV)

+ analogous columns for ECIN and ECOUT

Algorithm Benchmarks

- Typical event processing ~10-100 μs*
- Slowest algorithms:
  - Photon GBT filter
  - Pass-1 Fiducial filter (deprecated)
  - TrajLinker
- *cf.* reconstruction (via CLARA):
  - ~0.1-100 ms on GitHub CI

*Tested on my laptop; CI seems to be faster (better machines)

Relative times are more useful:
- compare algorithms to each other
- compare CI pipeline runs (e.g., do proposed changes make things slower?)

Benchmarks

Algorithm average execution time per event, for a file with 388722 events:

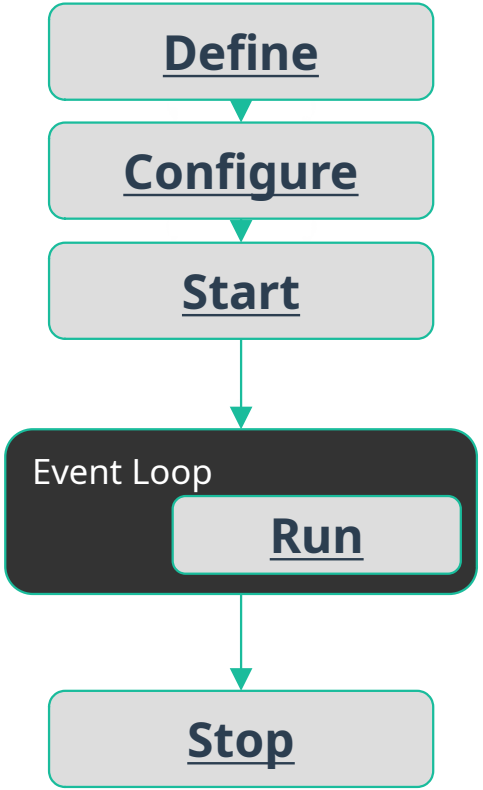
Algorithm	Single Threaded
`clas12::CalorimeterLinker`	61.458 μs/ev
`clas12::EventBuilderFilter`	42.91 μs/ev
`clas12::LeptonIDFilter`	29.455 μs/ev
`clas12::MatchParticleProximity`	17.056 μs/ev
`clas12::PhotonGBTFilter`	374.201 μs/ev
`clas12::SectorFinder`	52.017 μs/ev
`clas12::TrajLinker`	198.188 μs/ev
`clas12::ZVertexFilter`	26.986 μs/ev
`clas12::rga::FTEnergyCorrection`	6.509 μs/ev
`clas12::rga::FiducialFilterPass1`	372.554 μs/ev
`clas12::rga::FiducialFilterPass2`	19.062 μs/ev
`clas12::rga::MomentumCorrection`	112.06 μs/ev
`clas12::rga::ProtonEnergyLossCorrection`	14.149 μs/ev
`example::ExampleAlgorithm`	20.76 μs/ev
`physics::Depolarization`	19.577 μs/ev
`physics::DihadronKinematics`	22.33 μs/ev
`physics::InclusiveKinematics`	18.419 μs/ev
`physics::SingleHadronKinematics`	12.528 μs/ev

How to Use an Algorithm

Users can run multiple algorithms, of course; this explains how to run one

- Before event processing*
  - **Define** an algorithm object
    - Give it a unique name, if you want
  - **Configure** the algorithm
    - See next slide
    - Call any other algorithm-specific functions (rare)
  - Call the **Start** Function to “lock in” your algorithm configuration
- For each event, **Run** the algorithm by calling either
  - ‘Run()’ function
  - Action function(s)
- Call the **Stop** function after event processing (typically unnecessary)

* note: we *could* combine ‘Configure’ and ‘Start’ into the algorithm constructor, but then we lose configuration flexibility…



Algorithm Configuration

- Most algorithms are *configurable*; they include a configuration YAML file with *default* values
- Users can override values in several ways:
  - Use custom YAML file(s)
  - Call 'SetOption' in the code, for a single option
  - Print-out log levels can also be controlled
- Motivation for YAML file design:
  - Change parameters without recompilation
  - YAML was chosen over other options by comparing
    - User-friendliness
    - Flexibility for design needs
    - Available C++ parsing library capabilities
  - Possible Con: YAML is sensitive to indentation
    - Python users are familiar with this

Example YAML config file for Dihadron Kinematics algorithm:

```yaml
physics::DihadronKinematics:

  # the name of the particle bank, only used for users of `hipo::banklist`;
  # e.g., set it to 'MC::Particle' if you prefer
  particle_bank: 'REC::Particle'

  # list of hadrons used to pair dihadrons
  # - each hadron from `hadron_a_list` will be paired to each hadron from `hadron_b_list`
  hadron_a_list: [ 211 ]
  hadron_b_list: [ -211, 211, 2212 ]

  # calculation methods; see documentation and source code for details
  phi_r_method: RT_via_covariant_kT
  theta_method: hadron_a
```

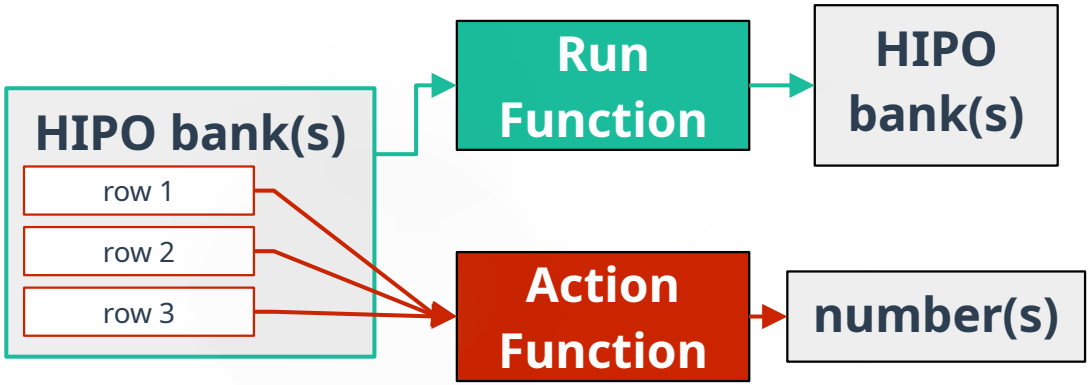
Use log levels to control algorithm print-out verbosity

Level	Description
`trace`	the most verbose level, used for fine-grained printouts for each event
`debug`	less verbose printout, expected to be less frequent than `trace`
`info`	the least verbose printout; this is the default level
`quiet`	use this level to only allow warnings and errors, silencing all other printouts
`warn`	an issue that may or may not be critical
`error`	an issue that is likely critical
`silent`	use this level to silence **all** printouts (use it at your own risk!)

Running an Algorithm

There are two options for running algorithms

- **Run functions**
  - Act on C++ hipo::bank objects
    - Not usable if the user does not have them
  - Simplest way to run



- **Action functions**
  - Simple parameters and simple return values
    - scalar parameters: int, double, string
    - vector parameters: list(int), list(double), list(string) [*i.e.*, std::vector in C++]
  - Usable by anyone: if you have the numbers, you can use them

Run Functions



There are 2 types of Run Functions

- **Run(bank1, bank2, …): acts on bank objects**
  - Parameters are hipo::bank objects (passed by lvalue reference)
  - Unique parameters for each algorithm
  - This is the “best” way to run an algorithm for users, since it acts on individual bank objects

- **Run(hipo::banklist): acts on an ordered list of banks**
  - One parameter, a list of banks (i.e., function signature is the same for each algorithm)
    - Great for test automation
    - Allows the usage of ‘AlgorithmSequence’, a convenience algorithm to run a list of algorithms in a specific order
  - Useful only when using the HIPO reader with a banklist
  - Not fully general in some subtle edge cases (hard to explain here…)

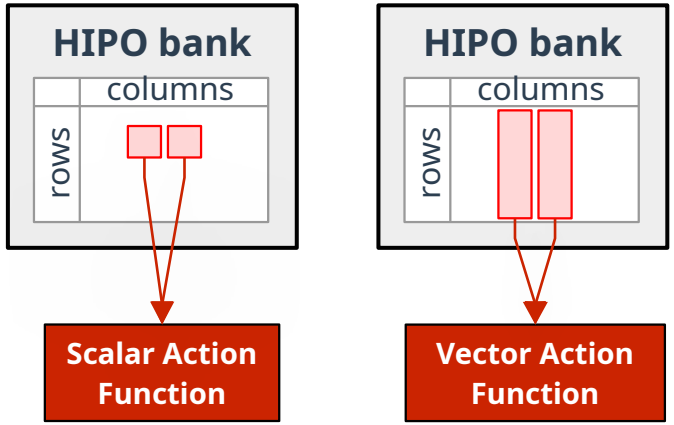
Run Functions (continued)

- Every algorithm has run functions
- Users must be mindful of the *order* in which they call each algorithm's Run functions
  - For example: do you do momentum corrections before or after fiducial cuts?
- **Con**: the user *needs* to have hipo::bank objects
  - Not everyone does, especially if they are not using C++ or Python (cppyy)
    - Alternative is Action Functions
  - Mutates the banks (filters filter, transformers transform)
    - Cannot run algorithms in parallel
    - On the other hand, this approach is convenient as long as you trust the mutation
    - Alternative: make every algorithm a creator algorithm, with immutable input banks
      - Our algorithms are so far very fast… likely better to run events in parallel (i.e., multithread upstream)

Action Functions

- Simple parameters and simple return values: numbers or lists of numbers

- There are 3 “ranks”
  - Scalar action functions
    - Inputs and outputs are single numbers
    - Act on values from individual bank rows
  - Vector action functions
    - Inputs and outputs are lists of numbers
    - Act on full bank columns
  - Hybrid: mix of scalar and vector parameters or outputs



- And there are 3 types, based on the algorithm type:

Algorithm Type	Action Function Output
Filter	Returns `bool` whether or not the filter passes
Transformer	Returns the transformed parameters
Creator	Returns a simple `struct` of parameters (corresponding to a created bank row)

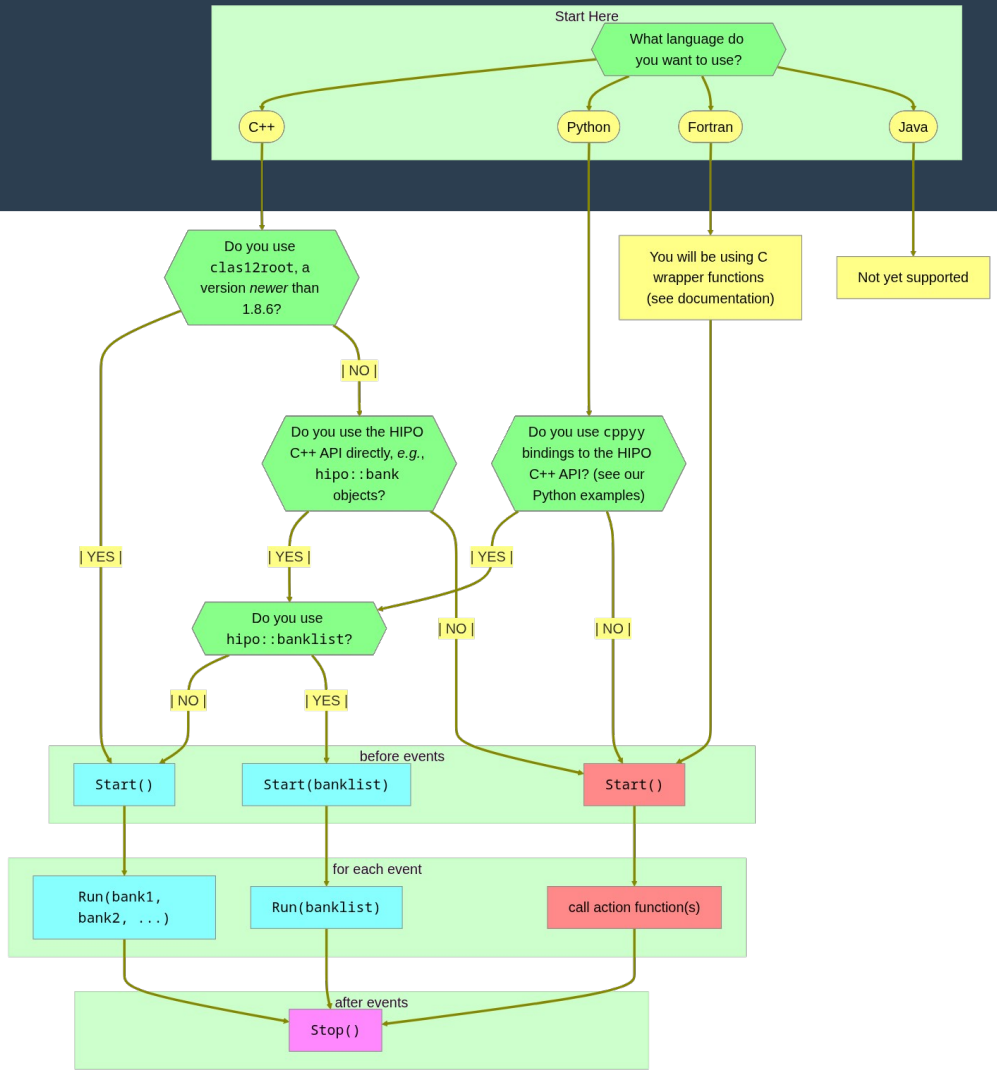
Action Functions (continued)

- Anyone can use them: if you have the numbers from a bank, you can call them
- Unique to each algorithm
  - Read its documentation
  - Some algorithms have multiple action functions which need to be called in a certain order
- As for Run functions, users must be mindful of the algorithm ordering
- Easy to bind to other languages (Java, Fortran, Python)
- **Cons:**
  - Some action functions have *many* parameters, and are therefore mildly uncomfortable to use
  - Some algorithms don’t yet have action functions

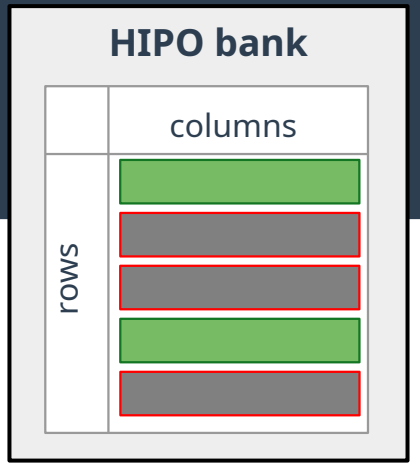
What should I use!?

Unfortunately, rather complicated to explain →

But only as complex as the initial choices of software for analysis…



Bank Filter Implementation



- Banks output from filter-type Run functions need special care by the user
- In the C++ library, hipo::bank objects contain a list of “active” rows
  - All of them are “active” by default
  - Certain rows can be “deactivated” by a filter

The user just needs to choose how they loop over rows:

For example, given a bank object named `particle_bank`:

```
for(auto const& row : particle_bank.getRowList()) {
    // loops over only the rows which pass the filter
}
```

```
for(int row = 0; row < particle_bank.getRows(); row++) {
    // loops over ALL rows, regardless of whether they pass the filter
}
```

- NOTE: Although this approach is “simple”…
  - It somewhat complicates framework integration (e.g., clas12root)
    - Other filter designs may be better or worse, but so far this one is sufficient…
  - The list of active rows is memory resident; we can’t “just write” a filtered bank to a file
    - We could only write the active rows, but need to update pindex values, for example

Algorithm Anatomy

```
src/iguana/algorithms/clas12/SectorFinder
├── Action.yaml
├── Algorithm.cc
├── Algorithm.h
├── Config.yaml
├── Validator.cc
└── Validator.h
```

- Each algorithm in its own source-code directory
- Algorithm implementation
  - **Algorithm.h:** has documentation (among other usual header stuff)
  - **Algorithm.cc:** has function implementations:

ConfigHook()	configuration parsing
StartHook(banklist)	additional startup stuff
RunHook(banklist)	calls Run(bank1, ...) functions
Run function(s)	
Action function(s)	
StopHook()	cleanup, etc.

- **Action.yaml:** describes action functions (for generating language bindings)
- **Config.yaml:** the default configuration file
- **Validator.h** and **Validator.cc:** validator algorithm implementation

Storing Trained Machine Learning Models

- **Every code.jlab.org (GitLab) repository can have a Model Registry**
  - For Iguana: https://code.jlab.org/hallb/clas12/iguana/-/ml/models

- **Provides a place to store trained ML models**
  - Version control of models
  - Some Iguana algorithms currently use pre-trained models
    - Mariana's Lepton ID and Gregory's Photon ID
  - These files are large relative to source code files
  - The Model Registry is a natural choice for hosting them
  - Models are automatically downloaded at Iguana build time

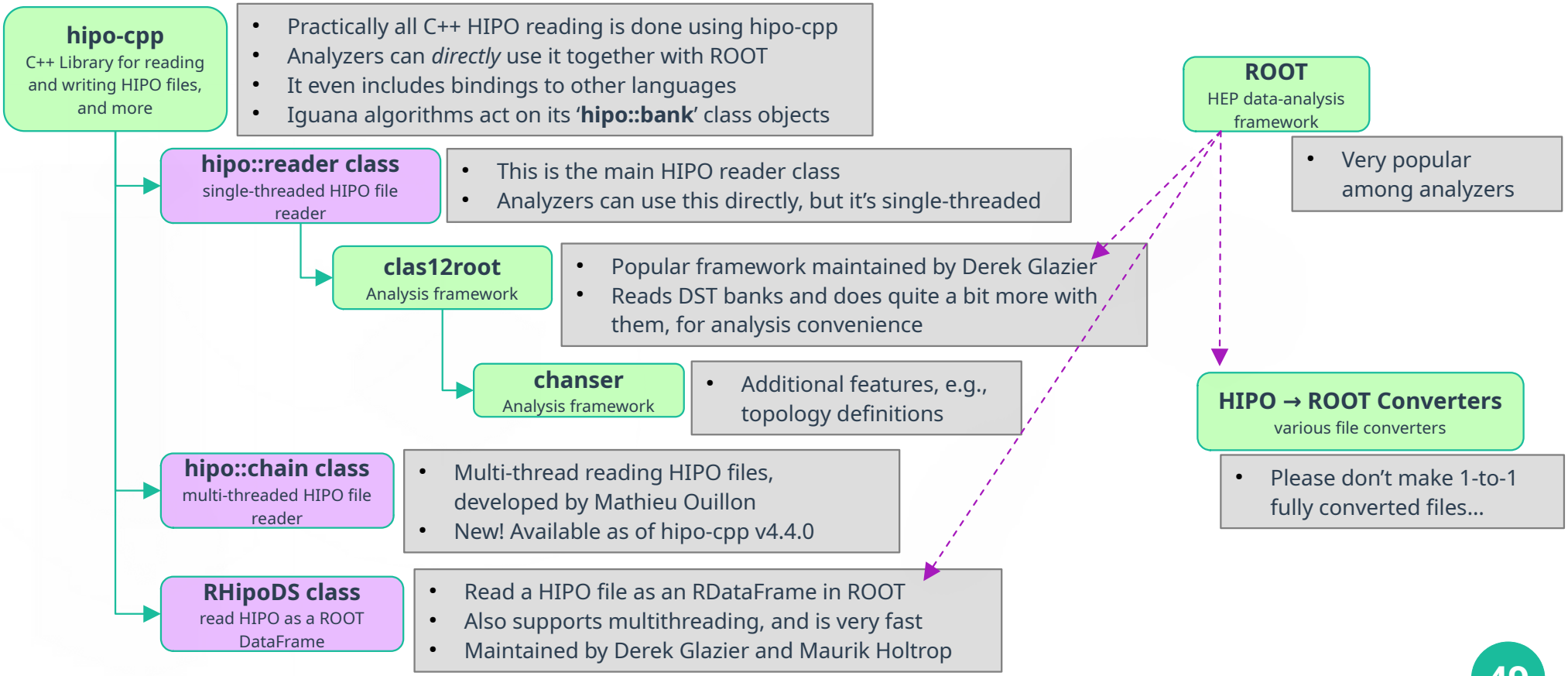
- **Integration with MLflow:** https://mlflow.org/
  - Experiments: run models and check metrics (training?)
  - Packaging for reproducibility and distribution
  - Serving, e.g. via a REST API
  - Compatible with frameworks such as TensorFlow and PyTorch

Software Support Status

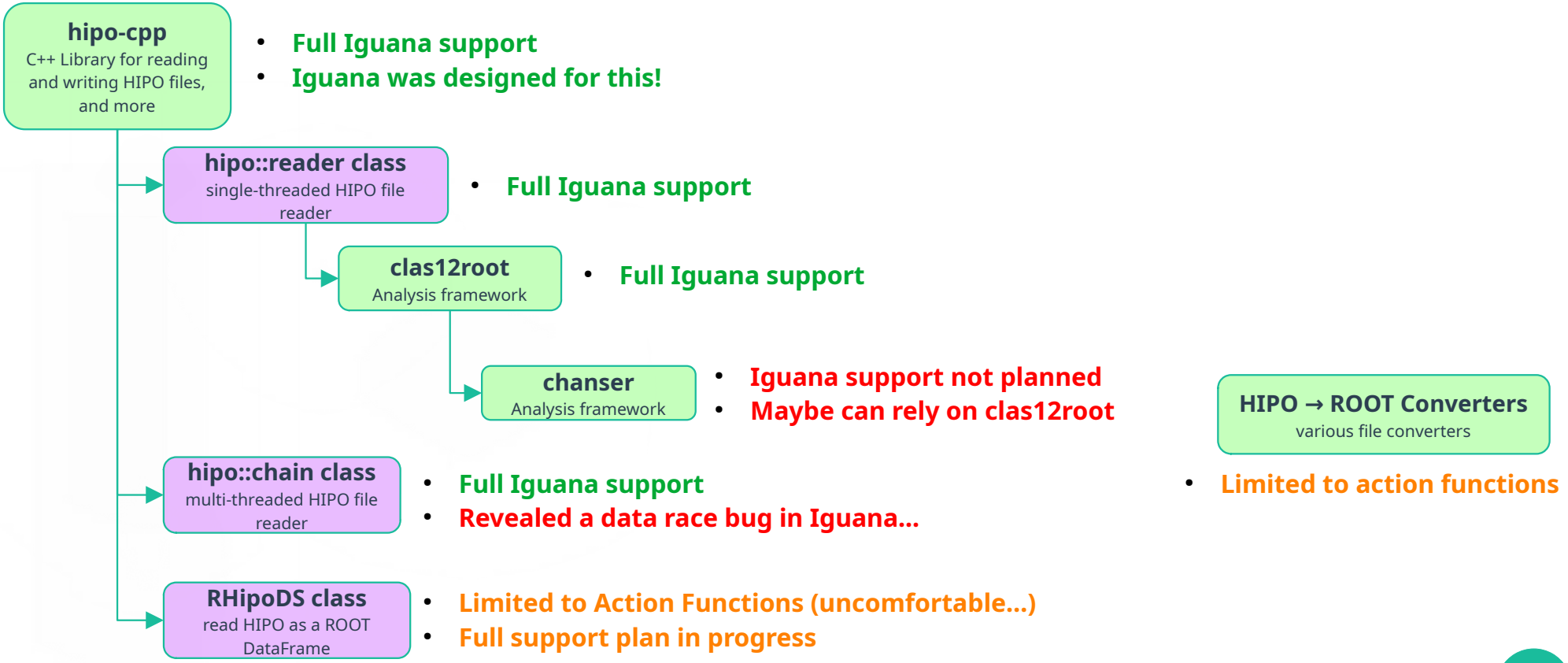
The following slides show tools for reading HIPO data, along with Iguana support status for each

Details are for reference

Analysis Software: C++



Analysis Software: C++



Analysis Software: Java

j4np, a.k.a. hipo-java
Java Library for reading and writing HIPO files, and more

- hipo-cpp is effectively a C++ port of this
- Closed source

coatjava
Reconstruction Framework and Data Model

- Uses jnp-hipo(4) to read and write HIPO files, and has wrappers for this
- Bank schema definitions (data model)
- Open source

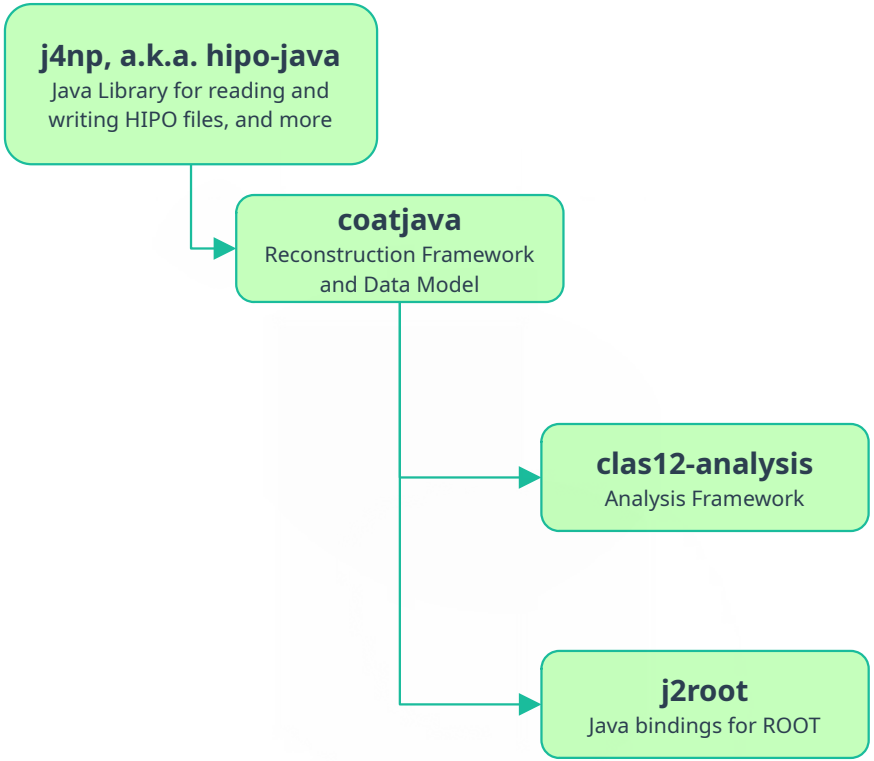
clas12-analysis
Analysis Framework

- No longer maintained? (last commit 5 years ago)
  - Depends on coatjava v6 (we are now on v13)
- 2023 Survey said 22% of respondents use it, but almost all of them also use ROOT or clas12root

j2root
Java bindings for ROOT

- Binds to a few ROOT classes: histograms, graphs, n-tuples, and trees
- Analyzers can read HIPO with Java, and write out custom ROOT files

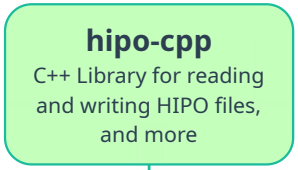
Analysis Software: Java



- **No Java Iguana support yet at all!**
  - We need to write C++ → Java bindings
- **Does anyone want it?**
  - All but 1 Java users also use C++/ROOT …

Aside: there may be some interest in calling certain COATJAVA engines from Iguana (Java → C++ bindings)

Analysis Software: Python



Python Bindings

- C++ → Python bindings via ctypes
- Part of hipo-cpp, in ‘extensions/python/’
- Uses ‘hipo::fusion’ (a ‘hipo::reader’ wrapper)

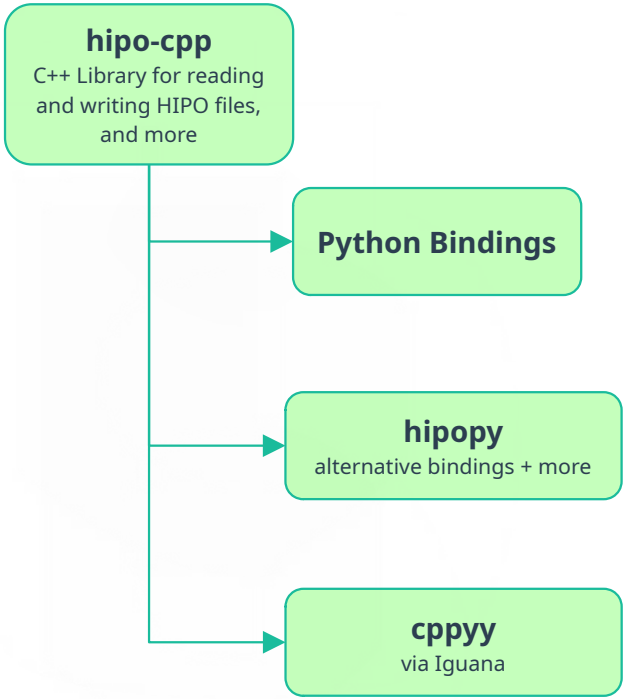
hipopy
alternative bindings + more

- Uproot-like I/O interface to HIPO files
- Written by Matthew McEneaney (but no longer maintained, since he’s moved to another position)
- C++ → Python bindings via pybind11 (packaged in hipopybind)
- Separate project that depends on hipo-cpp

cppyy
via Iguana

- cppyy generates automatic C++ ↔ Python bindings very conveniently
- ROOT’s PyROOT uses cppyy under the hood
- Iguana uses cppyy to generate its bindings, and for now it also uses cppyy to bind to hipo-cpp
- Has some issues, but is very powerful and has an optimistic future

Analysis Software: Python



- **Use Action functions**
- **Use Action functions**
- **Full Iguana support**

Using cppyy to generate C++ → Python bindings automatically

This is what ROOT uses for PyROOT

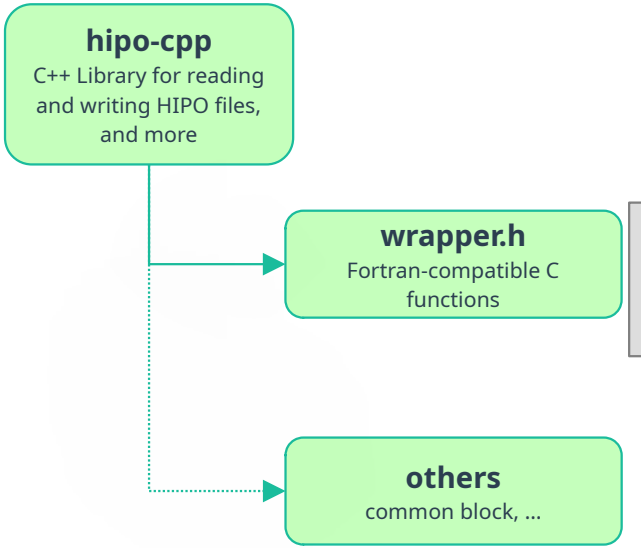
Sometimes a bit cumbersome or fragile, but *very* powerful

Optimistic future, with migration from CLING to CLANG

C++ interpreter , primarily for ROOT

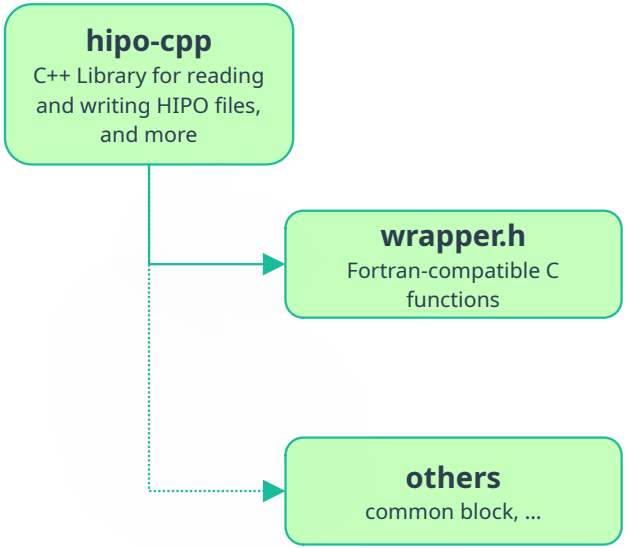
C++ compiler etc. , primarily for *everyone*

Analysis Software: Fortran



- Simple header file with C functions wrapping ‘hipo::reader’ functions
- Iguana uses the same technique for its Fortran bindings, via ISO_C_BINDING
- Used by Iguana Fortran examples

Analysis Software: Fortran



- **Use Action functions**
- **Use Action functions (not tested yet)**