A Repository for Common Analysis Criteria

Ideas for a Run Group A prototype

Christopher Dilks

RG-A Retreat

18 October 2023

Jefferson Lab

U.S. DEPARTMENT OF ENERGY | Office of Science

JSA

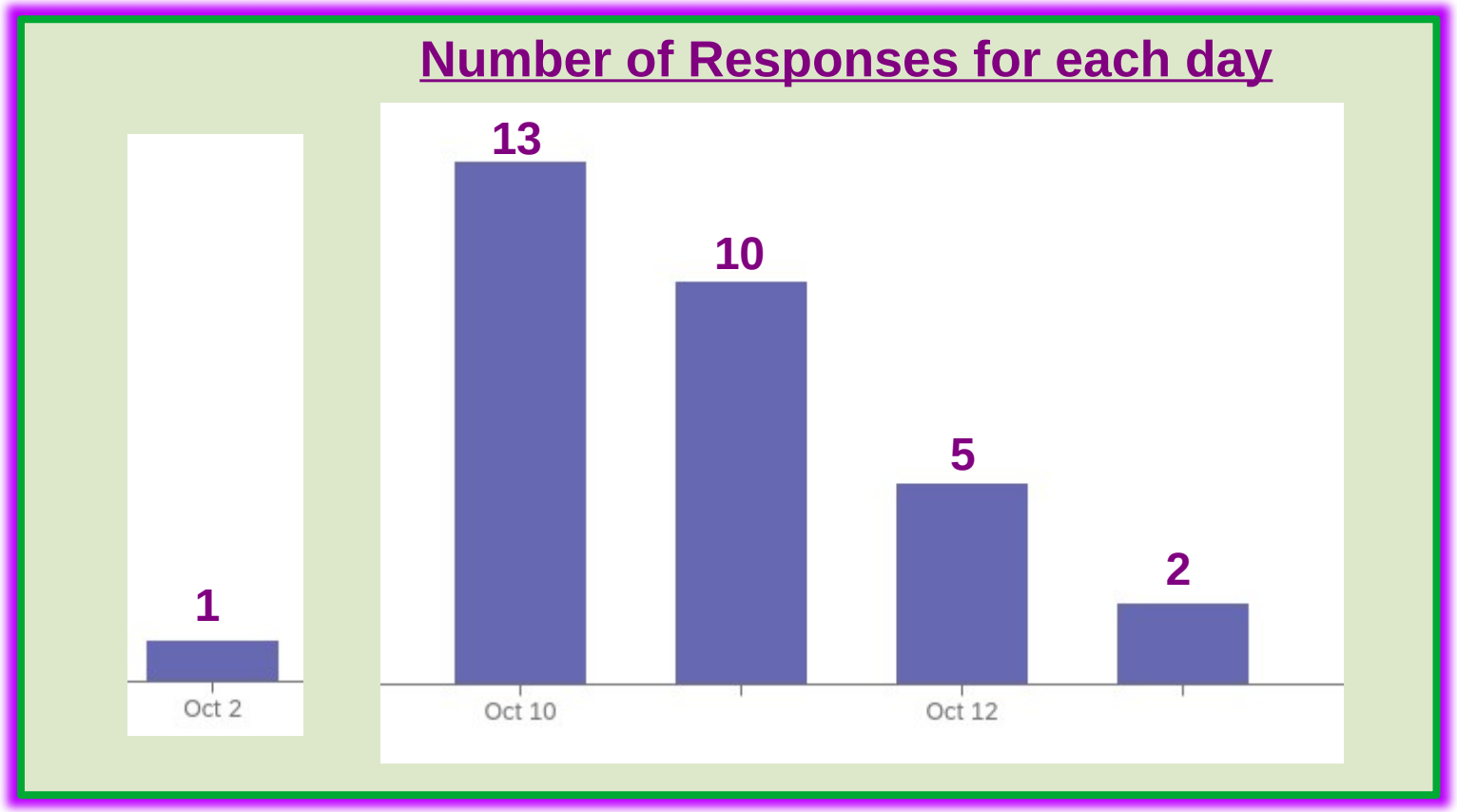
The Plan

- Implement a repository of common methods shared among physics analyses, such as fiducial cuts and enhanced PID criteria.
- This repository will aim to provide simple access to common techniques, and to preserve them under version control.
- User-centered design: the software survey

The Software Survey

- User experience is critical to designing such a repository
- Accumulating user feedback is the first step
- This was the primary goal of the recent software survey
- Additional questions of general interest to the CLAS Software Group were included

Survey Results



31 Responses
(as of Monday, 10/16)

Disclaimer:

- **Results in these slides might not include responses received after Monday, October 16th**
- **Later responses are certainly welcome and will be fully considered for the repository design and software group feedback.**
- **Focusing on the questions relevant for these slides**
- **Please take the survey if you haven't!**

Survey Results

What physics analyses are you working on?

- General responses shown
- Nothing surprising here…
- Broad range!

- (SI)DIS
- DVCS
- DVMP
- SRC
- (Spin) Structure
- KY
- Hyperons
- N*
- Exclusive
- J/psi
- MesonEx
- TCS
- Very Strange
- “None”

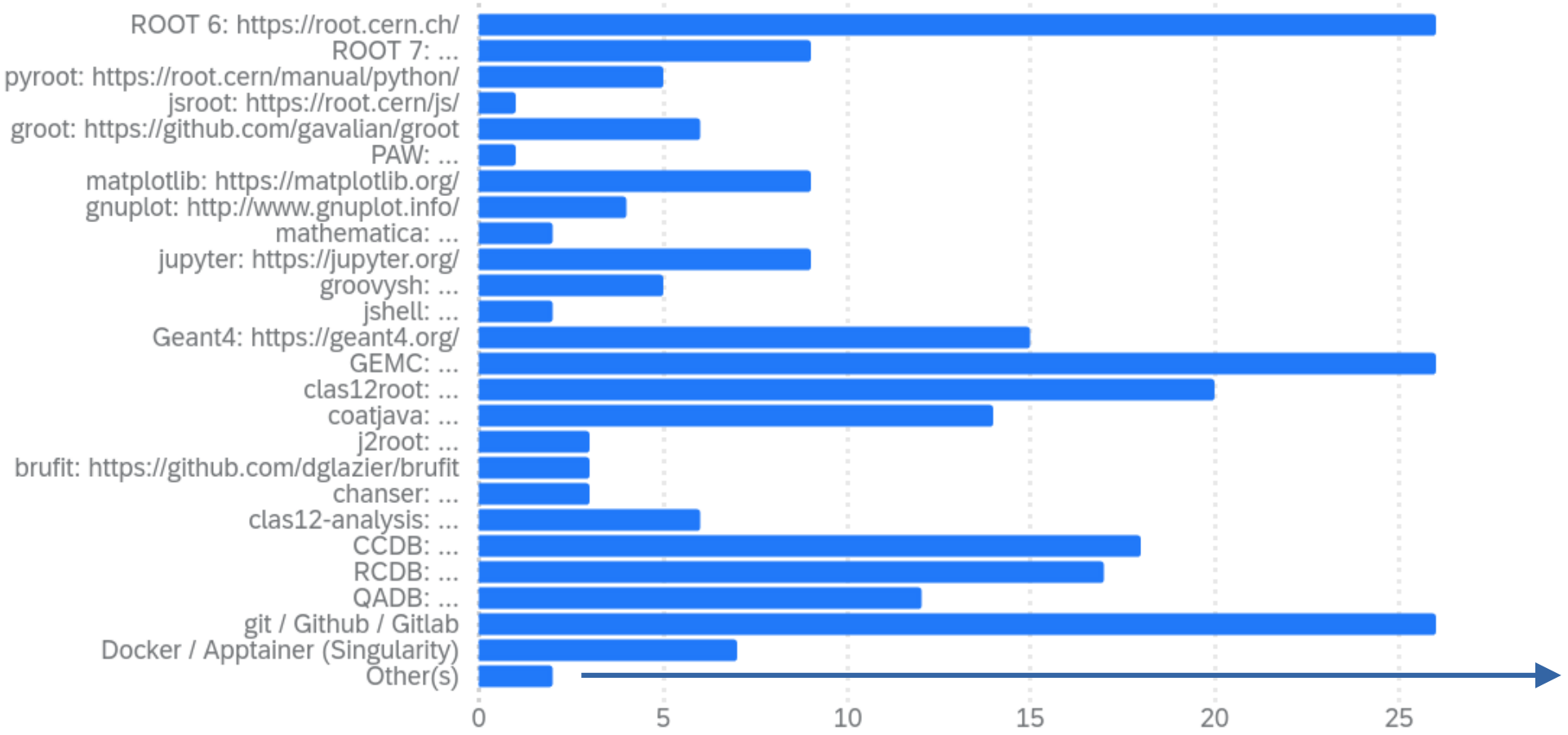
Survey Results

Are you working on anything else that uses CLAS12 software or data? If so, please list:

- COATJAVA
- Simulations
- Machine Learning
- Detectors
- Calibration
- Planning & Proposals
- **Analysis Criteria**
  - Momentum Corrections
  - PID refinements

Survey Results

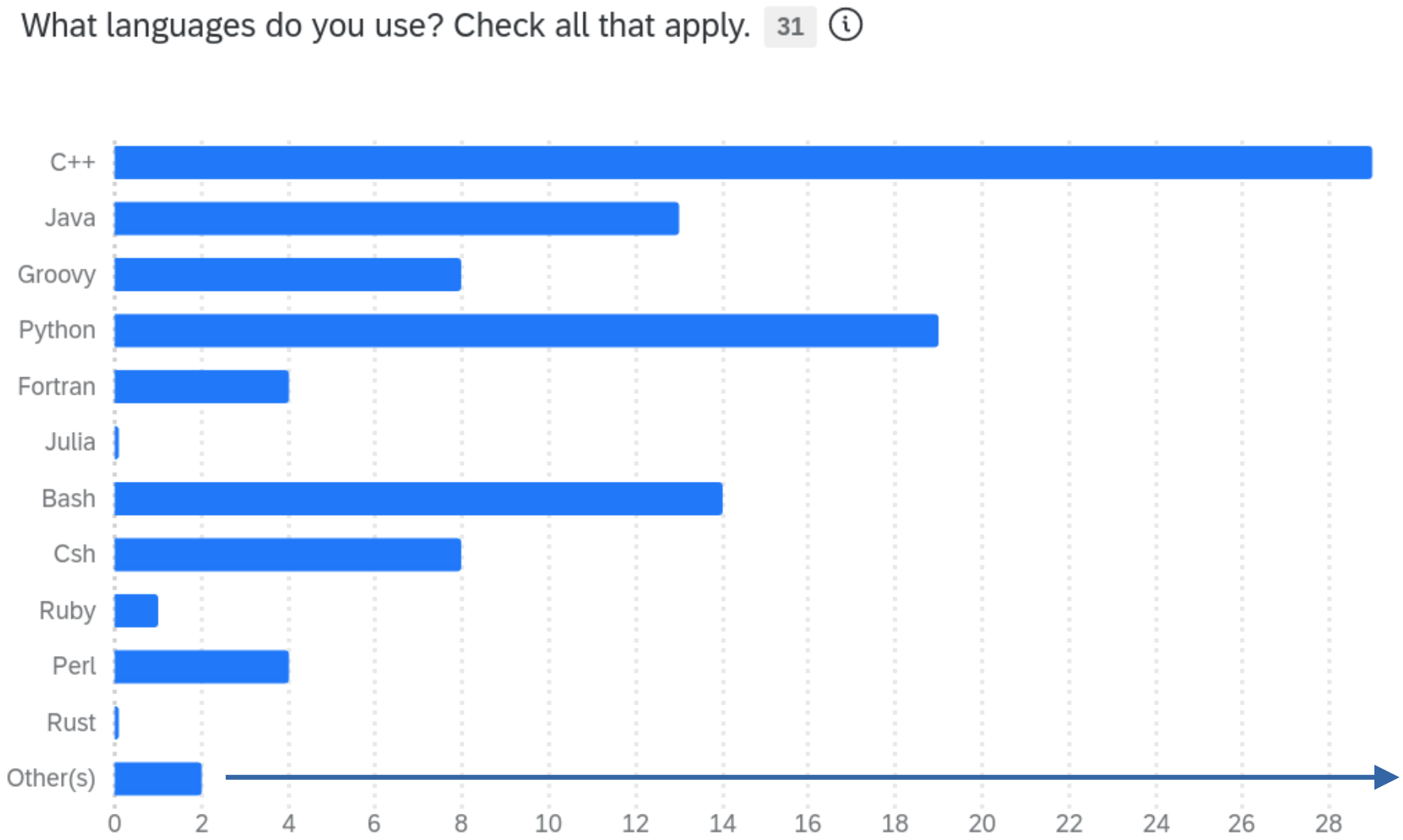
What tools do you use? Check all that apply. 31



Other(s):
- uproot
- pandas
- numpy

Jefferson Lab

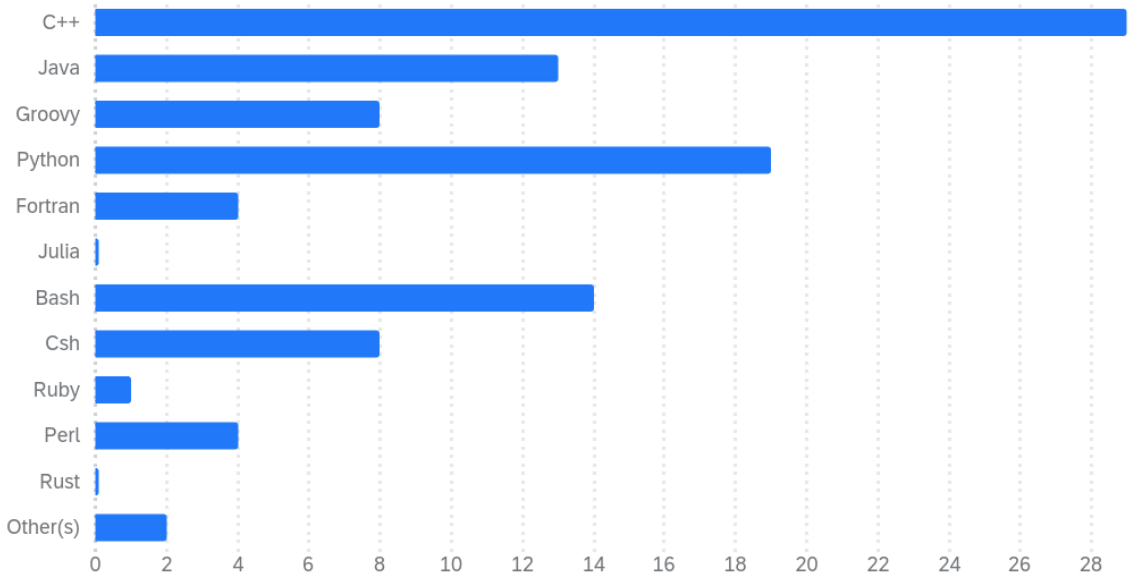
Survey Results



Other(s):
- zsh
- a note about relative usage

Jefferson Lab

About the Language…

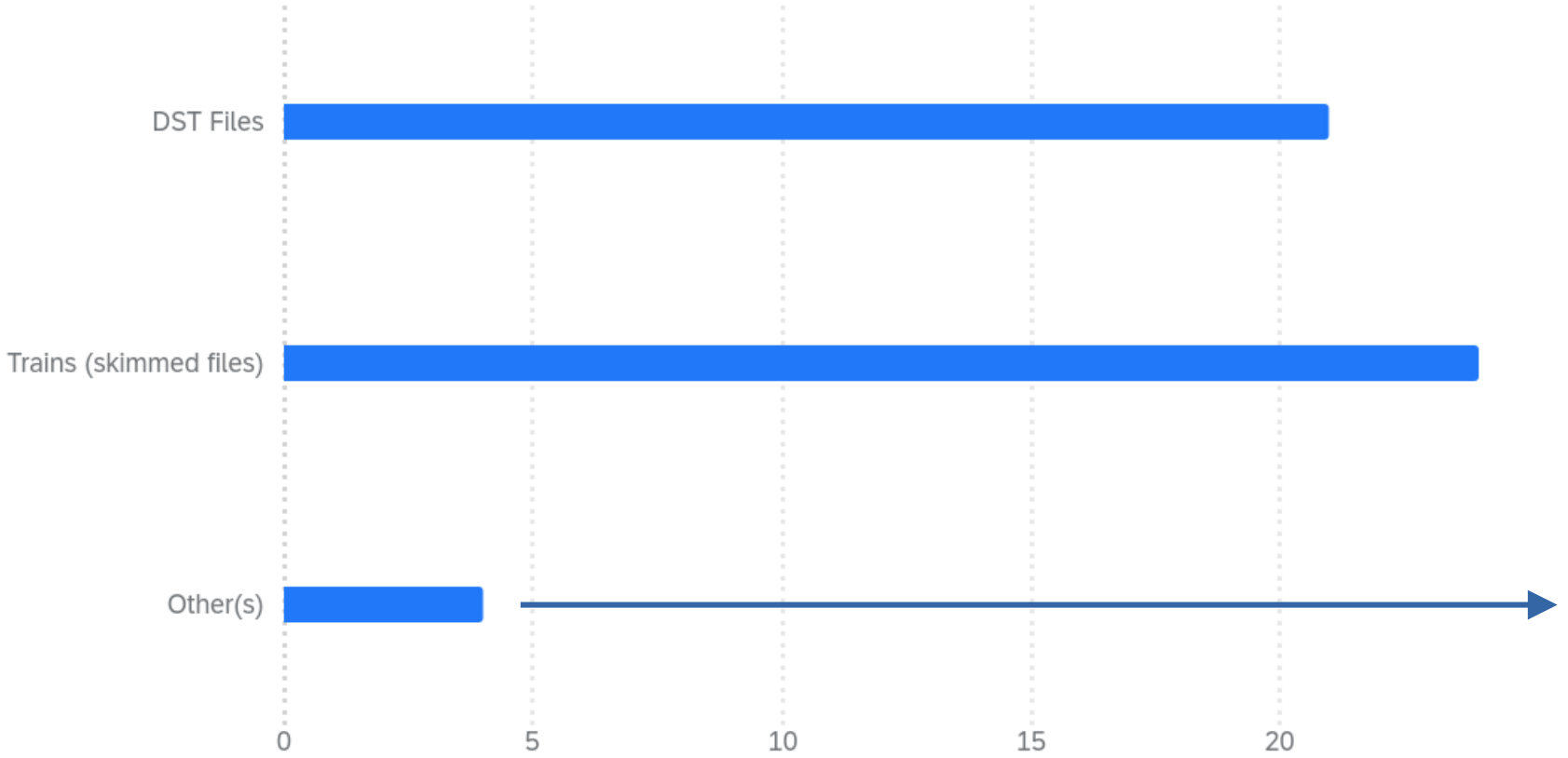


1st: C++
2nd: Python
3rd: Bash
4th: Java
5th: Groovy / Csh
6th: Fortran / Perl

- Most people who use Java/Groovy/Python also use C++
- But what are these being used for?
  - Very unlikely one does full analysis in Bash, but Bash is good for “glue”
  - Better question: What languages are the (HIPO) data processed in?
  - What language(s) are the existing common criteria in?

Survey Results

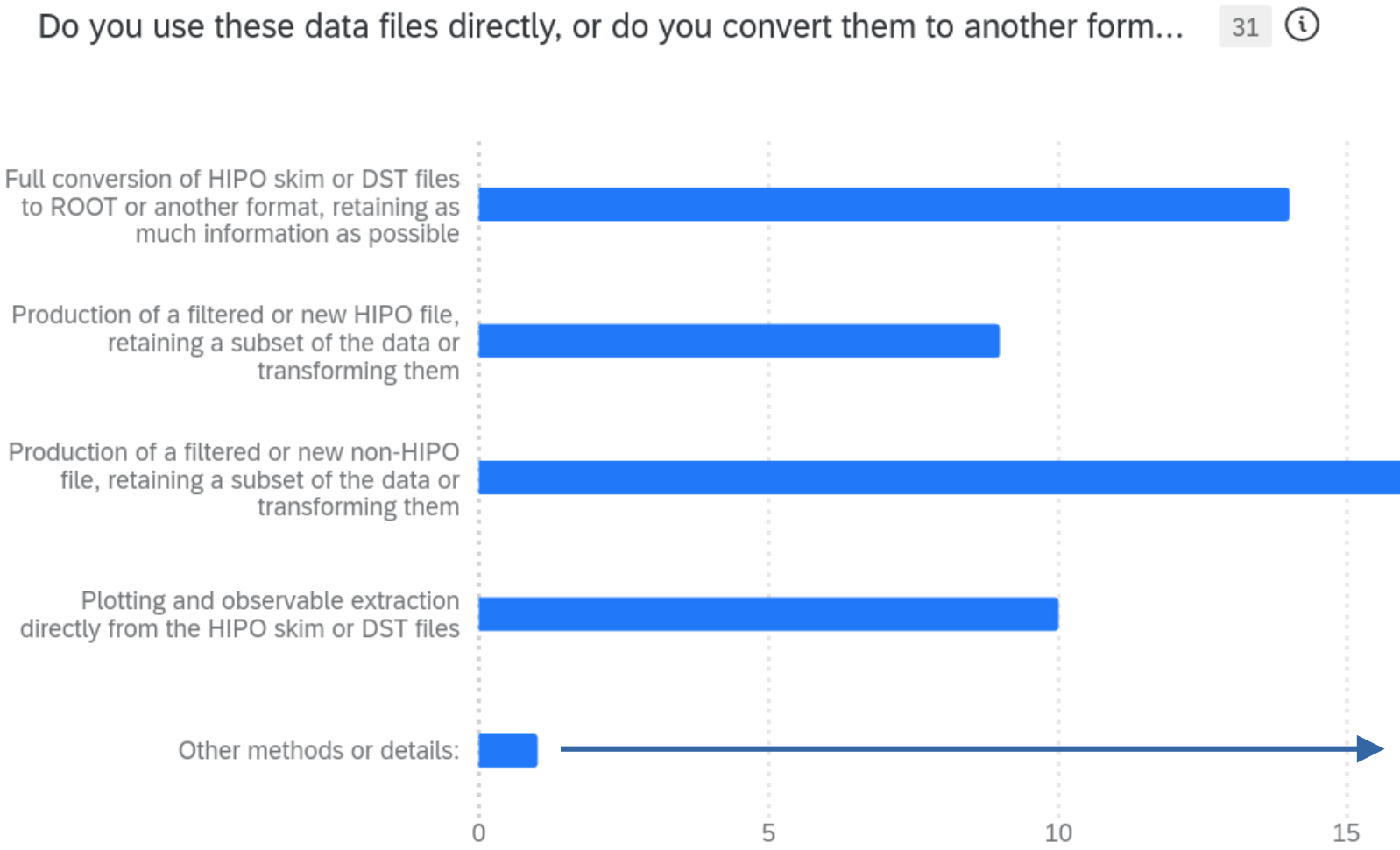
What type(s) of data files do you focus on? Check all that apply. 31



Other(s):
- PAW ntuples
- MC files
- Pre-DST banks
- EVIO

Jefferson Lab

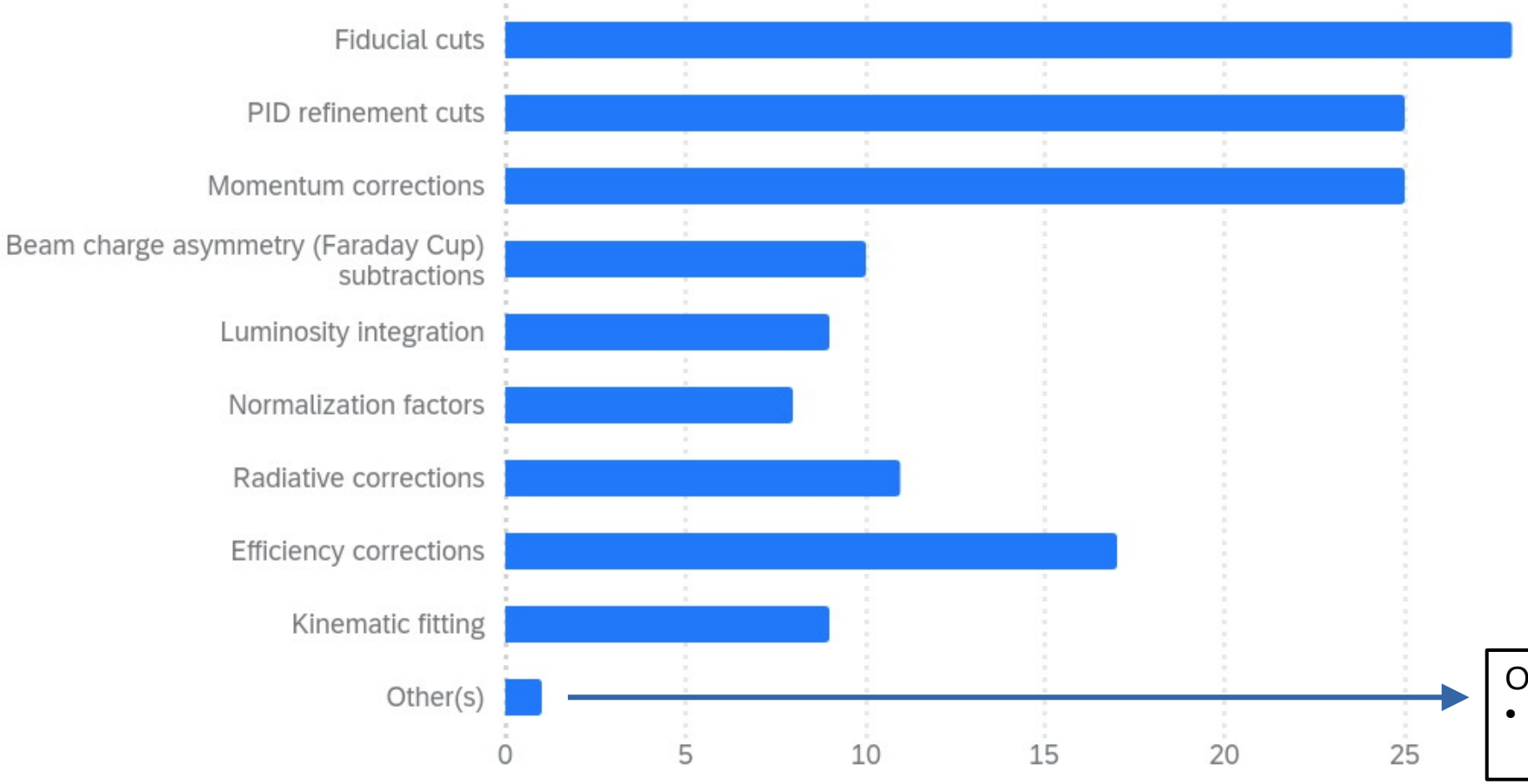
Survey Results



Other(s):
- Filter → event-by-event text files and ROOT trees with calculated kinematics

Survey Results

What methods do you use that would be considered commonly used? 30



Other(s):
- Forward Tagger corrections (part of momentum corrections?)

Jefferson Lab

Survey Results

Do you maintain any commonly used methods? If so, please describe, and include what language(s) are used and/or where the code can be found.

- RGM methods
- Ports of Fiducial cuts from C++ to:
  - Python
  - Groovy
  - Java?
- Common RGA methods in Chanser

- **Issue**: ports and code duplication
  - DRY: Don't Repeat Yourself!
  - If the C++ fiducial cuts are updated, who updates the ports?
  - Are the ports cross checked?
  - Automated testing?

- **Chanser**
  - Includes RGA common methods
    - Fiducial cuts
    - PID refinements
    - Vertex cuts
    - (maybe more)
  - Dependent on ROOT and clas12root (?)
  - C++
- Our goal for the common repository differs:
  - Primarily stay lightweight and as framework-independent as possible

Survey Results

What are your thoughts on the idea presented in the above mission statement? How would you like to interact with such a repository? What features would you like?

And some critical thoughts:

- Difficult to create one-size-fits-all methods
- Channel / observable / run period / analysis dependence is difficult
- Do not be opaque, black box
  - Stifle innovation
  - Does not educate students
  - May overlook a major issue in the code
- Do not force a framework, should be flexible
- Preference to do things themselves

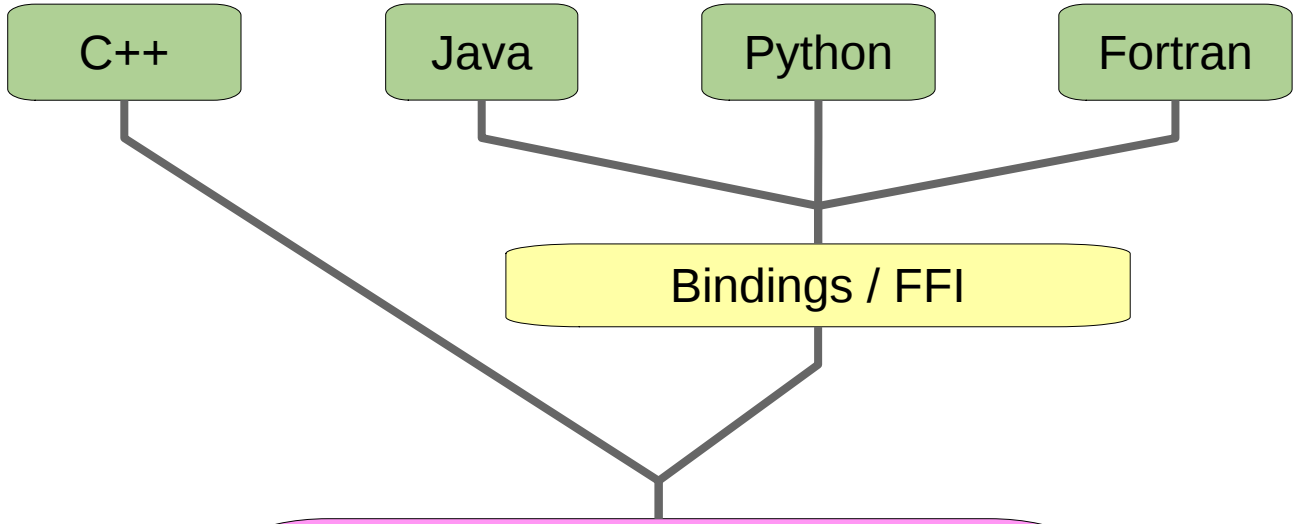
Mostly positive feedback!

- Analyses will need to test and adapt
- Website interaction desired
- Easier than searching through analysis notes
- One language
- Peer review
- Why not apply corrections during reconstruction?
- Compatibility with C++/ROOT/Chanser/etc.
- Run period dependence
- Ability to customize
- Executable on ifarm
- Up-to-date documentation
- Examples
- Easy for new users
- Kinematic calculations (e.g. particle → z, phi, etc.)
- Polarization from closest Moeller measurement
- C++ / Java / Python

Design

Dominant Language Model

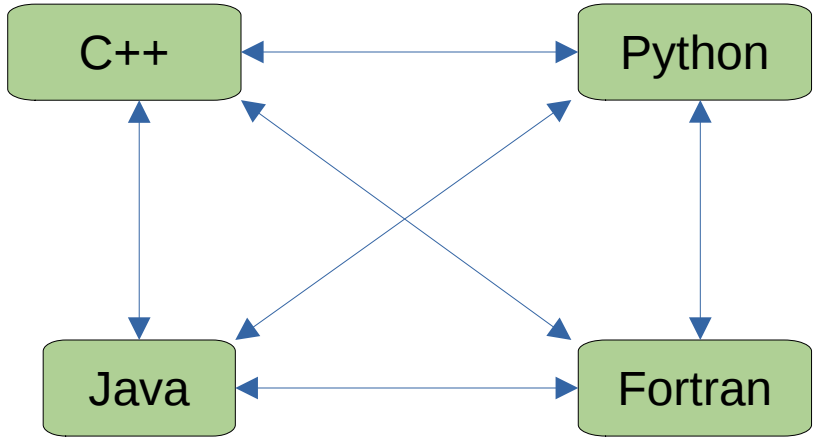
API Level:
For the Users



Criteria “Algorithm” level:
The Code

- Require all criteria (algorithms) to be in one “dominant” language, e.g., C++
- Consistent and maintainable
- If an algorithm is not in the dominant language, either:
  - Port it to the dominant language
  - Write a wrapper algorithm in the dominant language
- Use bindings / foreign function interfacing to expose API in other languages
  - SWIG
  - JNI
  - GraalVM
  - …

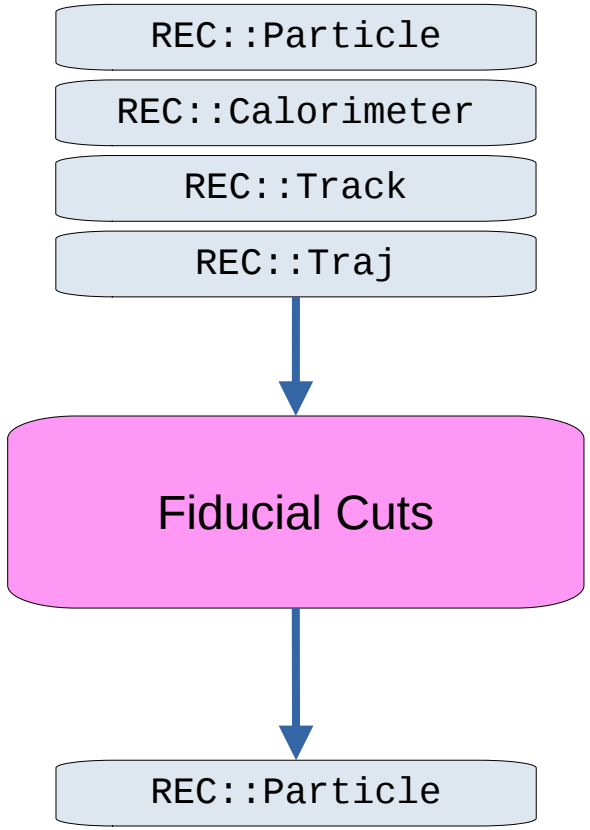
Free Model



- Allow algorithms to be in any language
  - No need to port or wrap any existing algorithms / criteria
- Need bidirectional bindings between all of them
  - 4 languages → 8 bindings
- Hard to implement
- Hard to maintain

Data Communication

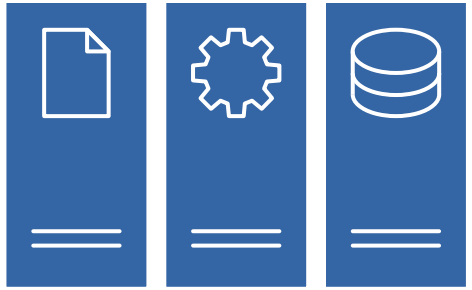
- Need a standard of communication of information
  - Users ↔ Algorithms
  - Algorithms ↔ Other Algorithms
  - Algorithm I/O should be banks
- HIPO data unit: HIPO bank
  - Implementations: preferred to be standalone
    - Java
    - C++ (?)
    - Python (?)
    - Fortran (?)
- Need bidirectional converters from the analysis “user” language to the dominant language (C++)
- Exploring ideas of “language independent banks”
  - JSON
  - Hopefully conversion is not slow...



- with only particles which pass the fiducial cuts
- maybe different name, but same structure

Services

- The algorithms will all have some basic common needs: “service singletons”
  - **Logging system**
    - Log-level control
    - Silence for production, verbose for debugging
    - Errors always print
  - **Unit system**
    - Define what is “1” in each system
    - For example, in Geant4: 1 = mm = MeV = ns
  - **Algorithm configuration**
    - For example: fiducial cut levels (loose, medium, tight)
    - Configuration file model
      - Default config file: the defaults for *all* algorithms
      - Handle run-period dependent configuration
      - Users may override any part (or all) of it with custom config files



Testing

- Needed to maintain stability
- Some Options for automated testing in Continuous Integration (CI):
  - Unit tests, requiring high coverage
  - clas12-validation: automated testing of full chain
    - event generation → simulation → reconstruction → analysis
    - no analysis step yet
    - https://github.com/JeffersonLab/clas12-validation
- Need also cross checks / peer review of algorithms

Documentation

- We have analysis notes
- The algorithm itself, although maybe hard to read, is effectively self-documenting
  - Comment your code!
  - Version control → algorithm is preserved
- Documentation of common repository usage is a separate issue
  - API documentation
  - Examples

Containerization

Multi-lingual support → difficult to setup (compile) for users!
Too many dependencies!

- Provide a Docker image with all dependencies + the common criteria repository, compiled and ready to use
  - Analysis code would run in containers, either locally or on clusters (*ifarm,* OSG)
- Customization:
  - Straightforward to replace software with no dependents
  - Replacing upstream software may require recompilation of dependent software
  - Adopt upstream package manager (e.g., Spack)
- Continuous Deployment: most recent version
  - Combined with a package manager makes replacing any piece of software an automated process
- Maintenance: everyone gets the same bugs

APPTAINER

docker

docker hub

Base image Layer

- Underlying Linux distribution
- Package updates
- Typical common software, e.g., vim, emacs
- Python, C++, Java, Groovy, Fortran

Maintained by JLab (?)

Common Physics Software Layer

- ROOT, PAW
- Geant4

Maintained by JLab (?)

CLAS Software Layer

- Clas12root
- Chanser
- Brufit
- … … …
- Common criteria repository

Maintained by CLAS

Pseudocode Prototyping

```cpp
// ANALYSIS PSEUDOCODE: C++ version
// -------------------------------

#include <CommonAnalysisCriteria.h>

// initialize
auto Criteria = CommonAnalysisCriteria();

// event loop pattern
for(event : events) {
  bankFiducialResult = Criteria.FiducialCuts(event.getBank("bank1"), event.getBank("bank2"));
}

// data frame pattern (assuming the elements of the frame are banks)
auto dataFrameFiducial = dataFrame.Define(
    "bankFiducialResult",
    [] (bank1, bank2) { return Criteria->FiducialCuts(bank1, bank2); },
    {"bank1", "bank2"}
    );
```

- Banks are in the analysis code's language
- CommonAnalysisCriteria is
  - In C++: the main class
  - In Python: the main class, wrapping the C++ algorithms (needs some thought how to design…)

```python
# ANALYSIS CODE: python version
# -----------------------------

import CommonAnalysisCriteria

# initialize
criteria = CommonAnalysisCriteria.CommonAnalysisCriteria()

# event loop pattern
for event in events:
    bankFiducialResult = criteria.FiducialCuts(event.getBank("bank1"), event.getBank("bank2"))
```

Pseudocode Prototyping

```
// API CODE: C++ version
// ---------------------
outputBankType FiducialCuts(inputBank1Type bank1, inputBankType bank2) {
  // convert input C++ banks to JSON
  auto json1 = BankToJSON(bank1);
  auto json2 = BankToJSON(bank2);
  // call algorithm
  auto jsonOut = FiducialCutsAlgorithm->Process(json1, json2);
  // convert output back to a C++ bank
  return JSONToBank(jsonOut);
}
```

- The API code will handle the conversion from the analysis code banks to language-independent banks, and call the appropriate underlying algorithm
- These API methods could be auto-generated
- Assumes JSON is the “language independent bank” (needs some thought and testing)

```
# API CODE: python version
# ------------------------
def FiducialCuts(bank1, bank2):
    # convert input python banks to JSON
    json1 = BankToJSON(bank1)
    json2 = BankToJSON(bank2)
    # call algorithm; e.g., a SWIG wrapper of the underlying C++ algorithm
    jsonOut = FidicualCutsAlgorithm.Process(json1, json2)
    # convert output back to a python bank
    return JSONtoBank(jsonOutput)
```

Pseudocode Prototyping

```
// THE ALGORITHM
// -------------
class FiducialCutsAlgorithm {
  public:

    FiducialCutsAlgorithm() { /* initialize services */ }

    // run before any events
    void Init(std::string configFile="") {
      // configuration (if specified an override)
      // initialize anything that needs it
    }

    // run on every event
    outputJsonType Process(inputJson1Type json1, inputJson2Type json2) {
      // the fiducial cuts algorithm
    }

    // run at the end of all events
    void End() {
      // cleanup
    }
}
```

- The algorithm itself follows the typical 3-methods pattern:
  - Init
  - Process
  - End
- A main CommonAnalysisCriteria can handle
  - Service initialization
  - Algorithm configuration
  - Cleanup at the end

Aside: Helicity Sign

```
// helFlip: if true, REC::Event.helicity has opposite sign from reality
def helFlip = false
if(RG=="RGA") helFlip = true
else if(RG=="RGB") {
  helFlip = true
  if(runnum>=11093 && runnum<=11283) helFlip = false // fall, 10.4 GeV period only
  else if(runnum>=11323 && runnum<=11571) helFlip = false // winter
};
else if(RG=="RGK") helFlip = false
```

```
int HelicityConvention() {
  return BSAWrong ? -1 : 1
}
```

- For pass 1 QA, it was requested by the run groups to correct the helicity so that the π+ beam spin asymmetry (BSA) timeline has the correct sign
- This will *no longer be done* for future QA timelines
  - The QA is for *finding* issues, not *fixing* them
- Instead: QA defect bit assigned for wrong BSA from REC::Event.helicity
  - Every RGA run will have this defect bit
  - No runs are “golden” (perfect), but they are still “OkForAsymmetry” and “OkForCrossSection” (new cut, to be implemented)
  - Add to QADB: “HelicityConvention()”: if this BSA is wrong → user must flip helicity sign
  - Automatically catches HWP issues

Aside: Other Run-Dependent and Run-Period-Dependent Values

- Beam (or target) Polarization and Error – TODO
- Trigger conditions – in RCDB (?)
- Faraday Cup Charge – in HIPO files, and in QADB (for QA-filtered charge)
- The *correct* beam energy (RCDB may not be “correct”) – proposed for CCDB
  - Under discussion in SW group

- Run-dependent values should go in RCDB or CCDB (e.g., beam polarization)
- Finer bins (e.g., time bins or DST files) can go in QADB (e.g., charge)
  - Pass 2 QA will be done in time bins, whereas Pass 1 was done by DST 5-files

- Common software repository could serve info from these databases, but…
  - This is a bit out of scope, since these databases already have APIs
  - Wrapping APIs with more APIs adds unnecessary complexity (unless the underlying API is user unfriendly…)
  - However, some algorithms may need to read the databases
  - If desired, we can implement it (would just be sugar for the underlying DB API)

Outlook and Plans

- Focus prototype design on:
  - Run Group A
  - Fiducial Cuts
  - PID Refinements

I have these in C++, but may be out of date, or even wrong (though they have been cross checked); they are also in Chanser

- Need maintainers of common methods
  - …Eventually… after the design and prototyping phase
- Anyone want to help test and design?
  - Service work opportunity?