Components
Why? How?

Nils Krumnack (Iowa State University)

Introduction

- already heard a bit about components yesterday
  - mostly a high level overview

- trying to give you a different approach to the topic
  - incorporate/use of components in analysis code
  - much smaller scale than full athena
  - still useful in such a much smaller context

- partly meant as introduction to common CP algorithms
  - explain why they look the way they look

- for your own code you don't have to go that complicated
  - just writing a single user algorithm is perfectly fine
  - mostly stick to code you can understand yourself

Basic Selection

- let's start with a simple example: select some good muons
  - do it in a simple way first
  - evolve on the next couple of slides
- all pseudo-code, won't compile 😜
  - also not checking status codes…

```
for (xAOD::Muon *muon : *muons) {
  ... // apply corrections to muon
  if (muon->pt() > 25 && fabs (muon->eta()) < 2.5 && ... ) {
     ... // use muon for something
  }
}
```

Basic Selection

- let's start with a simple example: select some good muons
  - do it in a simple way first
  - evolve on the next couple of slides
- all pseudo-code, won't compile 😜
  - also not checki

```
for (xAOD::Mu
  ... // apply cor
  if (muon->pt(
    ... // use mu
  }
}
```

problems:

- probably need muon definition in multiple places, shouldn't hard-code it everywhere
- shouldn't have lengthy code blocks inside single function
- probably copy-and-pasted definition from some CP group web page

Use Functions

- better to put corrections/selections into their own functions
  - easier to reuse (and easier to test)
  - main code a lot easier to read
  - easier to provide corrections/selections externally
  - easier to update muon definitions
- functions contain muon definition code from before

```
for (xAOD::Muon *muon : *muons) {
  correctMuon (*muon);
  if (isTightMuon (*muon)) {
     ... // use muon for something
  } else if (isLooseMuon (*muon)) {
     ... // use muon for something
  }
}
```

Use Functions

- better to put corrections/selections into their own functions
  - easier to reuse (and easier to test)
  - main code a lot easier to read
  - easier to provide corrections/selections externally
  - easier to update muon definitions
- functions contain

```
for (xAOD::Muo
  correctMuon (
  if (isTightMuo
    ... // use muo
  } else if (isLoos
    ... // use muo
  }
}
```

problems:

- might need to hold calibration constants for corrections
- no (easy) way to configure corrections and selections
- little reuse between tight and loose selection

Use Objects/Classes

- introduce objects/classes to handle corrections/selections
  - each handles one specific task (usually)
    - wrap/contain individual functions from before
  - object can hold calibration data if needed

```
for (xAOD::Muon *muon : *muons) {
  m_muonCorrection->applyCorrection (*muon);
  if (m_tightMuonSelector->accept (*muon)) {
     ... // use tight muon for something
  } else if (m_looseMuonSelector->accept (*muon)) {
     ... // use loose muon for something
  }
}
```

Using Tools

- haven't said what these objects are
  - didn't need to know to write this code
  - probably take them straight from the CP groups anyways

- perfect use case for tools in Athena/AnalysisBase
  - can be combined at compilation/run-time
  - CP groups can provide them preconfigured
  - can change configuration without changing C++ code

- most tools provide an interface class
  - only contains the methods you need
  - doesn't contain any "actual" code
  - decouples your code from tool code
- allows swapping tools easily
  - tight/loose can use same selector tool with different settings
  - tight/loose can use two different classes, e.g. cut-based and TMVA

Tool Configuration/Use

- define ToolHandle in algorithm class:

```
ToolHandle<IMyTool> m_tool {"MyTool/name", this};
```

- declare ToolHandle as property in algorithm constructor:

```
declareProperty ("tool", m_tool, "tool for doing xyz");
```

- retrieve tool in initialize (optional, but recommended):

```
ANA_CHECK (m_tool.retrieve());
```

- call tool in execute():

```
m_tool->doSomething (...);
```

- then in your python configuration:

```
addPrivateTool (alg, "tool", "MyTool")
alg.tool.property = value
```

Tool Configuration/Use

- define ToolHandle in algorithm class:

```
ToolHandle<IMyTool> m_tool {"MyTool/name", this};
```

- declare ToolHandle as property in algorithm constructor:

```
declareProperty ("tool", m_tool, "tool for doing xyz");
```

- retrie itiali mmended):

```
AN (m_
```

interface class

implementation class

- call tool in execute():

```
m_tool->doSomething (...);
```

- then in your python configuration:

```
addPrivateTool (alg, "tool", "MyTool")
alg.tool.property = value
```

Further Improvements

```
for (xAOD::Muon *muon : *muons) {
  m_muonCorrection->applyCorrection (*muon);
  if (m_tightMuonSelector->accept (*muon)) {
     ... // use tight muon for something
  } else if (m_looseMuonSelector->accept (*muon)) {
     ... // use loose muon for something
  }
}
```

- not ideal to use accept decision right in place
  - mixes your code with the selection code
  - may want to use selection decision in multiple places
  - may want to run corrections+selection in separate job
- can't (easily) do nested loops over all muon pairs
  - inner loop has to recheck selection over and over
- better: store selections for each muon

Selection Decorations

```
for (xAOD::Muon *muon : *muons) {
  m_muonCorrection->applyCorrection (*muon);
  muon->auxdata<char>("tight") =
    m_tightMuonSelector->accept (*muon);
  muon->auxdata<char>("loose") =
    m_looseMuonSelector->accept (*muon);

  if (muon->auxdata<char>("tight") {
    ... // use tight muon for something
  } else if (muon->auxdata<char>("loose") {
    ... // use loose muon for something
  }
}
```

- actually no need for a single loop over muons anymore
- can have separate loop for each task

Multiple Loops

```
for (xAOD::Muon *muon : *muons) {
  m_muonCorrection->applyCorrection (*muon);
 muon->auxdata<char>("tight") =
    m_tightMuonSelector->accept (*muon);
 muon->auxdata<char>("loose") =
    m_looseMuonSelector->accept (*muon);
}

for (xAOD::Muon *muon : *muons) {
   ... // do something with the muons
}
```

- complete separation between user code and CP code
- can split code into multiple algorithms
- only need access to the same muon container

Algorithms & Whiteboard

- can put muon corrections & selections into separate algorithms
  - only need to share `muons` variable
  - shared via whiteboard/event store (a.k.a. `evtStore()`)

- whiteboard variables take role of local variables in single algorithm
  - only thing algorithms need to know about each other

- what do we gain:
  - further reduced coupling between user code and CP tools
  - easier to add extra CP tools/algorithms to job
    - e.g. add more working points, add more object types
    - algorithms all look the same (unlike tools)
    - mere configuration change, no C++ needed
  - can run CP tools/algorithms in separate job from user algorithm
    - just pick objects up via `evtStore()` either way

Common CP Algorithms

Nils Krumnack (Iowa State University)

Introduction

- CP algorithms are part of the effort to harmonize user analysis
  - make analysis results more consistent
  - make it easier to reproduce/exchange physics configuration
  - increase code sharing for analysis code/frameworks

- should work standalone or with any analysis framework
  - will/may require some changes to framework/user code
  - will/may obsolete the simpler frameworks

- CP algorithms currently being rolled out
  - most of the infrastructure/algorithms available in release
  - had a few users successfully cross check with their framework
    - i.e. physics should be correct (or close to)
  - not supported by analysis frameworks yet
  - ultimately meant to be used by everyone

CP Algorithm Workflow

- no longer calling CP tools directly
  - instead add CP algorithms to your job
    - get run before your algorithm gets run
  - algorithm code & base configuration provided by CP group
    - distributed as part of the analysis release
  - any extra options set directly on CP algorithm by user

- directly retrieve corrected objects from event store
  - as if they directly come from your input file
  - extra information (e.g. weights) get added as decorations
  - systematic variations become multiple (shallow) copies

- can also write out mini-xAOD from CP algorithm output
  - and run your algorithm on that output file in a separate job
    - without needing to change your algorithm at all
  - also provide an algorithm to write out the mini-xAODs

Analysis Frameworks

- common CP algorithms replace a core part of any framework
  - e.g. for QuickAna that's pretty much all it does…
  - may be an opportunity for some frameworks to retire

- could incorporate CP algorithms in several ways, but:
  - they only communicate via the event store
  - the entire sequence only uses what's in the input file
- can just run all CP algorithms before the framework code
  - will look as if the corrected objects come from the input file
  - could indeed run (some) CP algorithms as separate job
  - separates framework code from CP recommendations
- note: can even run completely standalone (mostly for AnalysisTop)

- event store layout may require framework changes:
  - object/property names may change
  - how object/event properties are stored may change

Object Selection

- multiple aspects to object selection:
  - selection algorithms add selection decorations to each object
  - can specify object preselection for each algorithm
  - can create view containers containing only good objects
  - can combine multiple selections into single decoration

- can add your own object preselection to sequence as well
  - just add one/two more algorithms to sequence
  - more efficient, don't process objects you will cut later on
  - no example/instructions for doing this yet

- can add decorations for multiple working points to same object
  - e.g. to select loose-but-not-tight objects
- can produce object level cut flows as diagnostic
  - uses full selection bitmap produced by each selection tool

Writing Output Files

- will normally want to write output to mini-xAOD/n-tuple
  - CP algorithms (normally) slowest part of analysis
  - don't want to re-run them every time you change a histogram
  - your analysis framework may do this for you

- your choice whether you use mini-xAOD or n-tuple
  - fairly detailed discussion on pros and cons in the tutorial pages

- running on mini-xAODs may require (re)creating view containers
  - depends on details of the mini-xAOD

- facilities for writing output files not (quite) complete
  - have some basic mini-xAOD mechanisms so far
  - have done some larger scale testing for the n-tuple version

- also looking into including CP algorithms in DAODs (long term)

Python Configuration

- configuration is crucial when using the common CP algorithms
  - most of CP recommendation set via configuration
  - systematics handling relies on proper configuration

- configuration is done in python:
  - makes conditional configuration easier
    - i.e. can have if statements in configuration
  - integrates better with Athena

- best to treat these like other configuration files
  - i.e. like a series of "name = value" pairs
  - plus if statements as needed
- don't put complex logic into python configuration file
  - if needed, separate it from configuration values
  - (normally) don't read configuration values from other files

CP Base Configuration

- CP groups provide & maintain standard configuration files:
  - distributed as part of the analysis release
  - meant to improve physics harmonization across groups
  - should not require extra configuration by users
  - configuration files are meant to be readable by beginners

- users pick and choose the CP configuration they want
  - meant to be straightforward to do for beginners
  - tradeoff between brevity, clarity and flexibility
  - chose python to mesh better with other ATLAS configuration
- can override almost any aspect of tool/algorithm configuration
  - mostly to allow special studies

- can also maintain your own copy of central configuration file
  - allows to freeze/modify configuration at will
  - not yet sure if this should be a normal thing to do…

User Configuration

- user creates an algorithm sequence for each object type
  - e.g. electrons, muons, jets…
  - pre-configured for selected working point
  - user can override tool properties as needed
  - user can add/remove algorithms as needed
    - e.g. add object pre-selection to beginning of sequence
  - apply sequence post-configuration before adding to job
  - provide some debugging histograms (at least for now)

- MET & OR need to combine multiple object types
  - makes for more complex configuration
  - see the examples for details

- may want to merge sequences for multiple working points
  - e.g. to study loose-but-not-tight leptons
  - configure multiple sequences and combine them

Where To Find CP Algorithms

- (almost) all CP algorithms located at: https://gitlab.cern.ch/atlas/athena/tree/21.2/PhysicsAnalysis/Algorithms
- have one folder per object type:
  - AsgAnalysisAlgorithms
  - EgammaAnalysisAlgorithms
  - FTagAnalysisAlgorithms
  - JetAnalysisAlgorithms
  - …

- look much like the package you created in the beginners tutorial
- have a lot of very small algorithms in sequence
  - makes each one very simple in design
  - makes configuration more flexible (plus possibly more tedious)
  - apart from systematics handles much like your algorithm from the beginners tutorial

CP Algorithm Packages

- in each directory two sub-folders for configuration:
  - python: python configuration of algorithm sequence
  - share: example job options/macros/tests that you can run as is
- this configures one algorithm from the muon sequence:

```
alg = createAlgorithm( 'CP::AsgSelectionAlg',
                       'MuonKinSelectionAlg' + postfix )
addPrivateTool( alg, 'selectionTool',
                     'CP::AsgPtEtaSelectionTool' )
alg.selectionTool.minPt = 20e3
alg.selectionTool.maxEta = 2.4
selectList.append ('kin_select' + postfix + ',as_bits')
nCutsList.append (2)
alg.selectionDecoration = selectList[-1]
seq.append( alg, inputPropName = 'particles',
                 outputPropName = 'particlesOut',
                 stageName = 'selection' )
```

CP Algorithm Packages

- in each directory two sub-folders for configuration:
  - python: python configuration of algorithm sequence
  - share: example job that you can run as is
- this configures one algorithm sequence:

create algorithm

```
alg = createAlgorithm( 'CP::AsgSelectionAlg',
                       'MuonKinSelectionAlg' + postfix )
addPrivateTool( alg, 'selectionTool',
                     'CP::AsgPtEtaSelectionTool' )
alg.selectionTool.minPt = 20e3
alg.selectionTool.maxEta = 2.4
selectList.append ('kin_select' + postfix + ',as_bits')
nCutsList.append (2)
alg.selectionDecoration = selectList[-1]
seq.append( alg, inputPropName = 'particles',
                 outputPropName = 'particlesOut',
                 stageName = 'selection' )
```

CP Algorithm Packages

- in each directory two sub-folders for configuration:
  - python: python configuration of algorithm sequence
  - share: example job options/macros/tests that you can run as is
- this configures one algorithm from the muon sequence:

```
alg = createAlgorit                    ectionAlg',
                                       electionAlg' + postfix )
addPrivateTool( alg, 'selectionTool',
                     'CP::AsgPtEtaSelectionTool' )
alg.selectionTool.minPt = 20e3
alg.selectionTool.maxEta = 2.4
selectList.append ('kin_select' + postfix + ',as_bits')
nCutsList.append (2)
alg.selectionDecoration = selectList[-1]
seq.append( alg, inputPropName = 'particles',
                 outputPropName = 'particlesOut',
                 stageName = 'selection' )
```

configure tool

CP Algorithm Packages

- in each directory two sub-folders for configuration:
  - python: python configuration of algorithm sequence
  - share: example job options/macros/tests that you can run as is
- this configures one algorithm from the muon sequence:

```
alg = createAlgorithm( 'CP::AsgSelectionAlg',
                       'MuonKinSelectionAlg' + postfix )
addPrivateTool( alg, 'selectionTool',

alg.s
alg.s
selectList.append ('kin_select' + postfix + ',as_bits')
nCutsList.append (2)
alg.selectionDecoration = selectList[-1]
seq.append( alg, inputPropName = 'particles',
                 outputPropName = 'particlesOut',
                 stageName = 'selection' )
```

configure selection decoration
(remember name for later algorithms)

CP Algorithm Packages

- in each directory two sub-folders for configuration:
  - python: python configuration of algorithm sequence
  - share: example job options/macros/tests that you can run as is
- this configures one algorithm from the muon sequence:

```
alg = createAlgorithm( 'CP::AsgSelectionAlg',
                       'MuonKinSelectionAlg' + postfix )
addPrivateTool( alg, 'selectionTool',
                     'CP::AsgPtEtaSelectionTool' )
alg.selectionTool.minPt = 20e3
alg.selectionTool.maxEta = 2.4
sele                                                    )
nCu
alg.
seq.append( alg, inputPropName = 'particles',
                 outputPropName = 'particlesOut',
                 stageName = 'selection' )
```

add algorithm to sequence
(needs extra info for post-configuration)

About The Tutorial

- tutorial only uses muons
  - you are invited to try/add other object types as well
  - different object types should work very similarly
  - MET & OR more advanced, need at least jets and muons

- start with Athena job options/EventLoop run script from release
  - provided for all the object types
  - primarily used for testing, but also as an example

- also add user algorithm for looking at the output objects
  - give you some idea on how to access CP algorithm output
  - some introduction to using data handles for systematics
  - running in same job for simplicity

- feel free to use either EventLoop or Athena path
- tutorial still fairly new, may have some problems

CP Algorithms Summary

- common CP algorithms next logical harmonization step:
  - one abstraction level above CP tools
  - independent, self-contained units of code
  - completely separates CP code from analysis frameworks
  - avoids fair amount of code duplication

- CP algorithms (mostly) ready for use
  - CP algorithms themselves basically all there
  - early physics validation promising
- if you start a new analysis, consider using CP algorithms

- some things still missing:
  - physics validation not complete (you can help!!)
  - still working on some advanced aspects of configuration
  - still working on mini-xAOD output (n-tuple output mostly done)
  - not supported by any framework yet

Systematics Handling

Systematics Evaluation

- systematics are handled by introducing nuisance parameters into the likelihood of observing x:

$$L_{full}(x, \tilde{\boldsymbol{\theta}}|\mu, \boldsymbol{\theta}) = L_{phys}(x|\mu, \boldsymbol{\theta}) \prod_i L_{subs}(\tilde{\theta}_i|\theta_i)$$

- normally L_{phys} is calculated by comparing x to the sum of template histograms at the given point in parameter space
  - a.k.a. looking at a stack plot

- template histograms are sampled from full simulation at a couple of points in parameter space
- for all other parameter values the template histograms are calculated using interpolation algorithms

- adjusting the histograms to points in parameter space happens (mostly) by reconfiguring CP tools accordingly

Systematics Workflow A

- assume you already have an analysis without systematics
- step 1: make a list of all the systematics to incorporate
  - mostly dependent on CP tools used
    - though some tools have alternate sets of systematics
    - and some tools may recommend using only a subset
    - some analyzer choice involved here
  - systematics aware frameworks can collect this for the user
- step 2: make a list of nuisance parameter points to evaluate
  - traditionally just +/- 1 sigma
  - can do more for run 2:
    - e.g. vary by +/- 5 sigma or several systematics at once
  - basic tools for this part of PATInterfaces package
  - this quickly can get non-trivial
    - mostly if you want to pick a minimal set of NP points
    - stat. forum may provide a tool some day (volunteer needed)

Systematics Workflow B

- step 3 (slow): make n-tuples for different nuisance parameters
  - run analysis code repeatedly, at different points in NP space
  - can either produce several n-tuples, or a merged n-tuple
    - xAOD shallow copies can make merged n-tuples efficient
  - expect this to be done mostly on the grid
  - slowest step, but should not be ridiculously slow
- step 4: make histograms at each nuisance parameter point
  - process each n-tuple from step 3 with the analysis code
  - can add some NP points here:
    - can make analysis a lot more efficient to run
    - but: requires using CP tools & xAODs in this step
  - expect this to be done mostly on local resources

- how to split work and store data is analysis dependent:
  - this is the core of your analysis, so you need to like it
  - analysis needs vary, you may even choose more/fewer steps

Systematics Workflow C

- step 5: combine the histograms into a RooStats workspace
  - not necessarily RooStats, can be your favorite format
  - in extreme cases could have 100 samples and 100 NP points
    - i.e. could be 10000(!) histograms per channel
  - you are mostly on your own here
    - may provide a tool at some point
- step 6: use RooStats workspace to produce plots, limits, measurements, etc.
  - not necessarily using RooStats, e.g. BAT supports RooStats workspaces
  - really can be your favorite stat. tool
- step 7 (optional): update the list of nuisance parameter points and repeat from step 2 (or from step 1)
  - may find that original selection of systematics wasn't good
  - or may find to need a different sampling of NP space
  - rather expensive, don't want to do it frequently

Systematics Interface

- all relevant CP tools implement ISystematicsTool interface
  - allows frameworks/tools to handle most of this for you, as long as you register all the tools properly
  - also makes it easier to use manually

- tools can be queried for their systematics:
  - recommended vs. affecting: not all systematics are recommended
  - some tools have multiple systematics sets
    - selected via configuration (e.g. precision vs. minimal)

- have a sub-section on systematics in AnalysisBase tutorial:
  - you should do that after adding some CP tools
    - need something to apply systematics to
  - should give you a fair idea of the (low-level) how-to
  - actual use can get quite complicated
  - with common CP algorithms not quite as necessary anymore…

Systematics & CP Algorithms

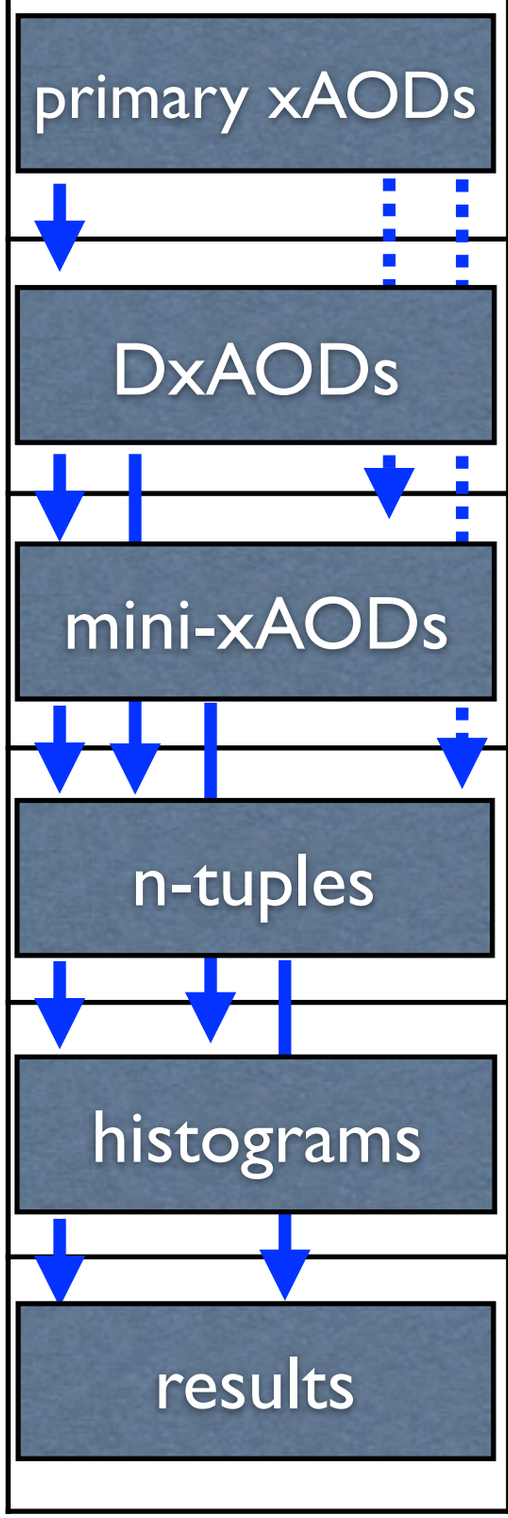
- systematics handling is very complicated under the hood
  - at least if you try to optimize it to reduce CPU usage
  - only giving the high-level view here

- have a special algorithm that loads the list of systematics
  - run in the beginning, other algorithms use its information
- put N copies of each container in event store, one per systematic
  - optimize away copies that would be identical
- algorithms that add new systematics make new copies
  - e.g. SF-algs add SF-systematics, that the correction-algs don't use
  - fair number of temporary copies in event store
  - (normally) only care about latest iteration of copies

- algorithms use data handles to access the event store
  - takes care of all the systematics handling under the hood
  - (typically) configured using special configuration helpers

Analysis Workflows

Run 2 Workflow (typical)

<table>
<tr><td rowspan="5">  </td><td>produced centrally, stored on grid</td><td>single definition</td><td rowspan="3">xAOD Format</td><td>no corrections</td></tr>
<tr><td>defined by physics group</td><td>only AODFix</td></tr>
<tr><td rowspan="2">produced by group/analyzer, stored locally</td><td rowspan="2">defined by group/analyzer</td><td>some/all corrections</td></tr>
<tr><td>TTree</td><td rowspan="3">all corrections</td></tr>
<tr><td>produced by analyzer, stored locally or personal machine</td><td>very analysis dependent</td><td>(mostly) TH1</td></tr>
<tr><td></td><td></td><td>analysis dependent</td></tr>
</table>

Run 2 Workflow (typical)

primary xAODs	produced centrally, stored on grid	single definition	xAOD Format	no corrections
DxAODs		defined by		only AODFix
mini-xAODs	produ group/a stored			some/all corrections
n-tuples				
histograms	produced by analyzer, stored locally or personal machine	very analysis dependent	(mostly) TH1	all corrections
results			analysis dependent	

Your analysis may deviate from this!

This merely reflects how most people do things…

Run 2 Workflow (typical)

primary xAODs	produced centrally, stored on grid	single definition	xAOD Format	no corrections
DxAODs		defined by physics group		only AODFix
mini-xAODs				some/all corrections
n-tuples			TTree	all corrections
histograms	produced by analyzer, stored locally or personal machine	very analysis dependent	(mostly) TH1	
results			analysis dependent	

CP algorithms run on this (mostly)

How Long Should It Take?

- mostly using multi-stage approach for performance
  - later stages (generally) run faster
  - later stages run far more often
- sometimes other reasons
  - use different frameworks for different stages
  - later stages run on smaller inputs → can often run locally

- DAODs take a couple of weeks to make
  - limited to about 1% of primary AOD size (ideally smaller)
- mini-xAODs (from DAODs) should be a few days (at most)
  - (with systematics) not always smaller than DAODs
  - often limited by grid turnaround time
- histogram making should be (at most) a few hours
  - mostly a productivity concern
  - that's the turnaround time for testing new ideas

Performance Improvements

- keep your input files small
  - apply tightest cuts you can when making input files
  - remove all objects you don't use
  - don't store variables you don't need
  - store actual variables of interest, not (multiple) input variables
- don't do things you don't need to
  - turn off code for studies you are not currently doing
  - don't run systematics if you don't use them
- tradeoff: faster later stage vs. rerunning early stages less often

- keep histogram files to tens of MB (at most)
  - often dominated by number of histogram bins
    - especially multi-dimensional histograms
  - #bins < #events: otherwise an n-tuple is faster/smaller
- a flat n-tuple with only ints/floats (no arrays) is very small/fast
  - also: very easy to make plots from

Backup Slides

Data Handles

- can use data handles to access whiteboard
  - replaces use of `evtStore()`
  - provide property to configure name in whiteboard
- removes some boiler plate code for properties

- AthenaMT requires use of data handles
  - allows tracking which algorithm depends on which
  - helps with scheduling algorithms in parallel
- standard data handles not available in 21.2

- common CP algorithms (see later) use systematics data handles
  - incorporate all the logic for systematics handling

CP Algorithms: How?

- know we can built an analysis frameworks around algorithms
  - very close to how some frameworks already work internally
  - personal experience from QuickAna design

- QuickAna algorithms typically simple to write:
  - normally wrap around a single tool (good level of granularity)
  - a couple of lines of algorithmic code
    - mostly a copy-paste of the CP group recipes
  - reasonable for a CP group to maintain an algorithm like this
    - plus whatever configuration is needed

- some open issues still to address:
  - algorithm configuration (dual-use, powerful, simple)
  - systematics handling (generic, efficient, simple)
  - selection mechanics (efficient, generic)
- some ideas/prototypes, but not all worked out yet

Job Configuration

- not quite settled yet on user job configuration
  - one option: full blown athena-style configuration
- don't have any idea/plan myself, mostly collecting requirements

- CP groups provide the full base configuration
  - either for algorithms or algorithm sequences
  - should reproduce recommendations out of the box
  - multiple/parametric configurations for multiple working points
- users pick the CP base configurations they need
- users override individual algorithm and tool properties
  - mostly needed for test runs
  - also needed to configure names in event store, etc.
- configuration will depend on dataset meta-data
  - at the very least data vs. mc vs. fast-sim
- beginners should be able to read, write & understand job config
  - i.e. have a simple, standard format for it

AnaAlgorithm

- recently introduced new class AnaAlgorithm
  - yet another Algorithm base class
  - will (hopefully) replace current EventLoop algorithms
  - design similar to AsgTool design

- project originally intended to provide common algorithms for
  - EventLoop
  - Athena
  - QuickAna

- some extra features (in AnalysisBase anyways):
  - (relatively) easy unit testing
  - lightweight, decoupled from framework
  - can be run without a framework
  - avoid various issues of EventLoop algorithms
  - no code changes for Athena (fully dual-use)

AnaAlgorithm

- recently introduced new class AnaAlgorithm
  - yet another Algorithm base class
  - will (hopefully) replace current EventLoop algorithms
  - design similar to AsgTool design

- project originally intended to provide common algorithms for
  - EventLoop
  - Athena
  - ~~QuickAna~~

- some extra features (in
  - (relatively) easy unit t
  - lightweight, decouple
  - can be run without a
  - avoid various issues of EventLoop algorithms
  - no code changes for Athena (fully dual-use)

can't do both (seamlessly)

could wrap QuickAna-like algorithms inside athena, but that creates issues as well

Writing AnaAlgorithms I

- if you work in Athena switching to AnaAlgorithm is easy:
  - mostly use a different base class for your algorithm
  - can't use any Athena services (not dual-use)
    - can still access event store and create histograms
  - need to create a dictionary for use in AnalysisBase

- not required to be dual-use, but easy if you can accept the limits
  - however in Athena, if you don't intend to make it dual-use then there is no advantage to AnaAlgorithm

Writing AnaAlgorithms II

- for EventLoop users switching is (somewhat) more involved:
  - signature for all member functions (slightly) changed
    - matches Athena now
  - configuration changed completely:
    - using declareProperty() as in Athena
  - no more streaming algorithms
    - old algorithms got created on the submission node and streamed to the worker node (also for local operation)
    - required users to design algorithms for root streaming
    - largest single source of user EventLoop problems
  - can't access wk() in your algorithm (not dual-use)
    - can access event store via evtStore()
    - can create/access histograms via book(), hist()
  - need to create a component library for use in Athena

- not required to be dual-use, but easy if you can accept the limits

Algorithm Configuration

- for dual-use algorithms need dual-use configuration:
  - otherwise welcome to cross-validation hell…
  - ask SUSYTools, they had that problem early on
- for EventLoop-only configuration, there are other options
  - probably won't advertise those too much though…

- constraints for dual-use configuration come from Athena:
  - must work nicely with its python configuration
- Attila made a very nice prototype for use with EventLoop:
  - introduces python configurables akin to Athena configurables
  - syntax very similar to Athena syntax
    - should be possible to have true dual-use configuration files
  - basic syntax straightforward enough for novice users
  - no support for (sub)tool configuration yet
  - no(?) support for stand-alone use yet
  - not tested on complex cases yet (e.g. overlap removal)

Systematics Handles I

```
class MyElectronAlg final : public EL::AnaAlgorithm {
  asg::AnaToolHandle<...> m_tool;



  // ...

  StatusCode initialize () {
    // tool initialization...

    return StatusCode::SUCCESS;
  }

  StatusCode execute () {

     xAOD::ElectronContainer *electrons = nullptr;
     ANA_CHECK (evtStore().retrieve (electrons, …));


     for (xAOD::Electron *electron : *electrons)
       ANA_CHECK (m_tool->apply (*electron));

  }
};
```

Systematics Handles II

```
class MyElectronAlg final : public EL::AnaAlgorithm {
  asg::AnaToolHandle<...> m_tool;
  EL::SysCopyHandle<xAOD::ElectronContainer> m_electronHandle {
    this, "electrons", "", "name of the electron container to use"};
  EL::SysVectorProperty m_systematics {this};

  // ...

  StatusCode initialize () {
    // tool initialization...
    m_systematics.addInputHandle (m_electronHandle);
    m_systematics.addAffectingSystematics (m_tool->affectingSystematics());
    ANA_CHECK (m_systematics.initialize());
    return StatusCode::SUCCESS;
  }

  StatusCode execute () {
    for (auto& sys : m_systematics.systematicsVector()) {
      xAOD::ElectronContainer *electrons = nullptr;
      ANA_CHECK (m_electronHandle.getCopy (electrons, sys));

      ANA_CHECK (m_tool->applySystematicVariation (sys));
      for (xAOD::Electron *electron : *electrons)
        ANA_CHECK (m_tool->apply (*electron));
    }
  }
};
```

algorithm granularity

Why aim for one tool per algorithm?

- grouping tools with different systematics requires multiple systematics loops, tricky in practice
- may want to run some algorithms multiple times, e.g. rerun selection for multiple working points
- may want to skip some algorithms, e.g. skip scale factors for data
- may want to insert extra algorithms between CP algorithm
  - particularly algorithms to apply cuts and skip further algorithms
  - also, may want to manipulate objects for studies, etc.
- may want to split algorithms between grid and local processing, e.g. corrections on the grid and scale factors locally
- different tools may be responsibility of different groups
- design-wise (arguably) preferable to split an algorithm consisting of multiple independent and consecutive sections into separate algorithms