



... for a brighter future

An AOD analysis example



Esteban Fullana Torregrosa
High Energy Physics Division
Argonne National Laboratory



U.S. Department
of Energy

UChicago ►
Argonne_{LLC}



A U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC

Introduction

■ What I am going to do

- This is short of continuing with the starting with Athena talk
- What I'll do now is to go into the details of the implementation, *i.e.* have a look at the *entrails of the code*
- I will propose several exercises (with the solutions) for you to get familiar with Athena
- I will be here to help you and answer any questions/problems you may have

Outline

- General comments about AODs
 - What we can do with them and what we cannot do with them
- The **Plain_Analysis** package
 - How it is organized and other important things
- The **DragonflyAlg** algorithm
 - A brief explanation of what it does and how it does it
- Proposed exercises
 - With some extra material to solve them

General comments about AODs

AODs in a nutshell

- An AOD is an object in *evolution*. The amount and organization of the information is *Athena-release-dependent*.
- However, the AOD was designed to be a **final analysis object**:
 - Useful to plot differential cross sections, applied cuts, trigger efficiencies: **i.e. physics analysis**
 - Limited to understand jet reconstruction/calibration.
- Objects (jets, tracks, etc.) are stored in **collections**, each collection has a different key that allow us to access the elements of the collection
- For example in a jet collection, each jet contains:

$$p_x, p_y, p_z, m, m^2, p, p^2, \eta, y, \phi, E, E_{\perp}, p_{\perp}, p_{\perp}^{-1},$$
$$\cos(\phi), \sin(\phi), \cos(\theta), \sin(\theta), \cot(\theta), \tan(\theta)$$

and some information regarding its constituents (basically b-tagging information and energy per layer).

The Jet collection keys

Jets

see also [JetAnalysis](#)

Note that there is a change from 14.0.1 and 14.1.0 merging these two classes. To read about it see [ParticleJetMerger](#)

In 15.3.0 and onward

Container Class	Location	Data Access Key
JetCollection	ESD & AOD	"Cone4H1TopoJets", "Cone4H1TowerJets", "Cone4TruthJets", "Cone7H1TowerJets", "AntiKt4H1TopoJets", "AntiKt4H1TowerJets", "AntiKt4TruthJets", "AntiKt6H1TowerJets"

- [Jet](#)

In 14.2.10 and onward

Container Class	Location	Data Access Key
JetCollection	ESD & AOD	"Cone4H1TopoJets", "Cone4H1TowerJets", "Cone4TruthJets", "Cone7H1TowerJets"

- [Jet](#)

In 14.1.0 and onward

Container Class	Location	Data Access Key
JetCollection	ESD & AOD	"Cone4H1TopoJets", "Cone4H1TowerJets", "Cone4TruthJets", "Cone7H1TopoJets", "Cone7H1TowerJets", "Cone7TruthJets", "Kt4H1TopoJets", "Kt4H1TowerJets", "Kt4TruthJets", "Kt6H1TopoJets", "Kt6H1TowerJets", "Kt6TruthJets"

- [Jet](#)

In 14.0.1

Container Class	Location	Data Access Key
ParticleJetContainer	AOD	"Cone4H1TopoParticleJets", "Cone4H1TowerParticleJets", "Cone4TruthParticleJets", "Cone7H1TopoParticleJets", "Cone7H1TowerParticleJets", "Cone7TruthParticleJets", "Kt4H1TopoParticleJets", "Kt4H1TowerParticleJets", "Kt4TruthParticleJets", "Kt6H1TopoParticleJets", "Kt6H1TowerParticleJets", "Kt6TruthParticleJets"

But to be sure please check the *twiki*:

<https://twiki.cern.ch/twiki/bin/view/Atlas/AODClassSummary>

Basic things about ESDs, AODs, DPDs

AOD or ESD ?

Only use ESD if you really needed, ESD will only be stored at the Tier 1 (BNL) and the processing time is slower

Do I need ESDs?

Yes if you want to run recalibration, build your own jet collection from reconstruction objects: in one sentence: if you need to understand your detector.

No if you want to measure cross sections, plot invariant masses, trigger efficiencies, etc; AOD is fine for that.

What is a DPD?

DPD is a reduced (and physics-group-customized) AOD. At first approximation everything I'll say about AODs works for DPDs.

How to access the information in an AOD (and analyze it!) is the goal of the next slides

The **Plain_Analysis** package

The *cmt* package structure

- The `Plain_Analysis` package is located here:
 - `/users/torregrosa/tutorial/Plain_Analysis.tgz`
 - Copy it into your `~/testarea/15.4.0/` directory and execute:
 - `tar -zxvf Plain_Analysis.tgz`
- Inside the main directory, these are the subdirectories you should worry about:
 - `Plain_Analysis/cmt` this is where the *makefile* and *requirements* file is.
 - `Plain_Analysis/src` this is where the `.cxx` files are
 - `Plain_Analysis/Plain_Analysis` this is where the `.h` files are
 - `Plain_Analysis/run` this is where the `.py` files are

Plain_Analysis/cmt

- Inside it you can find:
 - **requirements** : here you tell what Athena packages you are going to use
 - *You don't need to touch it, I only wanted to know where to do it if you want to add extra functionality in the future*
 - **Change_Version.sh** this is a script that helps you to change the version of the **DragonflyAlg** algorithm. There are six versions:
 - **V1.0** and **v2.0** : forget about them, these are working versions
 - **V3.0** : This is the default (and the basic) version, you must start with this one
 - **V4.0**, **v5.0** and **v6.0** : These versions contain the solutions to the exercises, have a look at them if you are lost while doing the exercises
 - **Display_Version.sh** this script displays the current version and gives information about it. You can read this information in the file **readme.txt**

You don't have to worry about anything else

Plain_Analysis/src

- Inside it you can find:
 - Six files like `DragonflyAlg.cxx_vi.0` where *i* goes from 1 to 6, each one contains the source for each version of the `Dragonfly` algorithm
 - `DragonflyAlg.cxx` is only a symbolic link to one of the files above

Plain_Analysis/Plain_Analysis

- Inside it you can find:
 - Six files like `DragonflyAlg.h_vi.0` where *i* goes from 1 to 6, each one contains the `header` for each version of the `Dragonfly` algorithm
 - `DragonflyAlg.h` is only a `symbolic link` to one of the files above

The script `Change_Version.sh` in the `cmt` directory changes the link to the version you want to use

Plain_Analysis/run

- Here is where you have to run athena.
 - The only thing you should worry about here are the job options file.
 - These are `Plain_Analysis_topOptions_vi.0.py` where *i* goes from 1 to 6
 - Later on I'll explain the three things you must know about them
-

I've explained the basic structure of an Athena package, but to summarize it:

- `Plain_Analysis/cmt` where you have to compile : `gmake`
- `Plain_Analysis/src` where the source files are
- `Plain_Analysis/Plain_Analysis` where the header files are
- `Plain_Analysis/run` where you have to run athena

Now I'll move to explain the basic things of an Athena algorithm

The DragonflyAlg algorithm

General things about the DragonflyAlg algorithm

- The goal is simply: **read** reconstructed objects (jets, electrons, etc.) from an AOD; **analyze** them and dump the **result** into an ntuple (root file) to make plots, i.e. it is basically an implementation of the analysis skeleton.
- The output ntuple format is `CBNT_AthenaAware`
 - I only mention it because it conditions the methods to define (see next slides)
 - It is **used by the e/gamma** group for their customized ntuples
 - It has limited functionality (e.g. you cannot store `TLorentzVector`) only singled valued variables or arrays (as vectors).
- At the end of the road `DragonflyAlg` algorithm is only a **c++ class** with a header file to define the methods and variables and a source file to write down the code for each method
 - I'll show you the methods of `v3.0` that is our starting version

The DragonflyAlg algorithm: compulsory methods

■ The constructor :

- `DragonflyAlg::DragonflyAlg(const std::string& name, ISvcLocator* pSvcLocator) : CBNT_AthenaAwareBase(name, pSvcLocator), m_trigDec("TrigDec::TrigDecisionTool")`
- The important thing to remember is that here you define the properties of the algorithm: i.e. a set of variables that you can set in the job Options e.g.:
`declareProperty("JetCollection", m_JetContainerName="ConeTowerJets");`

■ `CBNT_initializeBeforeEventLoop()`

- It is executed only once
- This is where I initialize the `TrigDecisionTool` :

```
if ( m_doTrigger ) {
    sc = m_trigDec.retrieve();
    if ( sc.isFailure() ){
        mLog << MSG::ERROR << "Can't get handle on TrigDecisionTool" << endreq;
    } else {
        mLog << MSG::DEBUG << "Got handle on TrigDecisionTool" << endreq;
    }
}
```

The DragonflyAlg algorithm: compulsory methods

■ CBNT_initialize : Again executed only once

```
StatusCode DragonflyAlg::CBNT_initialize() {
```

```
    MsgStream mLog( messageService(), name() );
```

```
    mLog << MSG::DEBUG << "Initializing DragonflyAlg" << endreq;
```

```
    /** get a handle of StoreGate for access to the Event Store */
```

```
    StatusCode sc = service("StoreGateSvc", m_storeGate);
```

```
    if (sc.isFailure()) {
```

```
        mLog << MSG::ERROR
```

```
            << "Unable to retrieve pointer to StoreGateSvc"
```

```
            << endreq;
```

```
        return sc;
```

```
    }
```

```
    /** get a handle on the NTuple and histogramming service */
```

```
    sc = service("THistSvc", m_thistSvc);
```

```
    if (sc.isFailure()) {
```

```
        mLog << MSG::ERROR
```

```
            << "Unable to retrieve pointer to THistSvc"
```

```
            << endreq;
```

```
        return sc;
```

```
    }
```

} The first line initializes the output text stream. The second line makes use of it

} These lines initialize the StoreGate (aka Event Store) . The Storegate is where all the things we want are stored

} These lines initialize the histogram service

I am telling you this for your information, but you don't need to touch these lines. The important things come now

The DragonflyAlg algorithm: compulsory methods

■ CBNT_initialize : Again executed only once

```
addBranch("NJets",          m_aan_njets, "NJets/i");
addBranch("JetsEta"        ,m_aan_JetEta);
addBranch("JetsPt"         ,m_aan_JetPt);
addBranch("JetsPhi"        ,m_aan_JetPhi);
addBranch("MissingET",     m_aan_ptMiss, "MissingET/d");
```

These lines **define** the branches of the **output root file**. Line 1 defines a single integer, lines 2,3,4 defines three vector branches. Line 5 defines a single double

```
/// ROOT histograms -----
```

```
/// jets -
m_h_jet_eta = new TH1F("jet_eta","Leading jet_eta",50,-5.,5.);
sc = m_thistSvc->regHist("/AANT/Jet/jet_eta",m_h_jet_eta);
```

```
m_h_jet_phi = new TH1F("jet_phi","Leading jet_phi",50,-3.2,3.2);
sc = m_thistSvc->regHist("/AANT/Jet/jet_phi",m_h_jet_phi);
```

```
m_h_jet_pt = new TH1F("jet_pt","Leading jet_pt",500,0.,600000.);
sc = m_thistSvc->regHist("/AANT/Jet/jet_pt",m_h_jet_pt);
```

```
/// missing ET
```

```
m_pxMis = new TH1F("MissingPx", "MissingPx",200,-500.0*GeV,500.*GeV);
sc = m_thistSvc->regHist("/AANT/MissingET/MissingPx", m_pxMis);
m_pyMis = new TH1F("MissingPy", "MissingPy",200,-500.0*GeV,500.*GeV);
sc = m_thistSvc->regHist("/AANT/MissingET/MissingPy", m_pyMis);
m_ptMis = new TH1F("MissingPt", "MissingPt",100,0.0,500.*GeV);
sc = m_thistSvc->regHist("/AANT/MissingET/MissingPt", m_ptMis);
```

These lines **define** the **histograms** to be stored in the **output file**. I hardly use them but I wanted you to know how do define them.

```
if (sc.isFailure()) {
    mLog << MSG::ERROR << "ROOT Hist registration failed" << endreq;
    return sc;
}
```

These lines just check that everything was fine

```
/// end ROOT Histograms -----
```

The DragonflyAlg algorithm: compulsory methods

- `CBNT_finalize` and `CBNT_clear`: Again executed only once

```
////////////////////////////////////  
/// Finalize - delete any memory allocation from the heap
```

```
StatusCode DragonflyAlg::CBNT_finalize() {  
    MsgStream mLog( messageService(), name() );  
  
    return StatusCode::SUCCESS;  
}
```

} This method is executed at the end of the loop over all the events. Useful to print out counting information for instance

```
////////////////////////////////////  
/// Clear - clear CBNT members
```

```
StatusCode DragonflyAlg::CBNT_clear() {  
    /// For Athena-Aware NTuple  
  
    m_aan_njets=0;  
    m_aan_JetEta->clear();  
    m_aan_JetPt->clear();  
    m_aan_JetPhi->clear();  
  
    //  
  
    m_aan_ptMiss = -1.;  
  
    return StatusCode::SUCCESS;  
}
```

} This method is important. If you want your arrays to be stored in the output root file. You **MUST** clear them in this method

The DragonflyAlg algorithm: compulsory methods

■ CBNT_execute: Executed in every event

```
StatusCode DragonflyAlg::CBNT_execute() {  
    MsgStream mLog( messageService(), name() );  
    mLog << MSG::DEBUG << "in execute()" << endreq;  
  
    StatusCode sc;
```

} The first line initializes the output text stream. The second line makes use of it

```
/** an minimal example using the TrigDecisionTool */  
if ( m_doTrigger ) {  
    sc = triggerSkeleton();  
    if (sc.isFailure()) {  
        mLog << MSG::WARNING << "The method triggerSkeleton() failed" << endreq;  
        return StatusCode::FAILURE;;  
    }  
}
```

} These lines called the **triggerSkeleton** method that's takes care of the trigger analysis. See next slides

The DragonflyAlg algorithm: compulsory methods

■ CBNT_execute: Executed in every event

```
/** get missing Et information */
```

```
sc = getMissingEt();  
if( sc.isFailure() ) {  
    mLog << MSG::WARNING  
        << "Failed to retrieve Et object found in TDS"  
        << endl;  
    return StatusCode::FAILURE;  
}
```

These lines call the `getMissingEt` method that's takes care of getting the Missing Et and dump it into the ntuple. See next slides

```
/** get Jet information */
```

```
sc = getJetInfo();  
if( sc.isFailure() ) {  
    mLog << MSG::WARNING  
        << "Failed to retrieve Jet object found in TDS"  
        << endl;  
    return StatusCode::FAILURE;  
}
```

These lines calle the `getJetInfo` method that's takes care of getting the Jet collection and dump it into the ntuple. See next slides

```
return StatusCode::SUCCESS;  
}
```

This is the end. That's it!!

Thus, the bulk of the analysis is inside these three methods:

`getMissingEt`, `getJetInfo`, `triggerSkeleton`

Now we'll have a look at them

The DragonflyAlg algorithm: non compulsory methods

■ triggerSkeleton

```
////////////////////////////////////  
/// Trigger method - called by execute() on event by event  
/// to be removed if not needed
```

```
StatusCode DragonflyAlg::triggerSkeleton() {  
    MsgStream mLog( messageService(), name() );  
    mLog << MSG::DEBUG << "in triggerSkeleton()" << endreq;  
  
    // for example, did event pass Event Filter ?  
    // needs to be changed to m_trigDec->isPhysicsPassed  
    mLog << MSG::INFO << "Pass state EF = " << m_trigDec->isPassed(TrigDec::EF) << endreq;  
  
    // for example, did event pass !L2_e25i chain?  
    // needs to be changed to m_trigDec->isPhysicsPassed  
    std::string mychain("L2_e25i");  
    const HLT::Chain* chain = m_trigDec->getHLTChain(mychain);  
    if (0 == chain){  
        mLog << MSG::INFO << "Chain " << mychain << " is not defined";  
    } else {  
        mLog << MSG::INFO << "Chain " << mychain << ": " << *chain << " passed: " << chain->chainPassed() << endreq;  
  
        mLog << MSG::DEBUG << "triggerSkeleton() succeeded" << endreq;  
  
        return StatusCode::SUCCESS;  
    }  
    return StatusCode::SUCCESS;  
}
```

This line prints out if the event passes the EF trigger

The line one defines the trigger we want to have a look at. Line 2 gets the trigger object

This method uses the TrigDecisionTool, for documentation about it, have a look at <https://twiki.cern.ch/twiki/bin/view/Atlas/TrigDecisionTool14>

The DragonflyAlg algorithm: non compulsory methods

■ getJetInfo

```
////////////////////////////////////  
/// Jet info object  
  
StatusCode DragonflyAlg::getJetInfo() {  
  
    MsgStream mLog(messageService(), name());  
    mLog << MSG::DEBUG << "getJetInfo()" << endreq;  
  
    StatusCode sc = StatusCode::SUCCESS;  
  
    // EFT retrieving the Jet collection from the storegate  
    const JetCollection* PartJetCont = 0;  
    sc = m_storeGate->retrieve( PartJetCont, m_JetContainerName );  
    if( sc.isFailure() || !PartJetCont ) {  
        mLog << MSG::ERROR  
            << "No AOD Particle Jet Container found in TDS"  
            << endreq;  
        return StatusCode::FAILURE;  
    }  
    //EFT Getting the size of the reco collection  
    int Number_of_reco_jets = (int) PartJetCont->size();  
  
    m_aan_njets = Number_of_reco_jets;  
}
```

Jets are stored in collections. There is a collection for each event. Line 1 gets a pointer to a jet collection. Line 2 links the just defined pointer to the storegate container through its key (m_JetContainerName). The following lines is to check that everything was fine and the last two lines get the number of jets in the event (size of the collection) and puts this information into the ntuple (m_aan_njets)

So we have the Jet collection, what can we do with it?

The DragonflyAlg algorithm: non compulsory methods

■ getJetInfo

```
//EFT Starting the loop over the jet collection
```

```
JetCollection::const_iterator JetItr = PartJetCont->begin();  
JetCollection::const_iterator JetItrE = PartJetCont->end();
```

```
double lead_jet_pt = (*JetItr)->pt();  
double lead_jet_eta = (*JetItr)->eta();  
double lead_jet_phi = (*JetItr)->phi();
```

```
for (; JetItr != JetItrE; ++JetItr)  
{  
    double local_pt = (*JetItr)->pt();  
    double local_eta = (*JetItr)->eta();  
    double local_phi = (*JetItr)->phi();
```

```
    m_aan_JetEta->push_back(local_eta);  
    m_aan_JetPhi->push_back(local_phi);  
    m_aan_JetPt->push_back(local_pt);  
}
```

```
/// fill missing jet histograms with the leading jet  
m_h_jet_eta->Fill(lead_jet_eta);  
m_h_jet_pt->Fill(lead_jet_pt);  
m_h_jet_phi->Fill(lead_jet_phi);  
return sc;
```

} Let's make a loop over them!! By getting the collection iterators

} The collection is sorted by Pt. The first jet is the hardest, these lines gets its Pt, eta and phi

} Here is where the loop takes place. For each Jet in the collection we get the Pt, phi and eta and we put into into the vector that is stored in the ntuple:
m_aan_JetXXX

} These lines only fill the histograms of the leading jet magnitudes

You can get more information about what you can do with a JetCollection and Jets here

<http://reserve02.usatlas.bnl.gov/lxr/source/atlas/Reconstruction/Jet/JetEvent/src/>

But that's it!!

The things you should know about Job Options files

```
# input directory
DIR='/afs/cern.ch/user/c/chekanov/public/data/mc08.106379.PythiaPhotonJet_AsymJetFilter.recon.AOD.e347_s462_r541/*'

print "##### INPUT FILES #####"
import glob
fileList = glob.glob(DIR)
for item in fileList:
    print item
print len(fileList) , " for analysis"

ServiceMgr.EventSelector.InputCollections =fileList;
```

These lines define the input files

```
# list of the algorithms to be executed at run time
from Plain_Analysis.Plain_AnalysisConf import DragonflyAlg
topSequence.CBNT_AthenaAware += DragonflyAlg()
DragonflyAlg = DragonflyAlg()
```

These lines tell Athena that we want to use the algorithm

```
##### The properties of the DragonflyAlg Algorithm
DragonflyAlg.MissingETObject = "MET_RefFinal"
DragonflyAlg.DeltaRMatchCut = 0.2
DragonflyAlg.MaxDeltaR = 0.9999
DragonflyAlg.MissingETCut = 20.0*GeV
DragonflyAlg.OutputLevel = INFO
```

These lines set the properties (options) of our algorithm

```
ServiceMgr.THistSvc.Output = [ "AANT DATAFILE='DragonflyAlg.aan.root' OPT='RECREATE'" ]
AANTupleStream.OutputName = 'DragonflyAlg.aan.root'
```

The output file

```
# Number of Events to process
theApp.EvtMax = -1
#theApp.EvtMax = 10
```

Number of events : 10 is 10; -1 is all

Proposed exercises

Exercise number cero

- Get the PlainAnalysis package coping it from:
`/users/torregrosa/tutorial/Plain_Analysis.tgz` to your
`$home/testarea/15.4.0/` directory and execute:
 - `tar -zxvf Plain_Analysis.tgz`
- Setup Athena (if not done yet):
 - http://atlaswww.hep.anl.gov/asc/ASC_working/index.php?n=Main.SettingUpAccount
- Go to the cmt directory and compile it :
 - `cmt config`
 - `source setup.sh` (or `source setup.csh`)
 - `gmake`
- Go to the run directory and run athena
 - `athena Plain_Analysis_topOptions_v3.0.py`
- Check the output ntuple. Have a look at the code:
 - `src/DragonflyAlg.cxx` and `Plain_Analysis/DragonflyAlg.h`
- Ask me any questions you may have about any part of it, get familiar with it

Exercise number one

- Add new information into the output ntuple. My proposal is
 - A vector that stores the mass of each jet in the collection
 - The invariant mass on the two leading jets
 - The $\text{Cos}(\theta)^*$ of the event
- I solve it making use of the CLHEP library. Have a look at the Jet.cxx class. You can get a HepLorentzVector out of each jet. Then is just question to use the proper methods of the HepLorentzVector class
- Have a look in the code where everything related with the ntuple is coded (both header and source file; the variables use to start with `m_aan`) just do the same for the new variables
- If you are lost, go to the `cmt` directory and do:
 - `./Change_Version.sh v5.0`
 - and have a look again at
 - `src/DragonflyAlg.cxx` and `Plain_Analysis/DragonflyAlg.h`
 - Compile it (as before) and execute it:
 - `athena Plain_Analysis_topOptions_v3.0.py`

Exercise number two

- One step further : create a new method that gets the jet and the track collection. Then for each jet loops over all the tracks and counts the number of tracks that match each jet (ΔR lower than some threshold). The store the number of tracks matched to each jet in a vector into the output tuple.
- The track collection class is: `Rec::TrackParticleContainer` the key that I use is : `TrackParticleCandidate`
- You can find information about this c++ class here
 - <http://reserve02.usatlas.bnl.gov/lxr/source/atlas/Reconstruction/Particle/?v=head>
- There is a function that help you to do the matching:
 - `DeltaR(double eta1, double phi1, double eta2, double phi2)`
- Again if you are lost, look at the solution doing exactly the same as in the exercise one but now : `./Change_Version.sh v6.0`
 - **Or ask me questions about it!!**

Exercise number three

- Now it comes the trigger issue: `triggerSkeleton` makes use of the `TrigDecisionTool`. I would like you to get familiar with it and to modify the `triggerSkeleton` method to count:
 - Events that passed L1_J70, L1_J120 trigger and L2_J150 triggers independently
 - Events that passed L1_J70 and L2_J150 triggers together
 - Events that passed L1_J120 and L2_J150 triggers together
 - Events that passed L1_J70 and L1_J120 triggers together
- It is not trivial, look carefully the `TrigDecisionTool` documentation:
 - <https://twiki.cern.ch/twiki/bin/view/Atlas/TrigDecisionTool14>
- Again if you are lost, look at the solution doing exactly the same as in the exercise one but now : `./Change_Version.sh v4.0`

Summary

- I tried to explained you the basic things as simple as possible
- Take it easy and please don't get discouraged. Athena requires dedication but pays the effort back
- Try the exercises, let me know if there is any problem, ask me doubts, now or by email. I'am here to help you

My strongest advise: take it as a game and enjoy it

