

Table of Contents

SoftwareTutorialGermanyAnalysis.....	1
Preface.....	2
Logging in on your working machine.....	2
NAF.....	2
lxplus.....	2
The xAOD sample.....	3
The analysis software release.....	3
Before we get started: CMake analysis environment setup.....	4
Part A: Using the xAOD in a ROOT session interactively.....	9
xAOD EDM software.....	9
Prepare your setup.....	9
Browsing the xAOD with the TBrowser.....	10
Interactive ROOT with the xAOD.....	10
You stand before a crossroad.....	12
Part B: Running Athena.....	13
1. Prepare your setup.....	13
2. Run Athena with a job option.....	13
Creating a jobOption.....	13
Specifying input files.....	14
3. Run Athena in interactive mode.....	15
Part C: Writing your own analysis code in Athena.....	17
1. Prepare your setup.....	17
2. Creating your analysis package and algorithm.....	17
Setting up your analysis skeleton.....	17
Investigate the package structure and first compilation.....	17
Interlude: how do I find the location of specific packages or code.....	18
(Optional) Break the package, for science!.....	18
Adding configurable properties to your algorithm.....	21
Add your algorithm.....	22
Interlude: joboptions in the package.....	22
3. Working with the StoreGateSvc.....	23
StoreGate and key names, and the content of the xAOD EDM.....	24
Example: Looping over jets.....	25
Accessing other EDM objects: Checklist.....	26
Manipulating xAOD collections: making a subset and adding decorations.....	26
Accessing xAOD MetaData information.....	27
4. Creating and saving trees and histograms.....	29
Using ROOT (trees and histograms).....	30
Using THistSvc (trees and histograms).....	30
Using AthHistogramFilterAlgorithm (only histograms).....	31
5. Using a tool.....	32
Interlude: How do I know which CP tools to use and how to use them?.....	32
Example: JetSelectorTool as an Athena Configurable.....	33
Interlude: and.....	35
6. Objects and tools for analysis.....	36
Muons.....	36
7. Advanced example: creating an algorithm for the muon systematic variations.....	36
Introduction and setup of the code skeleton.....	36

Table of Contents

Part C: Writing your own analysis code in Athena

Editing the header (.h) file.....	37
Editing the source (.cxx) file.....	38
Compiling your code.....	41
Editing the job options.....	42
8. Using ganga to send your job to the local batch system.....	42

Part D: Using existing Athena components.....44

1. Prepare your setup.....	44
2. Using existing features to make simple plots.....	44
Creating a jobOption.....	44
Specifying input files.....	45
Interlude: building packages from the athena repository on top of the release.....	46
Selecting muons and building Z-> $\mu\mu$ candidates.....	48
Create an output ROOT file for the resulting histograms.....	49
Booking simple cuts and simple histograms for every cut.....	49
Writing out a small xAOD.....	52
Writing out only selected events.....	54

SoftwareTutorialGermanyAnalysis

Preface

First, please note that the top of this page gives you the hierarchical navigation to other tutorial pages which have a lot of useful information:

- main page of the Germany tutorial
- main page of the general offline software tutorial.
- Atlas Software Documentation
- AthAnalysisBase twiki

The last link gives a lot of information on building an analysis in AthAnalysis (in which we will work for this tutorial). Don't get confused by the suffix `Base`, it was removed from the Athena analysis release naming scheme with release 21.

This tutorial will consist of three main parts and some extra information:

1. The first part will show you how you can easily open an existing xAOD and browse its content or make some simple plots using `TTree->Draw(...);`.
2. The second part will show you how to run a simple Athena job option.
3. The third part will show you how to use the xAOD event data model in an analysis.
4. The additional part will show you how to use existing Athena components to perform some simple analysis and make a few plots.

Logging in on your working machine

Note: Make sure you have X11 forwarding or you won't be able to open the TBrowser from ROOT. Passing `-XY` to ssh should work but ask us if you need help with this.

NAF

For this tutorial we assume that you'll be working on the NAF nodes at DESY. In order to connect to one of them, just do this in the terminal on your laptop:

```
ssh -XY yourusername@naf-atlas.desy.de
```

Here `yourusername` is your NAF account name.

lxplus

You can also do this tutorial on the lxplus nodes at CERN. In order to connect to one of them, just do this in the terminal on your laptop:

```
ssh -XY yourusername@lxplus.cern.ch
```

Assuming that you have a valid CERN computing account and used your correct CERN user name (instead of `yourusername` in the line above), you should be prompted to provide your password.

Note: if you encounter exceptionally slow performance on the particular node you're logged in, you might want to try to relog in to another one (the command above automatically assigns you to one, no need to specify a name).

The xAOD sample

The xAOD used in this example is coming from a Zmumu MC16 dataset (if you are on the NAF):

```
/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV.361107.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zm
```

In case you want to work on lxplus, you can also find it here:

```
/afs/cern.ch/work/n/nihartma/public/atlas-d-tutorial2019/mc16_13TeV.361107.PowhegPythia8EvtGen_AZ
```

The analysis software release

Running an analysis on xAODs as input files requires an environment which provides the various software components needed to process xAODs. We'll use one of the general Analysis Releases maintained by ASG. For this tutorial, we will use AthAnalysis with the release number 21.2.88. This is an Athena based version of the Analysis Release. You can set up the same release as a standalone ROOT version, where the release flavour would be AnalysisBase instead of AthAnalysis. We'll get back to some details about releases and how to choose them in the next section.

In order to simply browse the xAOD in ROOT you only need a relatively recent version of ROOT (the default with the AthAnalysis release should be fine), but to work interactively or using macros (described below) you need the additional ATLAS packages automatically included in the Analysis Release setup. Very brief information on an Analysis Release:

- it sits on cvmfs (so can be used on other sites with cvmfs access, like the Grid sites)
- has support in SLC6 (and MacOS for the AnalysisBase releases, i.e., without Athena)
- has a size of O(300 MB)
- can be checked out from cvmfs in full and run on one of the above supported platforms

Before we get started: CMake analysis environment setup

Since the main code building tool was changed when moving to release 21 of ATLAS software (from cmt to cmake), it's necessary to discuss the required steps to configure and compile your analysis environment. In principle not all steps that are mentioned here are mandatory if you don't want to compile code within the particular release you're using. You'll see that in Part A and this is also the case for the first couple of sections of Part D. However, it is worthwhile to incorporate the proper structure for your local environment right from the start, this makes anything you want to put on top of it afterwards a lot easier.

Note: While we use AthAnalysis here, the instructions can be taken for any other release flavour as well. In particular, you would do the same steps as well when using an AnalysisBase release with which you could do an EventLoop based analysis (i.e. an analysis in standalone ROOT).

In this section and throughout this tutorial, I will give very brief explanations on the cmake specific parts that are needed to develop your own analysis. However, I highly encourage you to have a look (at some point after this tutorial) at the basic introduction to cmake and, if you're interested in a more detailed explanation, also the advanced introduction to cmake. In the long run, you'll save time if you have at least a basic understanding of how this works, if you plan to do analysis in ATLAS.

If you're not logged in to a worker node, please do so now. Next, we will set up the ATLAS software environment so that you can load all the different software components and packages you might need in your projects and endeavors.

If you are working on lxplus, you can simply set up the ATLAS software environment by executing the alias

```
setupATLAS
```

If you are working on the NAF, you first need to create the necessary alias with these commands... [▶](#) If you are working on the NAF, you first need to create the necessary alias with these commands [▼](#)

If you are **not** working on lxplus, you will need the following commands:

```
export ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase
alias setupATLAS='source ${ATLAS_LOCAL_ROOT_BASE}/user/atlasLocalSetup.sh'
```

Next, we will create a proper directory structure for this tutorial and change to it:

```
cd      # make sure that we're in $HOME.
mkdir xAODSoftwareTutorial
cd xAODSoftwareTutorial
mkdir source build run
```

With cmake, we have three directories we usually work with:

- **source** This is where all your source files live. The build system will place a couple of steering files here, but apart from these, this directory is really only reserved for actual source files containing your code.
- **build** This is where the build system places all its created files. You are not supposed to put any of your own files into this directory. That way you can simply delete the directory and have your build system rebuild the entire setup.
- **run** This is where your final program actually runs and saves output files. There is no need for this directory to be anywhere near your `source` and `build` directory, it can be wherever you want, and you can create as many run-directories as you wish.

Some more comments on these directories...[▶](#) Some more comments on these directories[▼](#)

- Your `source` directory should in principle correspond to a git repository (or some version controlled area). This way, if you break your code, you can always revert to the latest working version.
- The naming of these directories is not set into stone, you can choose whatever names suit you best. In particular, you might have multiple `build` and `run` directories so you might want to give them more descriptive names.
- None of these directories need to be in the same parent directory. It is, however, often easier to keep them together. Your `build` directory, for example, might also go on a space that is not being backed up as you can easily rebuild the directory if you lose it. Your `run` directory, might go on a local disk (as opposed to a network disk) as your I/O might otherwise be limited by your network.

Now, we setup the Analysis Release (in the **build** directory) and run the cmake configuration:

```
cd build
asetup AthAnalysis,21.2.88,here
mv CMakeLists.txt ../source/
cmake $TestArea/../../source
source $TestArea/*/setup.sh
```

What is this \$TestArea?...[▶](#) What is this \$TestArea?[▼](#)

When you execute the `asetup` command, Athena will create an environment variable called `$TestArea`. The additional argument `,here` in the command line makes sure that `$TestArea` is set to the current directory. In our case it's the `xAODSoftwareTutorial/build` folder. You should always pass `,here` to the `asetup` command argument line, except when you make the default configuration below, then you can omit it. Using `$TestArea` in all the command is just a convenient way to execute the commands without having to care where you are in the folder structure (assuming you executed `asetup` in the `xAODSoftwareTutorial/build` folder), but you can of course always just pass the relative path to the folder instead (it's absolutely not mandatory to use this environment variable).

LPT: You will probably always want to make your `$TestArea` whatever the current working directory is when you call the `asetup` command. This is what the 'here' option is doing. If you add the following lines to a `.asetup` file in your home directory, you won't ever have to type 'here' again:

```
[defaults]
testarea = <pwd>
```

(Note: you may have done this already.)

A couple of things that are worth noting:

- More information regarding the `asetup` command can be found [here](#). Note that the choice of executing it in the `build` subdirectory is actually not crucial to the functionality of your setup. It's the convention of the official CERN tutorials to have `$TestArea` pointing to this folder in your environment, so we'll do that as well.
- Executing `asetup` generated a `CMakeLists.txt` file, which we can use as a template to compile and build our analysis environment. We move it into the `source` folder to have all the source code cleanly separated from the build environment. The source folder will always hold your own packages or ones you've pulled from the release to make local changes to (e.g. when you develop a package for the release).
- When executing `cmake $TestArea/../../source` for the **first time**, the entire build setup will be done in the folder we are currently in. That's why we have to do that in the `build` directory.
- **Most importantly:** The last line is a (unfortunately) crucial necessity to setup your local analysis environment, so that Athena also recognises your locally compiled and build code, not only what is

present in the particular release that you're using. It has to be executed **every time** you work in a new shell (after logging in or just opening a new terminal).

Finally, you are now ready to compile your code with:

```
cmake --build $TestArea
```

This is what you need to execute every time you make changes to your local code. The `--build` argument tells cmake to execute the building procedure, omitting it would trigger the configuration step as you've done before. There are some cases where you have to run the configuration step again (which can also be different between the `AthAnalysis` and `AnalysisBase` release flavours), but if you only make changes to your analysis packages you don't need to do that beyond the first time. As you can see, at the moment it does nothing, which is not surprising since we also don't have any code to compile yet. We'll get there soon enough though.

Whenever you start a new shell, you need to run the following commands:

```
setupATLAS
cd xAODSoftwareTutorial/build
asetup --restore
source $TestArea/*/setup.sh
```

If you don't do this properly, the next time you execute `cmake` with a folder as argument, cmake will try to do the build setup from scratch.

What does `asetup --restore` do?... [What does `asetup --restore` do?](#)

When you execute the `asetup` command for the first time, it will store a file called `.asetup.save` in the `$TestArea` folder. This holds the information of the specific release configuration you've set up, which allows you to simply type `asetup --restore` in the folder, to get the same setup again. So you don't have to remember whatever you typed the last time to get the release you are working with.

Final remarks before we move on:

So how did we know to setup `AthAnalysis`, 21.2.88? How do we find out what versions are available?

Choosing an athena release... [Choosing an athena release](#)

Type `showVersions` at the command line. It will give you a big list of all the versions of all the software you have access to over `cvmfs`.

Since this is quite a long list to go through by hand we can use a bit of bash convenience to filter the lines we are interested in:

```
showVersions | grep AthAnalysis
```

This will print the list of available `AthAnalysis` (formerly `AthAnalysisBase`) releases. For our case the line of interest would be

```
AthAnalysis-21.2.88-x86_64-slc6-gcc62-opt
```

By the time this tutorial takes place there most likely will be newer versions. In principle, there is no harm in using these, though it's quite possible that some of the examples shown here may not be compatible anymore.

The breakdown of the pieces is:

- the first part is the 'project' (other examples are `AtlasProduction` or `AtlasOffline` with aliases 'production' and 'offline')
- the second is the release number for that project

- the third is what architecture-os-compiler-compileMode you want. Collectively this third part is called the `$CMTCONFIG`, you'll find that you get an environment variable called that when you `asetup`. Some releases are compiled on lots of several different `$CMTCONFIG`, if you are on lxplus (or the NAF) you can use `x86_64-slc6` releases. For AthAnalysis you'll find that in general only one cmt configuration for each release number is provided. However, for other (older) projects you can use `i686` arch releases with the '32' option (`x86_64` is with the '64' option, which is done automatically if you are on a 64-bit machine. Type `arch` at the prompt to check this), alter the gcc version used with the 'gccXY' option, or switch to `slc5` with the 'slc5' option.

So if you were being completely explicit about this `asetup` release you would have typed:

```
asetup 21.2.88,AthAnalysis,slc6,gcc62,64,here
```

If all of that seems complicated, I suggest you try setting up different releases. E.g. if you wanted to setup `IBLProd-17.3.10.1.1-i686-slc5-gcc43-opt`, try doing:

```
asetup IBLProd,17.3.10.1.1,32,slc5,gcc43,here
```

But don't ever try to set up two releases in one shell! Always use a new clean shell for a new release.

So how do you decide which project and release to setup? For physics analyses it's quite straight forward, because you should just use the newest AthAnalysis (or AnalysisBase, for standalone ROOT) release, if possible (to guarantee that all packages you intend to use are up-to-date). But if in doubt of which version to use, just email someone (there are mailing lists out there, exactly for these kinds of questions, eg. atlas-sw-analysis-forum@cern.ch).

If this seems too cumbersome to you, fear not, there is fortunately a shorter version ...

Try using `acm` instead of bare `cmake` ... **Try using `acm` instead of bare `cmake`**

Some people would argue that CMake+Git can be annoying at times because it takes so many commands just to set everything up. Luckily, for standard CMake+Git commands that you will have to perform in a typical analysis/development setup, there is a fantastic command suite that simplifies things greatly!

It is called the `acm` command and is available after doing `setupATLAS`.

A comparison of CMake+Git and the respective `acm` commands is given in `CMTCMakeRosettaStone`. Here, I will simply show you, how easy it is to do all the above things with `acm` instead of `cmake`.

We'll go back to the point where you ran `setupATLAS` and created the `xAODSoftwareTutorial` folder with the necessary folder structure. These steps are the same as before, so in a new shell, just run:

```
setupATLAS
cd # make sure that we're in $HOME. This is not required, but in this tutorial, we assume this st
mkdir xAODSoftwareTutorial
cd xAODSoftwareTutorial
mkdir source build run
cd build
```

Now, off to the exciting stuff! Instead of the `cmake` ordeal you went through before, we'll just the following command inside the `build` folder:

```
acmSetup AthAnalysis,21.2.88
```

That's it, short and crisp. You are now ready to compile your code with:

```
acm compile
```

If you want to restore your setup on a new shell, you just navigate to the `build` folder, setup the ATLAS software environment with `setupATLAS`, and then simply run:

```
acmSetup
```

Very easy to remember. Then, why am I telling you this only now after you had to deal with naked CMake+Git? Well, it is of course very helpful to know your way around CMake before you start hiding it away with an additional layer of simplifying commands (but I don't really need to tell you this, right?).

Part A: Using the xAOD in a ROOT session interactively

This short hands-on tutorial gives you the opportunity to look at the xAOD event data model and the information available in the xAODs, using ROOT. Complete analysis examples are shown in Part C and Part D of this tutorial.

xAOD EDM software

For the moment the best way to understand what variables you can access for each object, you should look at the code.

The EDM code can be found from: [athena/Event/xAOD](#). One thing you should make sure is that you are actually in the correct tag of the athena repository, in order to see the versions of the different xAOD packages that are used by the release we are working in. Click on the drop-down menu on the top left above the directory table (where it currently says `master`), where you can just switch to the various branches/tags of the repository. In the search field type `21.2.88`, which should bring up `release/21.2.88` as an option (may take a couple of seconds). Click on the `release/21.2.88` tag, now all packages in the repository are at the revisions which are used by the 21.2.88 builds.

In most cases, it should be clear which package in the structure to look at. For example if you are interested in taus, you would look at the [xAODTau](#) package. For each xAOD EDM package you can either choose a particular tag of the repository (corresponding to the version in your Athena or Analysis Release versions), or you can look in the master branch to see the latest version.

Once you are in the particular version of the repository (which is always indicated in the drop-down menu), navigate to where the header files live (same as the package name). In there you will find yet another directory called `versions`, and inside there you will find the various versions of the xAOD EDM you are interested in.

Prepare your setup

We'll not do anything fancy yet with the AthAnalysis release, but still make sure that you've gone through the necessary steps to set up your analysis environment:

1. In case you're not logged in to a worker node anymore, do so as described [here](#).
2. In case that you haven't done the analysis environment setup as described [here](#), please do so now. Remember, if you want to use a new terminal and/or have to log in to a worker node again, go through the necessary steps to resume your previous setup:

```
setupATLAS
cd xAODSoftwareTutorial/build
asetup --restore
source $TestArea/*/setup.sh
```

Or if you feel adventurous, try `acm` ... Or if you feel adventurous, try `acm`

As mentioned before, `acm` simplifies CMake+Git. Simply run these commands instead:

```
setupATLAS
cd xAODSoftwareTutorial/build
acmSetup
```

This will however only work if you used `acm` before for the setup (otherwise you `acm` won't find a `.acmsetup.save` file).

Browsing the xAOD with the TBrowser

Note: Looking at graphic interfaces remotely is not particular fast if you don't have a very good internet connection. If you encounter painfully long loading times of the TBrowser, you can also just skip this part. However, we encourage to look at the contents of an xAOD if you have the opportunity to do so in the future (in a ROOT local setup).

Now let's open the xAOD in ROOT, and open a TBrowser for browsing:

```
root /nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV.361107.PowhegPythia8EvtGen_AZNLOCTEQ6
root[0] TBrowser b;
```

You will get lots and lots of warnings about dictionaries not available, don't worry about those for now.

Not reassured and still want to know what these warnings mean?... [▶](#) **Not reassured and still want to know what these warnings mean?** [▶](#)

These are warnings for all class-types that are stored in this AOD file that you are loading, but that are not part of the analysis releases. These classes are only usable from a full-blown Athena release and are not of the "xAOD" format. Usually, you don't need these for physics analysis purposes. But if you are working on the trigger or reconstruction, it is likely that you will encounter them. For these cases, you have to use a full-blown Athena release. This is actually another reason why we show you here `AthAnalysis` (and not `AnalysisBase`), i.e., to familiarize you also with the basics of Athena for the cases when you need/want to work on the reconstruction and/or trigger, e.g. for your qualification task.

The TTree containing the variables is called `CollectionTree`. Feel free to open that TTree and play with the items inside.

Note: The first time you open the `CollectionTree` folder in TBrowser, it will take some time to load everything into the cache.

Can you find and understand the variables we discussed in the morning lectures?

Can you see how the variables are organized?

Look and compare containers with and without Aux, representing the Auxillary store. When looking at the xAOD in the TBrowser we need to be aware of this, but when working interactively, or with a macro, or a full-blown compiled analysis, we will interact with the xAOD objects through an interface and don't need to explicitly worry about this Auxillary store business.

Interactive ROOT with the xAOD

Let's use interactive ROOT to look at the xAOD:

```
root [0] THtml::LoadAllLibs();
root [1] xAOD::Init();
root [2] f = TFile::Open( "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV.361107.PowhegPy
root [3] t = xAOD::MakeTransientTree( f );
root [4] t->Draw( "Electrons.pt() - Electrons.trackParticle().pt()" );
```

You'll see a couple of warnings and errors, which is not entirely surprising since we are working in a completely new release format. However, the interaction with xAOD objects should work fine.

Also, to exit the interactive ROOT prompt gracefully call this method:

```
root [5] xAOD::ClearTransientTrees()  
root [6] .q
```

This will avoid segmentation faults when ROOT tries to clean up (due to some features of the TransientTree functionality).

Also, once you've made the transient tree you can double click things nicely in the TBrowser by going to the following folder:

```
root/Root Memory/CollectionTree
```

You stand before a crossroad

Since this is only a one day tutorial we can't really go through all the different aspects of using ATLAS software to do analysis. The tutorial that is laid out on this twiki page is entirely done in the athena based software environment (AthAnalysis). It's absolutely worthwhile going through the different steps, even if you don't end up working in this environment (there is a lot of overlap with other approaches, both technically and from a conceptional point of view). However, you may already know that you will have to work with some code that is used in the standalone ROOT environment (AnalysisBase), i.e. based on EventLoop. In that case, if you prefer to rather go through a tutorial that is tailored to using EventLoop you can have a look at this link:

<https://atlassoftwaredocs.web.cern.ch/ABtutorial/>

It is tailored for lxplus, but will of course also work on NAF or any other Scientific Linux machine with CVMFS access.

You can of course also have a peek at both the athena and EventLoop setup, but it's unlikely that you have the time for both.

For completeness, you can also make an analysis completely in PyROOT, as shown here, but we encourage to stick to the athena or EventLoop based setup. You're very much an outlier in ATLAS if you only work in python, thus the support you can expect from others is somewhat reduced (when it comes to the python specific stuff).

Part B: Running Athena

In this part, you will learn how to run Athena and to use the interactive mode.

1. Prepare your setup

1. In case you're not logged in to a worker node anymore, do so as described here.
2. If you still have your terminal open from the previous section, you can skip this step. In case that you haven't done the first time analysis environment setup as described here, please do so now. Remember, if you want to use a new terminal and/or have to log in to a worker node again, go through the necessary steps to resume your previous setup:

```
setupATLAS
cd xAODSoftwareTutorial/build
asetup --restore
source $TestArea/*/setup.sh
```

2. Run Athena with a job option

In this section, you will learn how you can run Athena via job options. Job options are python scripts that tell Athena what algorithms should be scheduled and allow you to define the values of configurables in tools and/or algorithms. It also allows you to define input file(s) and to control meta parameters like verbosity etc.

Creating a jobOption

Let's create the most simple job option possible. Go to the `$TestArea/./run` directory, which is a good location to execute Athena from. Athena tends to create a few files, which we rather want to have in a separate folder.

```
cd $TestArea/./run
```

Now we are in the `run/` directory, so let's create a new `jobOptions` file (e.g. `myJobOptions.py`) with just one line in it:

```
theApp.EvtMax = 10
```

Recall that `theApp` is one of the objects Athena gives you for free. It's the application manager, which steers the execution of pretty much everything and it has an `EvtMax` property to say how many events should be processed. You could set it equal to `-1` to process all events (don't do that now though, because we haven't specified any input, so you'll just create an infinite loop).

Run Athena with these newly created job options:

```
athena myJobOptions.py 2>&1 | tee log.txt
```

Note the extension after `athena myJobOptions.py`, which pipes the console output in `log.txt` as well (`2>&1` just makes sure that `stdout` and `stderr` are merged). You'll find it very useful to produce log-files of your runs, since the athena output can get quite extensive.

As a side note, you can also steer the maximum number of events to be processed with a command line argument, which is particular convenient if you want to quickly test something:

```
athena --evtMax=100 myJobOptions.py 2>&1 | tee log.txt
```

The output you get is the minimal output from an Athena job. It's worth taking some time later on to familiarize yourself with the sections (ask if you're curious about any parts). But by far the **most important** line(s) to know about is this one and the ones which follow, which would print the members of the sub-AthSequencers if there are any:

```
AthMasterSeq          INFO Member list: AthSequencer/AthAlgSeq, AthSequencer/AthOutSeq, AthSequencer/AthRegSeq
```

This is telling you what algorithms will execute each event in the loop. For now you can ignore `AthOutSeq` and `AthRegSeq`, and focus on `AthAlgSeq`, which is the sequence where you usually add algorithms. If you take a look at the log file from a typical reconstruction job, you should now see why those jobs can be very slow.

We can try to change the behaviour of this empty loop job, by grabbing one of the Configurables that will be used by the loop (the `AthenaEventLoopMgr` specifically) and setting a property of it (Configurables are Athena algorithms, tools, or services that can be configured from the python side). Add the following to your job options (`myJobOptions.py`) and rerun to see what this property controls:

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

Click for some advanced commentary about the above line...[▢](#) [Hide▢](#)

The `AthenaEventLoopMgr` doesn't actually exist at the time the joboption is included. We are creating an instance of it. You can look at the `Configurable.allConfigurables.items()` list of instantiated configurables and see `AthenaEventLoopMgr` is not there (before this line). If you don't add (or otherwise reference) the instance we create to something that will survive past the end of the joboption file, then it will be automatically garbage collected by python. So to ensure this instance, with the non-default value of `EventPrintoutInterval` set (it's been set via the kwargs trick of the python constructor) survives to be used in the event loop, we have added it to the `ServiceManager` (`svcMgr`). This is appropriate, because `AthenaEventLoopMgr` is a service (print it to see). We will explain a little later in the tutorial what the `CfgMgr` is.

Your joboptions file should look like this now...[▢](#) **Your joboptions file should look like this now**[▢](#)

```
theApp.EvtMax = 10
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

Specifying input files

Files are input into athena via an `EventSelector`, which is a service. You need a magic incarnation to make the appropriate `EventSelector` appear for the input file you are using. For xAOD and POOL files (basically everything except raw NTuples) you would use the Athena service that deals with the POOL conversion from the stuff on disk to what will be in memory. Add the line below to **the top** (it tends to interfere with the previous printout interval line otherwise) of your python job option file:

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

Some more information on `ReadAthenaPool`[▢](#) [Hide▢](#)

As mentioned, the line above is what you include in your joboptions file in order to read POOL files (which xAODs are). There are also other read modes, which are more optimised for analysis processing patterns, i.e. are usually faster reading xAODs for analysis. However, with the current rel. 21 builds, these modes lead to segmentation violations when writing output xAODs, thus we'll stick to the general POOL read mode for now.

For reference, the more performant read mode would be set up like this:

```
import AthenaRootComps.ReadAthenaxAODHybrid
```


There are different read modes for Athena. A summary is provided here:

- **POOLAccess**: The slowest read mechanism, but can read any POOL file (EVNT, AOD, DAOD, ESD), and also must be used if writing out xAOD through MultipleStreamManager
- **AthenaAccess**: The next slowest read mechanism, and should only be used by experts in certain situations
- **ClassAccess**: The default read mode, which works with xAOD files (including CxAOD), and is much faster than POOLAccess. Use when writing out ntuples and histograms
- **BranchAccess**: Faster than ClassAccess but cannot read all parts of an xAOD
- **TreeAccess**: Use when reading a flat ntuple.

Click for the raw NTuple version... [Hide](#)

```
import AthenaRootComps.ReadAthenaRoot
svcMgr.EventSelector.TupleName="physics" #change if you want to read a different TTree from the i
```

This creates an EventSelector (with instance name EventSelector) in the svcMgr, which has a property that lets you specify the input files, as a list, again in your python job option file:

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
```

This is all you need to get the content of the input files made available to the StoreGateSvc (also known as the EventStore), which is the service your algorithms (and tools) will interact with to get at the needed information for every event (it's the equivalent of the TEvent object in standalone ROOT).

Your joboptions file should look like this now... [Your joboptions file should look like this now](#)

```
theApp.EvtMax = 10
```

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
```

3. Run Athena in interactive mode

You can start Athena also in interactive mode, which might be very useful for de-bugging or getting familiar with its workings.

To start athena in interactive mode, enter:

```
athena -i
```

You can then load a file, similar as we did in the job option above, by entering

```
import AthenaPoolCnvSvc.ReadAthenaPool
svcMgr.EventSelector.InputCollections = ["/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV.
```

Now let us initialize the application, and load the first event. As before, theApp/ is one of the objects that Athena gives you for free, so you have access to it and can initialize it and use it to load the first event:

```
theApp.initialize()
theApp.nextEvent()
```

What is available in the event? You can access the event content via the StoreGateSvc. We need to get a handle to the StoreGate:

```
sg = PyAthena.py_svc('StoreGateSvc')
```

Now we can take a look at what we have. You can do a dump of the StoreGate contents by entering:

```
sg.dump()
```

You will get a pretty large print-out of all the objects that are available in the event. These include physics objects like muons, electrons or jets, trigger information, truth level information (since the file does contain simulated events) and a lot more. Let's inspect the reconstructed muons in the event. From here on, the syntax is very similar to a python dictionary if you want to access things in the StoreGate. To access one of the things in the StoreGate, such as the reconstructed muons, enter

```
muons = sg['Muons']
```

If you want to inspect the transverse momentum of muon in the event, you can do so by entering

```
print(muons[0].pt())
```

Of course here we already know that the class representing the reconstructed muon has a method `pt` that retrieves the transverse momentum. If we hadn't know this, we would have needed to inspect the interface of the muon class in the Athena code base [↗](#)

Part C: Writing your own analysis code in Athena

In this part, you will learn how to write your own C++ source code to analyse your events in Athena.

1. Prepare your setup

1. In case you're not logged in to a worker node anymore, do so as described here.
2. If you still have your terminal open from the previous section, you can skip this step. In case that you haven't done the first time analysis environment setup as described here, please do so now. Remember, if you want to use a new terminal and/or have to log in to worker node again, go through the necessary steps to resume your previous setup:

```
setupATLAS
cd xAODSoftwareTutorial/build
asetup --restore
source $TestArea/*/setup.sh
```

2. Creating your analysis package and algorithm

Setting up your analysis skeleton

We will create a new Athena package to create and store the necessary scripts for our analysis code. Enter the following commands to create this new package (called MyAthenaxAODAnalysis, which will get by default the version number MyAthenaxAODAnalysis-00-00-01) in `$TestArea/./source`:

```
cd $TestArea/./source
acmd cmake new-skeleton MyAthenaxAODAnalysis
```

If you want to learn more about the above command, have a look at the twiki page [AthenaScripts](#).

Investigate the package structure and first compilation

Let's have a look at what was created. Switch to the package folder and execute `tree`:

```
cd $TestArea/./source/MyAthenaxAODAnalysis
tree
```

You can see that the `acmd` command created the `MyAthenaxAODAnalysis` package, already placed a source and header file for a new algorithm in `src/` (and made some necessary configuration in `src/components/`), and provided the `CMakeLists.txt` skeleton which tells `cmake` how this package is to be build.

The class, which was generated in the algorithm files (`MyAthenaxAODAnalysisAlg.h` and `MyAthenaxAODAnalysisAlg.cxx`), is of type `AthAnalysisAlgorithm`. `AthAnalysisAlgorithm` is a normal Athena algorithm (meaning it has an `initialize`, `execute`, and `finalize` method) with some additional analysis functionality (e.g. meta-data access). The location (`src`) is absolutely appropriate here. Only if you ever plan to inherit from your new code, you would need to move the header file into this new directory: `$TestArea/./source/MyAthenaxAODAnalysis/MyAthenaxAODAnalysis/`. But we won't do this here.

Is there a convenient way to add additional default algorithms to my package? Hide
There is indeed. You can do this (you have to be in the base folder of your package):

```
cd $TestArea/./source/MyAthenaxAODAnalysis
```

```
acmd cmake new-analysisalg MyOtherAlg
```

First, have a look at the CMakeLists.txt file **in the package**. Every package needs to have a CMakeLists.txt file which just defines how cmake builds this particular package. As a reminder, the CMakeLists.txt file in the `$TestArea/../../source` folder tells cmake about your analysis environment in general and how to stitch everything together. So, open `$TestArea/../../source/MyAthenaxAODAnalysis/CMakeLists.txt` in your favourite editor.

The documentation inside the file will give you the very bare-bones information you need, in order to make the required changes when modifying the already present algorithm or add new ones. I'll just very briefly summarise here some of the key points.

- If you're using some external code you declare this dependency with `find_package`. With external, we mean anything that is not ATLAS specific software. A prime example (that's also in the file) is `ROOT`.
- If you're using any ATLAS package code in the release or code from another local package, you declare this with `atlas_depends_on_subdirs`. If the header file of your algorithm is in the `src` subfolder, you write the dependency under `PRIVATE`. If the header file is in `$TestArea/../../source/MyAthenaxAODAnalysis/MyAthenaxAODAnalysis/` (for our example) you'd put it under `PUBLIC`. **Note:** This configuration is actually not entirely mandatory and something which will eventually be deprecated. You may find that your code works although you've omitted adding dependencies with this option. However, for the moment we'll try to be thorough and add it every time we introduce a new dependency to another package.
- You have to declare your package as a library with `atlas_add_library`. In doing so you need to specify the source files to be included in the library and also tell cmake what the dependencies for your new library are.
- If you want your tools/algorithms in your package to be configurable in joboptions, you can do so with `atlas_add_component`.

Now you can compile and build the package for the first time:

```
cmake --build $TestArea
```

Interlude: how do I find the location of specific packages or code

It can be quite annoying trying to find out where specific packages live inside the athena repository, at first glance at least. However, one of the countless benefits of moving to git as a repository for the ATLAS software code is the in repository search engine provided by gitlab.

You can simply go to the athena repository [🔗](#) and locate the `Find file` button at the top right above the repository directory table. The link points to the master branch of the athena repository, remember that you might want to switch to the tag relevant for the software release you are using. Press the `Find file` button and just put the name of the package that you want to locate in the search field, gitlab will then give you a long lists of file paths that match your search. From that you can easily see where the package lives. Btw, if you want to find algorithms or tools or whatever, this approach will work for most things you might want to find because usually the names of header and/or source files reflect the name of the classes in them.

You can also use the `lxr` search engine to either look for files in the athena repository, or search for key words inside the code: <http://acode-browser1.usatlas.bnl.gov/lxr/search?v=21.2> [🔗](#)

(Optional) Break the package, for science!

One interlude before we get the satisfaction of writing our own analysis code. Most of the Athena scripts to generate code skeletons (including the one above) take care of a couple of subtleties, which have to be kept in mind when adding new c++ code to a package or using other Athena tools in the package. For education purposes we will go through these steps, break some of the respective configurations and see what effect this has on the compilation of the package. It's useful to see what the compiler output is when omitting these configurations, but you can of course also skip ahead to the next section.

First, have another look at `$TestArea/../../source/MyAthenaxAODAnalysis/CMakeLists.txt` and comment out the lines that make `AthAnalysisBaseComps` known to the package (with `#` at the beginning of the lines). So it should look like this:

Show Hide

```
...
atlas_depends_on_subdirs (
    PUBLIC

    PRIVATE
#   Control/AthAnalysisBaseComps
)
...
atlas_add_library ( MyAthenaxAODAnalysisLib src/*.cxx
    PUBLIC_HEADERS MyAthenaxAODAnalysis
    INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
    LINK_LIBRARIES ${ROOT_LIBRARIES}
#
    AthAnalysisBaseCompsLib
)
...
```

Now try to compile the package (executing `cmake --build $TestArea`).

... It won't work ... you'll see some printout about trying to compile and install various things, but then at the end you'll see the following:

Show Hide

```
In file included from /afs/cern.ch/user/t/tmaier/xAODSoftwareTutorial/source/MyAthenaxAODAnalysis
/afs/cern.ch/user/t/tmaier/xAODSoftwareTutorial/source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnaly
#include "AthAnalysisBaseComps/AthAnalysisAlgorithm.h"
^
compilation terminated.
gmake[2]: *** [MyAthenaxAODAnalysis/CMakeFiles/MyAthenaxAODAnalysisLib.dir/src/MyAthenaxAODAnaly
gmake[1]: *** [MyAthenaxAODAnalysis/CMakeFiles/MyAthenaxAODAnalysisLib.dir/all] Error 2
gmake: *** [all] Error 2
```

This is the compiler telling you "I don't know what an `AthAnalysisBaseComps/AthAnalysisAlgorithm.h` is, which is one of the files that our header file is trying to include. As discussed before, we have to make `cmake` aware of which libraries we want to use in the package and declare that in the `CMakeLists.txt` file. Otherwise, you'll get a printout similar to the one above. For the future, if you want to figure out where certain pieces of code live, please refer to this section.

What does the `Comps` part in `AthAnalysisBaseComps` actually mean, i.e., what is a component? [Hide](#)

As I already alluded in the question, `Comps` is an abbreviation for "component". Here, a component is just a piece of software that performs a well defined task and has a well defined interface. Key aspect here is that it can seamlessly work together with other components to work on a larger task at hand. One can also say that one component is one part of a modular software design which can work together in a larger composition. Here, `AthAnalysisBaseComps` is the package that hold the base classes for individual components of `AthAnalysis` (algorithm, service, tool). If you want to get more general, the package `AthenaBaseComps` holds the base classes for individual components of all Athena release flavours. You can have a look at the Wikipedia article about component-based software engineering [for](#) some more background information.

(Optional) Break the package, for science!

Don't forget this before moving on: remove the `#` before the use `AthAnalysisBaseComps` statements and recompile (it should succeed now).

Apart from setting the necessary entries in the requirements file, the skeleton generator also made a couple of configurations in the `src/components` folder of your package.

You can see that two files were created in `src/components`. Before going into the purpose of these additional component files, let's break them and see what this does. Go to the `src/components` folder:

```
cd $TestArea/../../source/MyAthenaxAODAnalysis/src/components
```

Now, just comment out all lines in `MyAthenaxAODAnalysis_load.cxx` and `MyAthenaxAODAnalysis_entries.cxx` by adding `//` to the beginning of each line of code. For example the content of "`MyAthenaxAODAnalysis_load.cxx`" should look like this:

```
// #include "GaudiKernel/LoadFactoryEntries.h"
// LOAD_FACTORY_ENTRIES(MyAthenaxAODAnalysis)
```



Recompile your code:

```
cmake --build $TestArea
```

With this done, let's see if we could use our algorithm in a job option? Start athena in interactive mode (`athena -i`) and try creating an instance of `MyAthenaxAODAnalysisAlg` via the `CfgMgr`. If you can't remember how to do this just go back to here, where we created an instance of the `ParticleSelectionAlg`. Our new algorithm can be created in exactly the same way (just without the arguments in the brackets, also don't pass it to a sequence).

Show me please  Hide 

```
athena> myAthAlg = CfgMgr.MyAthenaxAODAnalysisAlg ()
```

What is this `CfgMgr`?  Hide 

Since you are already in the interactive Athena mode (`athena -i`), you can simply call `help` on `CfgMgr`. Since the interactive Athena mode is a Python prompt, this can be done as:

```
athena> help(CfgMgr)
```

You will see a short help. (Type `q` to leave the help again. Type `control-d` to leave the interactive athena/python prompt again.) What it tells you is basically the this "configurable manager" knows about all available configurables. A configurable is an Athena algorithm, tool, or service that can be configured from the python side.

.... it doesn't work, as you might expect from the fact that we removed configurations in `src/components` from the sight of the compiler.

```
...
Py:ConfigurableDb WARNING : Class MyAthenaxAODAnalysisAlg not in database
...
```

We can verify the problem by looking at the auto-generated python module (located at `$TestArea/MyAthenaxAODAnalysis/genConf/MyAthenaxAODAnalysisConf.py`) - there's no class in there corresponding to our algorithm. You can see now what the purpose of the additional files in `src/components` is, they basically tell cmake that `MyAthenaxAODAnalysisAlg` is indeed an algorithm and should have the necessary python generated for it.

Now, let's fix our package again. Go to the `src/components` folder and revert the changes you've done in `MyAthenaxAODAnalysis_load.cxx`, it should look like this:

```
#include "GaudiKernel/LoadFactoryEntries.h"
LOAD_FACTORY_ENTRIES(MyAthenaxAODAnalysis)
```

This declares your new package to the Athena factory such that its C++ components (algorithms, services, tools) are known. This is done exactly once (in our case from the code skeleton generator), after that you never touch this file again.

Now do the same with `MyAthenaxAODAnalysis_entries.cxx`, which should look like this:

```
#include "GaudiKernel/DeclareFactoryEntries.h"

#include "../MyAthenaxAODAnalysisAlg.h"

DECLARE_ALGORITHM_FACTORY( MyAthenaxAODAnalysisAlg )

DECLARE_FACTORY_ENTRIES( MyAthenaxAODAnalysis )
{
    DECLARE_ALGORITHM( MyAthenaxAODAnalysisAlg );
}
```

Here, you actually declare each individual algorithm (or tool or service) to the Athena framework such that it is known. You can learn a lot about what to put in this file by looking at the `_entries.cxx` files in the various packages already on git.

Now recompile your package again:

```
cmake --build $TestArea
```

Now you should be able to see your algorithm in athena (try it like before in interactive athena mode).

Note: if you can't manage to "fix" your package again, there is no harm in just starting from scratch, since we've not really done anything yet:

```
cd $TestArea/../../source
rm -r MyAthenaxAODAnalysis
acmd cmake new-skeleton MyAthenaxAODAnalysis
cmake --build $TestArea
```

Adding configurable properties to your algorithm

If you inspect your algorithm in interactive athena further, you'll see it has some properties that are common to all configurables. Here, configurable just means that you can configure it from the python side and you don't need to hard-code everything in the C++ code. In Athena, we effectively call every algorithm, tool, or service that we can configure from the python side (so pretty much all of them) a "configurable". You can inspect your algorithm, e.g., by first creating an instance of it and assigning a python pointer (in the example below `myAthAlg`) to this instance:

```
athena> myAthAlg = CfgMgr.MyAthenaxAODAnalysisAlg ()
```

Now, you can use the usual Python methods of getting information about your python object, e.g., `help(myAthAlg)` will show you the generated help documentation again. Another nice way to figure out what kind of methods can be called on any python object is the tab-completion feature. Simply type `myAthAlg.` and hit the TAB key to see what method calls are available. You will see that the method `getProperties()` is

available. Have a look at what it returns:

```
athena> myAthAlg.getProperties()
```

See what it returns [▢](#) [Hide▢](#)

```
{'ErrorMax': '<no value>', 'AuditExecute': '<no value>', 'AuditReinitialize': '<no value>', 'Hist
```

You will get back a python dictionary (a list of key-value pairs; a telephone book is a good analogy as it is names you look up (the keys) in order to find telephone numbers (the values)). It contains as keys the names of properties (the names of the things you can set from the python side) and as corresponding values their current value.

But what if you want to add a property yourself, so you can communicate from your joboptions to your c++ code? You need to use the `declareProperty` method in the constructor of the algorithm. We will show you how to do this later.

Add your algorithm

You should add your algorithm to the `AthAlgSeq` `AthSequencer`, to do that, add the following to your job option (`myJobOptions.py`):

```
algseq = CfgMgr.AthSequencer ("AthAlgSeq") # If not already done earlier
# This is another valid way to access your algorithm in your job options, it does the same as alg
from MyAthenaxAODAnalysis.MyAthenaxAODAnalysisConf import MyAthenaxAODAnalysisAlg
alg = MyAthenaxAODAnalysisAlg ()
algseq += alg
```

Run your joboptions again and look for what changed (reminder: you'll want to do that in the `run` directory):

```
cd $TestArea/./run
athena myJobOptions.py 2>&1 | tee log.txt
```

You should see the algorithm (instance is called `MyAthenaxAODAnalysisAlg`) is in the `AthAlgSeq`, and that we get the printout from the `initialize()` method and `finalize()` method. These lines have been picked out below:

```
AthAlgSeq          INFO Member list: MyAthenaxAODAnalysisAlg
MyAthenaxAODAnalysisAlg  INFO Initializing MyAthenaxAODAnalysisAlg...
...
MyAthenaxAODAnalysisAlg  INFO Finalizing MyAthenaxAODAnalysisAlg...
```

If you wanted to see the 'DEBUG' level output that was put in the `execute()` method of the algorithm (look at the `MyAthenaxAODAnalysisAlg.cxx` file), you need to set the `OutputLevel` property. Why not try adding a second instance of your alg, with the output level set to `DEBUG`:

```
algseq += CfgMgr.MyAthenaxAODAnalysisAlg ("CopyOfAlg", OutputLevel=DEBUG)
```

Note: After this quick test please comment this line out (or remove it entirely) ... we don't really need to run our algorithm a second time ... every time... (and in fact this will create problems later on if it is not commented out or removed).

Interlude: joboptions in the package

The package skeleton also already includes a joboptions file

`$TestArea/./source/MyAthenaxAODAnalysis/share/MyAthenaxAODAnalysisAlgJobOptions.py`, which includes all the ingredients to read an input file, setup your algorithm and run over a certain amount of events

(by default 10). The brief documentation in the file gives you an idea about the different components that are needed to run this minimal analysis, but we've gone over them already in detail in Part B (it is again referenced in the beginning of Part D).

You can execute this joboptions file like this:

```
athena MyAthenaxAODAnalysis/MyAthenaxAODAnalysisAlgJobOptions.py 2>&1 | tee log.txt
```

You might wonder why we could execute the joboptions file like this, instead of passing the proper relative or absolute file path in the command line. This is due to cmake installing the joboptions file of the package in your local analysis environment. This is done in

`$TestArea/../../source/MyAthenaxAODAnalysis/CMakeLists.txt` at the very bottom with `atlas_install_joboptions`. It's a convenient way to provide joboptions (and also other data/files) to your environment. Of course you could also do this to get the same result:

```
athena $TestArea/../../source/MyAthenaxAODAnalysis/share/MyAthenaxAODAnalysisAlgJobOptions.py 2>&1 |
```

In principle you can also use this joboptions file, but for the rest of the tutorial we'll stick to the one we created in `$TestArea/../../run/`.

3. Working with the StoreGateSvc

All `AthAlgorithms` come with an `evtStore()` method that provides a pointer to the `EventStore` storegate. You can `retrieve` and `record` information, just like with `TEvent`. Unlike with `TEvent` though, you can record to `StoreGateSvc` to pass information between algorithms/tools/services (it doesn't need to be written out).

The methods of `StoreGateSvc` return a `StatusCode` of `StatusCode::SUCCESS` or `StatusCode::FAILURE`. Generally, `StatusCodes` must not be left unchecked (you might even find this enforced when you try to run your code). Use the `ATH_CHECK` or `CHECK` functions (they are equivalent to each other) to automatically check this `statusCode`. You can only do this inside a function which itself returns a `StatusCode` though ... the `execute()` method does return `StatusCode`, so this is ok! For more details on `StatusCodes`, see `ReportingErrors`.

Let's modify our algorithm to read the `EventInfo` object, which, as the name indicates, holds various information about general event properties:

In our source code `$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx` near the top add the include to this `xAOD` class that we will access:

```
#include "xAODEventInfo/EventInfo.h"
```

And at the top of our `execute()` method add these lines to know if this event is data or MC:

```
//-----
// Event information
//-----
const xAOD::EventInfo* eventInfo = 0; //NOTE: Everything that comes from the storegate, directly

// ask the event store to retrieve the xAOD EventInfo container
//ATH_CHECK( evtStore()->retrieve( eventInfo, "EventInfo" ); // the second argument ("EventInfo"
ATH_CHECK( evtStore()->retrieve( eventInfo ) );
// if there is only one container of that type in the xAOD (as with the EventInfo container), yo
// the key name, the default will be taken as the only key name in the xAOD

// check if data or MC
if(!eventInfo->eventType(xAOD::EventInfo::IS_SIMULATION) ){
    ATH_MSG_DEBUG( "DATA. Will continue to the next event." );
    return StatusCode::SUCCESS; // go to next event, here only interested in MC
}
```

```
// extra event-level information you might need:
int datasetID = eventInfo->mcChannelNumber();
double eventWeight = eventInfo->mcEventWeight();
```

We will only continue with MC events for this Athena tutorial.

We have to add the dependency to the `xAODEventInfo` package under `atlas_depends_on_subdirs` and properly link its library under `atlas_add_library`, in

`$TestArea/./source/MyAthenaxAODAnalysis/CMakeLists.txt`:

```
...
atlas_depends_on_subdirs(
    PUBLIC

    PRIVATE
    Control/AthAnalysisBaseComps
    Event/xAOD/xAODEventInfo
)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
    PUBLIC_HEADERS MyAthenaxAODAnalysis
    INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
    LINK_LIBRARIES ${ROOT_LIBRARIES}
                  AthAnalysisBaseCompsLib xAODEventInfo
)
...
```

Note: The convention for ATLAS is actually that all libraries must have the suffix `Lib`, so in the future the library for `xAODEventInfo` will be called `xAODEventInfoLib`.

And finally time to compile.

```
cmake --build $TestArea
```

You can try running athena again from the `$TestArea/./run/` directory.

StoreGate and key names, and the content of the xAOD EDM

How do you know the StoreGate key names and container "types"?

Show Hide

Once you have setup Athena you can run `checkSG.py`. This helpful script can be run on an xAOD to see a printout of all available container names and types. For example:

```
checkSG.py /nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV.361107.PowhegPythia8EvtGen_AZN
```

The left column will show you the xAOD containers, and the column on the right shows the container "key names". When you are retrieving information from StoreGate you usually need to specify the container type (for example `CaloClusterContainer`) and the key name for the particular instance of that container you are interested in (for example `"egClusterCollection"`). In your analysis you can ignore the the "Aux" containers (for Auxiliary store), these hold some behind-the-scenes magic. You can also "mostly" ignore the versions like `_v1`.

Bonus: Understanding the xAOD EDM code lives.

Show Hide

All of the xAOD EDM code lives in athena/Event/xAOD [↗](#).

For the xAOD EventInfo class click on the link above and navigate down to the xAODEventInfo/ package, from there:

- click on the xAODEventInfo/ directory holding the header files
- now if you click on EventInfo.h you should see which version of the xAOD EventInfo EDM is being used in the latest version of the code (probably version 1, or v1), so let's find that!
- going back up one step, from the xAODEventInfo/ directory open the versions/ directory and find the version of the EventInfo.h file that matches the above (I'm assuming here the trunk is the same version used to make the xAOD we are using)
- from here you can see the EventInfo class and all the information you have access to in the xAOD
- you will find the variable EventType is an enum, and we want to know is the value of IS_SIMULATION to know if the event is data or MC

Example: Looping over jets

Let's define the jet collection we want to use and setup the preliminary loop over jets in the jet container. For this example we will use the AntiKt4EMTopoJets jet collection. The method that loops over the events in our Athena analysis code (MyAthenaxAODAnalysisAlg) is the execute() method, so in our source code, in the execute() method add these lines:

```
//-----
// jets
//-----
const xAOD::JetContainer* jets = 0;
ATH_CHECK( evtStore()->retrieve( jets, "AntiKt4EMTopoJets" ) );

for(const xAOD::Jet* jet : *jets) {
    ATH_MSG_INFO( " jet eta = " << jet->eta() );
} // end for loop over jets
```

This will very simply loop over the AntiKt4EMTopoJets collection and print out the jet eta. Note we have used a range-based for-loop, which is specific to c++11 only (to check you have this, look at your \$CMTCONFIG environment variable, if it has gcc47 or higher in it, you have c++11). You can of course also use iterators or just iterate over the index of the container entries like this (but why would you decline this very convenient feature 😊):

```
for(unsigned int iJet=0; iJet<jets->size(); ++iJet) {
    ATH_MSG_INFO( " jet eta = " << jets->at(iJet)->eta() );
} // end for loop over jets
```

Now of course we need to include this new xAOD class, so there are two things we need to do. First, at the top of this source code, add the appropriate include:

```
#include "xAODJet/JetContainer.h"
```

And then second, we need to make the appropriate changes in

```
$TestArea/.../source/MyAthenaxAODAnalysis/CMakeLists.txt:
```

Show Hide

```
...
atlas_depends_on_subdirs(
    PUBLIC

    PRIVATE
    Control/AthAnalysisBaseComps
    Event/xAOD/xAODEventInfo
    Event/xAOD/xAODJet
```

```

)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
                   PUBLIC_HEADERS MyAthenaxAODAnalysis
                   INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
                   LINK_LIBRARIES ${ROOT_LIBRARIES}
                                AthAnalysisBaseCompsLib xAODEventInfo xAODJet
)
...

```

Now recompile and rerun the code and have a look at the output. You might want to remove the message printing again afterwards (or change `ATH_MSG_INFO` to `ATH_MSG_DEBUG`). As you can see this tends to spam your output (and is also cut off after reaching the INFO message limit).

Accessing other EDM objects: Checklist

If you want to access and loop over other EDM containers (electrons, taus, etc.), there is a general prescription, similar to what has been shown above:

- **Source code** `$TestArea/./source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx`
 - ◆ in the `execute()` method (that loops event-by-event) retrieve the container type and key name, and iterate over the contents of that container
 - ◆ at the very top, include the header file to the xAOD EDM container class of interest
- **CMakeLists.txt** `$TestArea/./source/MyAthenaxAODAnalysis/CMakeLists.txt`
 - ◆ add the appropriate dependency and library linking to the xAOD EDM package (usually somewhere in `athena/Event/xAOD`)

Manipulating xAOD collections: making a subset and adding decorations

Now that you know how to read xAOD objects, you will probably want to manipulate them. Below is an example of how to create a new **non-constant** container (e.g. jet container), containing a subset of your objects, which you've decorated somehow, and then store this new collection into the StoreGate.

At the top of your source file

`$TestArea/./source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx` add:

```
#include "xAODJet/JetAuxContainer.h"
```

then in the `execute()` method:

```

xAOD::JetContainer* goodJets = new xAOD::JetContainer; //creates a new jet container to hold the
xAOD::JetAuxContainer* goodJetsAux = new xAOD::JetAuxContainer;

goodJets->setStore( goodJetsAux ); //gives it a new associated aux container
xAOD::Jet::Decorator<double> myDecoration("myOtherDouble"); //Object holding the decoration, to o

for(const xAOD::Jet* jet : *jets) {
    ATH_MSG_DEBUG( "   jet eta = " << jet->eta() );
    if(fabs(jet->eta()) > 2.5) continue; //example jet selection

    xAOD::Jet* newJet = new xAOD::Jet; //create a new jet object
    // Create private auxstore for the object, copying all values from old jet
    newJet->makePrivateStore(*jet);
    goodJets->push_back(newJet);
    newJet->auxdata<double>("myDouble") = 5.; //example decoration
    myDecoration(*newJet) = 10.0; //This is generally faster
} // end for loop over jets

```

Example: Looping over jets

```
//example record to storegate: you must record both the container and the auxcontainer
CHECK( evtStore()->record(goodJets, "GoodJets") );
CHECK( evtStore()->record(goodJetsAux, "GoodJetsAux.") );
```

So what happened here? The first line created a new jet container:

```
xAOD::JetContainer* goodJets = new xAOD::JetContainer; //creates a new jet container to hold the
```

xAOD containers need what we call an 'auxilliary container' in which the actual properties of the jets are held. So we create a new one of those too, and assign it to our new jet container.

```
xAOD::JetAuxContainer* goodJetsAux = new xAOD::JetAuxContainer;
goodJets->setStore( goodJetsAux ); //gives it a new associated aux container
```

When we create a new jet object in the loop, it is created without an auxilliary store (the place where its data is held). All xAOD objects have an auxilliary store, where the actual data is held. The xAOD object is just an interface to this data. We ensure we add the jet to the jet container straight away, as this makes the new jet use the auxilliary container as its auxilliary store.

```
xAOD::Jet* newJet = new xAOD::Jet; //create a new jet object
goodJets->push_back(newJet);
```

Finally, we use 'auxdata' to decorate the jet with a property:

```
newJet->auxdata<double>("myDouble") = 5.; //example decoration
```

At the very end, we also record our new jet collection, and its auxilliary container, to the storegate. This is so other algorithms as well as the outputstreams (used for writing out xAOD objects) can access our jet collection:

```
CHECK( evtStore()->record(goodJets, "GoodJets") );
CHECK( evtStore()->record(goodJetsAux, "GoodJetsAux.") );
```

Note: When you record something to the storegate, it takes 'ownership' of it, so it will delete the object for you. If you create something (using the 'new' keyword) you will have to delete it yourself unless you give it to storegate.

And as always, compile your code with:

```
cmake --build $TestArea
```

If you wanted to write this new collection out to a new output xAOD file, you can use the instructions given in the section about writing xAOD below. But to summarize that link, to create a new xAOD with only this new 'good' jet collection add the following to your job options:

```
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
outStream = MSMgr.NewPoolRootStream ( "MyXAODStream", "myXAOD.pool.root" )
outStream.AddItem([ 'xAOD::JetContainer#GoodJets' ])
outStream.AddItem([ 'xAOD::JetAuxContainer#GoodJetsAux.' ])
```

Accessing xAOD MetaData information

An xAOD file, besides holding the "main" CollectionTree, also has a MetaData tree, which contains general information about the file. Sooner or later you'll have to access this information. Fortunately, the algorithm template already inherits from the AthAnalysisAlgorithm class, which makes accessing meta-data information very easy. One feature (besides others) is a beginInputFile method, which is executed each time a new file is

opened.

As an example for retrieving information from the MetaData tree we'll get the initial number of events and sum-of-weights from the CutBookkeepers container (we'll get to what this is and why it's important for analysis in a minute).

Open the header file `$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.h` of your algorithm and add two member variables to private:

```
private:
    int m_totalInitialEvents;
    double m_totalInitialSumOfWeights;
```

Open the generated source file

`$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx` and initialise these two variables in the initialiser list of the constructor:

```
MyAthenaxAODAnalysisAlg::MyAthenaxAODAnalysisAlg( const std::string& name, ISvcLocator* pSvcLocator )
    : AthAnalysisAlgorithm ( name, pSvcLocator ),
      m_totalInitialEvents (0),
      m_totalInitialSumOfWeights (0)
{
    //declareProperty( "Property", m_nProperty = 0, "My Example Integer Property" ); //example prop
}
```

In the `beginInputFile` method add these lines:

```
const xAOD::CutBookkeeperContainer* bks = 0;
ATH_CHECK( inputMetaStore()->retrieve(bks, "CutBookkeepers" ) );
const xAOD::CutBookkeeper* all = 0;
int maxCycle=-1; //need to find the maximum cycle
for(const xAOD::CutBookkeeper* cbk : *bks){
    if((cbk->name() == "AllExecutedEvents") && (cbk->cycle()>maxCycle) && (cbk->inputStream() == "S
        maxCycle=cbk->cycle();
        all = cbk;
    }
}
m_totalInitialEvents += all->nAcceptedEvents();
m_totalInitialSumOfWeights += all->sumOfEventWeights();
```

Note that we used `inputMetaStore()` instead of `evtStore()` to access the CutBookkeepers. This is necessary, since `evtStore()` can only access containers from the CollectionTree of the input file.

In the `finalize` method add these two lines:

```
ATH_MSG_INFO ( "Number of total initial events: " << m_totalInitialEvents);
ATH_MSG_INFO ( "Number of total initial sum-of-weights: " << m_totalInitialSumOfWeights);
```

Finally, make sure that your algorithm knows what an `xAOD::CutBookkeeper` is by adding the proper includes at the top of `$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx`:

```
#include "xAODCutFlow/CutBookkeeper.h"
#include "xAODCutFlow/CutBookkeeperContainer.h"
```

and update `$TestArea/../../source/MyAthenaxAODAnalysis/CMakeLists.txt`:

Show Hide

```
...
atlas_depends_on_subdirs (
    PUBLIC
```

```

PRIVATE
Control/AthAnalysisBaseComps
Event/xAOD/xAODEventInfo
Event/xAOD/xAODJet
Event/xAOD/xAODCutFlow
)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
PUBLIC_HEADERS MyAthenaxAODAnalysis
INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
LINK_LIBRARIES ${ROOT_LIBRARIES}
                AthAnalysisBaseCompsLib xAODEventInfo xAODJet xAODCutFlow
)
...

```

Recompile and run your jobOptions again. You should find these two lines in your output:

```

MyAthenaxAODAna... INFO Number of total initial events: 10000
MyAthenaxAODAna... INFO Number of total initial sum-of-weights: 1.90765e+07

```

What are these CutBookkeepers and why do we need them? [Hide](#)

The CutBookkeeper class is a convenient way to keep track and propagate skimming(reduction of number of events by applying cuts) information. In Athena this can be done automatically by using the CutFlowSvc, which will create a CutBookkeeper for each algorithm, which applies a "filter" (i.e. removes events according to some cut(s)). You actually already encountered such an algorithm in this tutorial in form of the CutAlg here. It is of type `AthFilterAlgorithm` (which inherits from `AthAlgorithm`), if you want to write your own skimming alg, have a look at the code of CutAlg.

In our case here we only have one input file, but the implementation is done in a way that the information is gathered for each file (when running over a set of input files) and printed at the very end when we are done processing. And the number of your total MC statistics for a given MC dataset you use is of course very much needed to be able to normalise to the integrated luminosity of your data.

You might have the valid question why we need the CutBookkeepers for this. Our input file is a full xAOD, where we have each and every event available. So why not just count them by hand? For the full, original xAODs that's true, you could just count the events and calculate the sum-of-weights yourself. However, the standard procedure in ATLAS for Run2 is to use derived xAODs, which are centrally produced from the original xAODs and hold only a sub-set of their content. In particular they usually have some form of event skimming applied to remove unnecessary information and reduce their size. If you run over files from these derivations you have no way to get the information about initial MC statistics except via the CutBookkeepers.

If you want to test that the algorithm works as expected with multiple files, just add the test file we're using twice to the input file list. Also, make sure that EvtMax is set to -1 to run over all input events(this will take a bit though):

```

theApp.EvtMax = -1
...
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
...

```

You can add meta-data container to your output stream very similarly to how you add your other containers, just use `AddMetaDataItem` instead of `AddItem`. Have a look at [here](#) to get more information.

4. Creating and saving trees and histograms

So far we have seen how to get and print some information we are interested in, but now we will show you how to save some histograms and custom trees to a ROOT file.

Using ROOT (trees and histograms)

There's nothing to stop you just using some ROOT code to output trees and histograms. Typically you would create a TFile instance in `initialize()`, and create a set of histograms and any trees immediately after that, also in `initialize()`. For the trees you would connect a set of variables that would be set in the `execute()` method, and the trees and histograms would be filled there. In the `finalize()` method you would call `Write()` on the trees and histograms, and close the TFile. I will assume you can find the necessary ROOT functions for all of this elsewhere. The package skeleton builder also already made sure that most of the common ROOT components you might need are already included in the `CMakeLists.txt` file.

Using THistSvc (trees and histograms)

The histogram service THistSvc is designed to handle the TFile creation and writing part of what you would have done by hand with ROOT. It is a good idea to use it too because when you submit jobs to the grid, pathena can automatically detect the output files of THistSvc, ensuring that they go into the output dataset. All you need to do is when you create your histograms and trees in the `initialize()` step, do not create the associated TFile, instead and register the histograms and trees to the THistSvc. If your algorithm inherits from `AthHistogramAlgorithm` (like the generated skeleton does) you can access the THistSvc via `histSvc()`. Let's try to write out a TTree that contains the pt values of all jets. First we need to add the tree and the pt variable to the private members in the header file

```
$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.h:
```

```
private:
    TTree* m_myTree;
    Float_t m_jet_pt;
```

And to the initializer list in

```
$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx:
```

```
MyAthenaxAODAnalysisAlg::MyAthenaxAODAnalysisAlg( const std::string& name, ISvcLocator* pSvcLocator )
    : AthAnalysisAlgorithm( name, pSvcLocator ),
      m_myTree(0),
      m_jet_pt(0),
      ...
```

Next, we register the tree to the THistSvc and add our branch in the `initialize()` method:

```
m_myTree = new TTree("myTree", "myTree");
CHECK( histSvc()->regTree("/MYSTREAM/SubDirectory/myTree", m_myTree) ); //registers tree to output
```

In a similar way you could add histograms as well:

```
TH1D *myHist = new TH1D ("myHist", "myHist", 10, 0, 10);
CHECK(histSvc()->regHist("/MYSTREAM/myHist", myHist));
```

And finally fill the tree in `execute()`:

```
for(const xAOD::Jet* jet : *jets) {
    ...
    m_jet_pt = jet->pt();
    m_myTree->Fill();
    ...
} // end for loop over jets
```

The important part of the above code is the first parameter of `regHist` and `regTree`, which takes the form: `/<streamName>/<objectName>` where `<objectName>` is the name of the thing you are registering, as you want it to appear in the output file (can include a path, then THistSvc puts the object inside the relevant

TDirectory), and is a unique label for the file you want to write to, which becomes relevant when you configurable the THistSvc in the joboptions:

```
svcMgr += CfgMgr.THistSvc ()
svcMgr.THistSvc.Output += ["MYSTREAM DATAFILE='myfile.root' OPT='RECREATE'"]
//svcMgr.THistSvc.Output += ["ANOTHERSTREAM DATAFILE='anotherfile.root' OPT='RECREATE'"]
```

Run your joboptions with the snippet above added to the end. You should get a file `myfile.root` that contains your tree.

Using AthHistogramFilterAlgorithm (only histograms)

If your algorithm will output lots of histograms, you can change your algorithm to inherit from `AthHistogramAlgorithm` (which itself inherits from `AthAlgorithm`) or `AthHistogramFilterAlgorithm` (which itself inherits from `AthAlgorithm` as well as `AthFilterAlgorithm`) instead, which automatically handles a lot of the histogram creation, filling, and memory deletion for us.

Let's make a new algorithm using the `acmd` skeleton builder:

```
cd $TestArea/../../source/MyAthenaxAODAnalysis
acmd cmake new-analysisalg MyAthHistogramAlg
```

Now, both in the header file (`$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthHistogramAlg.h`) and the source file (`$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthHistogramAlg.cxx`) change all occurrences of `AthAnalysisAlgorithm` to `AthHistogramFilterAlgorithm`. The reason we are using the filter variant is simply because the skeleton builder already put the filter functionalities inside the code.

Note: the `AthHistogramFilterAlgorithm` doesn't live in the `AthAnalysisBaseComps` package but rather in the `AthenaBaseComps` package, thus we make sure that this is taken into account at the relevant places. In the header file, make sure that the include statement looks like this (i.e. exchange it with whatever is already there for the algorithm):

```
#include "AthenaBaseComps/AthHistogramFilterAlgorithm.h"
```

And since our package now depends on a new package, make sure to make the appropriate changes in the `CMakeLists.txt` file (the package path is `Control/AthenaBaseComps` and the library is called, for now, `AthenaBaseComps`).

Now that our algorithm inherits from `AthHistogramFilterAlgorithm`, it gives us the following methods:

`book()` : creates histograms and registers them with the `THistSvc`, which manages writing the hists to a root file.

`hist()` : returns a pointer to a histogram, e.g. to fill it

You should `book` histograms in the `initialize()` method:

```
ATH_CHECK( book( TH1F ("hist_nMuons", "Number of muons", 10, 0.0, 10.0) ) );
ATH_CHECK( book( TH1F ("hist_nCombinedMuons", "Number of muons with the type Combined", 10, 0.0,
ATH_CHECK( book( TH1F ("hist_muonPt", "Muon Pt (GeV)", 30, 0.0, 300.0) ) );
```

... and probably fill them in the `execute()` method, e.g:

Show Hide

```
int numCombinedMuons = 0;
```

```
// Retrieve the muons:
```

```
const xAOD::MuonContainer* muons = 0; // a const here as we will not modify this container
ATH_CHECK( evtStore()->retrieve( muons, "Muons" ) );
```

```
// Loop over them:
for(const xAOD::Muon* muon : *muons ) {
    ATH_MSG_INFO( "    Selected muon: eta = " << muon->eta() );
    hist("hist_muonPt")->Fill( (muon->pt())/1000 );

    if ( muon->muonType() == xAOD::Muon::Combined) numCombinedMuons++;
} // end for loop over muon container

hist("hist_nMuons")->Fill( muons->size() );
hist("hist_nCombinedMuons")->Fill( numCombinedMuons );
```

Note: Since we are using muons here, you of course also have to add the necessary include statement again:

```
#include "xAODMuon/MuonContainer.h"
```

Because we are inheriting from `AthHistogramFilterAlgorithm` it is really that simple to define and fill histograms.

You can compile your package and add these lines to the bottom of your `myJobOptions.py` file to include this histogram algorithm to your athena processing:

Show Hide

```
# Make histogram algorithm
alg = CfgMgr.MyAthHistogramAlg ()
algseq += alg
# Define your output file name and stream name
rootStreamName = "MyFirstHistoStream"
rootFileName    = "myHistosAth.root"
alg.RootStreamName = rootStreamName
alg.RootDirName   = "/MyHists"
# =====
# Define your output root file using MultipleStreamManager
# =====
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
MyFirstHistoXAODStream = MSMgr.NewRootStream ( rootStreamName, rootFileName )
```

Now to test it in Athena, go back to your `run/` directory and run:

```
athena myJobOptions.py 2>&l | tee log.txt
```

After the job completes (successfully!) you should have a new ROOT file `myHistosAth.root`. If you browse that in ROOT you will see a TDirectory `MyHists` that contain all of your histograms.

5. Using a tool

Interlude: How do I know which CP tools to use and how to use them?

This is a question I lost countless hours to in the past, so I find it fairly important to point out this fairly straight forward default course of action you should stick to when encountered with it. The problem is that the search system of the ATLAS twiki is, let's say, not optimal, so you will find yourself lost without the correct link already provided. Fortunately, when it comes to physics CP recommendations, the respective links to the various twiki pages you need are all fairly easily accessible from the `AtlasPhysics` twiki. So I think the best way to get to the analysis recommendations you want can be boiled down to a couple of points:

1. Go to `AtlasPhysics`
2. Identify the CP group in question (e.g. if you want to do something with jets, go to the `Jet/EtMiss` group twiki)

3. Go to their twiki page and find the link to their physics recommendation twiki page(s)
4. Save a lot of time you would have spent using the search engine, desperately trying to find the respective twiki pages (not impossible, but more than often in vain)

Try to navigate your way to the twiki page for the jet cleaning tool we use in the next section. Since we are in transition to rel. 21, this may be not entirely unambiguous:

Twiki for rel. 20.7 (2016) recommendations:

<https://twiki.cern.ch/twiki/bin/view/AtlasProtected/JetEtmisRecommendations2016>

Twiki for rel. 21 (2017) recommendations:

<https://twiki.cern.ch/twiki/bin/view/AtlasProtected/JetEtmisRecommendationsR21>

Optionally, you can also have a look at the PhysicsAnalysisWorkbooks that list all available CP tools for a given release period. Again, since we are in transition between rel. 20.7 and rel. 21, you should look at the twiki pages for both:

Twiki for rel. 20.7 (2016) workbook:

<https://twiki.cern.ch/twiki/bin/view/AtlasProtected/PhysicsAnalysisWorkBookRel20CPRec>

Twiki for rel. 21 (2017) workbook:

<https://twiki.cern.ch/twiki/bin/view/AtlasProtected/PhysicsAnalysisWorkBookRel21CPRec>

Example: JetSelectorTool as an Athena Configurable

The JetSelectorTool is used to select "clean" jets, as recommended on this page:

https://twiki.cern.ch/twiki/bin/view/AtlasProtected/HowToCleanJets2016#Jet_Cleaning_Tool. It is applied to both data and MC events. Here we will show you how to apply this jet cleaning to MC jets in Athena on xAODs.

Now we'll try to make use of the other half of the dual-usefulness of the xAOD Tools. Here we use a `ToolHandle` to an **interface** of the tool. The interface class (naming convention dictates they begin with an "I") may or may not reside in the same athena package as the tool (sometimes the tool itself is also the interface!), but the convention is actually to have tool interfaces collected in one (or a couple of) dedicated package(s). In this case the tool lives in `PhysicsAnalysis/JetMissingEtID/JetSelectorTools`, but the interface it implements lives in `Reconstruction/Jet/JetInterface`, and is called `IJetSelector.h`. To see this, look at the header file of the `JetCleaningTool` [we plan to use](#), and you'll see it inherits from `IJetSelector`, and you can search for that in a code-browser to see where it lives (or look at the include statement a few lines above).

Also, you should note that the interface implements `keep()` method, whereas when we used the tool directly, we used the `accept()` method. You won't be able to see `accept()` via the `ToolHandle`, because it's not in the interface. But thankfully these lines [in the JetCleaningTool](#) tell you that they're basically the same function.

We shouldn't need to check out this `JetInterface` package, it's already in the release.

To use this tool, we need to do the following:

Header file `$TestArea/.../source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.h`

To the header file, at the top with all the includes add the include to the tool interface:

```
#include "JetInterface/IJetSelector.h"
```

Now within our analysis class under `private` let's get a handle on the tool by adding this line:

```
ToolHandle < IJetSelector > m_jetCleaningTool;
```

Interlude: How do I know which CP tools to use and how to use them?

We will use the ToolHandle to access the tool, as it gives a nice way of being able to configure the tool from the job options.

Source code \$TestArea/./source/MyAthenaxAODAnalysis/src/MyAthenaxAODAnalysisAlg.cxx

We will have to update the constructor to look something like this:

```
MyAthenaxAODAnalysisAlg::MyAthenaxAODAnalysisAlg( const std::string& name, ISvcLocator* pSvcLocator
    AthAnalysisAlgorithm ( name, pSvcLocator ),
    m_jetCleaningTool( "JetCleaningTool/JetCleaningTool", this )
{
    //declareProperty( "Property", m_nProperty = 0, "My Example Integer Property" ); //example prop
    declareProperty( "JetCleaningTool", m_jetCleaningTool );
}
```

The text "JetCleaningTool/JetCleaningTool" is basically the "ToolName/MyInstanceOfToolName", you could just as easily do "JetCleaningTool" or "JetCleaningTool/MyJetCleaningTool". The this part makes the tool private, if you don't provide it, the tool will be shared, and belong to the ToolSvc object. Adding the declareProperty is what will allow us to configure this tool from our job options of our algorithm.

In the initialize() method we can retrieve this tool by hand (which initialises it), by adding:

```
ATH_CHECK( m_jetCleaningTool.retrieve() );
```

This is optional though, since tools which are handled by ToolHandles in Athena are automatically initialised (if not done so before) the first time they're used.

In the excute() method in the loop over the jet collection, to access the decision about whether the jet passes the cleaning selection do something like:

```
ATH_MSG_INFO( " jet cleaning = " << (m_jetCleaningTool->keep( *jet ) ) );
```

CMakeLists.txt \$TestArea/./source/MyAthenaxAODAnalysis/CMakeLists.txt

Add the appropriate lines:

Show Hide

```
...
atlas_depends_on_subdirs(
    PUBLIC

    PRIVATE
    Control/AthAnalysisBaseComps
    Event/xAOD/xAODEventInfo
    Event/xAOD/xAODJet
    Event/xAOD/xAODCutFlow
    Reconstruction/Jet/JetInterface
)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
    PUBLIC_HEADERS MyAthenaxAODAnalysis
    INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
    LINK_LIBRARIES ${ROOT_LIBRARIES}
                    AthAnalysisBaseCompsLib xAODEventInfo xAODJet xAODCutFlow Jet
)
...
```

Job options \$TestArea/./run/myJobOptions.py

To configure the tool, and let it know which cleaning criteria you want, you can do it from your job options,

by something like:

```
alg.JetCleaningTool.CutLevel = "LooseBad" # options: "LooseBad" , "TightBad"
```

The things you can configure in your job options are those items listed as `declareProperty` in the constructor of the `JetSelectorTool` [code](#).

You can recompile and rerun your athena job to see the new `INFO` line we have added for each jet in every event.

You could also create an instance of the `JetCleaningTool` in the joboption, and pass that to the `alg` instead (try it instead of the line above, and the line that sets up `alg`):

```
ToolSvc += CfgMgr.JetCleaningTool ("MyTool", CutLevel = "LooseBad")
alg = CfgMgr.MyAthenaxAODAnalysisAlg (JetCleaningTool = ToolSvc.MyTool)
```

... this is an example of a shared tool, meaning you passed the ownership of the tool to the `ToolSvc` after you created it. The algorithm will still pick it up, although you've specified the tool in the algorithm to be private (with the `this` statement), but will make a private copy of it. You can test this behaviour yourself by changing the configuration of the tool after you passed it to the algorithm. So at the very end of your job options add this line and run the code again:

```
ToolSvc.MyTool.CutLevel = "TightBad"
```

Look out for this line in your output (remember you have created a log file, which makes it easier to search):

```
MyAthenaxAODAna... INFO Configured with cut level LooseBad
```

As you can see the `CutLevel` didn't change from its default `LooseBad`. Play around with this, for example remove the `this` statement from the toolhandle in your algorithm (making it public) and after recompiling and rerunning the code you should find that the `CutLevel` which is used has changed to `TightBad`.

Interlude: and

The steps to follow when you try to work with tools and services from inside a configurable (e.g. algorithm) are:

1. `#include "GaudiKernel/ServiceHandle.h"` or `#include "GaudiKernel/ToolHandle.h"`. Note that you actually didn't have to do the include for `ToolHandle` in the example above, since this is already done in the `AthAnalysisAlgorithm` base class definition.
2. `#include "SomePackage/IInterface.h"` - the interface you wish to utilize for the tool you want to instantiate (the tool must inherit from the interface). If needed you could technically also use the header of the tool itself (sometimes necessary when interfaces are not properly updated...).
3. Create a `ServiceHandle< IInterface >` or `ToolHandle< IInterface >`. The constructors are:

```
ServiceHandle< IInterface > myService("ServiceType/ServiceName",name());
ToolHandle< IInterface > myTool("ToolType/ToolName" [, this]);
```

The optional `[,this]` will define a private tool, owned by the algorithm. When we created them as data members of the algorithm class, we initialized them immediately in the constructor, but you can also leave them as an empty toolhandle, via the default constructor. Then it relies on you passing an instance of the tool to the algorithm (the tool is shared if it is empty... see the next section for an example).

4. Optional: In the `initialize()` method of your algorithm, call the handle's `retrieve()` method ... this needs a dot, not an arrow .. it's a method of the handle, not the interface:

```
CHECK( myService.retrieve() ); CHECK( myTool.retrieve() );
```

5. Use the methods of the interface class by pretending `myTool` / `myService` are pointers to the tool/service.

6. Objects and tools for analysis

Muons

Let's create a loop over the muon container for each event, in our source code in the `execute()` method add the following lines:

```
// Retrieve the muons:
const xAOD::MuonContainer* muons = 0; // a const here as we will not modify this container
ATH_CHECK( evtStore()->retrieve( muons, "Muons" ) );

// Loop over them:
for( const xAOD::Muon* muon : *muons ) {
    ATH_MSG_INFO( "    Selected muon: eta = " << muon->eta() );
} // end for loop over muon container
```

At the top of our `cxx` file add the include to the muon container header:

```
#include "xAODMuon/MuonContainer.h"
```

And, as always, add the appropriate lines to

```
$TestArea/../../source/MyAthenaxAODAnalysis/CMakeLists.txt:
```

Show Hide

```
...
atlas_depends_on_subdirs(
    PUBLIC

    PRIVATE
    Control/AthAnalysisBaseComps
    Event/xAOD/xAODEventInfo
    Event/xAOD/xAODJet
    Event/xAOD/xAODCutFlow
    Reconstruction/Jet/JetInterface
    Event/xAOD/xAODMuon
)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
    PUBLIC_HEADERS MyAthenaxAODAnalysis
    INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
    LINK_LIBRARIES ${ROOT_LIBRARIES}
    AthAnalysisBaseCompsLib xAODEventInfo xAODJet xAODCutFlow Jet
)
...
```

You can recompile that and take a look at the output.

7. Advanced example: creating an algorithm for the muon systematic variations

Introduction and setup of the code skeleton

Since we are running on Monte Carlo simulation, we need to also apply final calibrations and also systematic variations thereof to the muons that we use. The variations of the four-momentum are stored using the shallow-copy feature of the xAOD. This feature is referring back to the original object's auxiliary container

for every requested attribute, except for an attribute that was specifically set for the shallow copy container. In our case, this would of course be the pt of the muon that we set for every variation. This technique can save considerable amount of size in the output file, depending on the setup. There is one issue that you have to keep in mind though: you must not attempt to change the order of the muons in the original or in the shallow copy container nor must you try to remove objects from either. If you do this, you will break the correct association between the two.

We need to write some code to do this analysis (that's right)! And for this part, we will not add it to the existing algorithm that we wrote before, but rather write a new one. The reason is that this is an encapsulated task that we may want to use in different places (and others may want to reuse). And if it is fully encapsulated in its own algorithm, this becomes very easy since one can configure and schedule it wherever you want in the python job option configuration.

Let's make our life easy again and use again the handy AthenaScripts to automatically generate the skeleton of the code. For this, go to the `base` directory of your package:

```
cd $TestArea/../../source/MyAthenaxAODAnalysis
acmd cmake new-analysisalg MyAthMuonCalibrationSmearingAlg
```

This will generate the source code file (`MyAthMuonCalibrationSmearingAlg.cxx`) and the header file (`MyAthMuonCalibrationSmearingAlg.h`), both in the `src/` directory of your package. This location is absolutely appropriate here. As before, only if you ever plan to inherit from your new code, you would need to move the header file into this new directory:

`$TestArea/../../source/MyAthenaxAODAnalysis/MyAthenaxAODAnalysis`. But we won't do this here.

The shell printout will give some items that you'd have to take care of, i.e. things to add to the `CMakeLists.txt` file. However, the way the package skeleton was build earlier, we actually don't have to change anything for now in order to properly compile our new code. So simply compile with

```
cmake --build $TestArea
```

Editing the header (.h) file

Now, lets fill our code with some life. Let's first edit the header file. Open

`$TestArea/../../source/MyAthenaxAODAnalysis/src/MyAthMuonCalibrationSmearingAlg.h` and make sure you have the following includes:

Show Hide

```
// STL includes
#include <string>
#include <vector>
#include <utility>

// FrameWork includes
#include "PATInterfaces/SystematicSet.h"

// Tool    interfaces
#include "PATInterfaces/ISystematicsTool.h"
#include "MuonAnalysisInterfaces/IMuonCalibrationAndSmearingTool.h"
```

Next, still in the header file, you need to create a few new private members:

Show Hide

```
////////////////////////////////////
// Private data:
////////////////////////////////////
private:
    /// @name The properties that can be defined via the python job options
```



```

/// @{

/// The input container name
StringProperty m_inCont;

/// The output container name
StringProperty m_outCont;

/// The names of all systematic variations to be applied
StringArrayProperty m_muCalibSmearSysNames;

/// The ToolHandle for the muon calibration and smearing tool
ToolHandle<CP::IMuonCalibrationAndSmearingTool> m_muCalibSmearTool;

/// The ToolHandle for the systematic interface of the muon calibration and smearing tool
ToolHandle<CP::IMuonCalibrationAndSmearingTool> m_muSysCalibSmearTool;

/// @}

private:

/// @name Truly private internal data members
/// @{

/// The vector of all momentum systematics and the corresponding container-name post-fixes
std::vector< std::pair< CP::SystematicSet, std::string > > m_p4SystVarNameVec;

/// @}

```

Some explanations are in order!

1. First, we use `"/" /` to denote code comments and also have these block delimiters `"/" / @{"` and `"/" / @}"`. These are doxygen commands. Doxygen is a code documentation engine. Doing this is just good practice. All code that ends up in an Athena release is automatically documented using these Doxygen commands. The page where all Doxygen code documentation resides is [here](#). More information on useful things can be found [here](#).
2. Second, we created a few private data members of type `StringProperty`. These, we will use to declare what we want to be able to configure from the python job option side. This will be done in the constructor. The `StringProperty` will simply hold a `std::string` while the `StringArrayProperty` will hold a `std::vector< std::string >`.
3. Third, we create two tool handles. Tool handles are there to give you a handle to an instance of a tool. The first one is a tool handle to the `CP::IMuonCalibrationAndSmearingTool` interface, i.e., the interface that the muon group defined for their calibration and smearing tool. We also create a tool handle to the `CP::ISystematicsTool` interface, i.e., the common interface defined by the Analysis Software Group (ASG) for handling systematics. We will use these two tool handles as public tool handles where both will use the SAME tool instance. This is because the muon calibration and smearing tool implements these two interfaces.
4. Fourth: we have this vector of pairs of systematic sets and strings. This, we will use to be able to figure out during the initialization of the algorithm which systematics we want to run and which strings we want to use as a suffix for the name of the output collection for the muons on which this systematic variation was applied. We will write out one muon collection (as a shallow copy container) per systematic variation.

Editing the source (.cxx) file

Next, lets start editing the source code in the

`$TestArea/./source/MyAthenaxAODAnalysis/src/MyAthMuonCalibrationSmearingAlg.cxx` file. Here, we need to include the following headers:

Editing the header (.h) file

Show Hide

```
// EDM includes
#include "xAODCore/ShallowCopy.h"
#include "xAODMuon/Muon.h"
#include "xAODMuon/MuonContainer.h"
#include "xAODMuon/MuonAuxContainer.h"
#include "xAODParticleEvent/IParticleLink.h"

// Tool includes
#include "PATInterfaces/SystematicCode.h"
#include "PATInterfaces/SystematicVariation.h"
```

Your constructor should look like this:

Show Hide

```
// Constructors
//////////
MyAthMuonCalibrationSmearingAlg::MyAthMuonCalibrationSmearingAlg( const std::string& name, ISvcLocator* pSvcLocator ) :
    AthAnalysisAlgorithm ( name, pSvcLocator ),
    m_muCalibSmearTool("CP::IMuonCalibrationAndSmearingTool"), // This is a handle to a public tool
    m_muSysCalibSmearTool("CP::ISystematicsTool")                // This is a handle to a public tool
{

    declareProperty("InputContainer", m_inCont="", "Input container name" );
    declareProperty("OutputContainer", m_outCont="",
                    "The name of the output container with the deep copy of input objects" );

    declareProperty("MuonCalibrationTool", m_muCalibSmearTool,
                    "The ToolHandle for the muon calibration and smearing tool" );

    declareProperty("MuonCalibrationSystematicsTool", m_muSysCalibSmearTool,
                    "The ToolHandle for the systematic interface of the muon calibration and smearing tool" );

    declareProperty("MomentumSystematicVariations", m_muCalibSmearSysNames,
                    "The names of all systematic variations to be applied" );

}
```

Your initialize method should look like this:

Show Hide

```
// Athena Algorithm's Hooks
//////////
StatusCode MyAthMuonCalibrationSmearingAlg::initialize()
{
    ATH_MSG_DEBUG ("Initializing " << name() << "...");

    // Print the configuration to the log file
    ATH_MSG_DEBUG( "Using: " << m_inCont );
    ATH_MSG_DEBUG( "Using: " << m_outCont );
    ATH_MSG_DEBUG( "Using: " << m_muCalibSmearSysNames );
    ATH_MSG_DEBUG( "Using: " << m_muCalibSmearTool );
    ATH_MSG_DEBUG( "Using: " << m_muSysCalibSmearTool );

    // Perform some sanity checks on the given container names
    if ( m_inCont.value().empty() || m_outCont.value().empty() ) {
        ATH_MSG_ERROR("Wrong user setup! You need to give a valid name for both the InputContainer and OutputContainer");
        return StatusCode::FAILURE;
    }

    // Abort on an unchecked systematics code
    // CP::SystematicCode::enableFailure();

    // Retrieve the tools
    ATH_CHECK(m_muCalibSmearTool.retrieve());
```

```

// Figure out what systematics are available and recommended
if ( msgLvl(MSG::DEBUG) || msgLvl(MSG::VERBOSE) ) {
    CP::SystematicSet affSys = m_muSysCalibSmearTool->affectingSystematics();
    ATH_MSG_DEBUG("Have " << affSys.size() << " affecting systematics with name "
                  << affSys.name() << " for tool " << m_muSysCalibSmearTool->name() );
    CP::SystematicSet recSys = m_muSysCalibSmearTool->recommendedSystematics();
    ATH_MSG_DEBUG("Have " << recSys.size() << " recommended systematics with name "
                  << recSys.name() << " for tool " << m_muSysCalibSmearTool->name() );
}

// Set up the internal vector of systematics and container name post-fixes,
// starting with the nominal one. First, clear it. Then, add the nominal, then systematics
m_p4SystVarNameVec.clear();
m_p4SystVarNameVec.push_back( std::make_pair( CP::SystematicSet(""), "" ) );
for ( const auto& sysName : m_muCalibSmearSysNames.value() ) {
    CP::SystematicVariation sysVar = CP::SystematicVariation(sysName);
    if ( m_muSysCalibSmearTool->isAffectedBySystematic(sysVar) ) {
        CP::SystematicSet sysSet{sysVar};
        m_p4SystVarNameVec.push_back( std::make_pair( sysSet, "__"+sysName ) );
        ATH_MSG_DEBUG("Adding systematic variation with name " << sysName );
    }
    else {
        CP::SystematicSet affSys = m_muSysCalibSmearTool->affectingSystematics();
        ATH_MSG_WARNING("Couldn't find systematic variation with name " << sysName
                       << " amongst the affected systematics: " << affSys.name() );
        return StatusCode::FAILURE;
    }
}
} // End: adding all systematic variations to be processed

return StatusCode::SUCCESS;
}

```

In finalize, we can explicitly clean up and release the two tool, which is optional though:

Show Hide

```

// Release the tools
ATH_CHECK(m_muCalibSmearTool.release());
ATH_CHECK(m_muSysCalibSmearTool.release());

```

And finally, the meat is in the execute method that should look like this:

Show Hide

```

StatusCode MyAthMuonCalibrationSmearingAlg::execute()
{
    ATH_MSG_DEBUG ("Executing " << name() << "...");

    // Open the input container
    const xAOD::MuonContainer* inCont;
    ATH_CHECK( evtStore()->retrieve( inCont, m_inCont.value() ) );

    // Now, we will loop over the 4-momentum systematics to be applied
    for ( const auto& systVarAndContName : m_p4SystVarNameVec ) {
        const CP::SystematicSet& systSet = systVarAndContName.first;
        const std::string& contSuffix = systVarAndContName.second;

        // Let's create a shallow copy of the const input container
        // auto = std::pair< xAOD::MuonContainer*, xAOD::ShallowAuxContainer* >;
        auto inContShallowCopy = xAOD::shallowCopyContainer( *inCont );
        ATH_CHECK( evtStore()->record( inContShallowCopy.first, m_outCont.value() + contSuffix ) );
        ATH_CHECK( evtStore()->record( inContShallowCopy.second, m_outCont.value() + contSuffix + "Au" );

        // Set the tool state to apply a systematic variation, if it is not empty
        if ( !(contSuffix.empty()) ) {
            if( m_muSysCalibSmearTool->applySystematicVariation( systSet ) != CP::SystematicCode::Ok )

```

```

    ATH_MSG_ERROR("Cannot configure MuonCalibrationAndSmearingTool for systematic variation "
                  << systSet.name() );
    return StatusCode::FAILURE;
}
ATH_MSG_DEBUG("Going to run muon p4 systematic variation " << systSet.name() );
}

// Create accassors for the ElementLink to the original muon.
// Unfortunately, the missing ET tool will need this
static SG::AuxElement::Accessor< xAOD::IParticleLink > accSetOriginLink ("originalObjectLink")

// Loop over all Muons in the shallow-copy container
for ( xAOD::Muon* muon : *(inContShallowCopy.first) ) {
    ATH_MSG_VERBOSE("Now iterating over the muon shallow copy container... at index=" << muon->index());
    const double originalPt = muon->pt();
    if ( m_muCalibSmearTool->applyCorrection(*muon) == CP::CorrectionCode::Error ) {
        ATH_MSG_ERROR("MuonCalibrationAndSmearingTool reported a CP::CorrectionCode::Error");
        return StatusCode::FAILURE;
    }
    // Create the ElementLink to the original muon. Unfortunately, needed my met tool
    const xAOD::IParticleLink originLink( *inCont, muon->index() );
    accSetOriginLink(*muon) = originLink;
    ATH_MSG_VERBOSE("Original pt: " << originalPt << " MeV, new pt: " << muon->pt()
                  << " MeV, original-new pt: " << originalPt - muon->pt() << " MeV " );
} // End: loop over Muons

} // End: loop over systematic variations

return StatusCode::SUCCESS;
}

```

Compiling your code

We added a lot of stuff to our new algorithm, so we need to make sure that cmake is aware of it. Make sure that your CMakeLists.txt file has the appropriate lines:

Show Hide

```

...
atlas_depends_on_subdirs (
    PUBLIC

    PRIVATE
    Control/AthAnalysisBaseComps
    Event/xAOD/xAODEventInfo
    Event/xAOD/xAODJet
    Event/xAOD/xAODCutFlow
    Reconstruction/Jet/JetInterface
    Event/xAOD/xAODMuon
    Event/xAOD/xAODParticleEvent
    Event/xAOD/xAODCore
    PhysicsAnalysis /AnalysisCommon/PATInterfaces
    PhysicsAnalysis /MuonID/MuonIDAnalysis/MuonMomentumCorrections
)
...
atlas_add_library( MyAthenaxAODAnalysisLib src/*.cxx
    PUBLIC_HEADERS MyAthenaxAODAnalysis
    INCLUDE_DIRS ${ROOT_INCLUDE_DIRS}
    LINK_LIBRARIES ${ROOT_LIBRARIES}
                  AthAnalysisBaseCompsLib xAODEventInfo xAODJet xAODCutFlow Jet
)
...

```

You should be able to compile the new code now.

Editing the job options

Since we want to calibrate and smear our muons before we perform a pt selection on them, we need to add the calibration and smearing of the muons BEFORE we schedule our `MyMuonSelectionAlg`, but of course AFTER we create our sequence `AthAlgSeq`. There, add the creation of the muon calibration and smearing tool instance, add it to the tool service, and create and configure also an instance of our new algorithm:

Show Hide

```
# Create an instance of the MuonCalibrationAndSmearingTool and configure it
ToolSvc += CfgMgr.CP__MuonCalibrationAndSmearingTool( "MuonCalibSmearTool",
                                                       OutputLevel = INFO
                                                       )

# Create the algorithm that will apply the calibrations/smearings provided by
# the tool and write out the new container(s)
algseq += CfgMgr.MyAthMuonCalibrationSmearingAlg ( "HWWMuonCalibrationSmearingAlg",
                                                    #OutputLevel                = VERBOSE,
                                                    MuonCalibrationTool            = ToolSvc.MuonCa
                                                    MuonCalibrationSystematicsTool = ToolSvc.MuonCa
                                                    InputContainer                  = "Muons",
                                                    OutputContainer                 = "CalibMuons",
                                                    MomentumSystematicVariations   = [ "MUON_ID__1do
                                                                               "MUON_MS__1up"
                                                                               "MUON_SAGITTA_
                                                                               "MUON_SAGITTA_

                                                    )
```

Make sure that your subsequent `MyMuonSelectionAlg` uses as an `InputContainer` the "CalibMuons" now instead of the "Muons"!

8. Using ganga to send your job to the local batch system

NOTE: As of ATLAS-D 2015, this section is not updated anymore. The instructions in general probably work, but the given example most likely will not. You can have a look at these instructions, which cover batch and grid submission with ganga.

You can use ganga not only to submit jobs to the grid, but also to submit jobs to your local batch system. The batch system at the NAF is using the SGE (Sun Grid Engine) batch system. After having setup your Athena release with `asetup`, you need to source the ganga setup:

```
localSetupGanga
```

In order to submit your `myJobOptions.py` Athena job to the batch system at the NAF using ganga, create a new file in your `$TestArea/./run` folder. Let's call it `submitSGE.py` with this content:

```
#!/usr/bin/env python

import user # look for .pythonrc.py for user-defined initial settings
import os, sys

# Get the right job options that athena should use
jobOpts = os.getenv("TestArea") + '/run/myJobOptions.py'

# Note that, as we have the Athena environment currently installed,
# you will need to do the following to allow the merger to work:
config['ROOT']['path'] = os.environ['ROOTSYS']

# This is so that all your resulting output files will end up in one directory
config['Athena']['SingleDirForLocalOutput'] = True
```

```

# This, you HAVE TO set such that ganga knows that you are working at DESY!
# Alternatively, you can put this (or any of the above configs) into your
# ${HOME}/.gangarc file.
config['DQ2']['DQ2_LOCAL_SITE_ID'] = 'DESY-HH_SCRATCHDISK'

# Set up the ganga job
j = Job()
j.name = "SGETest_01"

# Define what the job should do
j.application = Athena()
j.application.atlas_release = os.getenv("AtlasVersion")
j.application.option_file = jobOpts
j.application.athena_compile = False
j.application.prepare()

# Tell the job which input dataset to use. Here, we use a DC14 Z-mumu example
# NOTE: This must be available at the DESY Tier-2
j.inputdata = DQ2Dataset ()
j.inputdata.dataset = "mc14_8TeV.147807.PowhegPythia8_AU2CT10_Zmumu.merge.AOD.e1852_s1896_s1912_r

# Define what the output dataset should be
j.outputdata = ATLASOutputDataset ()
j.outputdata.outputdata = [ 'TutoHistFile.root', 'myXAOD.pool.root' ]
#j.outputdata.location = os.getenv("TestArea")+'/run/output/'
j.outputdata.location = '/nfs/dust/atlas/user/kkoeneke/TutoOut/'

# Set up the splitting (and merging) of the job into smaller sub-jobs (parallelize)
j.splitter = AthenaSplitterJob ()
j.splitter.numsubjobs = 20

# Tell the job to use the local batch system (SGE at DESY)
j.backend = SGE()
#j.backend.extraopts='-l distro=sld6 -l cvmfs -l h_vmem=2G'
j.backend.extraopts=' -l cvmfs -l h_vmem=2G'

# Actually submit the job
j.submit()

```

Then, go back to your `run/` directory. Create the directory where you schedule your resulting files to go (`/nfs/dust/test/atlas/user/kkoeneke/test1/` in the above example) and execute ganga with this new submit script:

```

cd $TestArea/./run
mkdir <YOUR/OUTPUT/PATH/>
ganga

```

Once you are in the ganga command prompt, execute your newly created submit script:

```
In [1]:execfile("./submitSGE.py")
```

Once all subjobs are completed, they will appear in your `<YOUR/OUTPUT/PATH/>` directory.

Part D: Using existing Athena components

This hands-on tutorial will lead you through some typical analysis tools used for analysing an xAOD in Athena (or ROOT-standalone, for the most parts). We will show you how to setup Athena, setup your analysis skeleton, loop over xAOD events and access content, use CP recommended tools, and output histograms and smaller xAODs.

We will setup the ATLAS software using CVMFS, and access xAODs that are accessible from the cern afs.

1. Prepare your setup

1. In case you're not logged in to a worker node anymore, do so as described here.

2. If you still have your terminal open from the previous section, you can skip this step. In case that you haven't done the first time analysis environment setup as described here, please do so now. Remember, if you want to use a new terminal and/or have to log in to a worker node again, go through the necessary steps to resume your previous setup:

```
setupATLAS
cd xAODSoftwareTutorial/build
asetup --restore
source $TestArea/*/setup.sh
```

2. Using existing features to make simple plots

In this section, you will learn how you can simply schedule existing algorithms and tools in order to: **1.** place a few simple cuts on the muons from the input file, **2.** build $Z \rightarrow \mu\mu$ candidates out of the selected muons, **3.** make a simple event selection, **4.** and make some plots for every stage of your event selection.

Creating a jobOption

Since Athena is steered using the so-called python job options, you have to create a simple one to load your input file(s) and schedule what you need. Go to the `$TestArea/./run` directory, which is a good location to execute Athena from. Athena tends to create a few files, which we rather want to have in a separate folder.

```
cd $TestArea/./run
```

Now we are in the `run/` directory, so let's create an empty jobOptions file (e.g. `myJobOptions.py`) with just one line in it:

```
theApp.EvtMax = 10
```

Recall that `theApp` is one of the objects Athena gives you for free. It's the application manager, which steers the execution of pretty much everything and it has an `EvtMax` property to say how many events should be processed. Set it equal to `-1` to process all events (don't do that now though, because we haven't specified any input, so you'll just create an infinite loop).

Run Athena with these newly created job options:

```
athena myJobOptions.py 2>&1 | tee log.txt
```

Note the extension after `athena myJobOptions.py`, which pipes the console output in `log.txt` as well (`2>&1` just makes sure that stdout and stderr are merged). You'll find it very useful to produce log-files of your runs,

since the athena output can get quite extensive.

As a side note, you can also steer the maximum number of events to be processed with a command line argument, which is particular convenient if you want to quickly test something:

```
athena --evtMax=100 myJobOptions.py 2>&1 | tee log.txt
```

The output you get is the minimal output from an Athena job. It's worth taking some time later on to familiarize yourself with the sections (ask if you're curious about any parts). But by far the **most important** line(s) to know about is this one and the ones which follow, which would print the members of the sub-AthSequencers if there are any:

```
AthMasterSeq          INFO Member list: AthSequencer/AthAlgSeq, AthSequencer/AthOutSeq, AthSequencer/AthRegSeq
```

This is telling you what algorithms will execute each event in the loop. For now you can ignore `AthOutSeq` and `AthRegSeq`, and focus on `AthAlgSeq`, which is the sequence where you usually add algorithms. If you take a look at the log file from a typical reconstruction job, you should now see why those jobs can be very slow.

We can try to change the behaviour of this empty loop job, by grabbing one of the Configurables that will be used by the loop (the `AthenaEventLoopMgr` specifically) and setting a property of it (Configurables are Athena algorithms, tools, or services that can be configured from the python side). Add the following to your job options (`myJobOptions.py`) and rerun to see what this property controls:

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

Click for some advanced commentary about the above line...[▢](#) [Hide▢](#)

The `AthenaEventLoopMgr` doesn't actually exist at the time the joboption is included. We are creating an instance of it. You can look at the `Configurable.allConfigurables.items()` list of instantiated configurables and see `AthenaEventLoopMgr` is not there (before this line). If you don't add (or otherwise reference) the instance we create to something that will survive past the end of the joboption file, then it will be automatically garbage collected by python. So to ensure this instance, with the non-default value of `EventPrintoutInterval` set (it's been set via the kwargs trick of the python constructor) survives to be used in the event loop, we have added it to the ServiceManager (`svcMgr`). This is appropriate, because `AthenaEventLoopMgr` is a service (print it to see). We will explain a little later in the tutorial what the `CfgMgr` is.

Your joboptions file should look like this now...[▢](#) **Your joboptions file should look like this now**[▢](#)

```
theApp.EvtMax = 10
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

Specifying input files

Files are input into athena via an `EventSelector`, which is a service. You need a magic incarnation to make the appropriate `EventSelector` appear for the input file you are using. For xAOD and POOL files (basically everything except D3PD, i.e. raw NTuples) you would use the Athena service that deals with the POOL conversion from the stuff on disk to what will be in memory. Add the line below to **the top** (it tends to interfere with the previous printout interval line otherwise) of your python job option file:

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

Some more information on `ReadAthenaPool`[▢](#) [Hide▢](#)

As mentioned, the line above is what you include in your joboptions file in order to read POOL files (which xAODs are). There are also other read modes, which are more optimised for analysis processing patterns, i.e. are usually faster reading xAODs for analysis. However, with the current rel. 21 builds, these modes lead to

segmentation violations when writing output xAODs, thus we'll stick to the general POOL read mode for now.

For reference, the more performant read mode would be set up like this:

```
import AthenaRootComps.ReadAthenaxAODHybrid
```

Click for the raw NTuple version... [Hide](#)

```
import AthenaRootComps.ReadAthenaRoot
svcMgr.EventSelector.TupleName="physics" #change if you want to read a different TTree from the i
```

This creates an EventSelector (with instance name `EventSelector`) in the `svcMgr`, which has a property that lets you specify the input files, as a list, again in your python job option file:

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
```

This is all you need to get the content of the input files made available to the `StoreGateSvc` (also known as the `EventStore`), which is the service your algorithms (and tools) will interact with to get at the needed information for every event (it's the equivalent of the `TEvent` object in standalone ROOT).

Your joboptions file should look like this now... [Your joboptions file should look like this now](#)

```
theApp.EvtMax = 10
```

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
```

Interlude: building packages from the athena repository on top of the release

Currently the migration to rel. 21 is still ongoing, which introduces some technical hurdles for the next couple of subsections of this tutorial. To keep it brief, some of the utilities packages that we will use are currently not included in the rel. 21 build of `AthAnalysis`. However, they compile and work perfectly fine. This would be a prime opportunity to introduce you to git and how you properly setup your athena repository in order to checkout certain packages and compile them on top of the `AthAnalysis` release that we are using. I very much encourage to have a look at this at some point, because it's almost certain that you have to deal with it sooner rather than later. However, for beginners the various steps you have to go through can be quite a hurdle to take and possibly cost a lot of time until it's working properly (which we don't really have during a one day tutorial).

So there are two options to proceed. If you want to give the "proper" setup a go or even already went through the necessary steps to work with the athena git repository, have a look at the drop-down button in the next line.

Let's give git a try [Let's give git a try](#)

If this is the first time you want to use git in ATLAS, you should log in to <https://gitlab.cern.ch/> with your normal CERN account credentials. This makes sure that you are properly registered on the CERN gitlab.

First, load git with the `lsetup` command that is available after `setupATLAS` was executed:

```
lsetup git
```


Furthermore, make sure that you configure git according to these recommendations^[7] and also fork the athena repository according to the instructions on the top of this page^[7].

What we're going to do is setting up a "sparse build" of athena inside the `source` folder of our setup (this is for **lxplus**, see the drop-down button for the NAF):

```
cd $TestArea/./source
git atlas init-workdir https://:@gitlab.cern.ch:8443/atlas/athena.git
# at this point, if you get a message "remote: HTTP Basic: Access denied", try updating your kerb
```

In case we are not on lxplus^[7] In case we are not on lxplus^[7]

If we are not on lxplus, it might not be the best idea using the kerberos authentication. I would recommend to store ssh keys in gitlab to authenticate yourself from your laptop/machine, but for now you can also just use the https git link and authenticate directly by typing in your password. This is a bit annoying because you have to provide your password every time you want to checkout packages or make commits. But for this tutorial this will suffice. Instead of the last line above, just type this:

```
git atlas init-workdir https://yourusername@gitlab.cern.ch/yourusername/athena.git
```

Here, `yourusername` is your CERN account user name (note that it's twice in the command line).

As a note on what happens here. Assuming we are on lxplus, we are using the kerberos credentials to authenticate ourselves to the git server. Also, although the git link points to the `atlas/athena` repository, the `git atlas` command will actually set up your fork of the athena repository. However, since it's a sparse setup, you'll find that the created athena folder only has one directory `athena/Projects`.

We'll now make sure that we're actually in the correct tag of the repository (`$AtlasVersion` is an environment variable set by `asetup`, in our case `21.2.88`):

```
cd $TestArea/./source/athena
git checkout release/$AtlasVersion
```

Now, add the two packages we need:

```
git atlas addpkg EventUtils
git atlas addpkg HistogramUtils
```

We also need to add a file `package_filters.txt` in the `source` directory, since we actually don't want to build the contents of the `Projects` folder. So open `$TestArea/./source/package_filters.txt` in an editor and add the line

```
- athena/Projects/.*
```

That's all, now we just need to configure our environment again, compile and build:

```
cmake $TestArea
cmake --build $TestArea
```

This is actually one of the cases where you have to run the `cmake` configuration again (`cmake $TestArea`). This is necessary every time you checkout a package from the athena repository, that you want to build locally.

Otherwise, here are the shortcut instructions on what you have to do before moving on (assuming you're on the NAF):

```
cd $TestArea/./source/
cp -r /nfs/dust/atlas/user/brendlik/Tutorial/Aux/athena .
```

```
cp /nfs/dust/atlas/user/brendlik/Tutorial/Aux/package_filters.txt .
cmake $TestArea
cmake --build $TestArea
```

In case we are on lxplus  In case we are on lxplus 

```
cd $TestArea/../../source/
cp -r /afs/cern.ch/work/t/tmaier/public/TutorialAux/athena .
cp /afs/cern.ch/work/t/tmaier/public/TutorialAux/package_filters.txt .
cmake $TestArea
cmake --build $TestArea
```

What we've done here is simply copy the necessary Athena packages into our `source` directory (in the folder structure that is also in the athena repository) and build them on top of the release that we're using. This is actually one of the cases where you have to run the cmake configuration again (`cmake $TestArea`). This is necessary every time you checkout a package from the athena repository, that you want to build locally.

Selecting muons and building Z-> $\mu\mu$ candidates

First, we need to create a sequence that will run all our algorithms. We will fetch for this task the `AthAlgSeq`, i.e., one of the existing master sequences where one should attach all algorithms. Add this at the bottom of your `myJobOptions.py` file:

```
# Fetch the AthAlgSeq, i.e., one of the existing master sequences where one should attach all alg
algseq = CfgMgr.AthSequencer ("AthAlgSeq")
```

To this sequence, we want to attach an instance of an existing algorithm class to select muons above a certain pt threshold, say 15 GeV. Add these lines at the bottom of your `myJobOptions.py` file:

```
# Select muons above a pt threshold and
# create an output muon container only with the selected muons
algseq += CfgMgr.ParticleSelectionAlg ( "MyMuonSelectionAlg",
                                         InputContainer      = "Muons",
                                         OutputContainer      = "SelectedMuons",
                                         Selection             = "Muons.pt > 15.0*GeV"
                                         )
```

What happens here is that you create an instance of the `ParticleSelectionAlg` algorithm class, give it an instance name "MyMuonSelectionAlg", configure it to read from the input file the muon collection with name "Muons", place a selection on all of its muons to have a transverse momentum larger than 15.0 GeV, and finally, create for every event an output muon container with only the selected muons from the original container, called "SelectedMuons". In the end, this configured algorithm instance is added to your sequence. And all of this in one go. The "Selection" here is using the string parsing engine that is also used in the generic skimming tool from the Derivation Framework.

Let's now use these selected muons to build viable Z-> $\mu\mu$ candidates. For this, we must make all possible combinations for every event and create finally a list of Z boson candidates and record it. This can be done using the existing `ParticleCombinerAlg`. Once again, at the end of your `myJobOptions.py` file, add these lines:

```
# Build all possible di-muon combinations and call the result viable Z-boson candidates
algseq += CfgMgr.ParticleCombinerAlg ( "MyZmumuBuilderAlg",
                                         InputContainerList = [ "SelectedMuons", "SelectedMuons" ],
                                         OutputContainer    = "ZmumuCands",
                                         SetPdgId            = 23 # This is a Z boson
                                         )
```

Here, we choose to feed this algorithm twice the selected muons since we are trying to reconstruct a two-body decay. If we wanted to build candidates for a three-body decay, the list of input containers would have three

names. We call the list of viable Z bosons "ZmumuCands" and assign a PDG ID of 23 to it.

Run your myJobOptions.py file again. You will find that the two algorithms we've added to the AthAlgSeq show up in it's member list (and thus are executed when this sequence is run):

```
AthAlgSeq          INFO Member list: ParticleSelectionAlg/MyMuonSelectionAlg, ParticleCombinerAlg
```

Your joboptions file should look like this now...  **Your joboptions file should look like this now** 

```
theApp.EvtMax = 10
```

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 5)
```

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV
```

```
# Fetch the AthAlgSeq, i.e., one of the existing master sequences where one should attach all alg
algseq = CfgMgr.AthSequencer ("AthAlgSeq")
```

```
# Select muons above a pt threshold and
```

```
# create an output muon container only with the selected muons
```

```
algseq += CfgMgr.ParticleSelectionAlg ( "MyMuonSelectionAlg",
                                         InputContainer      = "Muons",
                                         OutputContainer      = "SelectedMuons",
                                         Selection             = "Muons.pt > 15.0*GeV"
                                       )
```

```
# Build all possible di-muon combinations and call the result viable Z-boson candidates
```

```
algseq += CfgMgr.ParticleCombinerAlg ( "MyZmumuBuilderAlg",
                                         InputContainerList = [ "SelectedMuons", "SelectedMuons" ],
                                         OutputContainer      = "ZmumuCands",
                                         SetPdgId              = 23 # This is a Z boson
                                       )
```

Create an output ROOT file for the resulting histograms

We will have to create here an output ROOT file that will contain our histograms. Open your myJobOptions.py file again, if you don't have it still open from the previous section. At the end of this file, add these lines:

```
# =====
# Define your output root file holding the histograms using MultipleStreamManager
# =====
rootStreamName = "TutoHistStream"
rootFileName=   "TutoHistFile.root"
rootDirName=    "/Hists"
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
MyFirstHistoXAODStream = MSMgr.NewRootStream ( rootStreamName, rootFileName )
```

The MultipleStreamManager is a python component that can manage to write out several independent files at the same time. Here, we create a basic ROOT output file with name "TutoHistFile.root" and the python object (the output stream object) that manages the file writing is called "TutoHistStream". Since we are at it, we also declare in which ROOT TDirectory the histograms that we will produce should end up, i.e., in the TDirectory called "Hists".

Booking simple cuts and simple histograms for every cut

Since it is often very useful in an analysis to fill the same kind of histograms after each cut stage (in order to see how certain shapes evolve during the analysis selection), we will show you here how this is done automatically. This management will be done using an interplay of the HistogramManager and the

AthAnalysisSequencer (more on this below).

First, we will create an instance of a helper class that will manage all the histograms that we create (actually, all the histogram tools). We tell this class that to which output ROOT file the histograms should be written and to which TDirectory inside that output ROOT file by adding these lines at the end of your myJobOptions.py file:

```
# Now, import the new histogram manager and histogram tool
from HistogramUtils.HistogramManager import HistogramManager as HistMgr
histMgr = HistMgr ( "MyHistMgr",
                    RootStreamName = rootStreamName,
                    RootDirName    = rootDirName
                    )
```

Note that we used a python feature to rename HistogramManager to HistMgr since we want to be lazy in terms of typing.

We also need to declare the histograms that we want to fill and how to fill them.

```
# Import the 1-d and 2-d histograms from ROOT
from ROOT import TH1F, TH2F
# Adding a few histograms to the histogram manager
# Note the different methods of doing this (they are all equivalent)
from HistogramUtils.HistogramTool import HistogramTool as HistTool
histMgr += ( TH1F ("mueta", "#eta^{#mu}", 50, -2.7, 2.7), "Muons.eta" )
histMgr.add( TH1F ("mupt", "p_{t}^{#mu}", 50, 0.0, 100.0), "Muons.pt / GeV " )
histMgr.add( TH1F ("Zmass", "m^{Z}", 50, 50.0, 150.0), "ZmumuCands.m / GeV " )
histMgr += HistTool ( TH2F ("muptvsmueta", "p_{t}^{#mu} vs. #eta^{#mu}", 50, 0.0, 100.0, 50, -2.7
```

What we actually did here in the first two lines of the last block is to create a template consisting of a histogram (any ROOT histogram works here) and a string defining how this particular histogram should be filled (again, using the string parsing engine developed for the generic skimming tool from the Derivation Framework). The HistogramManager will then create an appropriate histogram tool automatically. The last line above actually shows how to explicitly create an instance of a histogram tool.

Now, the HistogramManager that we created (with the name "MyHistMgr") has a list of histogram tools that we can pass on to whoever needs it.

Let's now declare a sub-sequence and attach it to the main sequence. But here, we won't declare just any kind of sub-sequence, no, it is a special one. The AthAnalysisSequencer behaves as any other sequence. But in addition, one can schedule histograms with it that get cloned and automatically filled for every algorithm that you attach to this instance of the sub-sequence. We ask the HistogramManager to provide the list of histogram (tools) here:

```
# =====
# Create a subsequence:
# Remember that a subsequence stops its execution for a given event after an algorithm
# that declares that that event doesn't pass a certain selection. This special type of
# sub-sequence additionally handles histogram booking, cloning, and scheduling.
# =====
subSeq = CfgMgr.AthAnalysisSequencer ("AnaSubSeq",
                                      HistToolList = histMgr.ToolList()
                                      )
algseq += subSeq
```

Remember that the sub-sequence actually stops processing the current event if any algorithm in the subsequence declares that it doesn't accept the current event. We will use this feature to fill the same histograms after several cut stages, each time in their own sub-TDirectory in the resulting output ROOT file. This way, you will be able to easily compare how a physics distribution changes from one cut to the next.

Finally, lets actually create some cuts; the scheduled histograms will be filled automatically after each of those. Again, the cut algorithms here are using the string parsing.

```
# Make a cut: check that we have at least one Z-boson candidate
subSeq += CfgMgr.CutAlg ("CutZExists", Cut = "count(ZmumuCands.pt > -100.0*GeV) >= 1" )

# Make another cut: check the invariant mass of the di-muon system
subSeq += CfgMgr.CutAlg ("CutZMass", Cut = "count(ZmumuCands.m > 70.0*GeV) >= 1" )
```

Save your work and run athena with your job options (from the `run` directory of your setup). You probably also want to change the number of events to process to, e.g., 500 events. Also, you can change the `EventPrintoutInterval` to 100.

```
cd $TestArea/./run
athena myJobOptions.py 2>&1 | tee log.txt
```

Have a look at the resulting `TutoHistFile.root` file. This now has your histograms in several different `TDirectories`.

If you want to have further ideas on how to generate your histograms, using convenient Athena features, you might want to look at this section. However, you should probably first get to the section on how to create your own package and algorithm first.

Your joboptions file should look like this now... [Your joboptions file should look like this now](#)

```
theApp.EvtMax = 500
```

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 100)
```

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV" ]
```

```
# Fetch the AthAlgSeq, i.e., one of the existing master sequences where one should attach all alg
algseq = CfgMgr.AthSequencer ("AthAlgSeq")
```

```
# Select muons above a pt threshold and
```

```
# create an output muon container only with the selected muons
```

```
algseq += CfgMgr.ParticleSelectionAlg ( "MyMuonSelectionAlg",
                                         InputContainer      = "Muons",
                                         OutputContainer      = "SelectedMuons",
                                         Selection             = "Muons.pt > 15.0*GeV"
                                       )
```

```
# Build all possible di-muon combinations and call the result viable Z-boson candidates
```

```
algseq += CfgMgr.ParticleCombinerAlg ( "MyZmumuBuilderAlg",
                                         InputContainerList = [ "SelectedMuons", "SelectedMuons" ],
                                         OutputContainer    = "ZmumuCands",
                                         SetPdgId            = 23 # This is a Z boson
                                       )
```

```
# =====
```

```
# Define your output root file holding the histograms using MultipleStreamManager
```

```
# =====
```

```
rootStreamName = "TutoHistStream"
```

```
rootFileName=   "TutoHistFile.root"
```

```
rootDirName=    "/Hists"
```

```
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
```

```
MyFirstHistoXAODStream = MSMgr.NewRootStream ( rootStreamName, rootFileName )
```

```
# Now, import the new histogram manager and histogram tool
```

```
from HistogramUtils.HistogramManager import HistogramManager as HistMgr
```

```
histMgr = HistMgr ( "MyHistMgr",
                    RootStreamName = rootStreamName,
```

```

RootDirName      = rootDirName
)

# Import the 1-d and 2-d histograms from ROOT
from ROOT import TH1F, TH2F
# Adding a few histograms to the histogram manager
# Note the different methods of doing this (they are all equivalent)
from HistogramUtils.HistogramTool import HistogramTool as HistTool
histMgr += ( TH1F ("mueta", "#eta^{#mu}", 50, -2.7, 2.7), "Muons.eta" )
histMgr.add( TH1F ("mupt", "p_{t}^{#mu}", 50, 0.0, 100.0), "Muons.pt / GeV " )
histMgr.add( TH1F ("Zmass", "m^{Z}", 50, 50.0, 150.0), "ZmumuCands.m / GeV " )
histMgr += HistTool ( TH2F ("muptvsmueta", "p_{t}^{#mu} vs. #eta^{#mu}", 50, 0.0, 100.0, 50, -2.7

# =====
# Create a subsequence:
# Remember that a subsequence stops its execution for a given event after an algorithm
# that declares that that event doesn't pass a certain selection. This special type of
# sub-sequence additionally handles histogram booking, cloning, and scheduling.
# =====
subSeq = CfgMgr.AthAnalysisSequencer ("AnaSubSeq",
                                     HistToolList = histMgr.ToolList()
                                     )

algseq += subSeq

# Make a cut: check that we have at least one Z-boson candidate
subSeq += CfgMgr.CutAlg ("CutZExists", Cut = "count(ZmumuCands.pt > -100.0*GeV) >= 1" )

# Make another cut: check the invariant mass of the di-muon system
subSeq += CfgMgr.CutAlg ("CutZMass", Cut = "count(ZmumuCands.m > 70.0*GeV) >= 1" )

```

Writing out a small xAOD

Let's actually write out a small xAOD that contains the created Z-boson candidates and the muons. We will use the MultipleStreamManager to do this, just like we used it above to create a ROOT output file.

Let's use our job options (myJobOptions.py); at the end, add these lines:

```

# =====
# Create a new xAOD:
# =====
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
xAODStreamName = "MyFirstXAODStream"
xAODFileName = "myXAOD.pool.root"
MyFirstXAODStream = MSMgr.NewPoolRootStream ( xAODStreamName, xAODFileName )

```

Now we need to know what information we want added to the output xAOD. Here we will only show you how to copy full containers. There is a method that allows you to select exactly what variables you want in each container. We will get to that in a second (at least partially, since it's unfortunately not trivial to do from scratch). To copy full containers you need to do something like this after MyFirstXAODStream has been defined (shown for Muons and the Z-boson candidates):

```

MyFirstXAODStream.AddItem (['xAOD::MuonContainer#Muons'])
MyFirstXAODStream.AddItem (['xAOD::MuonAuxContainer#MuonsAux.'])
MyFirstXAODStream.AddItem (['xAOD::CompositeParticleContainer#ZmumuCands'])
MyFirstXAODStream.AddItem (['xAOD::CompositeParticleAuxContainer#ZmumuCandsAux.'])

```

Where you are specifying the container type and the key name of that container:

```

AddItem (['xAOD::ContainerType#KeyName'])
AddItem (['xAOD::ContainerAuxType#KeyNameAux.'])

```

You have to list both the non-Aux and Aux-type of the containers and key names, as shown above. Also something very subtle, at the end of the `Aux` key name, put a period! If you can't remember the exact container type and key names refer "forward" up to this part: StoreGate and key names, and the content of the xAOD EDM

You can use wildcards to copy all containers/collections of specified container type to your output xAOD file, something like:

```
MyFirstXAODStream.AddItem (['xAOD::TauJetContainer#*'])
MyFirstXAODStream.AddItem (['xAOD::TauJetAuxContainer#*'])
```

for all tau jet containers.

After you rerun your athena job, you will see a new output xAOD file, and you can browse that in a TBrowser in ROOT to see what that looks like. You will see a couple of warnings coming from the file writing, but this is again another symptom of the still developing rel. 21 builds (the output file should be fine though).

What to do when I only want specific variables to be written to my output xAOD? [Hide](#)

Let me start with the reason why this is not straight forward to do with the original xAODs as input. The actual payload of the xAOD objects is stored in the auxiliary store, so the "aux" containers. Most quantities are stored in dedicated branches, so in principal we should be able to just omit these branches and there we go, right? Unfortunately, it's not so easy. Basically, for most of the different xAOD object types the variables associated to them are hard wired into their respective auxcontainer classes. So, for example, if you store the payload of my Muons, which is `MuonsAux.`, as a `MuonAuxContainer` type object then you can't strip variables from it when writing out your muons. The way to work around this problem is to store `MuonsAux.` as an `AuxContainerBase` type (the "base" class from which all specific object auxcontainer classes inherit from), then all the object specific variables can be added as "dynamic" auxiliary branches (indicated as `MuonsAuxDyn.`). There is a very quick way to ensure that you can do that and that is to use the centrally produced xAOD Derivations. If you want to produce valid physics results you anyway must use them (corrections are applied to address errors made during reconstruction of the xAODs). The reason why you can slim down containers (remove variables from the objects) with derivations as input is simply that this "breaking up" of the auxcontainers I described, is already done in the derivations. Long story short, if the necessary requirements are fulfilled, instead of doing the above (for writing the full containers), you should be able to do something like this:

```
MyFirstXAODStream.AddItem (['xAOD::MuonContainer#Muons'])
MyFirstXAODStream.AddItem (['xAOD::AuxContainerBase#MuonsAux.eta.pt.phi'])
```

Note: There is one exception with jets, for them the auxcontainer is actually already maximally split. In general, you want to check the type of the aux stores of the various collections with the approach described here to be sure what to add to your output-stream itemlist. Another fast way to get the branches associated to an object container is actually to just use `TTree::Print()` on the `CollectionTree`. In interactive Root you for example do this to get the list of Muons aux branches:

```
CollectionTree->Print ("*Muons*")
```

Later in this tutorial you will learn to create your own containers and if you want to be able to slim them just do this

```
xAOD::MuonContainer *myMuons = new xAOD::MuonContainer();
xAOD::AuxContainerBase* myMuonsAux = new xAOD::AuxContainerBase();
myMuons->setStore( myMuonsAux );
CHECK( evtStore()->record(myMuons, "MyMuons") );
CHECK( evtStore()->record(myMuonsAux, "MyMuons." ) );
```

instead of this


```

xAOD::MuonContainer *myMuons = new xAOD::MuonContainer();
xAOD::MuonAuxContainer* myMuonsAux = new xAOD::MuonAuxContainer();
myMuons->setStore( myMuonsAux );
CHECK( evtStore()->record(myMuons, "MyMuons" ) );
CHECK( evtStore()->record(myMuonsAux, "MyMuons." ) );

```

This ensures that `MyMuonsAux.` is of type `AuxContainerBase` and can be slimmed.

I will not go into detail on what to do when you use original xAODs and want to modify the input collections to be able to write them with slimming applied to a new file. You could in principle dig into the Derivation code to see how this is done there, but you will probably never find the need to do this anyway.

Writing out only selected events

Since you have already scheduled some algorithms that perform an event selection above (the `CutAlgs`), it is now very easy to only write out events that pass, say the Z-boson candidate minimum mass selection. Since it is the output stream object that manages the writing of the output file, you need to tell it which algorithm(s) it should use to make the decision which events to write out and which ones to omit. This is done in the following way:

```

# Only events that pass the filters listed below are written out
# AcceptAlgs = logical OR of filters
# RequireAlgs = logical AND of filters
# VetoAlgs = logical NOT of filters
MyFirstXAODStream.AddAcceptAlgs ( ["CutZMass" ] )

```

Rerun Athena and you will see upon inspection of the produced mini-xAOD output file that not all events made it, as desired.

Your joboptions file should look like this now...  **Your joboptions file should look like this now** 

```
theApp.EvtMax = 500
```

```
import AthenaPoolCnvSvc.ReadAthenaPool
```

```
svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 100)
```

```
svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/mc16_13TeV" ]
```

```

# Fetch the AthAlgSeq, i.e., one of the existing master sequences where one should attach all alg
algseq = CfgMgr.AthSequencer ("AthAlgSeq")

```

```

# Select muons above a pt threshold and
# create an output muon container only with the selected muons
algseq += CfgMgr.ParticleSelectionAlg ( "MyMuonSelectionAlg",
                                       InputContainer      = "Muons",
                                       OutputContainer      = "SelectedMuons",
                                       Selection              = "Muons.pt > 15.0*GeV"
                                       )

```

```

# Build all possible di-muon combinations and call the result viable Z-boson candidates
algseq += CfgMgr.ParticleCombinerAlg ( "MyZmumuBuilderAlg",
                                       InputContainerList = [ "SelectedMuons", "SelectedMuons" ],
                                       OutputContainer    = "ZmumuCands",
                                       SetPdgId            = 23 # This is a Z boson
                                       )

```

```

# =====
# Define your output root file holding the histograms using MultipleStreamManager
# =====
rootStreamName = "TutoHistStream"
rootFileName=   "TutoHistFile.root"
rootDirName=    "/Hists"

```



```

from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
MyFirstHistoXAODStream = MSMgr.NewRootStream ( rootStreamName, rootFileName )

# Now, import the new histogram manager and histogram tool
from HistogramUtils.HistogramManager import HistogramManager as HistMgr
histMgr = HistMgr ( "MyHistMgr",
                    RootStreamName = rootStreamName,
                    RootDirName    = rootDirName
                    )

# Import the 1-d and 2-d histograms from ROOT
from ROOT import TH1F, TH2F
# Adding a few histograms to the histogram manager
# Note the different methods of doing this (they are all equivalent)
from HistogramUtils.HistogramTool import HistogramTool as HistTool
histMgr += ( TH1F ("mueta", "#eta^{#mu}", 50, -2.7, 2.7), "Muons.eta" )
histMgr.add( TH1F ("mupt", "p_{t}^{#mu}", 50, 0.0, 100.0), "Muons.pt / GeV " )
histMgr.add( TH1F ("Zmass", "m^{Z}", 50, 50.0, 150.0), "ZmumuCands.m / GeV " )
histMgr += HistTool ( TH2F ("muptvsmueta", "p_{t}^{#mu} vs. #eta^{#mu}", 50, 0.0, 100.0, 50, -2.7

# =====
# Create a subsequence:
# Remember that a subsequence stops its execution for a given event after an algorithm
# that declares that that event doesn't pass a certain selection. This special type of
# sub-sequence additionally handles histogram booking, cloning, and scheduling.
# =====
subSeq = CfgMgr.AthAnalysisSequencer ( "AnaSubSeq",
                                       HistToolList = histMgr.ToolList()
                                       )

algseq += subSeq

# Make a cut: check that we have at least one Z-boson candidate
subSeq += CfgMgr.CutAlg ( "CutZExists", Cut = "count(ZmumuCands.pt > -100.0*GeV) >= 1" )

# Make another cut: check the invariant mass of the di-muon system
subSeq += CfgMgr.CutAlg ( "CutZMass", Cut = "count(ZmumuCands.m > 70.0*GeV) >= 1" )

# =====
# Create a new xAOD:
# =====
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
xAODStreamName = "MyFirstXAODStream"
xAODFileName = "myXAOD.pool.root"
MyFirstXAODStream = MSMgr.NewPoolRootStream ( xAODStreamName, xAODFileName )

MyFirstXAODStream.AddItem ( ['xAOD::MuonContainer#Muons'])
MyFirstXAODStream.AddItem ( ['xAOD::MuonAuxContainer#MuonsAux.'])
MyFirstXAODStream.AddItem ( ['xAOD::CompositeParticleContainer#ZmumuCands'])
MyFirstXAODStream.AddItem ( ['xAOD::CompositeParticleAuxContainer#ZmumuCandsAux.'])

# Only events that pass the filters listed below are written out
# AcceptAlgs = logical OR of filters
# RequireAlgs = logical AND of filters
# VetoAlgs = logical NOT of filters
MyFirstXAODStream.AddAcceptAlgs ( ["CutZMass"] )

```

Major updates:

```

-- KarstenKoenke - 11 Sep 2014
-- ThomasMaier - 2015-09-08
-- ThomasMaier - 2017-09-05

```

Responsible: KarstenKoenke

Last reviewed by: **Never reviewed**

This topic: AtlasComputing > SoftwareTutorialGermanyAnalysis

Topic revision: r83 - 2019-09-17 - EricSchanet



Copyright &© 2008-2019 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.

Ideas, requests, problems regarding TWiki? Send feedback