Table of Contents

The LHCb Starterkit lessons 

These are the lessons taught during the LHCb Starterkit. If you'd like to join the next workshop, visit the website to find out when that will how and how to sign up.

If you'd just like to learn about how to use the LHCb software, read on!

Contributing

starterkit-lessons is an open source project, and we welcome contributions of all kinds:

- New lessons;
- Fixes to existing material;
- Bug reports; and
- Reviews of proposed changes.

By contributing, you are agreeing that we may redistribute your work under these licenses. You also agree to abide by our contributor code of conduct.

Getting Started

1. We use the fork and pull model to manage changes. More information about forking a repository and making a Pull Request.
2. To build the lessons please install the dependencies.
3. For our lessons, you should branch from and submit pull requests against the `master` branch.
4. When editing lesson pages, you need only commit changes to the Markdown source files.
5. If you're looking for things to work on, please see the list of issues for this repository. Comments on issues and reviews of pull requests are equally welcome.

Dependencies

To build the lessons locally, install the following:

1. Gitbook

Install the Gitbook plugins:

```
$ gitbook install
```

Then (from the starterkit-lessons directory) build the pages and start a web server to host them:

```
$ gitbook serve
```

You can see your local version by using a web-browser to navigate to `http://localhost:4000` or wherever it says it's serving the book.

First Analysis Steps

This is the LHCb Starterkit, a series of lessons for getting analysts working confidently with LHCb data and software. The lessons are best approached one after the other, as most lessons build on the knowledge gained from the previous ones.

If you have any problems or questions, you can send an email to `lhcb-starterkit@cern.ch`.

Prerequisites

Before starting, you should be familiar with using a shell, like `bash`, and with programming in Python.

The analysis essentials course has an introduction to these topics, as does the Software Carpentry workshop, which includes many other useful computing tools.

Pre-workshop checklist

Learning Objectives

- You will be ready for the workshop!

Please read and try the following steps **before** arriving. For some of the steps the solution requires waiting for a day or so. So please try them before arriving and try to fix what ever does not work.

> Follow this guide before arriving; we will not have time to help you with problems on these issues during the workshop. This means you will end up watching instead of participating.

This will be an interactive workshop, so you will need to bring a computer. There will be no machines for you to use in the room.

> Follow all the steps using the computer you plan to bring, not your desktop or someone else's computer.

If this is the first time you are bringing your laptop to CERN, you will have to register it before it can access the internet.

Please bring your power supply, as well as a plug adaptor to Swiss and European plugs.

Windows

In the following we assume you use Mac OS X or Linux. If you are running Windows, step 2 is replaced by a list of instructions given at the bottom of this page.

Try the following steps with the computer you will use at the workshop:

1. In your browser try and access the web-based book keeping. If you need help with your grid certificate there is the Grid certificate FAQ and you can ask questions on `lhcb-distributed-analysis@cern.ch` .
2. From a terminal (`xterm` on Linux or `Terminal` on Mac OS X) connect to lxplus with `ssh -X lxplus.cern.ch` . If your local username is different from your `lxplus` one use `ssh -X mylxplusname@lxplus.cern.ch` . Please try exactly this command even if you usually use an alias or other shortcut.

   > If, just below the `Password:` line, you get a message `Warning: untrusted X11 forwarding setup failed: xauth key data not generated` :
   >
   > - Logout (using `logout` or `Ctrl-d`)
   > - Login using `-Y` instead of `-X`
   > - This will switch to trusted X11 forwarding and you may see a message like `Warning: No xauth data; using fake authentication data for X11 forwarding.`
3. Once connected, check your grid certificate works by typing `lhcb-proxy-init` . If you need help with your grid certificate there is the Grid certificate FAQ and you can ask questions on `project-lcg-vo-lhcb-admin@cern.ch` . `lhcb-proxy-init` will ask you for the password of your grid certificate and then print something like:

```
Generating proxy...
Enter Certificate password:
Added VOMS attribute /lhcb/Role=user
Uploading proxy for lhcb_user...
Uploading proxy for private_pilot...
Proxy generated:
subject      : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=thead/CN=667505/CN=Timothy Daniel Head/CN=proxy/CN=proxy
issuer       : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=thead/CN=667505/CN=Timothy Daniel Head/CN=proxy
identity     : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=thead/CN=667505/CN=Timothy Daniel Head
timeleft     : 23:53:59
DIRAC group  : lhcb_user
path         : /tmp/x509up_u25636
username     : thead
properties   : NormalUser
VOMS         : True
VOMS fqan    : ['/lhcb/Role=user']

Proxies uploaded:
DN                                                                      | Group         | Until (GMT)
/DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=thead/CN=667505/CN=Timothy Daniel Head | lhcb_user     | 2015/08/25 08:05
/DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=thead/CN=667505/CN=Timothy Daniel Head | private_pilot | 2015/08/25 08:05
```

4. Check that X11 forwarding works by typing `xeyes` on lxplus. A set of eyes following your mouse should appear on your screen. Press `Ctrl-C` from the terminal to exit.

   > If you're not connected to the CERN network at CERN, do not worry if the X11 forwarding is slow--this is normal.

If you can successfully execute all of the above steps, you are ready to go for the workshop!

Using Bash

The Bash shell will be used throughout the workshop. The default for new LHCb computing accounts is now Bash. If you have an older account, the default used to be a shell called `tcsh` ("tee-cee-shell"), which has subtly different ways of doing things in comparison with Bash.

It is recommended to change your default shell to Bash if this is the case, which is much more widely used than `tcsh` and also supported by LHCb, by visiting the your CERN account page, then clicking "Resources and services", then "List services", "LXPLUS and Linux", "Settings", then change "Unix shell" to `/bin/bash` , and click "Save Selection".

If you don't want to change your default shell, just execute the `bash` command when you login to lxplus.

Windows-specific Instructions

On Windows, some additional steps are required before you can connect via SSH.

Set up steps (you only have to perform this once):

1. Download the Xming installer.
2. Run the installer.
3. Download PuTTY.

The following steps have to be executed each time you want to connect:

1. Start PuTTY.
2. In the list on the left, unfold `Connection` and `SSH` , then click the `X11` item.
3. In the window that appears, make sure the check box labeled `Enable X11 forwarding` is checked.

4. Return to the previous window by selecting `Session` int he list on the left.
5. In the text box labeled `Host Name (or IP address)` , type `lxplus.cern.ch` .
6. Make sure the `Port` text box contains the number `22` .
7. Click the `Open` button on the bottom of the screen.
8. A window appears with the text `login as:` . Type your CERN username, followed by Enter.
9. The window should say `Using keyboard-interactive authentication. Password:` . Type your password, again followed by Enter.
10. You now have a remote SSH session at an lxplus server node!

Goals of the course

Learning Objectives

- Understand what we'll be doing, and why we'll be doing it.

The first LHCb Starterkit is all about getting you the data you need to do your physics analysis.

We'll start from the very beginning, explaining how proton-proton collisions make their way to long-term storage, and will end with having a ROOT ntuple containing all the variables you might want to look at for making your measurement.

We want this course to give you the confidence to be able to start by yourself, to understand what the code you'll be writing does and why you're writing it, and to teach others how to do the same. Each lesson follows on from the previous one, but each can also serve as a standalone reference when you need to revisit a particular topic.

We'll be looking at:

- How data flows through the LHCb processing chain, what software is involved, and how the data are stored;
- How the data flow is different for simulated (Monte Carlo) events;
- Where the data ends up, how it's indexed, and how you can find and access the data you need;
- How to get candidate decays from the LHCb data format to ROOT ntuples; and
- How to add more variables to the ntuples.

We'll also cover how to efficiently run software locally and on the Grid using Ganga, as well as how to ask good questions when you're stuck and where to ask them.

The lessons will start with a lot of explaining, but then we'll get in to the hands-on stuff.

These lessons have been put together by a group of people who are passionate about teaching good software practices and demystifying code. Over the course of 2015, these lessons have been written on GitHub, a code sharing and collaboration website. You can find the source code of these lessons in the `lhcb/first-analysis-steps` repository, and *you* can contribute! Please submit an issue if you spot a mistake or you think something isn't clear enough, or you can make the changes yourself and open a pull request. If you're not already familiar with git, you could check out our analysis essentials course.

So, enough with the introduction, let's dive in!

Physics at LHCb

Learning Objectives

- Understand how detector signals are turned into objects in software
- Learn how decay chains are built.

Note that this is a very high level introduction to a high-energy physics experiment, and misses many important details.

The software that these lessons will cover was designed to make the types of physics analyses most commonly performed at LHCb as easy as possible. When using that software, it is then important to understand how those analyses are done in order to understand why the software works the way it does.

At the highest level, a physics analysis tries to measure one or more properties of objects produced through some process. At LHCb, this is typically some production or decay properties of heavy flavour hadrons. Because these objects have short lifetimes they decay before interacting with the detector, and so we must infer their properties through their decay products. Particles such as charged kaons and pions have lifetimes large enough that a large fraction can traverse the full detector, and so we typically consider these particles as 'stable'. Unstable objects, with much shorter lifetimes, are formed as combinations of these 'stable' particles[2]:

1. Charged pions $$\pi^{\pm}$$
2. Charged kaons $$K^{\pm}$$
3. Protons $$p/\bar{p}$$
4. Electrons $$e^{\pm}$$
5. Muons $$\mu^{\pm}$$
6. Photons $$\gamma$$
7. Deuterons (deuterium nuclei)

Many properties of these objects, such as their momentum and charge, are intrinsic, but we have to infer them based on the signals they create in the detector as they traverse through it.

The reconstruction

The properties of particles produced in collisions are known to Nature, but we must reconstruct those properties based on the readout of the detector. The process of inferring all properties is called *reconstruction*. For a given proton-proton collision the reconstruction of all the objects is performed simultaneously, e.g. one does not try to reconstruct one kaon, but reconstructs all charged tracks at the same time, considering all available information.

An important output of the reconstruction is a set of *tracks*, each representing the trajectory of a particle through the detector. Given that all stable charged particles listed above have charge ±1, we can assign a momentum estimate to a track based on its curvature, as induced by the LHCb dipole magnet. We know the polarity of the magnet, as well as its field strength, so can also infer the associated particle's charge.

A *vertex* represents some source of particle production in space. When proton-proton bunches collided, several *primary vertices* can be created in the interaction region. We can reconstruct primary vertices by looking for intersections in space of large numbers of tracks.

Much more detail on the LHCb reconstruction can be found in the LHCb detector description and Run 1 performance papers (these are both rather long, but worth taking the time to read). It is recommended that you become familiar with at least the aspects of the reconstruction that are relevant to your analysis. For now, we only need to understand tracks and vertices, and how decays can be reconstructed and selected from these.

Building decay candidates

All objects created by the reconstruction contain contributions from detector effects, such as resolution. In addition, not all created objects may correspond to real particles; we could for example create 'ghost' tracks, trajectories created from combinations of random hits that together *look like* a real trajectory.

Given this, we can never know anything with complete certainty. Instead we must infer properties *statistically* based on ensembles of objects and events.

Let's say we want to count the number of $$J/\psi$$ mesons that decay to two muons, $$\mu^{+}\mu^{-}$$. This proceeds via three steps:

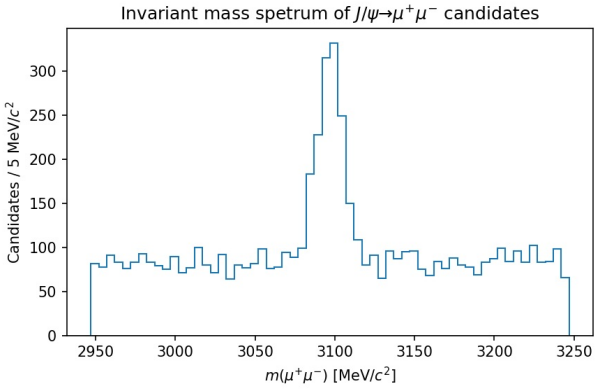
1. Select tracks created by the reconstruction;
2. Create pairs of oppositely-charged tracks;
3. Fit each pair under the hypothesis that they originate from a common point in space.

We'll go over each of these steps, starting with tracks produced by the reconstruction. Naturally, there are many of these in any given event, typically hundreds, so we begin be applying *selections* on the tracks based on our physics understanding. For example, the $$J/\psi$$ is quite heavy (at around $$3.1\,\mathrm{GeV}/c^{2}$$), so we might expect its decay products to have a higher momentum on average than objects produced from soft processes in the collision. The reconstruction gives us some probability-like information for a given track to have been created by a true muon, so we might require that the tracks we use have some minimum 'muon probability'.

With a reduced set of a tracks, we can create all pairs of opposite-sign muons. We know that a real particle decay happens at a point in space, so we could require that the distance of closest approach between the two muons does not exceed some maximum value. We could further require that the invariant mass of the dimuon combination is close to the known mass of the $$J/\psi$$, invoking the conservation of momentum.

With selected dimuon pairs, we can *fit* a $$J/\psi \to \mu^{+}\mu^{-}$$ decay vertex. This is done by expressing the hypothesis that there is a common origin vertex of both tracks as an optimisation problem, and then varying the measured $$\mu^{+}$$ and $$\mu^{-}$$ four-momenta within their measured uncertainties to best fit that hypothesis. The result is a vertex object which has, for example, a fit $$\chi^{2}$$ associated to it. The quality of the fit can be used in a selection.

Finally, with the fitted muon four-vectors, we can form the four-vector of the $$J/\psi$$ as their sum, creating the $$J/\psi$$ *candidate*[1]. Now we get back to our original goal of measuring the number of true $$J/\psi \to \mu^{+}\mu^{-}$$. By plotting the $$J/\psi$$ invariant mass values as a histogram, we might hope to see a signal component. An example is shown in the following plot, created using simulated toy data.

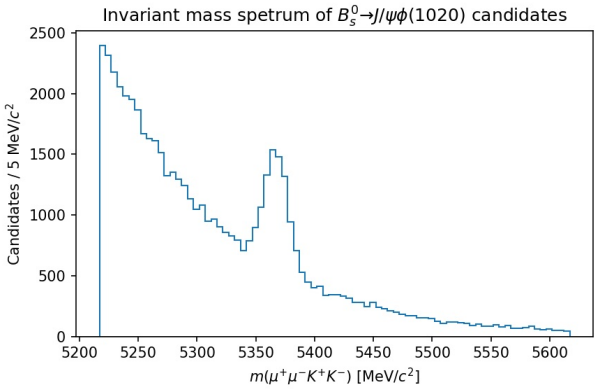


We could choose to model the components using probability density functions, and fit the total model to this histogram. The relative normalisations of the components can be used to deduce the fraction of the sample which contains true decays, within some uncertainty.

Building more complex decays

With a set of $$J/\psi$$ candidates, we could build more complex decay chains such as $$B_{s}^{0} \to J/\psi\phi(1020)$$. The $$J/\psi$$ and the $$\phi$$ meson can decay in various ways, but we might choose to reconstruct them in the $$\mu^{+}\mu^{-}$$ and

$$K^{+}K^{-}$$ final states, respectively. The $$\phi \to K^{+}K^{-}$$ candidates can then be reconstructed in a similar manner to that described previously for the $$J/\psi$$ decay. With a set of $$J/\psi$$ and $$\phi$$ candidates, we can then build $$B_{s}^{0}$$ meson candidates by combining the two decay products in another vertex fit. If our selection is clean enough, we may then see a $$B_s^{0}$$ signal peak, shown below (again, using toy data).



As we'll see in the following lesson, lots of selections like these are run centrally and produce output datasets that contain a mixture of decay chain candidates. This course will show you how to extract the candidates and their properties that are relevant for your analysis.

> [1]. 'Candidate' because, again, we never know anything with complete certainty; this could be combination of muons that just happen to pass our selection criteria. ↩

> [2]. Other 'stable' particles under this definition include neutrons $$n$$ and the long-lived neutral kaon weak eigenstate $$K_{\mathrm{L}}^{0}$$, but these are not part of standard reconstruction output. ↩

The LHCb data flow

Learning Objectives

- Understand the LHCb data flow
- Learn the key concepts on the stripping

The Large Hadron Collider provides proton-proton collisions to LHCb 40 million times a second. This results in a *huge* amount of data. In fact, if we were to store all the data coming in to LHCb, we would be recording ~ 1 TB *every second*.

That's too much data for us to be able to keep all of it, the price of storage is just too high, so instead we need to *filter* the data and try to keep only the events which contain something interesting. This raises its own problems:

- How do we filter and process the recorded data quickly and accurately?
- How do we manage all the complex tasks required to work with collision data?
- How do we organize all the data of a single bunch crossing in a flexible way?
- How do we configure our software flexibly without having to recompile it?
- Can you think of more?

These questions arise mostly due to two key points: the data must be processed very quickly because it's arriving very quickly, and the data is complex so there's a lot that can be done with it. In the last lesson we saw how many steps it took just to reconstruct a single decay, but there are thousands of possibilities that we might be interested in! Being able to perform such combinations in a flexible manner is then very important.

Collisions recorded by the LHCb detector go through a specific data flow designed to maximise the data-taking efficiency and data quality. This consists of several steps, each one being controlled by an ‘application’ that processes the data event-by-event, using the data from the previous step and creating the results ready for the next. These steps are as follows:

1. Data from the detector are filtered through the *trigger*, which consists of the L0, implemented in hardware, and the high level trigger (HLT), implemented in software. The application responsible for the software trigger is Moore, discussed in the trigger lesson in second-analysis-steps.
2. Triggered, raw data are reconstructed to transform the detector hits into objects such as tracks and clusters. This is done by the Brunel application. The objects are stored into an output file in a ‘DST’ format.
3. The reconstructed DST files are suitable for analysis, but they are not accessible to users due to computing restrictions. Data are filtered further through a set of selections called the *stripping*, controlled by the DaVinci application which write out data either in the DST or ‘µDST’ (micro-DST) format. To save disk space and to speed up access for analysts, the output files are grouped into *streams* which contain similar selections. By grouping all of the fully hadronic charm selections together, for example, analysts interested in that type of physics don't waste time running over the output of the dimuon selections.

   ### The output format

   A DST file is a ROOT file which contains the full event information, such as reconstructed objects and raw data. Each event typically takes around 150kB of disk space in the DST format. The µDST format was designed to save space by storing only the information concerning the build *candidates* (that is, the objects used to construct particle decays like tracks); the raw event, which takes around 50kB per event, is discarded.

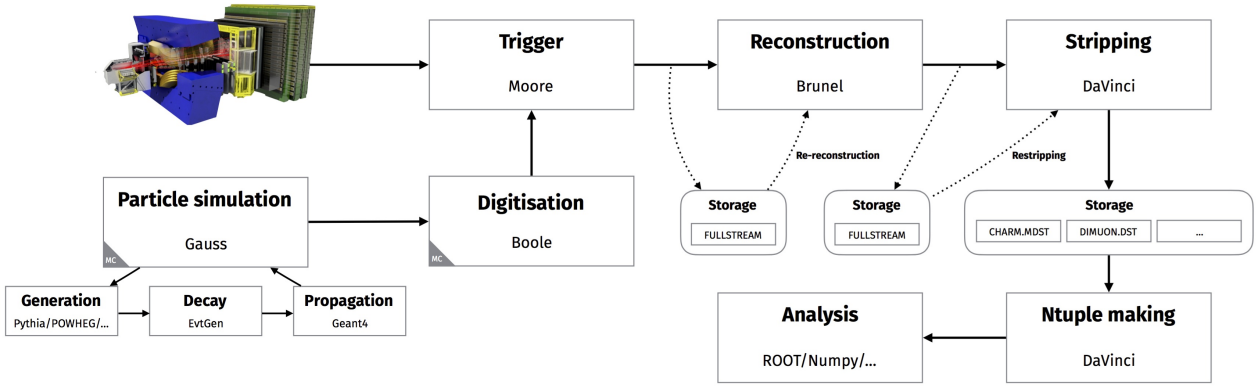
4. Users can run their own analysis tools to extract variables for their analysis with the DaVinci application. The processing is slightly

different between DST and µDST, since some calculations need the other tracks in the event (not only the signal), which are not available in the latter format.

We also produced lots of simulated events, often called Monte Carlo data, and this is processed in a very similar way to real data. This similarity is very beneficial, as the simulated data is subject to the same deficiencies as in the processing of real data. There are two simulation steps which replace the proton-proton collisions and the detector response:

1. The simulation of proton-proton collisions, and the hadronisation and decay of the resulting particles, are ultimately controlled by the Gauss application. Gauss is responsible for calling the various Monte Carlo generators that are supported such as Pythia (the default in LHCb) and POWHEG, and for controlling EvtGen and Geant4. EvtGen is used to describe the decays of simulated particles, whilst Geant4 is used to simulate the propagation and interaction of particles through and with the detector.
2. The simulated hits made in the virtual detector are converted to signals that mimic the real detector by the Boole application. The output of Boole is designed to closely match the output of the real detector, and so the simulated data can then be passed through the usual data processing chain described above, beginning with the trigger.

So, the data flow and the associated applications look like this:



Knowing this flow is essential in selecting your data! Different application versions can produce very different physics, so it's very useful to know how each application has manipulated the data you want to use.

Why are there multiple applications?

It's often simpler to create and visualise a single, monolithic program that does *everything*, but that's not how the data flow is set up in LHCb. Why not? What are the advantages of splitting up the software per task? What are the disadvantages?

With the exception of a few specific studies, it is only the DaVinci application that is run by users, everything else is run 'centrally' either on the computing farm next to the detector or on the Grid.

The reconstruction, Brunel, is rarely performed as it is very computationally intensive. It is only done when the data are taken and when a new reconstruction configuration is available. The stripping can be performed more often, since it runs on reconstructed data.

Stripping 'campaigns', when the stripping selections are centrally run, are identified by a version as `SXrYpZ` :

- The digit `X` marks the *major* stripping version. This marks all major *restrippings*, in which the full list of selections are processed.
- The digit `Y` is the release version, which was used during Run 1 to mark the *data type*, which corresponds to the year the data were taken: `0` was used for 2012 and `1` for 2011. The latest stripping for Run 1 data is called `S21` for 2012 and `S21r1` for 2011.
- The digit `Z` marks the *patch* version, which correspond to *incremental strippings*; campaigns in which only a handful of selections are run, either to fix bugs or to add a small number of new ones.

Knowing the reconstruction and stripping versions is often the most important part in choosing your data, because the selections generally always change between major stripping versions, and variables can look very different between reconstruction versions.

The list of stripping selection largely defines the reconstructed decays that are available to you, but the list is very long as there's a lot we can do with our data. If you don't know the stripping line you need, it's usually best to ask the stripping coordinators of the working group

you'll be presenting your work to. To learn more about the stripping, the best resource is the stripping page on the LHCb TWiki. In it we can find:

- The status of the current stripping, e.g. for Stripping `S28`.
- The configuration of all past stripping campaign, e.g. for Stripping `S21r1`.

Additionally, the information on all strippings can be found in the stripping project website, where you can see all the algorithms run and cuts applied in each *line*. For example, if we wanted to understand the `D2hhPromptDst2D2KKLine` line, which we will use in the exploring a DST lesson later on, we would go here.

Changes to the data flow in Run 2

Learning Objectives

- Understand how the LHCb data flow differs between Run 1 and Run 2

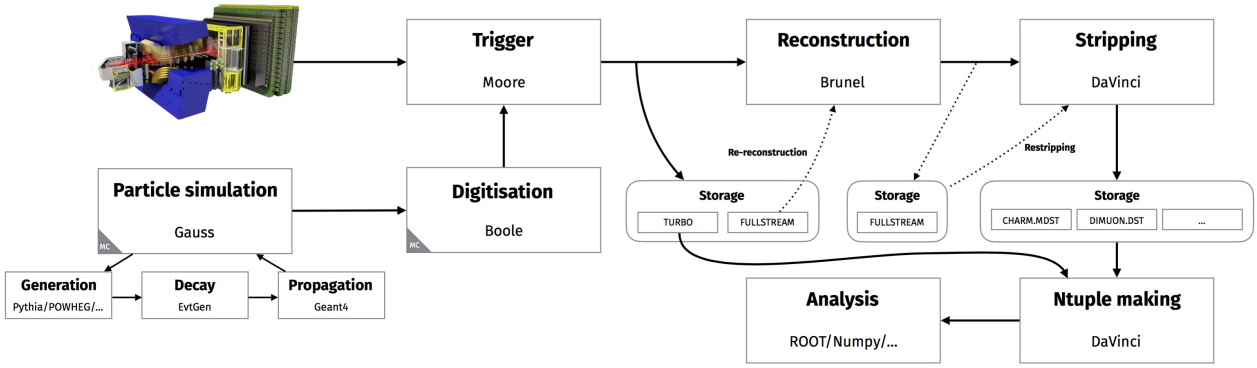
Run 1 of the LHC ran from 2010 to the end of 2012. Run 2 began in the middle of 2015 and finished at the end of 2018. During Run 1 the LHC provided proton-proton collisions at a centre-of-mass energy of 7 and 8 TeV, and in Run 2 the energy was increased to 13 TeV. With an increase in energy comes an increase in production cross-sections, and so a much higher rate of interesting events.

So now we have many more events that we would *like* to store, but as ever we're limited by computing resources. To help overcome this, LHCb introduced the *Turbo stream* in 2015, whereby the selection of candidates made in the second stage of the high level trigger, HLT2, is saved to disk and used *directly* by analysts, with no further offline reconstruction by Brunel.

Omitting the offline reconstruction is usually a Bad Idea because it's usually of a much better quality than the reconstruction performed in the HLT (the 'online reconstruction') as there's more time to run it. But, thanks to an enormous effort improving the reconstruction software both online and offline in between Run 1 and Run 2, the two reconstructions now perform identically. This means if HLT2 performs all the reconstruction you need in your analysis, there's no need to wait for the offline reconstruction to run! This saves a lot of time (it's Turbo, after all), and hence money.

This saves time and money, but you still have many more events to save. To overcome this, events saved to the Turbo stream contain *only* the candidates that were reconstructed in the trigger. That is to say, any tracks or detector responses that don't form part of the decay that the trigger line uses to evaluate its selection is thrown away. This is quite pragmatic, because a lot of analyses don't use this information anyway.

The Turbo stream runs in parallel with the regular data flow, and so everything now looks like this:



The change here is the addition of the Turbo stream in the storage output of the trigger. This stream cannot be re-reconstructed because the information needed to do that is thrown away to save disk space.

For the topics covered in this course, though, this doesn't change things much because we can run DaVinci over the Turbo stream in exactly the same manner as for the stripping output. We will just need to look in a different place to find the selection definitions, this time for *trigger lines* rather than stripping lines.

It's best to ask the trigger coordinator in your working group if your analysis has a trigger line that outputs to the Turbo stream, and if so where to find the selection definitions.

The different types of Turbo

As the Turbo data flow model has evolved throughout Run 2, the capabilities of Turbo have changed. Each capability has a different name, which can be useful to know.

1. **Turbo**: Saving of the candidate that fired the trigger line. The entire decay tree of the object is saved (i.e. including descendants).
2. **Turbo++**, or **PersistReco**: In addition to Turbo, the rest of the reconstruction performed in HLT2 is also saved when the trigger line fires. This includes all long and downstream tracks, all tracks associated to the primary vertex reconstruction (VELO tracks), and all particle identification objects. This allows you to perform arbitrary combinatorics offline.
3. **TurboSP**: For 'Turbo with selective persistence', in addition to Turbo you can specify additional objects to save. These additional objects are usually the subset of the reconstruction that's relevant for your physics analysis, such as the set of particles that combine with the trigger candidate and form a good-quality vertex. TurboSP is a compromise between Turbo, where you can only do analysis on what you used in the trigger but use little space, and Turbo++, where you can do many things offline but use a lot of space.

TurboSP is considered as the primary data flow model for the planned LHCb upgrade in Run 3.

An introduction to LHCb Software

Learning Objectives

- Learn the key concepts needed to work with the LHCb software
- Learn how to launch the LHCb software with `lb-run`

Imagine you want to design and run a new particle detector. Apart from organizing a collaboration, creating the design and specification, and several other tasks, you will also have to find solutions to many computational challenges. It's worth thinking about these for a second:

- How do we collect data as it is recorded by the detector?
- How do we filter and process the recorded data efficiently?
- How do we manage all the complex tasks required to work with collision data?
- How do we organize all the data of a single bunch crossing in a flexible way?
- How do we configure our software flexibly without having to recompile it?
- Can you think of more?

How would you go about solving these? The decisions you make will affect the performance of your experiment during datataking and analysis.

At LHCb, we base our software on the Gaudi framework, which was specifically designed with the above questions in mind. It's worth getting an idea of some of the most important Gaudi concepts at this point. After this, we will jump right into running the software and getting useful things done.

Event Loop Because the individual bunch crossings (events) are almost completely independent of each other, it makes sense to process them one by one, without holding them all in memory at once. Gaudi provides a global EventLoop, which allows you to process events one by one.

Transient Event Store A single event contains lots of different data objects (Particles, Vertices, Tracks, Hits, ...). In Gaudi, these are organized in the Transient Event Store (TES). You can think of it as a per-event file system with locations like `/Event/Rec/Track/Best` or `/Event/Phys/MyParticles`. When running over the event stream, Gaudi allows you to get and put from/to these locations. The contents of the TES are emptied at the end of the processing of each event.

Algorithms An *Algorithm* is a C++ class that can be inserted into the EventLoop. These allow you to perform a certain function for each event (like filtering according to trigger decision, reconstructing particles, ...).

Tools Often, algorithms will want to make use of some common function (vertex fitting, calculating distances, associating a primary vertex, ...). These are implemented as *Tools*, which are shared between Algorithms.

Options To make all of this configurable, Gaudi allows you to set properties of *Algorithms* and *Tools* from a Python script, called an *option* file. In an option file, you can specify which Algorithms are run in which order, and set their properties (strings, integers, doubles, and lists and dicts of these things can be set). You can then start the Gaudi EventLoop using this option file, and it will set up and run the corresponding C++ objects with specified settings.

You can find comprehensive documentation in the Gaudi Doxygen or the Gaudi Manual.

Usually, you will work with one of the LHCb software projects that are based on Gaudi. One of the most important ones is *DaVinci*, which provides lots of *Algorithms* and *Tools* for physics analysis.

You can run DaVinci using the following command on lxplus:

```
lb-run DaVinci/v44r6 gaudirun.py
```

This will run the `gaudirun.py` command using version v44r6 of DaVinci. (`lb-run` sets the specified environment for `gaudirun.py` to run in.) `gaudirun.py` is a script that sets up the EventLoop. You should get the following output:

```
# setting LC_ALL to "C"
ApplicationMgr    SUCCESS
====================================================================================================================================
======
                                                  Welcome to DaVinci version v44r6
                                        running on lxplus069.cern.ch on Wed Nov 21 17:53:58 2018
====================================================================================================================================
======
ApplicationMgr       INFO Application Manager Configured successfully
HistogramPersis...WARNING Histograms saving not required.
ApplicationMgr       INFO Application Manager Initialized successfully
ApplicationMgr       INFO Application Manager Started successfully
EventSelector        INFO End of event input reached.
EventLoopMgr         INFO No more events in event selection
ApplicationMgr       INFO Application Manager Stopped successfully
EventLoopMgr         INFO Histograms converted successfully according to request.
ToolSvc              INFO Removing all tools created by ToolSvc
ApplicationMgr       INFO Application Manager Finalized successfully
ApplicationMgr       INFO Application Manager Terminated successfully
```

During this run, DaVinci didn't do anything: We didn't specify any algorithms to run or any data to run over. Usually, you will write an option file (e.g. `options.py`) and specify it as an argument to `gaudirun.py` :

```
lb-run DaVinci/v44r6 gaudirun.py options.py
```

An `option.py` is just a regular Python script that specifies how to set things up in the software. Many of the following lessons will teach you how to do something with DaVinci by showing you how to write or extend an `options.py` . You can use the above command to test it. You can also specify several option files like this:

```
lb-run DaVinci/v44r6 gaudirun.py options1.py options2.py
```

They will then both be used to set up DaVinci.

Do you want to get an overview of which versions of DaVinci exist? Use

```
lb-run --list DaVinci
```

Which version of DaVinci should I use?

All available versions of DaVinci are given on the DaVinci releases page. Which one should you use? There are a couple of guidelines to follow:

1. When starting a new analysis, use the latest version suitable for the data you would like to analyse.
2. When continuing an analysis, use the same version of DaVinci consistently throughout. This includes cases where you want to repeat an existing analysis but with modified settings, e.g. re-running a Stripping line.

There are a set of DaVinci versions for Upgrade studies (versions v50r1 and above) and a set for everything else. Generally, you will want the latest version in the latter set, such as when making ntuples from Run 1 or Run 2 data.

These lessons use DaVinci v44r6, which was the latest Run 1/2 version at the time the text was last revised.

Do you want to start a shell that already contains the LHCb environment, so you don't have to use `lb-run` ? Execute

```
lb-run DaVinci/v44r6 $SHELL
```

Note that sometimes this environment can result in failing scripts due to struggles with your shell's rc file (e.g., `~/.bashrc`). Using

```
lb-run DaVinci/v44r6 bash --norc
```

avoids this, but means you won't be able to use any aliases, etc, included in the ignored rc file.

A simple `gaudirun.py` should work as well now. Typing `exit` or using `Ctrl-d` will close the shell and leave the LHCb environment behind.

Using SetupProject instead of lb-run

When reading through other tutorials, you will come across `SetupProject`. This is an older way of setting up a shell that is configured to run LHCb software. `lb-run` is the new way of doing things and has some nice benefits over `SetupProject`. For most purposes, `SetupProject DaVinci v44r6` is equivalent to

```
lb-run DaVinci/v44r6 $SHELL
```

but you should really avoid doing things this way as this method is no longer supported for the latest project releases. (The environment for DaVinci v44r6, for example, cannot be started this way.)

Finding data in the Bookkeeping

Knowing how data flows through the various Gaudi applications is crucial for knowing where to look for your data.

Data are catalogued in ‘the bookkeeping’, and are initially sorted in broad groups such as ‘real data for physics analysis’, ‘simulated data’, and ‘data for validation studies’. After this, a tree of various application and processing versions will eventually lead to the data you need.

So, before we can run our first DaVinci job we need to locate some events. In this tutorial we will use the decay $$D^{* +} \to D^{0}\pi^{+}$$ as an example, where the $$D^{0}$$ decays to $$K^{-} K^{+}$$.

Learning Objectives

- Find MC in the bookkeeping
- Find data in the bookkeeping
- Find the decay you want

Navigate to the bookkeeping which lets you find both simulated and real data.

At the bottom of the "Bookkeeping tree" tab there is a drop-down menu labelled `Simulation Condition`, open it and change it to `Event type`. This changes the way the bookkeeping tree is sorted, making it easier for us to locate files by event.

We will analyse 2016 data, and correspondingly use simulation for 2016 data. To navigate to the simulation, expand the folder icon in the "Bookkeeping tree" window. Navigate to the `MC/2016` folder. This will give you a very long list of all possible decay types for which there is simulated data. We are looking for a folder which is named `27163002 (Dst_D0pi,KK=DecProdCut)`. The number is a numerical representation of the event type. The text is the human readable version of that.

This sample of simulated events will only contain events where a $$D^{* +} \to D^{0}(\to K^{-}K^{+})\pi^{+}$$ was generated within the LHCb acceptance, although the decay might not have been fully reconstructed. (Not all simulated samples have the same requirements made on the signal decay.)

If you expand the `27163002 (Dst_D0pi,KK=DecProdCut)` folder you will find a couple different subfolders to choose from. The names of these subfolders correspond to different data-taking conditions, such as magnet polarity (`MagDown` and `MagUp`), as well as different software versions used to create the samples that are available. We will use `Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8`.

So much choice!

Often there are only one or two combinations of data-taking conditions and software versions to choose from, but sometimes there can be very many. Generally newer versions are the best bet, but you should always ask the Monte Carlo liason of your working group for advice on what to use if you're not sure.

Next we need to choose what version of the simulation you want to use. There are two available in our case, `Sim09b` and `Sim09c`, but usually the latest available version is the best when there are more than one. We also have to choose what configuration of the trigger (`Trig0x6138160F` in our case) and reconstruction we want to have in the simulated sample. Usually there is only one choice for these, which makes choosing easier.

Two final steps are versions of `Turbo` (only for Run 2 samples, as discussed in the previous lesson) and `Stripping`. In order to make the samples as useful as possible for the largest number of analyses, each of the Run 2 samples is processed via both Turbo and Stripping frameworks, so it is usable for either of two choices. Usually, there is only one version of `Turbo`, but there can be multiple versions of the `Stripping`. Choose any as long as it contains the word `Flagged`.

Flagged and filtered samples

In the usual data-taking flow, the trigger and stripping are run in *filtering* mode, whereby events that don't pass any trigger line or any stripping line are thrown away. In the simulation, it's often useful to keep such events so that the properties of the rejected events can be studied. The trigger and stripping are then run in *flagging* mode, such that the decisions are only recorded for later inspection. Filtered Monte Carlo can be produced for analyses that need lots of events.

After all this, you will be presented with a `ALLSTREAMS.DST` entry. By clicking on it we finally see a list of files that we can process. At the bottom right of the page there is a “Save” button which will let us download a file specifying the inputs that we'll use for running our DaVinci job. Click it, select “Save as a python file”, and add `.py` to the end of the text in “Save As...”. Clicking “Save” once again in the pop-up menu will start the download. Save this file somewhere you can find it again.

A copy of the file we just downloaded is available here.

Shortcut

Once you get a bit of experience with navigating the bookkeeping you can take a shortcut! At the bottom of your browser window there is a text field next to a green "plus" symbol. You can directly enter a path here to navigate there directly. For example you could go straight to: `evt+std://MC/2016/27163002/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/` by typing this path and pressing the `Go` button.

Find your own decay!

Think of a decay and try to find a Monte Carlo sample for it. You could use the decay that your analysis is about, or if you don't have any ideas you could look for the semileptonic decay

If you would like to find out more about how the event types define the signal decay, you can look at the documentation for the DecFiles package.

Downloading a file from the grid

Learning Objectives

- Obtain a DST file from the grid

In the previous section, we obtained a file called `MC_2016_27163002_Beam6500GeV2016MagDownNu1.625nsPythia8_Sim09c_Trig0x6138160F_Reco16_Turbo03_Stripping28r1NoPrescalingFlagged_ALLSTREAMS.DST.py` which contains the following section:

```
IOHelper('ROOT').inputFiles(['LFN:/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000001_7.AllStreams.dst',
'LFN:/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000002_7.AllStreams.dst',
'LFN:/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000003_7.AllStreams.dst',
'LFN:/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000004_7.AllStreams.dst',
...
], clear=True)
```

which is just a collection of **L**ogical **F**ile **N**ames on the grid.

This is a list of files that make up the dataset we are interested in. Each of the files contains a number of individual events, so if we just want to take a quick look at the dataset, it is sufficient to just obtain one of those files.

Before we can download the file, we need to set up our connection with the grid and load the Dirac software:

```
lhcb-proxy-init
```

Initialisation of the proxy might take a while and should ask you for your certificate password.

Once we have a working Dirac installation, getting the file is as easy as

```
lb-run LHCbDIRAC dirac-dms-get-file LFN:/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000001_7.AllStreams.dst
```

Again this will take a while but afterwards you should have a file called `00070793_00000001_7.AllStreams.dst` in the directory where you called the command.

Downloading the file during a Starterkit lesson

Lots of people downloading the same file at the same time can be very slow. As a workaround, the file is also available on EOS, and can be downloaded to your current directory with the following command:

```
$ xrdcp root://eosuser.cern.ch//eos/user/l/lhcbsk/data-sets/00070793_00000001_7.AllStreams.dst .
```

Since these files tend to be quite large, you might want to use your AFS work directory instead of your AFS user directory to store files.

Alternative: read files remotely instead of downloading them

To avoid filling up your AFS quota with DST files, you can also pass Gaudi an XML catalog such that it can access them remotely.

First generate the XML catalog with

```
lb-run LHCbDIRAC dirac-bookkeeping-genXMLCatalog --Options=MC_2016_27163002_Beam6500GeV2016MagDownNu1.625nsPythia8_Sim0
9c_Trig0x6138160F_Reco16_Turbo03_Stripping28r1NoPrescalingFlagged_ALLSTREAMS.DST.py --Catalog=myCatalog.xml
```

and add

```
from Gaudi.Configuration import FileCatalog
FileCatalog().Catalogs = [ "xmlcatalog_file:/path/to/myCatalog.xml" ]
```

to your options file. See the bookkeeping twiki.

Warning: the replicas of an LFN may change, so first try to regenerate the XML catalog in case you cannot access a file using this recipe.

If you want to obtain all the files, you can copy and paste the list of file names from the file you got from the bookkeeping and paste them into the following python script for convenience.

```
# Your list of file names here
FILES = []

if __name__ == '__main__':
    from subprocess import call
    from sys import argv

    n_files = len(FILES)
    if len(argv) > 1:
        n_files = int(argv[1])

    files = FILES[:n_files]
    for f in files:
        print('Getting file {0}.'.format(f))
        call('dirac-dms-get-file {0}'.format(f), shell=True)
    print('Done getting {0} files.'.format(n_files))
```

Save it as `getEvents.py` and use it via `lb-run LHCbDIRAC python getEvents.py [n]`. If you specify `n`, the script will only get the first n files from the grid.

Such a clever script!

`dirac-dms-get-file` (and the other `dirac-dms-*` scripts) is actually able to extract the LFNs from any file and download them for you. So a simple

```
 lb-run LHCbDIRAC dirac-dms-get-file --File=MC_2016_27163002_Beam6500GeV2016MagDownNu1.625nsPythia8_Sim09c_Trig0x613816
0F_Reco16_Turbo03_Stripping28r1NoPrescalingFlagged_ALLSTREAMS.DST.py
```

would do to download them all!

Interactively exploring a DST

Learning Objectives

- Open a DST in an interactive python session
- Print all nodes in a DST
- Explore the contents of the TES
- Inspect a track
- Inspect a stripping location

Data is stored in files called DSTs, which are processed by DaVinci to make nTuples. However you can also explore them interactively from a python session.

This is particularly useful if you want to quickly find something out, or the more complex processing in DaVinci is not working as expected.

The file we downloaded from the grid contains simulated data, with stripping and trigger decisions and so on. Here we assumed the file you downloaded is called `00070793_00000001_7.AllStreams.dst`. To take a look at the contents of the TES, we need to write a small Python file:

```
import sys

import GaudiPython as GP
from GaudiConf import IOHelper
from Configurables import DaVinci

dv = DaVinci()
dv.DataType = '2016'
dv.Simulation = True

# Pass file to open as first command line argument
inputFiles = [sys.argv[-1]]
IOHelper('ROOT').inputFiles(inputFiles)

appMgr = GP.AppMgr()
evt = appMgr.evtsvc()

appMgr.run(1)
evt.dump()
```

Place this into a file called `first.py` and run the following command in a new terminal:

```
$ lb-run DaVinci/v44r6 ipython -i first.py 00070793_00000001_7.AllStreams.dst
```

This will open the DST and print out some of the TES locations which exist for this event. We are now ready to explore the TES, which is accessible via the `evt` variable. For example you could look at the properties of some tracks for the first event by typing inside the python session:

```
tracks = evt['/Event/Rec/Track/Best']
print tracks[0]
```

The next question is, how do you know what TES locations that could exist? As we saw `evt.dump()` prints a few of them, but not all. In addition there are some special ones that only exist if you try to access them. The following snippet allows you to discover most TES locations that are interesting:

```
def nodes(evt, node=None):
    """List all nodes in `evt`"""
    nodenames = []

    if node is None:
        root = evt.retrieveObject('')
        node = root.registry()

    if node.object():
        nodenames.append(node.identifier())
        for l in evt.leaves(node):
            # skip a location that takes forever to load
            # XXX How to detect these automatically??
            if 'Swum' in l.identifier():
                continue

            temp = evt[l.identifier()]
            nodenames += nodes(evt, l)

    else:
        nodenames.append(node.identifier())

    return nodenames
```

The easiest way to use it is to add it to your `first.py` script and re-run it as before. Then, in your iPython session, enter `nodes(evt)`. This will list a large number of TES locations, but even so there are some which you have to know about. Another oddity is that some locations are "packed", for example: `/Event/AllStreams/pPhys/Particles`. You can not access these directly at this location. Instead you have to know what location the contents will get unpacked to when you want to use it. Often you can just try removing the small `p` from the location (`/Event/AllStreams/Phys/Particles`).

You can also inspect the particles and vertices built by your stripping line. However not every event will contain a candidate for your line, so the first tool we need is something that will advance us until the stripping decision was positive:

```
def advance(decision):
    """Advance until stripping decision is true, returns
    number of events by which we advanced"""
    n = 0
    while True:
        appMgr.run(1)

        if not evt['/Event/Rec/Header']:
            print 'Reached end of input files'
            break

        n += 1
        dec=evt['/Event/Strip/Phys/DecReports']
        if dec.hasDecisionName('Stripping{0}Decision'.format(decision)):
            break

    return n
```

Add this to your script and restart `ipython` as before.

Detecting file ends

It is not easy to detect that the input file has ended. Especially if you want to get it right for data and simulation. Checking that `/Event/Rec/Header` exists is a safe bet in simulation and data if your file has been processed by `Brunel` (the event reconstruction software). It might not work in other cases.

Using the name of our stripping line we can now advance through the DST until we reach an event which contains a candidate:

```
line = 'D2hhPromptDst2D2KKLine'
advance(line)
```

The candidates built for you can now be found at `/Event/AllStreams/Phys/D2hhPromptDst2D2KKLine/Particles` :

```
cands = evt['/Event/AllStreams/Phys/{0}/Particles'.format(line)]
print cands.size()
```

This tells you how many candidates there are in this event and you can access the first one with:

```
print cands[0]
```

Which will print out some information about the Particle. In our case a $$D^{* +}$$ (particle ID number 413). You can access its daughters with `cands[0].daughtersVector()[0]` and `cands[0].daughtersVector()[1]` , which will be a $$D^{0}$$ and a $$\pi^{+}$$.

There is a useful tool for printing out decay trees, which you can pass the top level particle to and it will print out the daughters etc:

```
print_decay = appMgr.toolsvc().create(
  'PrintDecayTreeTool', interface='IPrintDecayTreeTool'
)
print_decay.printTree(cands[0])
```

With our candidates in hand, it would be nice to be able to retrieve and compute the variables we need for an analysis. On to LoKi functors!

Fast DST browsing

While here we have discussed for pedagogical reasons all the configuration options needed in order to browse a `DST` file, in your daily life as a physicist it is often useful to use the `bender` application that belongs to the `Bender` project.

For example, to explore the `DST` we could have simply done:

```
lb-run Bender/latest bender 00070793_00000001_7.AllStreams.dst
```

This leaves us in a prompt in which we can proceed as discussed in this lesson, with the advantage that some functions are already provided for us, such as `seekStripDecision` (which replaces our `advance`) or `ls` and `get` , which allow to list and get TES locations. Other examples of useful functions are listed in the `bender` starting banner.

`Bender` also provides a useful command `dst-dump` , which is a quick way of figuring out what objects are present on a `DST` and where. Try out:

```
lb-run Bender/latest dst-dump -f -n 100 00070793_00000001_7.AllStreams.dst
```

The `-f` option tells `Bender` to try and "unpack" the locations such as `/Event/AllStreams/pPhys/Particles` that we mentioned above, while `-n 100` tells it to only process the first 100 events on the `DST` . Give this a try if you're ever stuck figuring out where your candidates are hiding!

Running a minimal DaVinci job locally

Looping event-by-event over a file and inspecting interesting quantities with LoKi functors is great for exploration: to checking that the file contains the candidates you need, that the topology makes sense, and so on. It's impractical for most cases, though, where you want *all* the candidates your trigger/stripping line produced, which could be tens of millions of decays. In these cases we use DaVinci, the application for analysing high-level information such as tracks and vertices, which we'll look at in this lesson to produce a ROOT ntuple.

Learning Objectives

- Run a DaVinci job over a local DST
- Inspect the ntuple output
- Set up the job to run in Ganga

With some stripped data located, it's useful to store the information on the selected particles inside an ntuple. This allows for quick, local analysis with ROOT, rather than always searching through a DST that contains lots of things we're not interested in.

As well as being the application that runs the stripping, DaVinci allows you to access events stored in DSTs and copy the information to ROOT ntuples. You tell DaVinci what you want it to do through *options files*, written in Python. There are lots of things you can do with DaVinci options files, as there's lots of information available on the DST, but for now we'll just work on getting the bare essentials up and running.

Our main tool will be the `DecayTreeTuple` object, which we'll create inside a file we will call `ntuple_options.py` :

```
from Configurables import DecayTreeTuple
from DecayTreeTuple.Configuration import *

# Stream and stripping line we want to use
stream = 'AllStreams'
line = 'D2hhPromptDst2D2KKLine'

# Create an ntuple to capture D*+ decays from the StrippingLine line
dtt = DecayTreeTuple('TupleDstToD0pi_D0ToKK')
dtt.Inputs = ['/Event/{0}/Phys/{1}/Particles'.format(stream, line)]
dtt.Decay = '[D*(2010)+ -> (D0 -> K- K+) pi+]CC'
```

This imports the `DecayTreeTuple` class, and then creates an object called `dtt` representing our ntuple. Once DaVinci has run, the resulting ntuple will be saved in a folder within the output ROOT file called `TupleDstToD0pi_D0ToKpi` .

The `Inputs` attribute specifies where `DecayTreeTuple` should look for particles, and here we want it to look at the output of the stripping line we're interested in.

As stripping lines can save many decays to a DST, the `Decay` attribute specifies what decay we would like to have in our ntuple. If there are no particles at the `Input` location, or the `Decay` string doesn't match any particles at that location, the ntuple will not be filled.

Decay descriptors

There is a special syntax for the `Decay` attribute string, commonly called 'decay descriptors', that allow a lot of flexibility with what you accept. For example, `D0 -> K- X+` will match any D0 decay that contains one negatively charged kaon and one positively charged track of any species. More information the decay descriptor syntax can be found on the LoKi decay finders TWiki page.

Now we need to tell DaVinci how to behave. The `DaVinci` class allows you to tell DaVinci how many events to run over, what type of data is being used, what algorithms to run over the events, and so on.

There are many configuration attributes defined on the `DaVinci` object, but we will only set the ones that are necessary for us.

```
from Configurables import DaVinci

# Configure DaVinci
DaVinci().UserAlgorithms += [dtt]
DaVinci().InputType = 'DST'
DaVinci().TupleFile = 'DVntuple.root'
DaVinci().PrintFreq = 1000
DaVinci().DataType = '2016'
DaVinci().Simulation = True
# Only ask for luminosity information when not using simulated data
DaVinci().Lumi = not DaVinci().Simulation
DaVinci().EvtMax = -1
DaVinci().CondDBtag = 'sim-20170721-2-vc-md100'
DaVinci().DDDBtag = 'dddb-20170721-3'
```

Nicely, a lot of the attributes of the `DaVinci` object are self-explanatory: `InputType` should be `'DST'` when giving DaVinci DST files; `PrintFreq` defines how often DaVinci should print its status; `DataType` is the year of data-taking the data corresponds to, which we get from looking at the bookkeeping path used to get the input DST; `Simulation` should be `True` when using Monte Carlo data, `Lumi` defines whether to store information on the integrated luminosity the input data corresponds to; and `EvtMax` defines how many events to run over, where a value of `-1` means "all events".

The `CondDBtag` and `DDDBtag` attributes specify the exact detector conditions that the Monte Carlo was generated with. Specifying these tags is important, as without them you can end up with the wrong magnet polarity value in your ntuple, amongst other Bad Things. You can find the values for these tags in the bookkeeping file we downloaded earlier.

Database tags

Generally, the `CondDB` and `DDDB` tags are different for each dataset you want to use, but will be the same for all DSTs within a given dataset.

For real collision data, you shouldn't specify these tags, as the default tags are the latest and greatest, so just remove those lines from the options file.

However, when using simulated data, *always* find out what the database tags are for your dataset!

There are several ways to access the database tags used for a specific production, but the most reliable one consists of the following steps:

- Find the bookkeeping location of any DST for your desired event type and conditions (e.g. `/lhcb/MC/2016/ALLSTREAMS.DST/00070793/0000/00070793_00000002_7.AllStreams.dst`).
- The number after `ALLSTREAMS.DST` is the number of the production: in this case, `00070793`.
- Go to the transformation monitor. Put this number in the field `ProductionID(s):` and press "Submit". You will see the details of the production to the right.
- Right click on these details, and press "Show request". The new tab "Production Request manager" will appear to the right of the "LHCb Transformation Monitor". Go to that tab.
- You will see the details of the MC request. Right click on it, and press "View".
- A new window will pop up with the complete details of the request. You have to find the "Step 1" section, and the following line in it `DDDB: dddb-20170721-3 Condition DB: sim-20170721-2-vc-md100` contains your database tags.

Note that the Condition DB tags for different magnet polarities are different: `-md100` should be replaced by `-mu100` for the MagUp conditions.

This method can also be used to find other details about how any data was processed by DIRAC, such as the options files and

application versions.

In order to run an algorithm that we have previously created, we need to add it to the `UserAlgorithms` list. The `TupleFile` attribute defines the name of the ROOT output file that DaVinci will store any algorithm output in, which should be our ntuple.

Being smart and efficient

Typical stripping lines take only a small part of the stripped stream - so, a small fraction of events in the DST: actually, usually you care about a single TES location! At the same time, event unpacking and running the DecayTreeTuple machinery for each event is time-consuming. Consequently, DSTs can be processed much faster if before unpacking we select *only* events which are likely to accomodate the desired TES location. This can be achieved, for example, by requiring a prefilter checking whether event passes a stripping requirement. You may also filter on trigger decisions - this is an idea behind the Turbo stream. As a conclusion, it is *strongly* recommended to exploit the `EventPreFilters` method offered by `DaVinci` : this feature can save a lot of processing time and collaboration's computing resources when running over millions of events. To require events to pass a specific stripping line requirement, one should add these lines to the options file:

```
from PhysConf.Filters import LoKi_Filters
fltrs = LoKi_Filters (
    STRIP_Code = "HLT_PASS_RE('StrippingD2hhPromptDst2D2KKLineDecision')"
)
DaVinci().EventPreFilters = fltrs.filters('Filters')
```

Here we use the LoKi functor `HLT_PASS_RE` which checks for a positive decision on (in this case) the stripping line. You may investigate some of more advanced examples of `EventPreFilters` usage here and here.

All that's left to do is to say what data we would like to run over. As we already have a data file downloaded locally, we define that as our input data.

```
from GaudiConf import IOHelper

# Use the local input data
IOHelper().inputFiles([
    './00070793_00000001_7.AllStreams.dst'
], clear=True)
```

This says to use the `.dst` file that is in the same directory as the options file, and to clear any previous input files that might have been defined.

That's it! We're ready to run DaVinci.

In the same folder as your options file `ntuple_options.py` and your DST file ending in `.dst` , there's just a single command you need run on `lxplus` .

```
$ lb-run DaVinci/v44r6 gaudirun.py ntuple_options.py
```

The full options file we've created, `ntuple_options.py` , is available here. A slightly modified version that uses remote files (using an XML catalog as described here) is available here.

Using a microDST

A microDST (or µDST) is a smaller version of a DST. Some stripping lines go to µDSTs, and some go to DSTs. There are two

things that need changing in our options file in order to have it work when it is used with a stripping line that goes to a µDST:

1. The `DecayTreeTuple.Inputs` attribute should start at the word `Phys` ; and
2. The `RootInTES` attribute on the `DaVinci` object has to be set to `/Event/$STREAM`

In context, the changes look like

```
dtt.Inputs = ['Phys/{0}/Particles'.format(line)]
# ...
DaVinci().RootInTES = '/Event/{0}'.format(stream)
```

Fun with LoKi Functors

Learning Objectives

- Find out how the physics information can be obtained from the DST
- Understand what LoKi functors are
- Use LoKi functors interactively
- Be able to find functors that do what we want

LoKi functors are designed to flexibly compute and compare properties of the current decay, from simple quantities such as the transverse momentum of a particle to complicated ones like helicity angles. Internally, functors are implemented as C++ classes that take an object of type `TYPE1` and return another of `TYPE2`. They can be used both in C++ and in Python code, and can be combined with each other using logical operations.

According to `TYPE2` there are 3 types of functors:

- *Functions*, which return `double`.
- *Predicates*, which return a `bool`.
- *Streamers*, which return a `std::vector` of some other type `TYPE3`.

When filling tuples, the most used functors are functions, while predicates are typically used for selections.

According to `TYPE1`, there are many types of functors, the most important of which are (you can find a full list in the LoKi FAQ):

- *Particle functors*, which take `LHCb::Particle*` as input.
- *Vertex functors*, which take `LHCb::VertexBase*` as input.
- *MC particle functors*, which take `LHCb::MCParticle*` as input.
- *MC vertex functors*, which take `LHCb::MCVertex*` as input.
- *Array particle functors*, which take a `LoKi::Range_` (an array of particles) as input.
- *Track functors*, which take `LHCb::Track` as input.

C++ classes

Things like `LHCb::Particle` are C++ classes that usually represent some physical object. You will interact with the C++ objects directly very rarely, if ever.

To understand what we can do with LoKi functors, we will pick up from where we left off exploring a DST interactively. Open the DST and get the first candidate in the `D2hhPromptDst2D2KKLine` line:

```
cands = evt['/Event/AllStreams/Phys/D2hhPromptDst2D2KKLine/Particles']
cand = cands[0]
```

We can now try to get very simple properties of the $$D^{* +}$$ candidate. Let's start from the components of its momentum. This can be done calling the function `momentum()` for our candidate in the following way:

```
p_x = cand.momentum().X()
p_y = cand.momentum().Y()
p_z = cand.momentum().Z()
print p_x, p_y, p_z
```

This is inconvenient when running DaVinci with Python options files: there's no way of calling the `momentum()` method. Instead, we can use the corresponding LoKi particle functors:

```
from LoKiPhys.decorators import PX, PY, PZ
print PX(cand)
print PY(cand)
print PZ(cand)
```

You will see an error when loading the functors:

```
LoKiSvc.REPORT      ERROR LoKi::AuxDesktopBase:     loadDesktop(): unable to load IPhysDesktop! StatusCode=FAILURE
LoKiSvc.REPORT      ERROR The   ERROR message is suppressed : 'LoKi::AuxDesktopBase:     loadDesktop(): unable to load IPhysDe
sktop!' StatusCode=FAILURE
```

This is related to the fact that some functors need to run in the `DaVinci` 'scope', and they are all loaded in the `LoKiPhys.decorators` module. It's harmless in the examples we will use. If the import is made *before* the instantiation of the `ApplicationMgr`, there will be no warnings.

Does it make sense?

Compare the output of `PX` functor with the result of calling the function `cand.momentum().X()`.

Math operations are also allowed:

```
p_components_sum = PX + PY + PZ
p_components_sum(cand)
```

There exist specific LoKi functors for all the most important properties of the particle. For example, the transverse momentum and mass:

```
from LoKiPhys.decorators import PT, M
print PT(cand)
print M(cand)
```

Some practice

Retrieve the momentum magnitude using functors `PX`, `PY` and `PZ`. There is also a specific functor `P` which does the job. Compare the results.

Now, retrieve the transverse momentum and invariant mass (you will probably need the energy functor `E`), and see if it matches what the `PT` and `M` functors return.

A note about units

By the convention, the LHCb default units are MeV, millimeters and nanoseconds. It is easy to print the values of interest in other units:

```
from LoKiPhys.decorators import GeV
print PT(cand)/GeV
```

If we want to get the properties of the $$D^{* +}$$ vertex, for example its fit quality ($$\chi^2$$), we need to pass a vertex object to the vertex functor.

```
from LoKiPhys.decorators import VCHI2
print VCHI2(cand.endVertex())
```

Again, this is inconvenient when running DaVinci with Python options files, since in that case we don't have any way of calling the `endVertex` method. Instead, we can use the `VFASPF` *adaptor* functor, which allows us to use vertex functors as if they were particle functors (note how the functor is built by combining two functors).

```
from LoKiPhys.decorators import VFASPF
VCHI2(cand.endVertex()) == VFASPF(VCHI2)(cand)
```

Functions of functions of functions of…

Make sure you understand what `VFASPF(VCHI2)(cand)` means. It may help to play around in Python, creating a function that takes another function as an argument, for example:

```
def create_greeting(salutation):
    def greet(name):
        print '{0}, {1}!'.format(salutation, name)
    return greet
```

What would `create_greeting('Hello')` return? What about `create_greeting('Howdy')('partner')` ? Why is doing this useful?

Calculation of some of the properties, such as the impact parameter (IP) or cosine of the direction angle (DIRA), requires the knowledge of the primary vertex (PV) associated to the candidate. In `GaudiPython` , we can get the PVs ourselves.

```
pv_finder_tool = appMgr.toolsvc().create(
    'GenericParticle2PVRelator<_p2PVWithIPChi2, OfflineDistanceCalculatorName>/P2PVWithIPChi2',
    interface='IRelatedPVFinder'
)
pvs = evt['/Event/Rec/Vertex/Primary']
best_pv = pv_finder_tool.relatedPV(cand, pvs)
```

Now, we can get the cosine of the direction angle for the candidate given the primary vertex:

```
from LoKiPhys.decorators import DIRA
print DIRA(best_pv)(cand)
```

Given that this is a very common operation, we have the possibility of using, in the context of a `DaVinci` application (Stripping, for example), a special set of functors, starting with the `BPV` prefix (for Best PV), which will get the PV for us. Some functors also end with the suffix `DV` , which means they can only be used in the `DaVinci` context.

To get the quality of impact parameter of the candidate, one needs as well to call a distance calculator:

```
from GaudiPython.Bindings import gbl
distCal = appMgr.toolSvc().create("LoKi::DistanceCalculator", interface=gbl.IDistanceCalculator)
ipTool = gbl.LoKi.Vertices.ImpactParamTool(distCal)
```

Now, we evaluate the quality of impact parameter of the candidate, given the primary vertex, and using the provided calculator:

```
from LoKiPhys.decorators import IPCHI2
print IPCHI2(best_pv, ipTool)(cand)
```

In the context of `DaVinci` application, e.g. the Stripping, the things become much simplier since the calculator instances are loaded automatically, and the syntax for calling the `IPCHI2` functor becomes `IPCHI2(best_pv,geo())(cand)`, where `geo()` is the geometry calculator tool.

Finding LoKi functors

The full list of defined LoKi functors can be found in the `LoKi::Cuts` namespace in the doxygen. They are quite well documented with examples on how to use them. The list can be overwhelming, so it's also worth checking a more curated selection of functors in the TWiki, here and here.

So far we've only looked at the properties of the head of the decay (that is, the $$D^{* +}$$), but what if we want to get information about its daughters? As an example, let's get the largest transverse momentum of the final state particles. A simple solution would be to navigate the tree and calculate the maximum $$p_{\text{T}}$$.

```
def find_tracks(particle):
    tracks = []
    if particle.isBasicParticle():
        proto = particle.proto()
        if proto:
            track = proto.track()
            if track:
                try:
                    tracks.append(particle.data())
                except AttributeError:
                    tracks.append(particle)
    else:
        for child in particle.daughters():
            tracks += find_tracks(child)
    return tracks

max_pt = max([PT(child) for child in find_tracks(cand)])
```

A note about the try/except

If you import LoKi before running this example, it magically removes the `.data()` function and allows the particle to be used directly. The code above is made general using the `try` / `except` block and will work in either case.

However, LoKi offers functions for performing such operations, namely `MAXTREE` and `MINTREE`, which get as parameters the selection criteria, the functor to calculate and a default value. In our example,

```
from LoKiPhys.decorators import MAXTREE, ISBASIC, HASTRACK
MAXTREE(ISBASIC & HASTRACK, PT, -1)(cand) == max_pt
```

In this example, we have used two selection functors, `ISBASIC` and `HASTRACK`, which return true if the particle doesn't have children and is made up by a track, respectively. We can see that they do the same thing as `particle.isBasicParticle()` and `particle.proto().track()` in a more compact way.

Combining LoKi cuts

You might have noticed above we used the `&` operator ("bitwise AND") to combine the `ISBASIC` and `HASTRACK` cuts above. This is because Python doesn't allow LoKi to override the behaviour of `and` and `or` ("logical AND/OR"), so if we use them the Python interpreter tries to combine the two cuts straight away, before we have even passed in our candidate:

```
In [1]: ((M>1200) or (PT > 500))
Out[1]:  (M>1200)
```

the result is that our `PT` cut vanishes! If we use the `|` operator ("bitwise OR") then LoKi correctly builds a functor representing the `OR` of our cuts:

```
In [2]: ((M>1200) | (PT > 500))
Out[2]:  ( (M>1200) || (PT>500) )
```

This is why you should **always** use `&` and `|` when combining LoKi functors, and **never** use `and` and `or`.

Similarly, the `SUMTREE` functor allows us to accumulate quantities for those children that pass a certain selection:

```
from LoKiPhys.decorators import SUMTREE, ABSID
print SUMTREE(321 == ABSID, PT)(cand)
print SUMTREE('K+' == ABSID, PT)(cand)
```

In this case, we have summed the transverse momentum of the charged kaons in the tree. Note the usage of the `ABSID` functor, which selects particles from the decay tree using either their PDG Monte Carlo ID or their name. If you would like to consider only the kaons of one specific charge in the selection requirement, consider the `ID` functor which does exactly the same thing, however has a sign which is positive for particles and negative for antiparticles.

Another very useful LoKi functor is `CHILD`, which allows us to access a property of a single child of the particle. To specify which child we want, its order is used, so we need to know how the candidate was built. For example, from

```
In [10]: cand.daughtersVector()
Out[10]:

 0 |->D0                          M/PT/E/PX/PY/PZ: 1.8653/ 2.5848/ 31.32/ 2.508/-0.6267/ 31.15 [GeV]  #  0
                                      EndVertex  X/Y/Z: 1.053/-0.2006/-29.13 [mm]  Chi2/nDoF 0.2349/1 #  0
 1    |->K+                        M/PT/E/PX/PY/PZ: 0.4937/ 2.1334/ 20.26/ 2.129/0.1371/ 20.14 [GeV]  #  5
 1    |->K-                        M/PT/E/PX/PY/PZ: 0.4937/ 0.8534/ 11.06/0.3795/-0.7643/ 11.01 [GeV]  # 10
 0 |->pi+                         M/PT/E/PX/PY/PZ: 0.1396/ 0.2558/ 3.101/0.2451/-0.0733/ 3.088 [GeV]  #  4
```

we know that `D0` is the first child and `pi+` is the second. Therefore, to access the mass of the $$D^{0}$$ we have 2 options:

```
from LoKiPhys.decorators import CHILD
# Option 1
mass = M(cand.daughtersVector()[0])
# Option 2
mass_child = CHILD(M, 1)(cand)
# Do they agree?
mass == mass_child
```

Child vertex?

Evaluate the quality of the D0 decay vertex.

In the similar way, we may access properties of child of the child: for example, a kaon from the $$D^{0}$$ decay:

```
from LoKiPhys.decorators import CHILD
mass_kaon = CHILD(CHILD(M, 1),1)(cand)
```

Tracks and PID

For the particles having tracks, we may exploit track functors to get the corresponding track properties. For instance, the track quality is given by functor `TRCHI2`.

What happens if we call `TRCHI2(cand)`? Explain the result.

Evaluate the track quality for the first and second kaon, also independently of that retrieve (in a single line) the worst of two.

Then, evaluate the probability that each kaon is really a kaon (`PROBNNk`) or rather a misidentified pion (`PROBNNpi`).

The usage of LoKi functors extends much further than in the interactive `GaudiPython` world we've been exploring here.

They constitute the basis of particle filtering in the *selection framework*, discussed in the Building your own decay chain lesson in second-analysis-steps. Selecting particles means using LoKi *predicates*, functors that give a `bool` output, like `ISBASIC` and `HASTRACK`. Amongst these, a key functor is `in_range`, which returns `True` if the value of the given *function* functor (that is, the functor that returns a `double`) is within the given lower and upper limit. It helps writing CPU-efficient functors and thus is very important when building time-critical software like trigger or stripping lines.

```
from LoKiCore.functions import in_range
in_range(2000, M, 2014)(cand)
in_range(1860, CHILD(M, 1), 1870)(cand)
```

Understanding the cuts in the stripping lines

Have a look at the stripping line D2hhPromptDst2D2KKLine which is used in our example. Open a `CombineParticles/D2hhPromptDst2D2KKLine` section, and explain which requirements are coded in the 'MotherCut', 'DaughterCuts' and 'CombinationCut' sections. (More details about `CombineParticles` algorithm are explained in the lesson of second analysis steps.)

Additionally, LoKi functors can be used directly inside our `DaVinci` jobs to store specific bits of information in our ntuples without the need for a complicated C++-based algorithms. This second option will be discussed in the TupleTools and branches lesson.

Debugging LoKi functors

If you write complicated LoKi functors, typically in the context of selections, you need functions for debugging when things go wrong. LoKi provides wrapper functors that evaluate a functor (or functor expression), print debugging information and return the result; the most important of these are:

- `dump1`, which prints the input object and returns the calculated functor value,

```
from LoKiCore.functions import dump1
debug_p_components_sum = dump1(p_components_sum)
debug_p_components_sum(cand)
```

- `monitor` which prints the input the functor string and returns the calculated functor value,

```
from LoKiCore.functions import monitor
monitor_p_components_sum = monitor(p_components_sum)
monitor_p_components_sum(cand)
```

TupleTools and branches

Learning Objectives

- Add extra TupleTools to the default DecayTreeTuple
- Configure the extra TupleTools
- Use branches
- Find useful TupleTools
- Learn how to use LoKi functors in a DecayTreeTuple

Usually, the default information stored by `DecayTreeTuple` as shown in our minimal DaVinci job is not enough for physics analysis. Fortunately, most of the information we need can be added by adding C++ tools (known as `TupleTools`) to `dtt`; there is an extensive library of these, some of which will be briefly discussed during the lesson.

Default DecayTreeTuple tools

The default tools added in `DecayTreeTuple` are:

- `TupleToolKinematic`, which fills the kinematic information of the decay.
- `TupleToolPid`, which stores DLL and PID information of the particle.
- `TupleToolANNPID`, which stores the new NeuralNet-based PID information of the particle.
- `TupleToolGeometry`, which stores the geometrical variables (IP, vertex position, etc) of the particle.
- `TupleToolEventInfo`, which stores general information (event number, run number, GPS time, etc) of the event.

In order to add `TupleTools` to `dtt`, we have to use the `addTupleTool` method of `DecayTreeTuple` (only available when we have `from DecayTreeTuple.Configuration import *` in our script). This method instantiates the tool, adds it to the list of tools to execute and returns it. For example, if we want to fill the tracking information of our particles, we can add the `TupleToolTrackInfo` tool in the following way:

```
track_tool = dtt.addTupleTool('TupleToolTrackInfo')
```

Some tools can be configured. For example, if we wanted further information from the tracks, such as the number of degrees of freedom of the track fit, we would have to turn on the verbose mode of the tool:

```
track_tool.Verbose = True
```

If we don't need to configure the tool or we want to use the defaults, there's no need for storing the returned tool in a variable. For example, if we wanted the information of the PV associated to our particle, we could just add the `TupleToolPrimaries` with no further configuration:

```
dtt.addTupleTool('TupleToolPrimaries')
```

The way the `DecayTreeTuple.Decay` is written in in our minimal DaVinci job,

```
dtt.Decay = '[D*(2010)+ -> (D0 -> K- K+) pi+]CC'
```

means that the configured `TupleTools` will only run on the head of the decay chain, that is, the `D*(2010)+`. In order to select the particles for which we want the information stored, we need to mark them with a `^` symbol in the decay descriptor. For example, if we want to fill the information of the `D0` and its children, as well as the soft `pi+`, we would modify the above line to look like this:

```
dtt.Decay = '[D*(2010)+ -> ^(D0 -> ^K- ^K+) ^pi+]CC'
```

This will run all the configured `TupleTools` on the marked particles, with the caveat that some tools are only run on certain types of particles (eg, tracking tools on particles that have an associated track). This configuration is not optimal, since there may be tools which we only want to run on the D's and some only on the children. Enter `Branches`, which allow us to specify which tools get applied to which particle in the decay (in addition to the `TupleTools` configured at the top level).

Branches let you define custom namespaces in your ntuple by means of a `dict`. Its keys define the name of each branch (and, as a consequence, the prefix of the corresponding leaves in the ntuple), while the corresponding values are decay descriptors that specify which particles you want to include in the branch.

```
dtt.addBranches({'Dstar' : '[D*(2010)+ -> (D0 -> K- K+) pi+]CC',
                 'D0'    : '[D*(2010)+ -> ^(D0 -> K- K+) pi+]CC',
                 'Kminus': '[D*(2010)+ -> (D0 -> ^K- K+) pi+]CC',
                 'Kplus' : '[D*(2010)+ -> (D0 -> K- ^K+) pi+]CC',
                 'pisoft': '[D*(2010)+ -> (D0 -> K- K+) ^pi+]CC'})
```

Note that in order to use branches, we have to make sure that all particles we want to use are marked in the main decay descriptor (`dtt.Decay`). DaVinci will ignore branches for particles that have not been marked in `dtt.Decay`!

Once the branches have been configured, they can be accessed as `dtt.PARTICLENAME` and `TupleTools` can be added as discussed before. For example, if we want to store the proper time information of the D0, we would do

```
dtt.D0.addTupleTool('TupleToolPropertime')
```

Do I really have to type my decay descriptor that many times?

No! You can use the (new) `dtt.setDescriptorTemplate()` method to set up your decay descriptor and branches in just one line! Well, nearly: because this is a new feature it is not available in most released versions of `DaVinci`, but this snippet will add it to an older version. With that out of the way, you can simply use

```
dtt.setDescriptorTemplate('${Dstar}[D*(2010)+ -> ${D0}(D0 -> ${Kminus}K- ${Kplus}K+) ${pisoft}pi+]CC')
```

This will set up both `dtt.Decay` and `Branches` for you.

The usage of `Branches` is very important (and strongly encouraged) to keep the size of your ntuples small, since it prevents us from storing unneeded information (for example trigger information, which will be discussed at a later lesson).

Where to find TupleTools

One of the most difficult things is to know which tool we need to add to our `DecayTreeTuple` in order to get the information we want. For this, it is necessary to know where to find `TupleTools` and their code. `TupleTools` are spread in 9 packages under `Analysis/Phys` (see the master branch in `git` here), all starting with the prefix `DecayTreeTuple`, according to the type of information they fill in our ntuple:

- `DecayTreeTuple` for the more general tools.

- `DecayTreeTupleANNPID` for the NeuralNet-based PID tools.
- `DecayTreeTupleDalitz` for Dalitz analysis.
- `DecayTreeTupleJets` for obtaining information on jets.
- `DecayTreeTupleMC` gives us access to MC-level information.
- `DecayTreeTupleMuonCalib` for muon calibration tools.
- `DecayTreeTupleReco` for reconstruction-level information, such as `TupleToolTrackInfo`.
- `DecayTreeTupleTracking` for more detailed tools regarding tracking.
- `DecayTreeTupleTrigger` for accessing to the trigger information of the candidates.

The `TupleTools` are placed in the `src` folder within each package and it's usually easy to get what they do just by looking at their name. However, the best way to know what a tool does is check its documentation, either by opening its `.h` file or be searching for it in the latest `doxygen`. Most tools are very well documented and will also inform you of their configuration options. As an example, to get the information on the `TupleToolTrackInfo` we used before we could either check its source code or its web documentation. In case we need more information or need to know *exactly* what the code does, the `fill` method is the one we need to look at.

As a shortcut, the list of tupletools can also be found in doxygen at the top of the pages for the `IParticleTupleTool` and the `IEventTupleTool` interfaces (depending on whether they fill information about specific particles or the event in general).

The updated options can be found here.

Test your ntuple

Run the options in the same way as in the minimal DaVinci job lesson. You will obtain a `DVntuple.root` file, which we can open and inspect with `ROOT`'s `TBrowser`:

```
$ root DVntuple.root
root [0]
Attaching file DVntuple.root as _file0...
root [1] TBrowser b
root [2]
```

Try to locate the branches we have added, which are placed in the `TupleDstToD0pi_D0ToKpi/DecayTree`, and plot some distributions by double-clicking the leaves.

Picking up with the LoKi functors lesson, let's store some specific bits of information discussed there in our ntuple. To add LoKi-based leaves to the tree, we need to use the `LoKi::Hybrid::TupleTool`, which is configured with 3 arguments:

1. Its *name*, specified in the `addTupleTool` call after a `/`. This is very useful (and recommended) if we want to have different `LoKi::Hybrid::TupleTool` for each of our branches. For instance, we may want to add different information for the D*, the D0 and the soft $$\pi$$:

   ```
   dstar_hybrid = dtt.Dstar.addTupleTool('LoKi::Hybrid::TupleTool/LoKi_Dstar')
   d0_hybrid = dtt.D0.addTupleTool('LoKi::Hybrid::TupleTool/LoKi_D0')
   pisoft_hybrid = dtt.pisoft.addTupleTool('LoKi::Hybrid::TupleTool/LoKi_PiSoft')
   ```

2. The `Preambulo` property, which lets us perform preprocessing of the LoKi functors to simplify the code that is used to fill the leaves, for example creating combinations of LoKi functors or performing mathematical operations:

```
preamble = [
   'DZ = VFASPF(VZ) - BPV(VZ)',
   'TRACK_MAX_PT = MAXTREE(ISBASIC & HASTRACK, PT, -1)'
]
dstar_hybrid.Preambulo = preamble
d0_hybrid.Preambulo = preamble
```

3. The `Variables` property, consisting of a `dict` of (variable name, LoKi functor) pairs. In here, LoKi functors can be used, as well as any variable we may have defined in the `Preambulo`:

```
dstar_hybrid.Variables = {
   'mass': 'M',
   'mass_D0': 'CHILD(M, 1)',
   'pt': 'PT',
   'dz': 'DZ',
   'dira': 'BPVDIRA',
   'max_pt': 'MAXTREE(ISBASIC & HASTRACK, PT, -1)',
   'max_pt_preambulo': 'TRACK_MAX_PT',
   'sum_pt_pions': 'SUMTREE(211 == ABSID, PT)',
   'n_highpt_tracks': 'NINTREE(ISBASIC & HASTRACK & (PT > 1500*MeV))'
}
d0_hybrid.Variables = {
   'mass': 'M',
   'pt': 'PT',
   'dira': 'BPVDIRA',
   'vtx_chi2': 'VFASPF(VCHI2)',
   'dz': 'DZ'
}
pisoft_hybrid.Variables = {
   'p': 'P',
   'pt': 'PT'
}
```

In the code snippets specified above (available here), you can see that the `NINTREE` functor counts the number of particles that pass the specified criteria. While this is not very useful for ntuple-building (we can always do it offline), it's a very powerful functor to use when building decay selections.

Getting more practice

In the `LoKi::Hybrid::TupleTool` we've used some functors that have not been described previously. Find out what they do in the doxygen. To check `SUMTREE` and `CHILD`, run the code above and check that the `Dstar_max_pt` and `Dstar_max_pt_preambulo` and the `Dstar_mass_D0` and `D0_mass` branches have exactly the same values.

How do I use DecayTreeFitter?

Learning Objectives

- Add a kinematic fitter to a branch in the decay tree
- Apply a mass constraint
- Inspect the refitted decay tree

Once you have made a hypothesis on the chain of decays that lead to your final state, you then can incorporate the additional knowledge that comes with this hypothesis to get a new best estimate for the particle parameters -- in particular their momenta. The additional knowledge is represented as constraints, which your decay tree has to fulfill.

For example, for the decay

```
'[D*(2010)+ -> (D0 -> K- K+) pi+]CC'
```

you can make the assumption that the (K- K+) combine to form a D0 with a specific invariant mass. This results in a so called *mass constraint*. In addition the two kaons should originate from exactly the same point in space. If you know that your data only contains prompt D* candidates, you can constrain them to do come from the primary vertex. Boundary conditions like those are called *vertex constraints* (the last of which is known as a *primary-vertex constraint*).

Applying kinematic constraints leads to new best estimates for the track parameters of the final-state particles. The process of calculating those is called a *kinematic (re)fit* and the `DecayTreeFitter` is the algorithm that performs this task for us. Access to this tool is provided by the TupleTool with the name `TupleToolDecayTreeFitter`.

The physics and mathematics behind DecayTreeFitter

For details of the method see the paper on Decay chain fitting with a Kalman filter.

So how do we use a `TupleToolDecayTreeFitter` in our DaVinci script? Let's create a branch to add the tool to. We'll just name it `'Dstar'`:

```
dtt.addBranches({
    'Dstar': '[D*(2010)+ -> (D0 -> K- K+) pi+]CC',
})
```

To this branch we can now apply the `TupleToolDecayTreeFitter` with arbitrarily chosen name `consD`.

```
dtt.Dstar.addTupleTool('TupleToolDecayTreeFitter/ConsD')
```

Now we can proceed with the configuration of the fitter. We are going to constrain the D0 to have originated from the primary vertex. We want all the output available, so we set the `verbose` option. Finally we want to apply the mass constraint to the D0:

```
dtt.Dstar.ConsD.constrainToOriginVertex = True
dtt.Dstar.ConsD.Verbose = True
dtt.Dstar.ConsD.daughtersToConstrain = ['D0']
```

Note that you can constrain more than one intermediate state at once if that fits your decay.

When using the `TupleToolDecayTreeFitter` in a `DecayTreeTuple`, all the variables created by the other TupleTools are not affected by the change, but some new variables are created, one set per `DecayTreeFitter` instance. Depending on whether the `Verbose` option is specified, the new variables are created for the head particle only or for the head particle and its daughters too.

If the daughters are not stable particles and decay further, the daughters of the daughters have no new variables associated to them by default. Since in many cases this information might be useful, there is an option to store the information from those tracks

```
dtt.Dstar.ConsD.UpdateDaughters = True
```

DecayTreeFitter and LoKi functors

Alternatively, many of the operations described above can be done by using the `DecayTreeFitter` via LoKi functors, see the DaVinci tutorial for details.

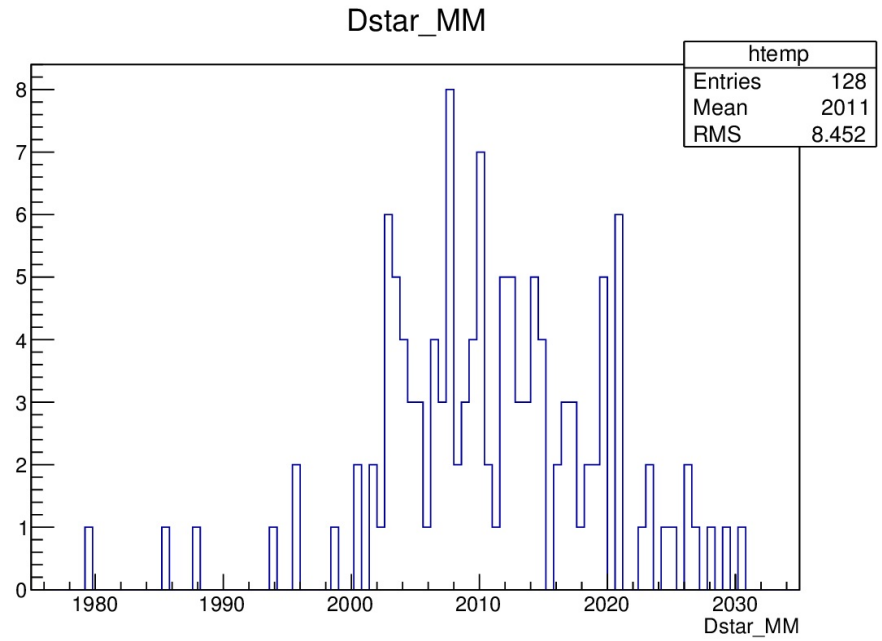
Which constraints to apply

It is important to be aware of the assumptions you make to build your ntuple. For example, after you require the vertex constraint you must be careful if using the `IPCHI2_OWNPV`, since the particle you are looking at is *forced* to point to the PV. Which constraints make most sense for you depends on the questions you want to ask in your analysis, so ask your supervisor/working group in case of doubt.

Once you have produced your ntuple you can have a look at the refitted variables.

```
root -l DVntuple.root
TupleDstToD0pi_D0ToKK->cd()
DecayTree->StartViewer()
```

Plotting the raw mass of the D* (without the fit) `Dstar_M`, you should see a broad signal around 2 GeV:



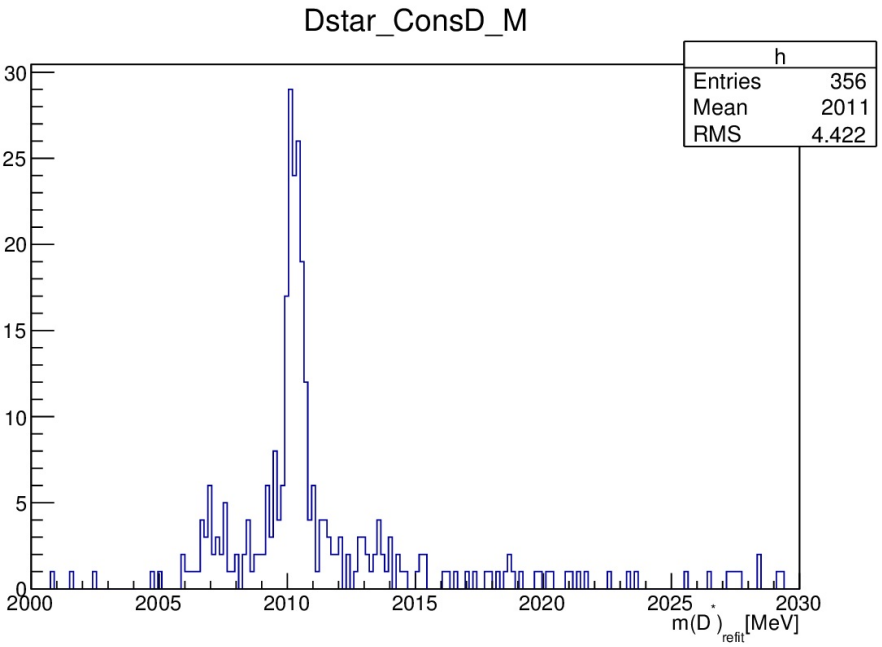
Which mass variable to use

In many ntuples you also find a mass variable called `_MM`. This (confusingly) refers to measured mass. However, it is usually better to use `_M`. `_MM` is the sum of the 4-momenta of the final state particles extrapolated back to the fitted vertex position, but not the result of the actual vertex fit. Remember that a vertex fit acts like a vertex constraint, improving the opening-angle resolution.

Now let us look at the refitted mass of the D*, with the D0 constrained to its nominal mass. It is stored in the variable `Dstar_ConsD_M`. If you plot this you will note that some values are unphysical. So, let's restrict the range we look at to something that makes sense.

On the root prompt use the `arrow-up` key to get the last draw command and modify it to pipe the output into a histogram:

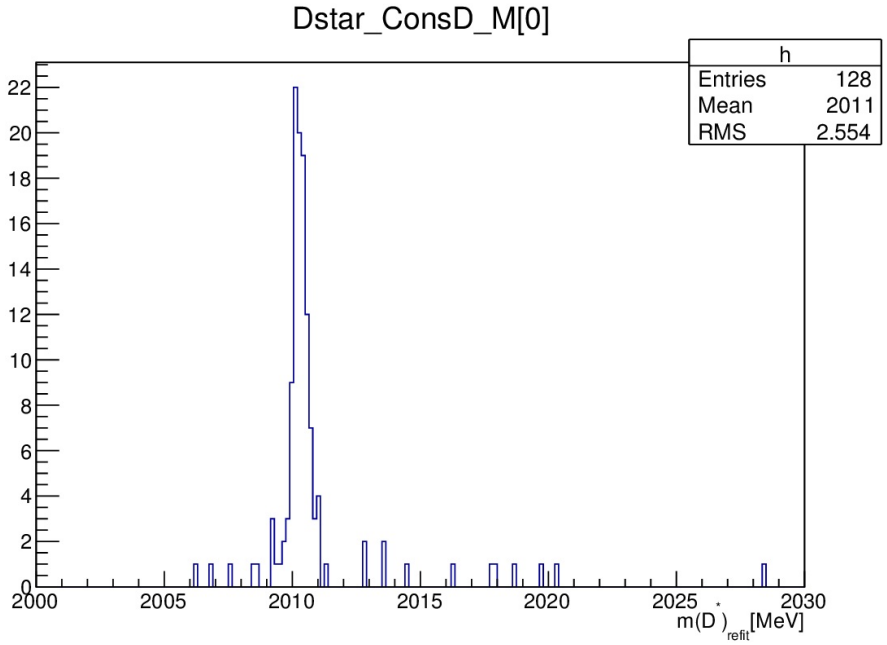
```
tv__tree->Draw("Dstar_ConsD_M>>h(200,2000,2030)","","");
```



Note that this plot has 356 entries, although we only have 128 candidates in the raw mass spectrum. The reason for this is that we typically have several primary vertices per event. When you use the vertex contraint, the fitter is run for each of the possible vertex hypotheses available in the event. So all the `Dstar_ConsD-xxx` variables are in fact arrays, where the first value corresponds to the *best PV* hypothesis. We can plot only those by doing

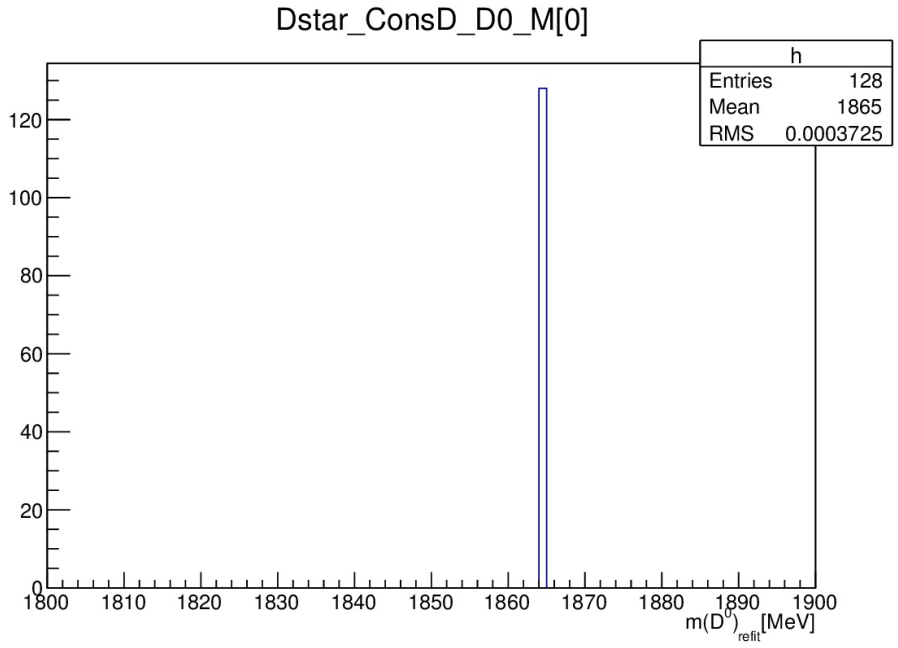
```
tv__tree->Draw("Dstar_ConsD_M[0]>>h(200,2000,2030)","","");
```

and we get the final kinematically refitted Dstar mass:



Finally, let's check how the D0 mass constraint has played out.

```
tv__tree->Draw("Dstar_ConsD_D0_M[0]>>h(100,1800,1900)","","", 128, 0);
```



As expected, the D0 candidates are forced onto their PDG mass value.

Explore

- Look at the `status` variable to check if the fits converged.
- Look at the chi2 distribution of the fit

`DecayTreeFitter` can be told to change some of the hypotheses in the decay tree. This is very useful if you want to slightly change which decays you want to look at. As an example let's say we want to examine the decay of the D0 into K- pi+ instead of K- K+. For this we add a second fitter, giving it a new name `ConsDKpi` :

```
dtt.Dstar.addTupleTool('TupleToolDecayTreeFitter/ConsDKpi')
dtt.Dstar.ConsDKpi.constrainToOriginVertex = True
dtt.Dstar.ConsDKpi.Verbose = True
dtt.Dstar.ConsDKpi.daughtersToConstrain = ['D0']
```

We now can tell the fitter to substitute one of the kaons in the D0 decay by a pion.

```
dtt.Dstar.ConsDKpi.Substitutions = {
    'Charm -> (D0 -> K- ^K+) Meson': 'pi+',
    'Charm -> (D~0 -> K+ ^K-) Meson': 'pi-'
}
```

In the dictionary that is passed to the `Substitutions` property of the fitter, the keys are decay descriptors, where the respective particle to be substituted is marked with a `^` . The values are the respective new particle hypotheses. The substitution will only work if you start from a decay descriptor that actually matches your candidates. However, you are allowed to generalise parts of the decay. Here we replaced `D*(2010)` with the more general `Charm` and the bachelor `pi-` is just represented by a `Meson` .

Note that the substitution mechanism does not understand the `CC` symbol. Both charge states have to be specified explicitely.

Running the ntuple script again with these additions gives you fit results for the re-interpreted decay.

Challenge

- Compare the outcome of the two fits with the different mass hypothesis
- Compare the fit quality between the correct and the the wrong hypothesis

The solution to this exercise `ntuple_DTF1.py` , is available here.

Running DaVinci on the grid

Learning Objectives

- Create a ganga job
- Submit a ganga job
- Waiting for ganga
- Find the job output

This lesson will teach you how to take our minimal DaVinci job and run it on the grid.

`ganga` is a program which you can use to interact with your grid jobs.

Before creating your first `ganga` job, open the script `ntuple_options.py`, obtained in the previous lesson, and comment out the lines taking the local input data: we will now use the data stored on grid.

Also, you need to know the path to your data from Bookkeeping. In our case the path is `/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09c/Trig0x6138160F/Reco16/Turbo03/Stripping28r1NoPrescalingFlagged/27163002/ALLSTREAMS.DST`. Note, that here the event type number should be located at the end of the path, which is not the case if you browse the bookkeping by `Event type`.

Finally, launch your grid proxy typing `lhcb-proxy-init` and enter your *grid certificate* password.

Great! Now you are ready to start ganga! Do it with:

```
$ ganga
```

After `ganga` has started you will be dropped into something that looks very much like an `ipython` session. `ganga` is built on top of `ipython` so you can type anything that is legal `python` in addition to some special commands provided by `ganga`.

To create your first `ganga` job, type the following:

```
j = Job(name='First ganga job')
myApp = prepareGaudiExec('DaVinci','v44r6', myPath='.')
j.application = myApp
j.application.options = ['ntuple_options.py']
bkPath = '/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09c/Trig0x6138160F/Reco16/Turbo03/Stripping28r1NoPrescalingF
lagged/27163002/ALLSTREAMS.DST'
data  = BKQuery(bkPath, dqflag=['OK']).getDataset()
j.inputdata = data[0:2]     # access only the first 2 files of data
j.backend = Dirac()
j.splitter = SplitByFiles(filesPerJob=1)
j.outputfiles = [LocalFile('DVntuple.root')]
```

This will create a `Job` object that will execute `DaVinci` configured with the option files given in `j.application.options` using a backend called `Dirac`, which is "the grid". Instead of specifying the files to process as part of the options file you have now to tell the `Job` about it. This allows `ganga` to split your job up by setting `j.splitter`, processing different files simultaneously. More details about the splitter are given in the next lesson. Note that data will be accessed using its path in the bookkeeping `bkPath`. In order to speed-up our job, only the first 2 elements (files) of `data` will be accessed; we don't need to look at much data here.

DaVinciDev folder

When you create a job using `prepareGaudiExec('DaVinci','v44r6', myPath='.')` you get the following message:

```
INFO     Set up App Env at: ./DaVinciDev_v44r6
```

`ganga` has created a folder with a local copy of the DaVinci v44r6 release. The content of it will be sent to the grid to ensure your job runs with exactly this configuration. We will use this folder for the following jobs and you will learn more about this in the Developing LHCb Software lesson.

Now you have created your first job, however it has not started running yet. To submit it type `j.submit()`. Now `ganga` will do the equivalent of `lb-run DaVinci/v44r6`, prepare your job and then ship it off to the grid.

Picking up a right platform

Early 2018, the default platform on most of lxplus machines was changed to `x86_64-slc6-gcc62-opt` (instead of `x86_64-slc6-gcc49-opt`), changing the version of the gcc compiler from 4.9 to 6.2. However, most of older DaVinci versions, anterior to v42r0, are not compiled for `x86_64-slc6-gcc62-opt`.

The list of platforms available for a certain DaVinci version (let's say `v38r0`), can be viewed by

```
$ lb-sdb-query listPlatforms DaVinci v38r0
```

In case you have a strong reason to use one of these DaVinci versions, few additional actions are needed to set up your ganga job properly.

When setting up your ganga job, add the following line after declaring the `j.application`:

```
j.application.platform = 'x86_64-slc6-gcc49-opt'
```

The default compiler platform for GaudiExec applications is `x86_64-slc6-gcc62-opt`.

While it runs, let's submit an identical job via slightly different method. Having to type in the details of each job every time you want to run it is error prone and tedious. Instead you can place all the lines that define a job in a file and simply run that.

Place the following in a file called `first-job.py`:

```
j = Job(name='First ganga job')
myApp = GaudiExec()
myApp.directory = "./DaVinciDev_v44r6"
j.application = myApp
j.application.options = ['ntuple_options.py']
bkPath = '/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09c/Trig0x6138160F/Reco16/Turbo03/Stripping28r1NoPrescalingF
lagged/27163002/ALLSTREAMS.DST'
data  = BKQuery(bkPath, dqflag=['OK']).getDataset()
j.inputdata = data[0:2]
j.backend = Dirac()
j.splitter = SplitByFiles(filesPerJob=1)
j.outputfiles = [LocalFile('DVntuple.root')]
j.submit()
```

Which you can execute and submit like so, from within a `ganga` session:

```
%ganga first-job.py
```

This will print an output similar to the one you saw when submitting the job from within `ganga`.

You can check on your jobs by typing `jobs` into a `ganga` console. This will list all of your jobs, their status, what kind of application they are and more.

You can get more detailed information about your job by typing `jobs($jobid)`, replacing `$jobid` with the `id` of the job you are interested in. For concretness we will assume you are interested in a job with jobid 787 in this example.

Once your job has finished its status will be `completed`. Check this by typing `jobs` or by printing out the status of one particular job:

```
print 'Status of my job:', jobs(787).status
```

The next thing to do is to find the output of your job. Two things can happen to files your job creates:

- They get downloaded by `ganga`, or
- they are stored "on the grid".

By default `ganga` will download most files below a size of XX MB. The rest will remain on the grid. Log files will almost always be downloaded.

To find where the files `ganga` downloaded are you can check the `outputdir` property of your job.

```
output = jobs(787).outputdir
print 'Job output stored in:', output
```

Take a look at the contents of this directory. Tip: this can be done from ganga using command `jobs(787).peek()`.

Using the Shell from IPython

IPython lets you execute shell commands from within the `ganga` session. This means you can list the contents of a directory without leaving ganga by typing `!ls /tmp/`. This will list the contents of the `/tmp` directory. In our case we can use this to list the contents of the job output directory with `!ls $output` as we stored the path in the variable `output`.

To look at the `root` file produced by the job start a new terminal, and type:

```
$ lb-run DaVinci/v44r6 $SHELL
$ root -l path/to/the/job/output
```

You need to setup `DaVinci` as we need ROOT version 6 to read the nTuple.

Getting help with ganga

To find out more take a look at the Ganga FAQ

Splitting a job into subjobs

Learning Objectives

- Learn how to process many files in parallel on the grid by splitting a job into many subjobs

In the previous lesson, you've submitted a job to the LHC grid. You will notice that the job will take a long time to finish. This is because it has to process many gigabytes of data.

`ganga` provides several *splitters* that implement strategies for processing data in parallel. The one we will use now is `SplitByFiles`, which spawns several subjobs, each of which only processes a certain number of files.

Apart from processing data faster, this will also allow you to work with datasets that are spread across several sites of the LHC grid. This is because a job can only access datasets that are available on the site that it runs on.

To activate a splitter, assign it to the `.splitter` attribute of your job:

```
j.splitter = SplitByFiles(filesPerJob=5)
```

Note that the specified number of files per job is only the allowed maximum. You will often get jobs with fewer files.

How do I choose the number of files per job?

Choose fewer files per job if possible, as this will allow you to finish sooner and reduces the impact of jobs failing due to grid problems. Setting `filesPerJob=5` should work well for real data, while `filesPerJob=1` should be good for signal MC.

Splitter arguments

The splitter has other useful arguments:

- `maxFiles` : the maximal total number of files. By default the splitter will run over all files in the dataset (which corresponds to the default value of -1)
- `ignoremissing` : boolean indicating whether it is appropriate to run if there are data files which are not accessible at the moment. This is important if it is necessary to make sure that the resulting ntuples correspond to the whole data/MC sample.

Now, when you run `j.submit()`, the job will automatically be split into several subjobs. These can be displayed by entering

```
jobs(787).subjobs
```

in `ganga`, where you have to replace 787 with the `id` of your main job.

You can access individual subjobs as in `jobs(787).subjobs(2)`. For example, to resubmit a failed subjob, you would run

```
jobs(787).subjobs(2).resubmit()
```

To access several subjobs at once, you can use the `.select` method:

```
jobs(787).subjobs.select(status='failed').resubmit()
```

This will resubmit all failed subjobs.

If you want to do something more complex with each subjob, a regular `for` -loop can be used as well:

```
for j in jobs(787).subjobs:
    print(j.id)
```

It's possible that some of your subjobs will be stuck in a certain state (submitting/completing/...). If that is the case, try to reset the Dirac backend:

```
jobs(787).subjobs(42).backend.reset()
```

If that doesn't help, try failing the job and resubmitting:

```
jobs(787).subjobs(42).force_status('failed')
jobs(787).subjobs(42).resubmit()
```

It can take quite a while to submit all of your subjobs. If you want to continue working in ganga while submitting jobs, you can use the `queues` feature to do just that. Simply call `queues.add` with the `submit` function of a job without adding parentheses, like this:

```
queues.add(j.submit)
```

Ganga will then submit this job (and its subjobs) in the background. Make sure not to close `ganga` before the submission is finished, or you will have to start submitting the rest of the jobs again later on.

Splitting your first job

Try splitting the `ganga` job from our previous lesson with `splitByFiles=1` (reference code) and submit it with `ganga` .

More Ganga

Learning Objectives

- Set the input data with BKQuery
- Use LHCbDatasets
- Set the location of the output of our jobs
- Set the location of your .gangadir
- Access output stored on the grid

As you already saw in the previous lesson, the input data can be specified for your job with the BKQuery tool. The path for the data can be found using the online Dirac portal and passed to the `BKQuery` to get the dataset. For example, to run over the Stripping 21 MagUp, Semileptonic stream

```
Ganga In [3]: j.inputdata = BKQuery('/LHCb/Collision12/Beam4000GeV-VeloClosed-MagUp/Real Data/Reco14/Stripping21r0p1a/90000000
/SEMILEPTONIC.DST').getDataset()
Ganga In [4]: j.inputdata
Ganga Out [4]:
 LHCbDataset (
   depth = 0,
   treat_as_inputfiles = False,
   persistency = None,
   files = [3717 Entries of type 'DiracFile'] ,
   XMLCatalogueSlice =    LocalFile (
     namePattern = ,
     compressed = False,
     localDir =
   )
 )
```

This is a list of `DiracFile` , the Ganga object for files stored on the grid. We can access one locally via the `accessURL` :

```
Ganga In [5]: j.inputdata[0].accessURL()
Ganga Out [5]: ['root://bw32-4.grid.sara.nl:1094/pnfs/grid.sara.nl/data/lhcb/LHCb/Collision12/SEMILEPTONIC.DST/00051179/0000/0
0051179_00006978_1.semileptonic.dst']
```

The returned path can be used by Bender to explore the contents of the DST, as in the Interactively exploring a DST lesson.

In the previous lesson we looked at the location of the ouput with `job(782).outputdir` . This location points us to the `gangadir` where ganga stores information about the jobs and their output. If we have lots of jobs with large files, the file system where the gangadir is located will quickly fill up.

Setting the gangadir location

The location of the `gangadir` can be changed in the configuration file '~/.gangarc'. Just search for the `gangadir` attribute and change it to where you like (on the CERN AFS the `work` area is a popular choice).

To avoid filling up the filespace, it is wise to put the large files produced by your job somewhere with lots of storage - the grid. You can do so by setting the `outputfiles` attribute:

```
j.outputfiles = [DiracFile('*.root'), LocalFile('stdout')]
```

The `DiracFile` will be stored in your user area on the grid (with up to 2TB personal capacity). The wildcard means that any root file produced by your job will stay on the grid. `LocalFile` downloads the file to your `gangadir`, in this case the one called `stdout`. You can access your files stored on the grid with the `accessURL()` function as before. For example, to access the location of the output `.root` file of a specific subjob, one can use

```
jobs(787).subjobs(15).outputfiles[0].accessURL()
```

This is a very useful feature: you do not have to download your files from the grid in order to merge them locally! This takes a lot of time and disk space. Instead, one can get a list of URLs from each subjob, and pass them to the `TChain` or use the `hadd` method of ROOT[1]. Ganga has a shortcut to access the list of all `.root` files from all the subjobs of a given job:

```
jobs(787).backend.getOutputDataAccessURLs()
```

Small files are downloaded as standard: `.root`, logfiles etc. Files that are expected to be large (with extensions `.dst` etc) are by default kept on the grid as Dirac files. In general, you are encouraged to keep your large files on the grid to avoid moving large amounts of data around through your work area.

More information on the `DiracFile` usage is provided in the next lesson.

Getting help with ganga

To find out more take a look at the Ganga FAQ

[1]. Merging your files with `hadd` will be significantly faster if you run it with the option telling ROOT to use the same compression level in that output file as is used for the input files. This can be done using the `-fk` option. If running on lxplus you will need to get a newer ROOT version that supports this option by using: `lb-run ROOT hadd -fk output.root input/*.root` ↩

Storing large files on EOS

Learning Objectives

- Run a ganga job which puts output onto EOS
- Open and view the files on EOS

During a real analysis the output of your jobs will quickly grow beyond what fits onto your AFS space. CERN provides you with 2TB of space on a set of hard drives called the EOS service and a grid storage quota of 2TB.

To retrieve a job outputfile, one can use two types of files:

- `LocalFile` : the standard one with the output file directly downloaded to the `gangadir` .
- `DiracFile` : the output file is stored directly on the grid and be accessed through the XRootD protocol.

In this lesson, we will focus on the use of `DiracFile` in `ganga` to manage big output files.

We can reuse what has been done to run a DaVinci job on the grid and adapt the `j.outputfiles` part.

To add the DiracFile in the configuration of the job we just need:

```
j = Job(name='First ganga job')
myApp = GaudiExec()
myApp.directory = "./DaVinciDev_v44r6"
j.application = myApp
j.application.options = ['code/davinci-grid/ntuple_options_grid.py']
bkPath = '/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09c/Trig0x6138160F/Reco16/Turbo03/Stripping28r1NoPrescalingF
lagged/27163002/ALLSTREAMS.DST'
data  = BKQuery(bkPath, dqflag=['OK']).getDataset()
j.inputdata = data[0:2]     # access only the first 2 files of data
j.backend = Dirac()
j.outputfiles = [
    DiracFile('DVntuple.root')
]
j.splitter = SplitByFiles(filesPerJob=5)
j.submit()
```

When the job is completed, no output is dowloaded but some interesting information are provided by typing `j.outputfiles[0]` :

```
DiracFile(namePattern='DVntuple.root',
          lfn='/lhcb/user/a/another/2018_11/146255/146255492/DVntuple.root',
          localDir='/afs/cern.ch/user/a/another/gangadir/workspace/another/LocalXML/129/output')
```

Apart from the `namePattern` which was set during the configuration of the job, we can retrieve the `localDir` which is the path in your gangadir to the output of the job and its `lfn` which stands for Logical File Name.

This LFN can then be given as an argument in order to download the file. But more important in most of the cases, you don't even need to download the file thanks to the `accessURL` function which will give you the URL of your output file by typing `j.outputfiles[0].accessURL()` :

```
['root://eoslhcb.cern.ch//eos/lhcb/grid/user/lhcb/user/a/another/2018_11/146255/146255492/DVntuple.root']
```

This URL can be directly used in your ROOT script with the help of the XRootD protocol as follows:

```
TFile::Open("root://eoslhcb.cern.ch//eos/lhcb/grid/user/lhcb/user/a/another/2018_11/146255/146255492/DVntuple.root")
```

Use of the XRootD protocol

In order to access files on every grid site with the XRootD protocol, be sure to have a valid proxy using `lhcb-proxy-init`.

On `lxplus` EOS is mount under `/eos`. If you list the contents of the `/eos` directory you should see various experiments areas, and the user area:

```
lhcb ship user
```

There are two useful workspaces on EOS:

- `/eos/user` which is your private EOS space, also accessible via CERNBox, and has 1TB of space;
- `/eos/lhcb` which is the LHCb area and contains the following sub-directories:
  - `/eos/lhcb/user` which is your user area, where you can store your tuples, and has 2TB of space;
  - `/eos/lhcb/wg` which contains the Working Group areas (e.g. for preserving tuples for completed analyses);
  - `/eos/lhcb/grid/` which is the path to the CERN grid site; and many others.

Let's inspect your user area:

```
$ ls -al /eos/lhcb/user/a/another
```

Files stored on EOS can be open in `ROOT` the usual way:

```
$ root /eos/lhcb/user/a/another/myfavouritefile.root
```

You may also use the following path which is independent of the EOS mount (the mount is in general not very reliable):

```
$ root root://eoslhcb.cern.ch//eos/lhcb/user/a/another/myfavouritefile.root
```

Developing LHCb Software

Learning Objectives

- Learn how to work with and modify LHCb software projects and packages
- Learn how to find and search the source code and its documentation

Prerequisites

Before starting, you should have a basic understanding of how to use `git`, similar to what has been taught during the Starterkit.

In this lesson, we'll show you a complete workflow for developing the LHCb software using the `git` version control system. At LHCb, we use Gitlab to manage our git repositories. Among other features, Gitlab allows you to browse the source code for each project, and to review new changes (called *merge requests*) in a convenient web interface. You can find the CERN Gitlab instance at https://gitlab.cern.ch.

In principle, there are multiple ways of interacting with the LHCb software repositories:

1. A vanilla git workflow using only the standard git commands. This requires you to `clone` and compile an entire LHCb project at a time.
2. An LHCb-specific workflow using a set of `lb-*` subcommands. This allows you to check out individual packages inside a project, and streamlines the modification of a few packages at a time. (This is closer to the previously used `getpack` command.)

Here, we want to focus on the second workflow. The first workflow will be discussed briefly at the bottom of this page. Note that, although the lb-git commands are much easier for small changes to existing packages where recompiling an entire project would be cumbersome, for any serious development the usage of vanilla git is much more stable. Please consider using it if you can spare the compilation time.

Initial setup

Before jumping in by creating a project in Gitlab, you should make sure that your local git configuration and your settings on Gitlab are sufficiently set up.

- Your name and email address should be set up in your local `git` configuration. To ensure that this is the case, run

  ```
  git config --global user.name "Your Name"
  git config --global user.email "Your Name <your.name@cern.ch>"
  ```

  and put in your information.

- Next, connect to https://gitlab.cern.ch and log in with your CERN credentials. Visit https://gitlab.cern.ch/profile/keys and add an SSH key.

- Finally, run this LHCb-specific configuration command:

  ```
  git config --global lb-use.protocol ssh
  ```

This makes sure the LHCb commands use the ssh protocol instead of https.

This lesson introduces you the commands:

- `lb-dev` for setting up a new development environment
- `git lb-use` and `git lb-checkout` for downloading LHCb software packages

If you want to make changes to a software package, you will need to set up a development environment. `lb-dev` is your friend here:

```
lb-dev --name DaVinciDev DaVinci/v44r6
```

The output should look similar to this:

```
Successfully created the local project DaVinciDev in .

To start working:

  > cd ./DaVinciDev
  > git lb-use DaVinci
  > git lb-checkout DaVinci/vXrY MyPackage

then

  > make
  > make test

and optionally (CMake only)

  > make install

You can customize the configuration by editing the files 'build.conf' and
'CMakeLists.txt' (see http://cern.ch/gaudi/CMake for details).
```

lb-dev created local projects are Git repositories

When `lb-dev` creates the local project directory and create the initial files there, it also calls `git init` and commits to the local Git repository the first version of the files (try with `git log` in there).

You can then use git to keep track of your development, and share your code with others (for example with a new project in gitlab.cern.ch).

Follow those instructions to compile the software:

```
cd DaVinciDev
git lb-use DaVinci
make
```

Once that's done, you can do

```
./run bash -l
```

inside the directory. This will (similar to `lb-run`) give you a new bash session with the right environment variables set, from which you can run project-specific commands such as `gaudirun.py`.

Your new development environment won't be very useful without any software to modify and build. So let's check out one of the existing LHCb packages! These are stored in the LHCb Git repositories.

In order to obtain the source code of the package you want to work on, we'll use the Git4LHCb scripts. These are a set of aliases, starting with `git lb-` , that are designed to make developing LHCb software easier. For example, if you want to write a custom stripping selection, execute the following in the `DaVinciDev` directory:

```
git lb-use Stripping
git lb-checkout Stripping/master Phys/StrippingSelections
make configure
```

Under the hood, `git lb-use` will add the `Stripping` repository as a remote in git – check this with `git remote -v` ! `git lb-checkout` will then perform a *partial* checkout of the master branch of the Stripping repository, only adding the files under `Phys/StrippingSelections` .

Which project to use in `git lb-use` ?

The project name to pass to `git lb-use` depends on the directories you want to check out and work on, and not on the project name you passed to `lb-dev` . Moreover you can call `git lb-use` several times for different remote projects in the same local project:

```
lb-dev --name DaVinciDev DaVinci/v44r6
cd DaVinciDev
git lb-use Analysis
git lb-use Stripping
git lb-use DaVinci
```

Not that in order for this to work, projects you specify in `lb-use` may not depend on the project you specify in `lb-dev` . In other words, the top-level project should be at the top of the dependency chain.

You can now modify the `StrippingSelections` package and run `make purge && make` to build it with your changes. You can test your changes with the `./run` script. It works similar to `lb-run` , without the need to specify a package and version:

```
./run gaudirun.py options.py
```

What if `git` asks for my password?

Make sure you succesfully completed the instructions under initial setup.

If you have made changes that you'd like to be integrated into the official LHCb repositories, you can use `git lb-push` to push it to a new branch in the central git repository. But please read the instructions in the TWiki page first.

Depending on the project, you may be required to document your changes in the release notes which are found in `doc/release.notes` .

Note that no-one has permission to push directly to the `master` branch of any project. In order to get your changes merged there from the branch to which you `lb-push` ed, you need to create a merge request, so the project maintainer can inspect your code. This can be done on the project repository web page, for example.

Quick link to create a merge request

When pushing to a branch in a project in Gitlab you will see a message like:

```
remote:
remote: Create merge request for my-branch:
remote:   https://gitlab.cern.ch/lhcb/Stripping/merge_requests/new?merge_request%5Bsource_branch%5D=my-branch
remote:
```

You can use the URL in the message to quickly create a merge request for the changes you just pushed.

When your merge request is approved (which can be after some additional commits on your part), your changes are part of the `master` branch of the respective project, and your contributions are officially part of the LHCb software stack. Congratulations!

Nightlies

It is advisable to test new developments on the so-called nightly builds. Each project is built overnight (hence the name), and all pending merge requests are applied. You can use a nightly build version of a project with:

```
lb-dev --nightly lhcb-head DaVinci/HEAD
```

A more detailed description of the command is found here:

- SoftwareEnvTools

Sometimes mistakes happen and the committed code is either not compiling or does not do what it is supposed to do. Therefore the nightly tests are performed. They first try to build the full software stack.

If that is successful, they run some reference jobs and compare the output of the new build with a reference file. The results of the nightly builds can be found here.

- Nightly builds summaries

If the aim of the commit was to change the ouput, e.g. because you increased the track reconstruction efficiency by a factor of two, mention it in the merge request description, such that the manager of the affected project can update the reference file.

If you want to take a look the source code, without checking it out, you can easily access the repository through the Gitlab web interface. This website also provides search functionality, but the output is not always easy to read, especially if it returns many hits. To search a project much quicker, you can use `Lbglimpse`. It allows you to search for a given string in the source code of a particular LHCb project.

```
Lbglimpse "PVRefitter" DaVinci v44r6
```

This works with every LHCb project and released version. Since it's a shell command, you can easily process the output using `less`, `grep`, and other tools.

To get an idea of how a certain component of the LHCb software works, you can also access the doxygen documentation. One set of doxygen web pages is generated for several related projects, and is linked in all the projects web sites, like for DaVinci. See also the LHCb Computing web page for a list of projects.

Working with a full project checkout

The `lb-git` commands are not strictly necessary, but they're very convenient if you just want to quickly edit one package.

Otherwise you'd have to build the entire project in which the package is residing, instead of using the precompiled version. However, if you develop across multiple packages, or want to use more sophisticated `git` commands, nothing prevents you from checking out an entire project – just don't be surprised if it takes O(hours) to compile!

To check out a project, run the following:

```
git clone https://:@gitlab.cern.ch:8443/lhcb/DaVinci.git
```

replacing `DaVinci` with the project name of your choice. Next, initialise and compile it:

```
lb-project-init make
```

optionally followed by `make test` to run the tests and/or `make install` to install it to the `InstallArea` directory. That's all! You now have a vanilla git repository containing all the source files of the project.

Asking good questions

Learning Objectives

- How to ask a good question
- Where to ask questions

Eventually you will get stuck when trying to do something. This lesson is about how to get help with getting unstuck.

People love helping others. Below some tips on how to improve your chances of getting a good answer. The answers you get will depend very much on the way you ask your question.

Mailing lists

For LHCb specific questions your best bet are the LHCb mailing lists like: lhcb-distributed-analysis@cern.ch, lhcb-davinci@cern.ch, and lhcb-soft-talk@cern.ch. A recent alternative to the mailing lists is the Mattermost chat having dedicated channels ('davinci', 'Distributed Analysis' etc). It is a great place to ask your question if you're not confident enough to write an email to the mailing list, or you'd just prefer a more informal setting.

For more general questions Stack Overflow and Google are good starting places.

LHCb Q&A

Another experimental alternative to the mailing lists is the LHCb Questions and Answers website. It works like Stack overflow, but focuses on LHCb-specific questions. You can post your question there and you should usually receive an answer within a few hours or days.

The title/subject is the first thing people will see of your question. If it is not interesting, they will not read the rest. If you are struggling with a good title, write it last! Having written the rest of your email will give you a better idea of what the one sentence summary is.

Grammar and spelling

This is a no brainer. You want to make a good impression: someone who made an effort and values other people's time. If you aren't comfortable with writing in English, ask a friend to proof-read your email for you.

Most people like hard problems and thought-provoking questions. So give them an interesting question to chew on, and they will love it.

Despite this, mailing lists have a reputation for a hostile tone and an air of arrogance. People are hostile towards people who seem unwilling to think for themselves or did not do their homework before asking their question.

Volunteers

Remember people are volunteering their time to help you. They have busy lives and there are a lot of questions. So they filter ruthlessly. Make sure your question is the most interesting one out there, and people will choose to help you.

It is OK that you are not technically competent, what you need to show is that you have the skills to become competent: alert, thoughtful, observant, willing to be an active partner in developing a solution.

Before posting your question try:

1. Searching the archives of the forum/mailing list
2. Searching the web
3. Finding the answer in a FAQ
4. Finding the answer by experimenting
5. Asking a skilled friend
6. Or reading the code

When asking your question, mention which of these steps you have tried. It will help demonstrate that you are not lazy and put in some effort. If you learned something from trying this, mention it!

The beginning of your email should explain what you are trying to do and why, as well as where the problem occurs. Often it is useful to describe the big goal, and not the particular step you are stuck with.

Help others reproduce your problem by including the necessary details. If the problem is with code you wrote, include it. However do not post all of your program. Try and make a minimal example that demonstrates the problem. Stack Overflow has a good guide on creating a minimal, complete, and verifiable example.

You have to be precise. Do not simply dump all possible information that might be relevant. Vague questions receive vague answers. Being precise is useful for at least three reasons. One: being seen to invest effort in simplifying the question makes it more likely you'll get an answer, Two: simplifying the question makes it more likely you'll get a useful answer. Three: In the process of refining your bug report, you may develop a fix or workaround yourself.

When people post replies or questions about your problem, follow them up. If you manage to solve your problem, tell the mailing list.

Love letters also known as private replies

Please do not reply to questions privately. We are trying to solve a technical issue, not writing lover letters. If you take the discussion off the mailing list future generations will just see the question and no solution. You will also miss out on help from others that did not chime in to your first question but might be able to help later on.

Once the problem is solved post a final message saying that the problem is solved and what the solution is.

A lot of the material in this topic was taken from the following, excellent guides to asking good questions:

- How To Ask Questions The Smart Way by Eric Raymond
- Writing the perfect question by Jon Skeet
- How do I ask a good question? on Stack Overflow

Early career, gender and diversity

Learning Objectives

- Know about the existence of the ECGD office, where ECGD means Early Career, Gender and Diversity.
- Meet the ECGD representatives
- Learn how/when the ECGD office can be useful for you.

The aim of the ECGD is to help LHCb achieve a working environment in which all LHCb members can thrive. This includes especially those experiencing discrimination on grounds of gender, sexual orientation, ethnicity, disability, creed, cultural background or other factors. The ECGD also helps early-career physicists who wish to, eventually, escape the precarious life of repeated short-term contracts and reach a permanent position. You can find more information and a lot of documentation in the ECGD web page.

The ECGD office

It's made up of two people. You'll meet one of them during the Starterkit. In general, the ECGD "officers" will be happy to receive your emails, phone calls or meet in person.

Contribute to this lesson

Learning Objectives

- Reporting a mistake
- How to look at the source of these lessons
- How to modify a lesson

These lessons are not really about software, they are about people. If you have followed along until this point you are more than qualified to edit the lessons. There are probably several mistakes in these lessons, or they will be outdated soon. Keeping the lessons working and fixing all mistakes is a monumental task for one single person.

We need you! You now know everything you need to in order to contribute. Take advantage of this.

The source of this lesson is hosted on GitHub: lhcb/starterkit-lessons.

Submitting a bug report

If you spot something that is wrong, create a bug report on the issue tracker This is super simple and makes it easy for everyone to keep track of what is broken and needs fixing. It also increases your chances of someone posting a solution.

You do not need anyone's permission to start making changes. You can start directly. If you want to edit something the first thing to do is to create a fork of the repository. Visit lhcb/first-analysis-steps and click the "Fork" button at the top right.

A fork is simply a copy of the original repository. It works just as well as the original. Clone the repository to your computer to start making changes:

```
$ git clone https://YOURUSERNAME@github.com/YOURUSERNAME/starterkit-lessons.git
```

As you can see each lesson has its own `.md` file. The source of this lesson is in `CONTRIBUTING.md`. It is a simple text file with a few clever lines with special meaning.

The format the files are written in is called Markdown. It is a very simple language, which adds some basic formatting to text files. `**Bold text**` leads to **Bold text**, `_Italic_` is *italic* and `[the search engine](http://google.com)` makes a link to the search engine.

Trying it out live

Try out Markdown live in your browser with Dillinger.

If you want to see what your changes look like, simply paste a lesson to Dillinger.

If you found something to improve, create a new branch with a fitting name (replace `fixing-typos`):

```
$ git checkout -b fixing-typos
```

Once you are done with your changes, commit them. To commit use `git add 00-lesson-you-edited.md` and then `git commit`. After `git push`'ing it, visit your copy of the repository on github: `https://github.com/YOURUSERNAME/starterkit-lessons`.

In order to test your changes, you can run the starterkit website locally. To do so, first install the required python packages. This can be done by passing the `requirements.txt` that is present in the git repository to pip

```
pip install -r requirements.txt --user
```

Besides these packages you need to install pandoc on your system. Once all requirements are satisfied, run

```
make preview
cd _site
python -m http.server
```

in the top level of the git repository.

This will start a webserver on your computer. Then open you web browser and navigate to

```
localhost:8000
```

to see the website.

Next you want to create a pull request. The github documentation is excellent, so we will not duplicate it here. Simply follow the guide: how to create a pull request.

Now we can see your proposed changes and will probably leave you some comments. Once everyone is happy, one of the main starterkit'ers will merge your pull request. Congratulations, you have successfully contributed!

Second Analysis Steps

These are the lessons for the second-stage workshop of the Starterkit series. They build on those from the first workshop, teaching LHCb software that's more advanced and more focused on specific tasks.

Unlike the first workshop, there may be some lessons here that aren't applicable to everyone's analysis, but all the lessons should still provide a useful insight in to how things work under the hood. It may also be that some lessons don't depend on any others; the prerequisites will be clearly stated at the beginning of each lesson.

If you have any problems or questions, you can open an issue on the GitHub repository where these lessons are developed, or you can send an email to `lhcb-starterkit@cern.ch`.

Prerequisites

Before starting, you should be familiar with the first analysis steps and satisfy all of its prerequisites.

Using git to develop LHCb software

Learning Objectives

- Learn how to clone specific LHCb packages to a local development directory
- Learn how to make changes and upload them to be reviewed by others

Prerequisites

Before starting, you should have a basic understanding of how to use `git`, similar to what has been taught during the starterkit.

In this lesson, we'll show you a complete workflow for developing the LHCb software using the `git` version control system. At LHCb, we use GitLab to manage our git repositories. Among other features, GitLab allows you to browse the source code for each project, and to review new changes (called *merge requests*) in a convenient web interface. You can find the CERN GitLab instance at https://gitlab.cern.ch.

In principle, there are multiple ways of interacting with the LHCb software repositories:

1. A vanilla git workflow using only the standard git commands. This requires you to `clone` and compile an entire LHCb project at a time. This approach is recommended when your development is expected to last multiple days or more or if you collaborate with others.
2. An LHCb-specific workflow using a set of `lb-*` subcommands. This allows you to check out individual packages inside a project, and streamlines the modification of a few packages at a time. This approach is mostly appropriate for quick and isolated changes. (This is closer to the previously used `getpack` command.)

Here, we want to focus on the second workflow.

Before jumping in by creating a project in GitLab, you should make sure that your local git configuration and your settings on GitLab are sufficiently set up:

Your name and email address should be set up in your local `git` configuration. To ensure that this is the case, run

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Name <your.name@cern.ch>"
```

and put in your information.

Next, connect to https://gitlab.cern.ch and log in with your CERN credentials.

Visit https://gitlab.cern.ch/profile/keys and add an SSH key.

Run this LHCb-specific configuration command:

```
git config --global lb-use.protocol ssh
```

This makes sure the LHCb commands use the ssh protocol instead of https.

In the GitLab web interface, create a new project by clicking on the "New project" button. Give your project the name "LHCbSK".

We will now set up a local LHCb development area that is connected to the git repository you just created. In order to create a new dev

environment, run

```
$ lb-dev --list LHCb
```

to see which versions of the LHCb project are available. Pick the newest one and run

```
$ lb-dev --name LHCbSK LHCb/<version>
```

where you need to insert the version you picked earlier. This will automatically run `git init` to create a new git repository with some initial files (`Makefile` , `CMakeLists.txt` , etc.). In order to connect your repository with the remote one, run

```
cd LHCbSK
git remote add origin ssh://git@gitlab.cern.ch:7999/<username>/LHCbSK.git
```

where you need to substitute your username. Run

```
git push -u origin master
```

once to define `master` as the remote branch that we should push to by default.

Pushing your dev project to GitLab

You don't necessarily need to create a remote git repository on GitLab for your local dev project, especially if its lifetime will be short. However, in case you plan to collaborate with someone, a remote repo where you can synchronize your work is a must. Pushing to GitLab is also good practice if you use this dev project for ganga's `GaudiExec` .

For the purpose of this tutorial, we've set up an `SKTest` project containing the package `TheTestPackage` . In order to be able to work on parts of the `SKTest` project, first run

```
git lb-use SKTest
```

This will define a remote `SKTest` and fetch the git repository from GitLab.

You can then run

```
git lb-checkout SKTest/master TheTestPackage
```

to check out the files inside the `SKTest` project that belong to the `TheTestPackage` package. This performs `git checkout` under the hood and commits the files into the local (synthetic) branch. You can check how this looks in the history with `git log` .

You can now run

```
make
```

to build the project, make some changes, run `make` again, etc. You can also use

```
make test
```

to run the tests. Once you're happy with the changes, run

```
git add <your-changed-files>
```

where you have to specify all files with changes that you want to store, and

```
git commit
```

to store them in a new commit. This will open up a text editor that will allow you to type in a commit message.

Committing often

There's no cost to committing often. You should try to make a new commit every time you've made modifications that can be considered a single unit of changes.

Now push your changes to your newly created GitLab project

```
git push
```

and check that they appear on the website.

Once you want to upload your commits to the SKTest project for review, run

```
git lb-push SKTest <username>-new-feature
```

to create a new `<username>-new-feature` branch on the main GitLab repository and upload your commits to it. This will calculate the changes between your local dev repository and the remote project repository, and create corresponding commits for the remote project. You can now create a merge request by going to https://gitlab.cern.ch/LHCb-SVN-mirrors/SKTest/merge_requests/new, selecting the `<username>-new-feature` branch as the source and `master` as the target. Add a title for your merge request and explain what you've done in the main text.

Merge conflicts

- Working with a partner, try to create a merge conflict by making changes to the same file. If you don't have someone else to work with, you can simulate this by creating a second development area.
- Now, try to resolve the merge conflict. This is a bit complicated because of the way the local repository is set up. Take a look at https://twiki.cern.ch/twiki/bin/view/LHCb/Git4LHCb#Replacement_for_svn_update_in_lo for pointers.

Working with entire projects

Working with entire projects is just as simple, but requires a little more care with the environment. Most importantly, be prepared that the project repo clones and the build products may need a few GB of disk space and that the compilation may take a long time. A step-by-step guide is available at https://twiki.cern.ch/twiki/bin/view/LHCb/Git4LHCb#Building_everything_locally_exam. A more integrated approach that automatically takes care of the environment is https://gitlab.cern.ch/lhcb/upgrade-hackathon-setup and a variation of it that lets you use some of the nightly builds to spare compiling many projects is at https://gitlab.cern.ch/lhcb-HLT/trigger-dev.

Building your own decay

Learning Objectives

- How existing containers of particles can be filtered.
- How new particles are made by combining existing particles.
- How to express particle selections and combinations in options files, the Stripping, and the trigger.

As you might imagine, combining reconstructed tracks under some physical hypothesis is quite a common operation for a particle physicist to perform.

We've already manipulated the result of such an operation, so-called composite particles, and in this short series of lessons we'll see how you can create such composites yourself. The knowledge you'll gain will give you the ability to understand the large body of existing particle combination and filtering code, as well as the ability to use the Stripping and HLT2 selection frameworks to write new combinations for your analysis.

Data processing flow

Some charged and neutral particles are created in 'the reconstruction', either Brunel or the beginning of HLT1 and HLT2. These include 'stable' particles like electrons, protons, and charged kaons and pions, and neutrals like photons and neutral pions.

Why aren't 'composite' particles, like D and B mesons, also created in Brunel? What are the advantages and disadvantages of creating particle combinations in a separate step?

Building your own decay

The Selection Framework

Learning Objectives

- Learn the concepts behind the LHCb selection framework
- Learn the advantages of the LHCb selection framework

In order to perform most physics analyses we need to build a *decay chain* with reconstructed particles that represents the physics process we want to study. In LHCb, this decay chain can be built through `LHCb::Particle` and `LHCb::MCParticle` objects that represent individual particles and contain links to their children, also represented by the same type of object.

We'll learn all the concepts involved by running through our usual full example of the $$D^\\ast\rightarrow D^0(\rightarrow K^{-} K^{+}) \pi$$ decay chain.

The LHCb approach to building decays is from the bottom up. Therefore, to build $$D^\\ast\rightarrow D^0(\rightarrow K^{-} K^{+}) \pi$$ we need to

1. Get input pions and kaons and filter them according to our physics needs.
2. Combine two kaons to build a $$D^0$$, and apply selection cuts to it.
3. Combine this $$D^0$$ with a pion to build the $$D^\\ast$$, again filtering when necessary.

To do that, we need to know a little bit more about how the LHCb analysis framework works.

As discussed in the Gaudi introduction, `Gaudi` is based on the event-by-event sequential (chained) execution of algorithms wrapped in a `GaudiSequencer`, which takes care of handling the execution order such that processing stops when an algorithm is *not passed*. However, it does not handle the data dependencies between these algorithms nor does it give easy access to them. To solve this problem, the Selection Framework was created, and it is based on two types of objects: `Selection` and `SelectionSequence`:

- The `Selection` is the basic unit of the framework. It uses DaVinci algorithms to process `LHCb::Particles` and writes them to a TES location easily findable through its `outputLocation` method. Additionally, it knows about other `Selections` that it requires to pass in order to obtain input particles through its `RequiredSelections` argument. A `Selection` requires *all* of its `RequiredSelections` to pass.
- The `SelectionSequence` takes a `Selection` object, resolves its `Selection` requirements, and builds a flat, chained and ordered list of `Selections`. It then exports (via the `selection` method) a self-contained `GaudiSequencer` with all the algorithm configurables necessary to run the selection. It also makes the output locations of the data written by the selection chain available via the `outputLocations` method.

Additionally, we need algorithms to perform the particle combination and filtering (or other data processing capabilities), since `Selection`s don't do any work by themselves, but just organize the dependencies properly. Two of the most important data processing algorithms are `CombineParticles` and `FilterDesktop`, which will be discussed in the next lesson.

The advantages of using this framework are several:

- Building the algorithms with bare Configurables and chaining their `Input` and `Output` is a responsibility of the users. This means the user needs to put the algorithms in the correct order, not only chaining properly the inputs/outputs but also executing the algorithms in sequence. The use of the Selection Framework places all algorithms in *correct sequences in the optimal order*, allowing to achieve the maximal possible CPU efficiency.
- Its *reusability*: one can use/re-use the same code for data analysis on "user"-level for MC processing for building of Stripping lines, since usually only the input particles need to be changed.

- *Easiness of debugging*: one can visualize the selection chain, for example using the `PrintSelection` algorithm for debugging of the data flow.
- Some (advanced) tasks are *virtually impossible* to do without it, such as accessing some features for MC µDST (MCTruth for inclusive lines, for example) or applying the momentum scaling in Turbo.

The LHCb singletons and usability

A big part of the reusability of the Selection objects is thanks to how the LHCb framework is designed: all LHCb algorithms need an explicit and unique name because they are *singletons* (a singleton is a software design pattern that restricts the instantiation of a class to one object). As a consequence of this, only one algorithm with a given name can be instantiated.

This allows to reuse and reload algorithms that have already been created in a configuration sequence. For example:

We could create a generic selection for building D0 with a known name, put it in a `build_d0.py` file and use (and even modify) it in another file loaded in the same `gaudirun.py` call. We could write the data-only parts of our selection in one file and the MC-only separately (typically inputs are different), but setting the same names for the algorithms. Then, our `DecayTreeTuple` code could be common, as the selection would be loaded "by-name".

This is very useful to build complicated configuration chains, but it's *very easy* to have problems if our selection/algorithm names are not unique. Therefore, it's very important to pay attention to the algorithm names (we will see how to minimize this problem in the final Selection Framework lesson).

Building your own decay

A Historical Approach

Learning Objectives

- Build a decay chain using the most basic tools
- Understand the limitations and problems of these tools

Lesson caveat

In this lesson we will explain how to work with the most basic building blocks of the Selection Framework. This is not the most optimal way to use it, but it is included here because their use is very generalized, for example in the Stripping, and understanding them is very useful for understanding most of our current code. At the end of this lesson its shortcomings will be highlighted and a better way to approach the problem will be presented in the following lesson.

Now we'll learn to apply the concepts of the Selection Framework by running through a full example: using the DST files from the Downloading a file from the Grid lesson, we will build our own $$D^\ast\rightarrow D^0(\rightarrow K^{-} K^{+}) \pi$$ decay chain from scratch. Get your LoKi skills ready and let's start.

Getting started

There's no need to download the files from the Grid for this lesson. We can simply open them using the `root` protocol thanks to the fact that they are replicated at CERN:

```
from GaudiConf import IOHelper
IOHelper().inputFiles([('root://eoslhcb.cern.ch//eos/lhcb/grid/prod/lhcb/MC/2016/ALLSTREAMS.DST/00062514/0000/00062514_
00000008_7.AllStreams.dst')],
                      clear=True)
```

The starting code for this exercise can be found here.

Our input pions and kaons can be imported from the `StandardParticles` package:

```
from StandardParticles import StdAllNoPIDsPions as Pions
from StandardParticles import StdAllLooseKaons as Kaons
```

This is an ideal way to get pre-made particles with the standard LHCb configuration.

Where do `StandardParticles` come from?

One important type of `Selection` is the `AutomaticData`, which builds objects from their TES location using a centrally predefined algorithm. The `StandardParticles` / `CommonParticles` packages (one imports the other), which you can find here, allow to access premade particles with reasonable reconstruction/selections for us to use with `AutomaticData`.

For example, in our specific case, we use the `AutomaticData` class with the `Phys/StdAllNoPIDsPions/Particles` and `Phys/StdAllLooseKaons/Particles` locations to access the output of the `StdAllNoPIDsPions` and `StdAllLooseKaons` algorithms, respectively (see here and here). Therefore, the following code would be equivalent to what we have used:

```
from PhysConf.Selections import AutomaticData
Pions = AutomaticData('Phys/StdAllNoPIDsPions/Particles')
Kaons = AutomaticData('Phys/StdAllLooseKaons/Particles')
```

Once we have the input kaons, we can combine them to build a $$D^0$$ by means of the `CombineParticles` algorithm. This algorithm performs the combinatorics for us according to a given decay descriptor and puts the resulting particle in the TES, allowing also to apply some cuts on them:

- `DaughtersCuts` is a dictionary that maps each child particle type to a LoKi particle functor that determines if that particular particle satisfies our selection criteria. Optionally, one can specify also a `Preambulo` property that allows us to make imports, preprocess functors, etc (more on this in the LoKi functors lesson). For example:

  ```
  d0_daughters = {
    'K-': '(PT > 750*MeV) & (P > 4000*MeV) & (MIPCHI2DV(PRIMARY) > 4)',
    'K+': '(PT > 750*MeV) & (P > 4000*MeV) & (MIPCHI2DV(PRIMARY) > 4)'
  }
  ```

- `CombinationCut` is a particle array LoKi functor (note the `A` prefix, see more here) that is given the array of particles in a single combination (the *children*) as input (in our case two kaons). This cut is applied before the vertex fit so it is typically used to save CPU time by performing some sanity cuts such as `AMAXDOCA` or `ADAMASS` before the CPU-consuming fit:

  ```
  d0_comb = "(AMAXDOCA('') < 0.2*mm) & (ADAMASS('D0') < 100*MeV)"
  ```

- `MotherCut` is a selection LoKi particle functor that acts on the particle produced by the vertex fit (the *parent*) from the input particles, which allows to apply cuts on those variables that require a vertex, for example:

  ```
  # We can split long selections across multiple lines
  d0_mother = (
    '(VFASPF(VCHI2/VDOF)< 9)'
    '& (BPVDIRA > 0.9997)'
    "& (ADMASS('D0') < 70*MeV)"
  )
  ```

  With all the selections ready, we can build a combiner as

```
from Configurables import CombineParticles
d0 = CombineParticles(
    'Combine_D0',
    DecayDescriptor='[D0 -> K- K+]cc',
    DaughtersCuts=d0_daughters,
    CombinationCut=d0_comb,
    MotherCut=d0_mother
)
```

A small question

- Do you understand this selection?
- Do you know what each of these LoKi functors does?

Now we have to build a `Selection` out of it so we can later on put all pieces together:

```
from PhysConf.Selections import Selection
d0_sel = Selection(
    'Sel_D0',
    Algorithm=d0,
    RequiredSelections=[Kaons]
)
```

We can already see that this two-step process (building the `CombineParticles` and the `Selection`) is a bit cumbersome. This can be simplified using a `SimpleSelection` object, which will be discussed in the next lesson.

For the time being, let's finish building our candidates. Now we can use another `CombineParticles` to build the $$D^\\ast$$ with pions and the $$D^0$$'s as inputs, and applying a filtering only on the soft pion:

```
dstar_daughters = {'pi+': '(TRCHI2DOF < 3) & (PT > 100*MeV)'}
dstar_comb = "(ADAMASS('D*(2010)+') < 400*MeV)"
dstar_mother = (
    "(abs(M-MAXTREE('D0'==ABSID,M)-145.42) < 10*MeV)"
    '& (VFASPF(VCHI2/VDOF)< 9)'
)

dstar = CombineParticles(
    'Combine_Dstar',
    DecayDescriptor='[D*(2010)+ -> D0 pi+]cc',
    DaughtersCuts=dstar_daughters,
    CombinationCut=dstar_comb,
    MotherCut=dstar_mother
)

dstar_sel = Selection(
    'Sel_Dstar',
    Algorithm=dstar,
    RequiredSelections=[d0_sel, Pions]
)
```

Building shared selections

In some cases we may want to build several decays in the same script with some common particles/selection; for example, in our case we could have been building D0->K pi in the same script, and then we would have wanted to select the soft pion in the same way when building the Dstar. In this situation, we can make use of the `FilterDesktop` algorithm, which takes a TES location and filters the particles inside according to a given LoKi functor in the `Code` property, which then can be given as input to a `Selection` :

```
from Configurables import FilterDesktop
soft_pion = FilterDesktop('Filter_SoftPi',
                          Code='(TRCHI2DOF < 3) & (PT > 100*MeV)')
soft_pion_sel = Selection('Sel_SoftPi',
                          Algorithm=soft_pion,
                          RequiredSelections=[Pions])
dstar = CombineParticles(
    'CombineDstar',
    DecayDescriptor='[D*(2010)+ -> D0 pi+]cc',
    CombinationCut="(ADAMASS('D*(2010)+') < 400*MeV)",
    MotherCut=(
        "(abs(M-MAXTREE('D0'==ABSID,M)-145.42) < 10*MeV)"
        '& (VFASPF(VCHI2/VDOF)< 9)'
    )
)
dstar_sel = Selection(
    'Sel_Dstar',
    Algorithm=dstar,
    RequiredSelections=[d0_sel, soft_pion_sel]
)
```

This allows us to save time by performing the filtering of the soft pions only once, and to keep all the common cuts in a single place, avoiding duplication of code.

We can now build build a `SelectionSequence` to add to the `DaVinci` execution sequence

```
from PhysConf.Selections import SelectionSequence
dstar_seq = SelectionSequence('Dstar_Seq', TopSelection=dstar_sel)
from Configurables import DaVinci
DaVinci().UserAlgorithms += [dstar_seq.sequence()]
```

Work to do

- Finish the script by adapting the basic `DaVinci` configuration from its corresponding lesson and check the output ntuple (run over 10000 events to make sure your tuple has enough entries). The solution can be found here).
- Add a `PrintSelection` algorithm in your selections and run again. It gets a `Selection` as input and it can be used the same way, except it will print the decay tree everytime making use of the `PrintDecayTree` algorithm which was discussed in the Exploring a DST lesson.
- Compare your selection with what is done in the actual Stripping, which can be found here. You can appreciate the power of the Selection Framework in the modularity of that Stripping.

By looking at the final script, there is one striking thing: there is a lot of repetition (`CombineParticles` - `Selection` sequence) which leads to complicated naming schemes, due to the fact that we want our `Selection` or `CombineParticle` objects to have a unique name. To help with these name clashes and to allow a much more streamlined `Selection` building, the `PhysConf.Selections` module offers a large set of more optimized classes, which we'll discuss in the next lesson.

Building your own decay

Modern Selection Framework

Learning Objectives

- Build a decay chain with the most optimized tools
- Learn the advantages of these specialized tools

As discussed previously, the Selection Framework can become a bit cumbersome in terms of the naming and construction of the `Selection` - `CombineParticles` repetitions. For this reason, the Selection Framework offers a more streamlined set of `Selection` s to deal with these issues.

This two-step process for building the `Selection` (creating an algorithm and building a selection with it) can be simplified by using a helper function in the `PhysConf.Selections` module, called `SimpleSelection` . It gets a selection name, the algorithm type we want to run, the inputs and any other parameters that need to be passed to the algorithm (as keyword arguments), and returns a `Selection` object build in the same two-step way. With that in mind, we can rewrite the previous two pieces of code as

```
import GaudiConfUtils.ConfigurableGenerators as ConfigurableGenerators
from PhysConf.Selections import SimpleSelection
d0_sel = SimpleSelection(
    'Sel_D0',
    ConfigurableGenerators.CombineParticles,
    [Kaons],
    DecayDescriptor='[D0 -> K- K+]cc',
    DaughtersCuts=d0_daughters,
    CombinationCut=d0_comb,
    MotherCut=d0_mother
)
```

Note how we needed to use the `CombineParticles` from `GaudiConfUtils.ConfigurableGenerators` instead of the `PhysConf.Selections` one to make this work. This is because the LHCb algorithms are configured as singletons and it is mandatory to give them a name, which we don't want to in `SimpleSelection` (we want to skip steps!).

Why `ConfigurableGenerators` ?

If we had tried to simply use `CombineParticles` inside our `SimpleSelection` , we would have seen it fail with the following error

```
NameError: Could not instantiate Selection because input Configurable CombineParticles has default name. This is too un
safe to be allowed.
```

The reason for this is that all LHCb algorithms need an explicit and unique name. The solution for our problem, in which we actually don't care about the `CombineParticles` name, is the `GaudiConfUtils.ConfigurableGenerators` package: it contains wrappers around algorithms such as `CombineParticles` or `FilterDesktop` allowing them to be instantiated without an explicit name argument.

In this case, we could also wonder about the need for the `CombineParticles` generator. While `SimpleSelection` will allow us to do

anything we would do with `Selection`, there exist a few specialized versions of it that allow us to address its most common usages:

- `CombineSelection` is used for the `Selection` – `CombineParticles` combination. The previous example would be written then:

```
from PhysConf.Selections import CombineSelection
d0_sel = CombineSelection(
    'Sel_D0',
    [Kaons],
    DecayDescriptor='[D0 -> K- K+]cc',
    DaughtersCuts=d0_daughters,
    CombinationCut=d0_comb,
    MotherCut=d0_mother
)
```

- `Combine3BodySelection` and `Combine4BodySelection` are the selection version of the `DaVinci::N{3,4}BodyDecays` DaVinci algorithms (you can see an example of their use as generators here), that allow to perform 3- and 4-body combinations with an improved CPU efficiency thanks to the existence of a set of cuts on the intermediate particle combinations (`Combination12Cut`, `Combination123Cut`). These are very useful in timing-critical environments, such as the Stripping.
- `TupleSelection` allows to build a `Selection` with `DecayTreeTuple` as an algorithm.
- `FilterSelection` is used to produce a `Selection` – `FilterDesktop` combination. It's used in a similar way as `CombineSelection`.

Work to do

- Rewrite the previous script to select our signal in terms of `SimpleSelections` and the other algorithms we just learned.
- Introduce `FilterSelection` by performing the soft pion selection outside the `CombineParticles` as discussed in the *Building shared selections* callout in the previous lesson. The solution can be found here.

To finalize, it is also very useful to know about the existence of certain selection algorithms specialized in filtering according to very commonly used criteria. These can be used as inputs for `SimpleSelection` to make sure the latter only run on those events that pass the filter. The most interesting are (see the code for the full list, along with examples on how to use them):

- `TriggerSelection`, including `L0` / `Hlt1` / `Hlt2` specific versions. These are used to filter on certain trigger decisions (NB: their job is simply to filter, so they cannot be used as particle input for another selection). The same interface can be used for filtering on Stripping decisions by using the `StrippingSelection` class. With it, the example from the Starterkit minimal DaVinci job, in which the output of a Stripping line was passed to `DecayTreeTuple`, could be written in a more CPU-efficient way:

```
line = 'D2hhPromptDst2D2KKLine'
strip_sel = StrippingSelection("Strip_sel",
                               "HLT_PASS('StrippingD2hhPromptDst2D2KKLineDecision')")
strip_input = AutomaticData('Phys/{0}/Particles'.format(line))
tuple_sel = TupleSelection('Tuple_sel',
                           [strip_sel, strip_input],
                           Decay='[D*(2010)+ -> (D0 -> K- K+) pi+]CC')
```

  This avoids running `DecayTreeTuple` on empty events, since the `strip_sel` stops processing before. This will only be helpful for rare Stripping lines, since the overhead of running `DecayTreeTuple` on empty events is small, but this has been proven useful in more complex workflows. Additionally, it takes care of `RootInTes` for you.

A small test

Try running the minimal DaVinci job with and without the `StrippingSelection` / `DecayTreeTuple` selections, and compare their performance

In this particular case, there is a simple way to achieve a CPU-efficient code with `DecayTreeTuple`, thanks to the use of `DaVinci` pre-event filters;

```
from PhysConf.Filters import LoKi_Filters
filter_ = LoKi_Filters(STRIP_Code="HLT_PASS('StrippingD2hhPromptDst2D2KKLineDecision')")

DaVinci().EventPreFilters = filter_.filters("FILTERS")
```

- Related to the previous ones, `TisTosSelection` are used to filter according to TIS/TOS. A whole range of them is available: `L0TOSSelection`, `L0TISSelection`, `Hlt1TOSSelection`, `Hlt1TISSelection`, `Hlt2TOSSelection` and `Hlt2TISSelection`.
- `ValidBPVSelection`, which is used to check that a valid associated primary vertex is present.

While it's hard to find simple examples in which the utility of these tools is clearly highlighted, it's important to note that they constitute a modular framework that allows to build very complex workflows from very simple pieces. In these situations, the Selection Framework is the ideal tool to keep the code bug-free and CPU-efficient.

What to do when something fails

Learning Objectives

- Learn how to read the logs to know where things are breaking
- Learn how to get a glimpse of where algorithms are writing in the TES

When chaining complex workflows (building particles, combining them, etc) we find that our ntuple is not written while we don't have any errors. The first step is to look at the logs. Let's first go back at what we learned when building our own decays and rerun again (saving the output to a log file!).

We can scroll through the log until we find our selections, where we will see something like this:

```
Sel_D0            SUCCESS Number of counters : 10
 |    Counter                                       |     #     |    sum     | mean/eff^* | rms/err^*  |     min     |     max
    |
 | "# D0 -> K-  K+ "                                |      1000 |         95 |   0.095000 |    0.29661 |      0.0000 |      2.0
000 |
 | "# D~0 -> K+  K- "                               |      1000 |         95 |   0.095000 |    0.29661 |      0.0000 |      2.0
000 |
 | "# K+"                                           |      1000 |        723 |    0.72300 |    0.89122 |      0.0000 |      6.0
000 |
 | "# K-"                                           |      1000 |        691 |    0.69100 |    0.85995 |      0.0000 |      6.0
000 |
 | "# Phys/StdAllLooseKaons"                        |      1000 |      20491 |     20.491 |     13.579 |      1.0000 |      90.
000 |
 | "# input particles"                              |      1000 |      20491 |     20.491 |     13.579 |      1.0000 |      90.
000 |
 | "# selected"                                     |      1000 |        190 |    0.19000 |    0.59321 |      0.0000 |      4.0
000 |
 |*"#accept"                                        |      1000 |         94 |(  9.40000 +- 0.922843 )%|   -------   |   ------
-   |
 |*"#pass combcut"                                  |      1308 |        234 |(  17.8899 +- 1.05974  )%|   -------   |   ------
-   |
 |*"#pass mother cut"                               |       234 |        190 |(  81.1966 +- 2.55434  )%|   -------   |   ------
-   |
SelFilterPhys_S...SUCCESS Number of counters : 1
 |    Counter                                       |     #     |    sum     | mean/eff^* | rms/err^*  |     min     |     max
    |
 |*"#passed"                                        |        94 |         94 |(  100.000 +- 1.06383  )%|   -------   |   ------
-   |
Sel_Dstar         SUCCESS Number of counters : 14
 |    Counter                                       |     #     |    sum     | mean/eff^* | rms/err^*  |     min     |     max
    |
 | "# D*(2010)+ -> D0  pi+ "                        |        94 |         30 |    0.31915 |    0.48844 |      0.0000 |      2.0
000 |
 | "# D*(2010)- -> D~0  pi- "                       |        94 |         31 |    0.32979 |    0.51340 |      0.0000 |      2.0
000 |
 | "# D0"                                           |        94 |         95 |     1.0106 |    0.10259 |      1.0000 |      2.0
000 |
 | "# D~0"                                          |        94 |         95 |     1.0106 |    0.10259 |      1.0000 |      2.0
000 |
 | "# Phys/Sel_D0"                                  |        94 |        190 |     2.0213 |    0.20518 |      2.0000 |      4.0
000 |
 | "# Phys/StdAllNoPIDsPions"                       |        94 |       3699 |     39.351 |     19.111 |      7.0000 |      95.
000 |
 | "# input particles"                              |        94 |       3889 |     41.372 |     19.174 |      9.0000 |      99.
000 |
 | "# pi+"                                          |        94 |       1767 |     18.798 |     8.8949 |      3.0000 |      42.
000 |
 | "# pi-"                                          |        94 |       1743 |     18.543 |     9.4381 |      3.0000 |      49.
000 |
 | "# selected"                                     |        94 |         61 |    0.64894 |    0.55940 |      0.0000 |      2.0
000 |
 |*"#accept"                                        |        94 |         57 |(  60.6383 +- 5.03902  )%|   -------   |   ------
-   |
 |*"#pass combcut"                                  |      3601 |        843 |(  23.4102 +- 0.705629 )%|   -------   |   ------
-   |
 |*"#pass mother cut"                               |       843 |         61 |(  7.23606 +- 0.892333 )%|   -------   |   ------
-   |
 | "Error from IParticleCombiner, skip the combinat|         5 |          5 |     1.0000 |     0.0000 |      1.0000 |      1.0
000 |
```

Here we have information of the input containers, types of particles, etc, with all the counters corresponding to our run on 1000 events.

Understanding the log

How many excited D* mesons do we expect in our ntuple? Can you check it? Can you change some cuts and see how the counters change? Try to free the D* mass window requirement and see if you get more of those.

Now, let's make the particle builder fail silently and see if we can debug this. For example, imagine we forgot to add the Kaons as inputs in `Sel_D0`, and instead by mistake added the Pions:

```
d0_sel = Selection('Sel_D0',
                   Algorithm=d0,
                   RequiredSelections=[Pions])
```

Then we get

```
Sel_D0            SUCCESS Number of counters : 10
 |    Counter                                       |     #     |    sum     | mean/eff^* | rms/err^*  |     min     |     max
    |
 | "# D0 -> K-  K+ "                                |      1000 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# D~0 -> K+  K- "                               |      1000 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# K+"                                           |      1000 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# K-"                                           |      1000 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# Phys/StdAllNoPIDsPions"                       |      1000 |      38848 |     38.848 |     21.700 |      1.0000 |      128
.00 |
 | "# input particles"                              |      1000 |      38848 |     38.848 |     21.700 |      1.0000 |      128
.00 |
 | "# selected"                                     |      1000 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 |*"#accept"                                        |      1000 |          0 |(  0.00000 +- 0.100000 )%|   -------   |   ------
-   |
 | "#pass combcut"                                  |         0 |          0 |     0.0000 |     0.0000 | 1.7977e+308 |-1.7977e+
308 |
 | "#pass mother cut"                               |         0 |          0 |     0.0000 |     0.0000 | 1.7977e+308 |-1.7977e+
308 |
```

It's easy to see we have 0 input kaons and we can see we only get input pions!

Another problem: we messed up with a cut, for example in building the $$D^*$$,

```
dstar_mother = (
    "(abs(M-MAXTREE('D0'==ABSID,M)-145.42) < 10*MeV)"
    '& (VFASPF(VCHI2/VDOF) < 0)'
)
```

Running this, we get

```
Sel_Dstar         SUCCESS Number of counters : 14
 |    Counter                                       |     #     |    sum     | mean/eff^* | rms/err^*  |     min     |     max
    |
 | "# D*(2010)+ -> D0  pi+ "                        |        94 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# D*(2010)- -> D~0  pi- "                       |        94 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 | "# D0"                                           |        94 |         95 |     1.0106 |    0.10259 |      1.0000 |      2.0
000 |
 | "# D~0"                                          |        94 |         95 |     1.0106 |    0.10259 |      1.0000 |      2.0
000 |
 | "# Phys/Sel_D0"                                  |        94 |        190 |     2.0213 |    0.20518 |      2.0000 |      4.0
000 |
 | "# Phys/StdAllNoPIDsPions"                       |        94 |       3699 |     39.351 |     19.111 |      7.0000 |      95.
000 |
 | "# input particles"                              |        94 |       3889 |     41.372 |     19.174 |      9.0000 |      99.
000 |
 | "# pi+"                                          |        94 |       1767 |     18.798 |     8.8949 |      3.0000 |      42.
000 |
 | "# pi-"                                          |        94 |       1743 |     18.543 |     9.4381 |      3.0000 |      49.
000 |
 | "# selected"                                     |        94 |          0 |     0.0000 |     0.0000 |      0.0000 |      0.0
000 |
 |*"#accept"                                        |        94 |          0 |(  0.00000 +- 1.06383  )%|   -------   |   ------
-   |
 |*"#pass combcut"                                  |      3601 |        843 |(  23.4102 +- 0.705629 )%|   -------   |   ------
-   |
 |*"#pass mother cut"                               |       843 |          0 |(  0.00000 +- 0.118624 )%|   -------   |   ------
-   |
 | "Error from IParticleCombiner, skip the combinat|         5 |          5 |     1.0000 |     0.0000 |      1.0000 |      1.0
000 |
```

And we would get suspicious about the `MotherCut` ...

The final trick is to use the `StoreExplorerAlg` , which shows us the state of the TES in the middle of execution. We can configure it very easily and insert it in the DaVinci sequence to see what is happening.

```
from Configurables import StoreExplorerAlg

DaVinci().UserAlgorithms += [StoreExplorerAlg("Before"),
                             dstar_seq.sequence(),
                             StoreExplorerAlg("After")]
```

And then run it:

```
EventSelector     SUCCESS Reading Event record 1. Record number within stream 1: 1
...
Before               INFO ========= /Event[0x2f009e70@EventDataSvc]:
Before               INFO +--> /Event [Address: CLID=0x1 Type=0x2]  DataObject
Before               INFO | +--> /Gen [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /PrevPrev [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /Prev [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /Next [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /NextNext [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /MC [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /Link [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /pSim [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before               INFO | +--> /Rec [Address: CLID=0x1 Type=0x2]  DataObject
Before               INFO | | +--> /Header [Address: CLID=0x69 Type=0x2]  LHCb::RecHeader
Before               INFO | | +--> /Status [Address: CLID=0x1389 Type=0x2]  (Unloaded)
Before               INFO | | +--> /Summary [Address: CLID=0x6a Type=0x2]  (Unloaded)
Before               INFO | | +--> /ProtoP [No Address]  DataObject
Before               INFO | | | +--> /Charged [No Address]  KeyedContainer<LHCb::ProtoPartic [0x1d]
Before               INFO | | +--> /Muon [No Address]  DataObject
Before               INFO | | | +--> /MuonPID [No Address]  KeyedContainer<LHCb::MuonPID,Con [0x17]
Before               INFO | | +--> /Track [No Address]  DataObject
Before               INFO | | | +--> /Best [No Address]  KeyedContainer<LHCb::Track,Conta [0x45]
Before               INFO | | | +--> /FittedHLT1VeloTracks [No Address]  KeyedContainer<LHCb::Track,Conta [0x20]
```

```
Before              INFO | | +--> /Rich [No Address]  DataObject
Before              INFO | | | +--> /PIDs [No Address]  KeyedContainer<LHCb::RichPID,Con [0x18]
Before              INFO | | +--> /Vertex [No Address]  DataObject
Before              INFO | |   +--> /Primary [No Address]  KeyedContainer<LHCb::RecVertex,C [0x3]
Before              INFO | +--> /Calo [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Unstripped [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /HC [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Tracker [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Velo [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Muon [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Rich [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Trigger [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | +--> /RawEvent [Address: CLID=0x3ea Type=0x2]  LHCb::RawEvent
Before              INFO | +--> /pRec [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | +--> /Track [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | | +--> /Best [Address: CLID=0x60e Type=0x2]  LHCb::PackedTracks
Before              INFO | | | +--> /FittedHLT1VeloTracks [Address: CLID=0x60e Type=0x2]  LHCb::PackedTracks
Before              INFO | | | +--> /Muon [Address: CLID=0x60e Type=0x2]  (Unloaded)
Before              INFO | | +--> /Rich [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | | +--> /PIDs [Address: CLID=0x619 Type=0x2]  LHCb::PackedRichPIDs
Before              INFO | | +--> /Muon [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | | +--> /MuonPID [Address: CLID=0x623 Type=0x2]  LHCb::PackedMuonPIDs
Before              INFO | | +--> /Calo [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | | +--> /ProtoP [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | | +--> /Charged [Address: CLID=0x610 Type=0x2]  LHCb::PackedProtoParticles
Before              INFO | | | +--> /Neutrals [Address: CLID=0x610 Type=0x2]  (Unloaded)
Before              INFO | | +--> /Vertex [Address: CLID=0x1 Type=0x2]  DataObject
Before              INFO | | | +--> /Primary [Address: CLID=0x611 Type=0x2]  LHCb::PackedRecVertices
Before              INFO | |   +--> /V0 [Address: CLID=0x612 Type=0x2]  (Unloaded)
Before              INFO | +--> /Turbo [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /Strip [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /AllStreams [Address: CLID=0x1 Type=0x2]  (Unloaded)
Before              INFO | +--> /DAQ [No Address]  DataObject
Before              INFO | | +--> /ODIN [No Address]  LHCb::ODIN
Before              INFO   +--> /Phys [No Address]  DataObject
Before              INFO   | +--> /StdAllLooseKaons [No Address]  DataObject
Before              INFO   | | +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0x9]
Before              INFO   | | +--> /decayVertices [No Address]  KeyedContainer<LHCb::Vertex,Cont [0]
Before              INFO   | | +--> /Particle2VertexRelations [No Address]  LHCb::Relation1D<LHCb::Particle,
Before              INFO   | | +--> /_RefitPVs [No Address]  KeyedContainer<LHCb::RecVertex,C [0]
Before              INFO   | +--> /Sel_D0 [No Address]  DataObject
Before              INFO   | | +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0]
Before              INFO     +--> /TupleDstToD0pi_D0ToKK [No Address]  DataObject
Before              INFO       +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0]
After               INFO ========= /Event[0x2f009e70@EventDataSvc]:
After               INFO +--> /Event [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | +--> /Gen [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /PrevPrev [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Prev [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Next [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /NextNext [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /MC [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Link [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /pSim [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Rec [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | +--> /Header [Address: CLID=0x69 Type=0x2]  LHCb::RecHeader
After               INFO | | +--> /Status [Address: CLID=0x1389 Type=0x2]  (Unloaded)
After               INFO | | +--> /Summary [Address: CLID=0x6a Type=0x2]  (Unloaded)
After               INFO | | +--> /ProtoP [No Address]  DataObject
After               INFO | | | +--> /Charged [No Address]  KeyedContainer<LHCb::ProtoPartic [0x1d]
After               INFO | | +--> /Muon [No Address]  DataObject
After               INFO | | | +--> /MuonPID [No Address]  KeyedContainer<LHCb::MuonPID,Con [0x17]
After               INFO | | +--> /Track [No Address]  DataObject
After               INFO | | | +--> /Best [No Address]  KeyedContainer<LHCb::Track,Conta [0x45]
After               INFO | | | +--> /FittedHLT1VeloTracks [No Address]  KeyedContainer<LHCb::Track,Conta [0x20]
After               INFO | | +--> /Rich [No Address]  DataObject
After               INFO | | | +--> /PIDs [No Address]  KeyedContainer<LHCb::RichPID,Con [0x18]
After               INFO | | +--> /Vertex [No Address]  DataObject
After               INFO | |   +--> /Primary [No Address]  KeyedContainer<LHCb::RecVertex,C [0x3]
After               INFO | +--> /Calo [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Unstripped [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /HC [Address: CLID=0x1 Type=0x2]  (Unloaded)
```

```
After               INFO | +--> /Tracker [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Velo [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Muon [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Rich [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Trigger [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | +--> /RawEvent [Address: CLID=0x3ea Type=0x2]  LHCb::RawEvent
After               INFO | +--> /pRec [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | +--> /Track [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | | +--> /Best [Address: CLID=0x60e Type=0x2]  LHCb::PackedTracks
After               INFO | | | +--> /FittedHLT1VeloTracks [Address: CLID=0x60e Type=0x2]  LHCb::PackedTracks
After               INFO | | | +--> /Muon [Address: CLID=0x60e Type=0x2]  (Unloaded)
After               INFO | | +--> /Rich [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | | +--> /PIDs [Address: CLID=0x619 Type=0x2]  LHCb::PackedRichPIDs
After               INFO | | +--> /Muon [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | | +--> /MuonPID [Address: CLID=0x623 Type=0x2]  LHCb::PackedMuonPIDs
After               INFO | | +--> /Calo [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | | +--> /ProtoP [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | | +--> /Charged [Address: CLID=0x610 Type=0x2]  LHCb::PackedProtoParticles
After               INFO | | | +--> /Neutrals [Address: CLID=0x610 Type=0x2]  (Unloaded)
After               INFO | | +--> /Vertex [Address: CLID=0x1 Type=0x2]  DataObject
After               INFO | | | +--> /Primary [Address: CLID=0x611 Type=0x2]  LHCb::PackedRecVertices
After               INFO | |   +--> /V0 [Address: CLID=0x612 Type=0x2]  (Unloaded)
After               INFO | +--> /Turbo [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /Strip [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /AllStreams [Address: CLID=0x1 Type=0x2]  (Unloaded)
After               INFO | +--> /DAQ [No Address]  DataObject
After               INFO | | +--> /ODIN [No Address]  LHCb::ODIN
After               INFO  +--> /Phys [No Address]  DataObject
After               INFO   | +--> /StdAllLooseKaons [No Address]  DataObject
After               INFO   | | +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0x9]
After               INFO   | | +--> /decayVertices [No Address]  KeyedContainer<LHCb::Vertex,Cont [0]
After               INFO   | | +--> /Particle2VertexRelations [No Address]  LHCb::Relation1D<LHCb::Particle,
After               INFO   | | +--> /_RefitPVs [No Address]  KeyedContainer<LHCb::RecVertex,C [0]
After               INFO   | +--> /Sel_D0 [No Address]  DataObject
After               INFO   | | +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0]
After               INFO     +--> /TupleDstToD0pi_D0ToKK [No Address]  DataObject
After               INFO       +--> /Particles [No Address]  KeyedContainer<LHCb::Particle,Co [0]
```

Here we can see where our decays are being put and we can debug problems with inputs and outputs. It can also be useful to know where things are written for accessing them interactively, if we want to further explore and debug.

Configuring the algorithm

The `StoreExplorerAlg` has the same print frequency as `DaVinci`, but it can be configured by modifying `PrintFreq` (fraction of events that are printed) or `PrintEvt`. Have a look at the class Doxygen to see what they do.

Run a different stripping line on simulated data

Learning Objectives

- Modify the minimal DecayTreeTuple example to apply a different stripping version to an MC sample

Ideally, our simulated samples should feature the same stripping cuts as the real data we want to work with. We can be sure of this if the same stripping version has been used when processing the simulated and real data.

But often, our simulated sample will have a different version of the stripping applied to it. For example, what if our data sample uses Stripping 28r1, while our MC sample uses Stripping 28?

In this case, we simply need to rerun our stripping line of choice from the correct stripping version with one caveat: the decisions of the stripping that ran during the central MC production are placed in a default location in the TES (`/Event/Strip/Phys/DecReports`), so if we try to run our custom stripping it will complain because the location it wants to write the new decisions to already exists. To solve this issue, we need to run an instance of `EventNodeKiller` to remove the decisions from the MC production so that our custom stripping can write there instead. This is nice, because most tools expect to read the stripping decisions from the default location, so we won't have to reconfigure anything.

This example is an extended version of the minimal DaVinci DecayTreeTuple job that additionally runs the corresponding stripping line from Stripping 28r1.

Take a look at the file and try to find out what has changed compared to the minimal DaVinci example.

The key changes are

- Removing the old stripping reports with a node killer

```
from Configurables import EventNodeKiller
event_node_killer = EventNodeKiller('StripKiller')
event_node_killer.Nodes = ['/Event/AllStreams', '/Event/Strip']
```

- Picking the right stripping line from Stripping 28r1 (which we prepare with `buildStreams`):
- Building a custom stream that only contains the desired stripping line

```
strip = 'stripping28r1'
streams = buildStreams(stripping=strippingConfiguration(strip),
                       archive=strippingArchive(strip))

custom_stream = StrippingStream('CustomStream')
custom_line = 'Stripping'+line

for stream in streams:
    for sline in stream.lines:
        if sline.name() == custom_line:
            custom_stream.appendLines([sline])
```

- Instantiating a `StrippingConf` for running the stripping

```
sc = StrippingConf(Streams=[custom_stream],
                   MaxCandidates=2000,
                   AcceptBadEvents=False,
                   BadEventSelection=filterBadEvents)
```

- Updating the path for `dtt.Inputs` : the output is now placed directly in `Phys`

```
dtt.Inputs = ['/Event/Phys/{0}/Particles'.format(line)]
```

- Inserting the node killer and the stripping selection sequence into the Gaudi sequence

```
DaVinci().appendToMainSequence([event_node_killer, sc.sequence()])
```

It is strongly recommended to run restripping with the same version of DaVinci as the one used for the corresponding stripping production: this ensures all the settings are configured the same exact way as for your data. For the Stripping 28r1, DaVinci v41r4p4 was used. Notice that this version is not available for the default platform `x86_64-slc6-gcc62-opt`. To work around this you can pick up the best suitable platform using `lb-run -c best DaVinci/v41r4p4 ...` or by setting the `application.platform` attribute in ganga. See the ProcessingPasses TWiki page to find which version of DaVinci was used for each stripping campaign.

Replace a mass hypothesis

Learning Objectives

- Create a new selection starting from the stripping line output
- Change the particle hypothesis made by the stripping line

There are many situations where you want to repurpose a stripping line to look for a new decay that is similar in topology but distinct from what was put into the stripping line. The easiest thing to do is to change some of the hypothesis on particle IDs made in the stripping. This lesson will show you how to do that.

Reinterpreting stripping selections

Note that with this method you can never find more candidates than the stripping has already found. However, you can reinterpret parts of the decay to look for new decay modes.

As an example we will switch the decays of the D0 from (K- K+) to (K- pi+).

There is an algorithm that allows us to replace parts of the decay descriptor called `SubstitutePID` :

```
# configure an algorithm to substitute the K+ (resp. the K-)
# in the D0 (resp the D~0) decay by a pion
from Configurables import SubstitutePID
subs = SubstitutePID(
    'MakeD02Kpi',
    Code = "DECTREE('[D*(2010)+ -> (D0 -> K- K+) pi+]CC')",
    # note that SubstitutePID can't handle automatic CC
    Substitutions = {
    'Charm -> (D0 -> K- ^K+) Meson': 'pi+',
    'Charm -> (D~0 -> K+ ^K-) Meson': 'pi-'}
)
```

The algorithm is configured with a name `MakeD02Kpi` . In the `Code` argument we need to specify the initial selection. This is done by using LoKi functors. Since we know we will be using an already prepared selection, we can simply use the `DECTREE` functor to search for candidates fulfilling this decay structure. See the lesson on LoKi for more info on what you can do here.

Now we are ready to specify which hypotheses to change. `Substitutions` is a dictionary where the keys are decay descriptors and the values are the names of the replacement particles. The particle that should be replaced is marked with a `^` . So in the example above

```
'Charm -> (D0 -> K- ^K+) Meson': 'pi+'
```

means: Look for a decay of a Charm-particle into D0 plus any meson, where the D0 decays to (K- K+) and replace the K+ with a pi+.

Note that `SubstitutePID` does not automatically handle complex conjugation via the `CC` operator. Therefore you have to specify all substitutions explicitely.

Next we have to handle the input and output of this algorithm. This is accomplished using particle selections. The input to our substitution algorithm will be the candidates produced by the stripping. In order to make them look like a selection we can use the `DataOnDemand` service:

```
from PhysSelPython.Wrappers import Selection
from PhysSelPython.Wrappers import SelectionSequence
from PhysSelPython.Wrappers import DataOnDemand

# Stream and stripping line we want to use
stream = 'AllStreams'
line = 'D2hhPromptDst2D2KKLine'
tesLoc = '/Event/{0}/Phys/{1}/Particles'.format(stream, line)

# get the selection(s) created by the stripping
strippingSels = [DataOnDemand(Location=tesLoc)]
```

The output of the algorithm has to be packaged into a new selection:

```
# create a selection using the substitution algorithm
selSub = Selection(
    'Dst2D0pi_D02Kpi_Sel',
    Algorithm=Subs,
    RequiredSelections=strippingSels
)
```

Note how the input stripping selection is daisy-chained to the output selection through the `RequiredSelections` (it has to be a list of selections) argument. The new selection is added into a `SelectionSequence` for further use by DaVinci:

```
selSeq = SelectionSequence('SelSeq', TopSelection=selSub)
```

We are now ready to produce an ntuple on our newly created selection. As usual we configure a `DecayTreeTuple`, which now is looking for the candidates, which have the redefined D0 decay:

```
# Create an ntuple to capture D*+ decays from the new selection
dtt = DecayTreeTuple('TupleDstToD0pi_D0ToKpi')
dtt.Inputs = [selSeq.outputLocation()]
# note the redefined decay of the D0
dtt.Decay = '[D*(2010)+ -> ^(D0 -> ^K- ^pi+) ^pi+]CC'
```

The input to the `DecayTreeTuple` is taken as the `outputLocations` of the `SelectionSequence` we just created.

Why use DaVinci selections?

Selections and SelectionSequences are an elegant way to organize the required algorithms that perform the job of selecting data. The particle selection toolkit uses python tricks to make the management of even complicated sequences of selections rather straight forward. In particular the toolkit ensures that all necessary algorithms are run in the correct order to produce the desired selections. It is therefore mainly a tool to manage dependencies. For technical details, have a look at the TWiki page.

Finally we add the `SelectionSequence` and the `DecayTreeTuple` to the DaVinci application. Since we are adding more than one algorithm we need a `GaudiSequencer` that takes care of calling everything in the right order:

```
# add our new selection and the tuple into the sequencer
seq = GaudiSequencer('MyTupleSeq')
seq.Members += [selSeq.sequence()]
seq.Members += [dtt]
DaVinci().appendToMainSequence([seq])
```

The solution to this exercise `ntuple_switchHypo.py`, is available here.

Think about it

Why can't we use this method to look for D* decaying to D0+photon ?

Reuse particles from a decay tree

Learning Objectives

- Learn how to extract particles from a decay tree
- Build a new particle from the extracted particles

Sometimes we want to extract a portion of the decay tree in order to build a different decay. To do that, we need to put the particles we're interested in in a new container so they can afterwards be used as inputs to a `CombineParticles` instance (as we saw in the selection framework lesson). To achieve this we can use the `FilterInTrees` algorithm, a simple variation of `FilterDesktop` (doxygen).

Let's start from the example in the selection framework lesson and let's check that the $$K^+$$ child of the $$D^0$$ does not come from a $$K^{*}(892)^{0} \to K^{+}\pi^{-}$$. To do that, we have to extract the $$K^+$$ from `([D0 -> K+ K-]CC)` and combine it with all pions in `Phys/StdAllNoPIDsPions/Particles` .

Using `FilterInTrees` is done in the same way we would use `FilterDesktop` :

```
from Configurables import FilterInTrees
from PhysSelPython.Wrappers import Selection, DataOnDemand

stream = 'AllStreams'
line = 'D2hhPromptDst2D2KKLine'
tesLoc = '/Event/{0}/Phys/{1}/Particles'.format(stream, line)

kaons_from_d0 = FilterInTrees('kaons_from_d0_filter', Code="('K+' == ABSID)")
kaons_from_d0_sel = Selection("kaons_from_d0_sel",
                              Algorithm=kaons_from_d0,
                              RequiredSelections=[DataOnDemand(Location=tesLoc)])
```

The output of `kaons_from_d0_sel` is a container with all the kaons coming from the $$D^0$$.

The final step is easy, very similar to building your own decay:

```
from Configurables import CombineParticles
from PhysSelPython.Wrappers import Selection, DataOnDemand

Pions = DataOnDemand('Phys/StdAllNoPIDsPions/Particles')
kst = CombineParticles('kst_particles',
                       DecayDescriptor="[K*(892)0 -> K+ pi-]cc",
                       CombinationCut="ADAMASS('K*(892)0') < 300*MeV",
                       MotherCut='(VFASPF(VCHI2/VDOF)< 9)')
kst_sel = Selection('kst_sel',
                    Algorithm=kst,
                    RequiredSelections=[kaons_from_d0_sel, Pions])
```

An interesting detail

One can use `FilterInTrees` and `FilterDecays` to select several particles at once and obtain a flattened list. For example, if we had a Stripping line that builds `[B- -> (^D0 -> ^K- ^pi+) ^pi-]cc` and we wanted to combine the `D0` and `pi-` with an external `pi0` to build `[B- -> D0 pi- pi0]cc` , we could do

```
flatlist = FilterInTrees ("FlatList", Code="('D0' == ABSID) | ('pi-' == ABSID)")
from Configurables import CombineParticles
add_pi0 = CombineParticles("MakeB",
                           DecayDescriptor = "[B- -> D0 pi- pi0]cc",
                           ...
                           Inputs=[flatlist, resolvedPi0])
```

`flatlist` contains both `D0` and `pi-`, which are then used to build the `B`.

The simulation framework

Requirements

- Know how to run LHCb applications using `lb-run`
- Now how to set up and use `lb-dev`
- Be able to produce nTuples using `MCDecayTreeTuple`

Gauss is LHCb's simulation framework that interfaces various generators to decay engines and then simulate the response of our detector. This lesson will give a basic introduction to Gauss, focussing on the most common use-cases analysts might encounter. Additional information and a collection of useful links can be found on the project's website: http://lhcbdoc.web.cern.ch/lhcbdoc/gauss/.

Learning Objectives

- Understand how a signal decay sample is produced in the LHCb framework
- Produce generator level Monte Carlo, print the decay tree and produce nTuples
- Read a DecFile and understand what it produces, including generator level cuts
- Generate a sample without/with modified generator cuts
- Raise awareness about the available options for fast simulation

Setup

Before we continue, please set up a Gauss v49r9 `lb-dev` :

- Change your `CMTCONFIG` to `x86_64-slc6-gcc49-opt` .
- Get the `DecFiles` package for later: `git lb-clone-pkg Gen/DecFiles`
- Build it!

Learning Objectives

- Understand how a signal decay sample is produced in the LHCb framework

What is Gauss?

Gauss is the LHCb simulation framework which manages the creation of simulated events by interfacing to multiple external applications. Most commonly, an event is created via the following procedure:

1. The `ProductionTool` (Pythia, GenXicc, ...) generates an event with the required signal particle. Either by generating minimum bias events until a matching particle is found or by ensuring one is produced in every event. The resulting event is comprised of either stable particles or unstable particles which are known to either EvtGen or Geant4 and can be decayed.
2. The signal particle is decayed using the `DecayTool` (EvtGen) to the desired final state, all remaining unstable particles are decayed independently.
3. The signal and its decay products might be required to pass generator level cuts implemented as a `CutTool`.
4. Particles are transported through the detector simulation.

Things to remember

1. The detector simulation is **by far** the most time consuming step (minutes, compared to seconds for the rest). So make sure your generator cuts remove events you cannot possibly reconstruct or select later on. Additional options are available to increase the speed, please talk to your MC liaisons!
2. The generator cuts are only applied to the signal that was forced to decay to the specific final state. *Any* other true signal candidate is not required to pass. Furthermore, if the signal candidate flies in negative `z` direction, the event is mirrored to optimise the use of available CPU resources.
3. The number of generated events refers to the number entering step 4 above, so those passing the generator level cuts. **Not** the number of events produced by the `ProductionTool` in the first step.

Learning Objectives

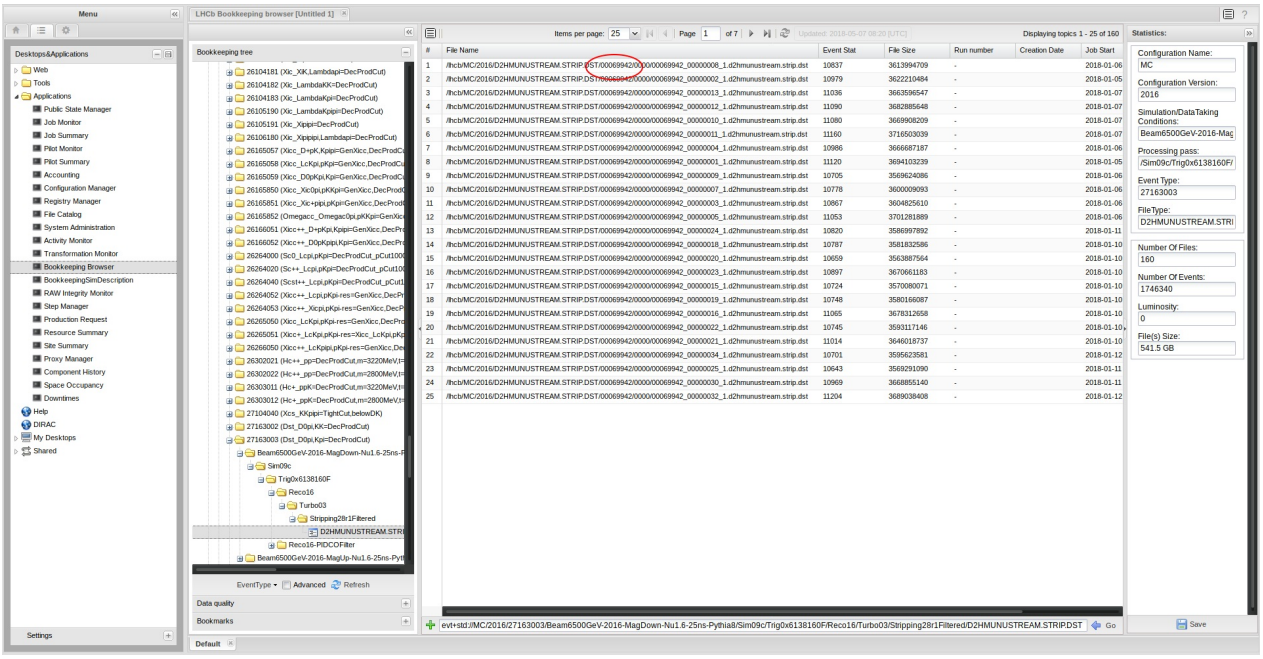
- Find the correct option files to run Gauss
- Produce generator level Monte Carlo, print the decay tree and produce nTuples

Which option files to use and how to run Gauss

Imagine you need to know the option files and software versions used for a simulated sample you have found in the bookkeeping, e.g.

```
/MC/2016/Beam6500GeV-2016-MagDown-Nu1.6-25ns-Pythia8/Sim09c/Trig0x6138160F/Reco16/Turbo03/Stripping28r1Filtered/27163003/D2HMU
NUSTREAM.STRIP.DST
```

First, find the ProductionID:



Search for this ID in the Transformation Monitor, right click the result and select "Show request". Right clicking and selecting "View" in the new window will open an overview about all the individual steps of the production with their application version and option files used.

Important: the order of the option files does matter!

`'$DECFILESROOT/options/27163003.py' '$LBPYTHIA8ROOT/options/Pythia8.py'` produces the sample using Pythia 8 while `'$LBPYTHIA8ROOT/options/Pythia8.py' '$DECFILESROOT/options/27163003.py'` uses Pythia 6.

Running Gauss and create a generator-only sample

The production system handles the necessary settings for initial event- and runnumber and the used database tags. In a private production, you need to set these yourself in an additional options file, containing, for example:

```
from Gauss.Configuration import GenInit

GaussGen = GenInit("GaussGen")
GaussGen.FirstEventNumber = 1
GaussGen.RunNumber = 1082

from Configurables import LHCbApp
LHCbApp().DDDBtag = 'dddb-20170721-3'
LHCbApp().CondDBtag = 'sim-20170721-2-vc-md100'
LHCbApp().EvtMax = 5
```

Assuming this is saved in a file called `Gauss-Job.py` and following the example above, the sample can then be produced by running

```
./run gaudirun.py '$APPCONFIGOPTS/Gauss/Beam6500GeV-md100-2016-nu1.6.py' \
        '$APPCONFIGOPTS/Gauss/EnableSpillover-25ns.py' \
        '$APPCONFIGOPTS/Gauss/DataType-2016.py' \
        '$APPCONFIGOPTS/Gauss/RICHRandomHits.py' \
        '$DECFILESROOT/options/27163003.py' \
        '$LBPYTHIA8ROOT/options/Pythia8.py' \
        '$APPCONFIGOPTS/Gauss/G4PL_FTFP_BERT_EmNoCuts.py' \
       Gauss-Job.py
```

This would take 5 to 10 minutes due to the detector simulation, which can be turned off by adding `'$GAUSSOPTS/GenStandAlone.py'` as one of the option files. In this case, all but `'$GAUSSOPTS/GenStandAlone.py'`, `'$DECFILESROOT/options/27163003.py'`, and `'$LBPYTHIA8ROOT/options/Pythia8.py'` are redundant

Only one option file

You can source the various options files from your `'Gauss-Job.py'` by adding at its top:

```
from Gaudi.Configuration import *
importOptions("$APPCONFIGOPTS/Gauss/Beam6500GeV-md100-2016-nu1.6.py")
# etc ...
```

Make an nTuple

The `.xgen` file can be processed into something more usable (copied together from here). A larger input file containing 10,000 generated events for event-type can be found on EOS: `root://eosuser.cern.ch//eos/user/l/lhcbsk/sim-lesson/Gauss-27163003-10000ev.xgen`.

```
"""Configure the variables below with:
decay: Decay you want to inspect, using 'newer' LoKi decay descriptor syntax,
decay_heads: Particles you'd like to see the decay tree of,
datafile: Where the file created by the Gauss generation phase is, and
year: What year the MC is simulating.
"""
# https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders
decay = "[D*(2010)+ => ^[([D0]cc => ^K- ^pi+)]CC ^pi+]CC"
datafile = "./Gauss-27163003-100ev-20180507.xgen"

mc_basic_loki_vars = {
    'ETA': 'MCETA',
    'PHI': 'MCPHI',
    'PT': 'MCPT',
    'P': 'MCP',
}

from Configurables import (
    DaVinci,
```

```
    MCDecayTreeTuple
)
from DecayTreeTuple.Configuration import *

# For a quick and dirty check, you don't need to edit anything below here.
###############################################################################

# Create an MC DTT containing any candidates matching the decay descriptor
mctuple = MCDecayTreeTuple("MCDecayTreeTuple")
mctuple.Decay = decay

mctuple.ToolList = []
mctuple.addTupleTool(
    'LoKi::Hybrid::MCTupleTool/basicLoKiTT'
).Variables = mc_basic_loki_vars

# Name of the .xgen file produced by Gauss
from GaudiConf import IOHelper
# Use the local input data
IOHelper().inputFiles([
    datafile
], clear=True)

# Configure DaVinci
DaVinci().TupleFile = "DVntuple.root"
DaVinci().Simulation = True
DaVinci().Lumi = False
DaVinci().DataType = '2016'
DaVinci().InputType = 'DIGI'
DaVinci().UserAlgorithms = [mctuple]

from Gaudi.Configuration import appendPostConfigAction
def doIt():
    """
    specific post-config action for (x)GEN-files
    """
    extension = "xgen"
    ext = extension.upper()

    from Configurables import DataOnDemandSvc
    dod  = DataOnDemandSvc ()
    from copy import deepcopy
    algs = deepcopy ( dod.AlgMap )
    bad  = set()
    for key in algs :
        if     0 <= key.find ( 'Rec'     )                  : bad.add ( key )
        elif   0 <= key.find ( 'Raw'     )                  : bad.add ( key )
        elif   0 <= key.find ( 'DAQ'     )                  : bad.add ( key )
        elif   0 <= key.find ( 'Trigger' )                  : bad.add ( key )
        elif   0 <= key.find ( 'Phys'    )                  : bad.add ( key )
        elif   0 <= key.find ( 'Prev/'   )                  : bad.add ( key )
        elif   0 <= key.find ( 'Next/'   )                  : bad.add ( key )
        elif   0 <= key.find ( '/MC/'    ) and 'GEN' == ext : bad.add ( key )

    for b in bad :
        del algs[b]

    dod.AlgMap = algs

    from Configurables import EventClockSvc, CondDB
    EventClockSvc ( EventTimeDecoder = "FakeEventTime" )
    CondDB  ( IgnoreHeartBeat = True )

appendPostConfigAction( doIt )
```

Learning Objectives

- Learn the basic structure of a decfile.

Controlling the decay: DecFiles

The `DecFile` controls the decay itself (i.e. what `EvtGen` does) as well as provide any event-type specific configuration (e.g. generator cuts). They exist in the `Gen/DecFile` package which we have already checked out and built in the beginning and are given as `.dec` files in `Gen/DecFiles/dkfiles`. For example, the `.dec` file for event-type `27163003` is `Dst_D0pi,Kpi=DecProdCut.dec`:

```
# EventType: 27163003
#
# Descriptor: [D*(2010)+ -> (D0 -> K- pi+) pi+]cc
#
# NickName: Dst_D0pi,Kpi=DecProdCut
#
# Cuts: DaughtersInLHCb
#
# Documentation: D0 decays to right-sign mode (K- pi+) with a D* tag.
# EndDocumentation
#
# PhysicsWG: Charm
# Tested: Yes
# Responsible: Mat Charles
# Email: matthew.charles@<nospam>cern.ch
# Date: 20101210
#

Alias MyD0 D0
Alias MyantiD0 anti-D0
ChargeConj MyD0 MyantiD0

Decay D*+sig
  1.000 MyD0  pi+    VSS;
Enddecay
CDecay D*-sig

Decay MyD0
  1.0   K-  pi+    PHSP;
Enddecay
CDecay MyantiD0
#
End
```

The commented actually contains vital information and is parsed during the compilation to create the `<event-type>.py` file that is given to `Gauss` as one of its options! The EventType is a series of flags which controls the generation. The rules for this are described in detail in LHCb-2005-034 and also a website that allows you to build/parse event-types. For example for the first digit: 1 = contains b quark, 2 = c quark, 3 = min bias ... Similarly, the document specifies the conventions for the "NickName" - which also has to be the filename. Note that once MC has been produced from a given DecFile, it is not allowed to be changed, so you never need to worry about which version of DecFiles you are looking at when trying to understand existing samples. The "Cuts" field specifies which one of a predetermined set of cut tools are used. Additional python code can be added to the resulting configuration file:

```
# InsertPythonCode:
# code ...
# EndInsertPythonCode
```

The bottom part of the decay file specifies the decay itself: This DecFile defines a signal `D*+` which decays 100% to `D0` `pi+`, and the

D0 in turn decays 100% into `K-` `pi+` . Important is the definition of "MyD0". If the decay was to "D0" rather than "MyD0", the D0 would decay via all of the decay modes implemented in DECAY.DEC. The final part of each decay is the actual physics model used - in this case "VSS", the generic matrix element for vector to scaler-scaler transitions, and "PHSP", which is phase space only (matrix element = constant). Note that with PHSP the daughters are completely unpolarized - for anything other than (spin0) to (spin0 spin0) this will get the angular momentum wrong!

Learning Objectives

- Learn advanced configuration options to configure `EvtGen`

Advanced: Controlling the decay

Two-body decays - getting angular momentum right

EvtGen has specific models for each two body spin configuration, for example Scalar to Vector+Scalar (SVS), and Vector to lepton+lepton(VLL)

```
#
Decay B+sig
  1.000     MyJ/psi  K+                  SVS;
Enddecay
CDecay B-sig
#
Decay MyJ/psi
  1.000     mu+  mu-                     PHOTOS  VLL;
Enddecay
```

For decays to two vectors, there is a more complicated polarization structure which needs to be specified - for example here the fraction and phase for each helicity are set according to measured values:

```
Define Hp 0.159
Define Hz 0.775
Define Hm 0.612
Define pHp 1.563
Define pHz 0.0
Define pHm 2.712
#
Alias      MyJ/psi    J/psi
Alias      MyK*0      K*0
Alias      Myanti-K*0 anti-K*0
ChargeConj MyK*0      Myanti-K*0
ChargeConj MyJ/psi    MyJ/psi
#
Decay B0sig
  1.000         MyJ/psi   MyK*0          SVV_HELAMP Hp pHp Hz pHz Hm pHm;
Enddecay
Decay anti-B0sig
  1.000         MyJ/psi   Myanti-K*0     SVV_HELAMP Hm pHm Hz pHz Hp pHp;
Enddecay
```

3+ multi-body decays

For 3+ bodies the physics models get more complicated. For a fully hadronic final state, typically a Dalitz model will be specified, e.g:

```
# D_DALITZ includes resonances contributions (K*(892), K*(1430), K*(1680))
Decay MyD-
  1.000    K+        pi-    pi-          D_DALITZ;
Enddecay
CDecay MyD+
```

Any time you see a 3+ body decay with the PHSP model, you know it will be very far from reality. If you have no other information sometimes this is the best you can do, though.

A semileptonic decay would typically be produced according to some form factor model, e.g

```
Decay B0sig
# FORM FACTORS as per HFAG PDG10
   1   MyD*-        mu+  nu_mu         PHOTOS  HQET 1.20 1.426 0.818 0.908;
  #
Enddecay
CDecay anti-B0sig
```

here the numbers correspond to measured values for the form factor parameters.

Cocktail decays

Often you will want to simulate more than one decay mode in a sample, e.g:

```
Decay MyD_s+
 0.0259 phi      mu+     nu_mu                    PHOTOS  ISGW2;
 0.0267 eta      mu+     nu_mu                    PHOTOS  ISGW2;
 0.0099 eta'     mu+     nu_mu                    PHOTOS  ISGW2;
 0.0037 K0       mu+     nu_mu                    PHOTOS  ISGW2;
 0.0018 K*0      mu+     nu_mu                    PHOTOS  ISGW2;
 0.0020 f_0      mu+     nu_mu                    PHOTOS  ISGW2;
 0.0059 mu+      nu_mu                            PHOTOS   SLN;
Enddecay
CDecay MyD_s-
```

Note that the fractions will always be renormalised to sum to 1 - you can directly use PDG branching fractions without having to rescale by hand.

Final state radiation

After generating the decay, final state radiation is added using PHOTOS. Note that PHOTOS is enabled by default, even though many decfiles explicitly specify it. It needs to be explicitly removed via "noPhotos"

Changing particle masses / lifetimes/ widths

Sometimes you need to change the mass or lifetime of a particle, either because the initial values are wrong, or the particle you actually want doesn't exist in EvtGen, and you need to adapt an existing particle. This can be done with python code inserted in the header:

```
# InsertPythonCode:
#from Configurables import LHCb__ParticlePropertySvc
#LHCb__ParticlePropertySvc().Particles = [
# "N(1440)+              636       12212   1.0      1.4400000      2.194041e-24               N(1440)+           21440
0.00",
# "N(1440)~-             637      -12212  -1.0      1.4400000      2.194841e-24                 anti-N(1440)-           -214
40      0.00",
#"N(1520)+              420        2124   1.0      1.52000000      5.723584e-24                 N(1520)+           21520
  0.00",
# "N(1520)~-             421       -2124  -1.0      1.52000000     5.723584e-24                 anti-N(1520)-           -215
20      0.00",
#"N(1535)+              713       22212   1.0      1.53500000      4.388081e-24                 N(1535)+           21535
  0.00",
#"N(1535)~-             714      -22212  -1.0      1.53500000      4.388081e-24                 anti-N(1535)-           -215
35      0.00",
#"N(1720)+              775       32124   1.0      1.72000000      2.632849e-24                 N(1720)+           21720
  0.00",
#"N(1720)~-             776      -32124  -1.0      1.72000000      2.632849e-24                 anti-N(1720)-           -217
20      0.00"
#]
# EndInsertPythonCode
```

The format is:

```
#                  GEANTID    PDGID  CHARGE   MASS(GeV)      TLIFE(s)                  EVTGENNAME    PYTHIAID    MAXWIDTH
```

Learning Objectives

- Learn how to modify/remove generator level cuts

Generator level cuts

Detector simulation is computationally expensive, and event generation is comparatively fast. Cuts at generator level save a huge amount of CPU and disk space (which means you can have more actually useful events) almost for free. At generator level you can only cut on pre-resolution quantities, so normally you want the generator cuts to be 100% efficient for selected events (within epsilon). The default example is to immediately remove events where the daughters are far outside the LHCb acceptance. This is implemented in "DaugthersInLHCb", aka "DecProdCut" in the NickName. This requires that each "stable charged particle" is in a loose region around the LHCb acceptance (10-400 mrad in Theta). Cut tools need to be implemented in C++ and reside in the package `Gen/GenCuts`.

Removing the generator cuts

The absolute efficiencies for generator cuts can be obtained from the respective website or the produced `GeneratorLog.xml` file, which contains:

```
<efficiency name = "generator level cut">
    <after> 5 </after>
    <before> 27 </before>
    <value> 0.18519 </value>
    <error> 0.074757 </error>
</efficiency>
```

However, if you need the efficiencies as functions of some other observables this is not sufficient. Instead one might want to create a generator-only sample without any cuts applied. Two possibilities exist:

1. Modify the DecFile and recompile the package
2. Overwrite the python configuration originally configured by `<event-type>.py`

The second option is usually easier and in the example used so far only requires one additional line of configuration

```
Generation().SignalPlain.CutTool = ""
```

which must be included after `27163003.py` is sourced (e.g. in `Gauss-Job.py`). You can convince yourself that this alters the observed distributions and leads to a generator level cut efficiency of 100%. A large sample can be found on EOS: `root://eosuser.cern.ch//eos/user/l/lhcbsk/sim-lesson/GaussNoGenCut-27163003-10000ev.xgen` Have a look at the pseudorapidity distribution of the head particle. This illustrates another default behavior of the generation of signal decays in Gauss: The generated events' z-axis if inverted if the selected signal particle's momentum along that axis is negative.

Modifying the cut tool

If you need to modify the cut tool, you generally can pick between multiple options with increasing complexity and time until in production:

1. Configure an existing cut-tool in `Gen/GenCuts` if possible.
2. Use `LoKi` functors for `GenParticle` (starting with a `G`) in a `LoKi::GenCutTool`.
3. Last resort for really special things: write your own C++ implementation of the `IGenCutTool` interface.

`LoKi::GenCutTool` are a good solution when you need to impose additional requirements beyond those provided by `DaughtersInLHCb`, for example a minimum for the transverse mometum of a `D0`. For local tests, this can be easily implemented by overwriting the default cut tool set by `27163003.py`:

```
from Configurables import LoKi__GenCutTool
from Gauss.Configuration import *
generation = Generation()
signal = generation.SignalPlain
signal.addTool(LoKi__GenCutTool, 'TightCut')
tightCut = signal.TightCut
tightCut.Decay = '[D*(2010)+ -> ^( D0 => ^K- ^pi+ ) pi+]CC'
tightCut.Preambulo += [
    'from GaudiKernel.SystemOfUnits import GeV',
    'inAcc         =  in_range ( 0.005 , GTHETA , 0.400 )',
    'goodD0        =  ( GPT > 2.0 * GeV )',
]
tightCut.Cuts = {
    '[D0]cc': 'goodD0',
    '[K+]cc': 'inAcc',
    '[pi+]cc': 'inAcc'
    }
```

You can again check that this works and a larger sample of 10,000 events can be found `root://eosuser.cern.ch//eos/user/l/lhcbsk/sim-lesson/GaussTightCut-27163003-10000ev.xgen`. You might also notice a slight slow-down in the rate at which events are produced: by default, an event failing the generator cut (which is applied after Pythia and EvtGen are done) triggers a reset of the entire generation phase of the simulation. Therefore, very tight generator level cuts in combination with a signal particle that only rarely occurs in minimum bias events can results in the generation phase taking manifold longer than the simulation of the detector response (and you might want to rethink your strategy for event generation).

Modifying cut tools for production

As the cut tools are to be configured in the DecFiles, they form an integral part of the event-type itself. Hence, any modification that changes the produced events usually requires the release of a new DecFile including a new event-type or a new simulation sub-version so events end up in a different bookkeeping location.

Learning Objectives

- Learn about existing fast simulation options

Why fast simulation is crucial

Our production queues are beyond full and nominal simulation for all requests is unsustainable for the future. Using fast simulation or being smart about how you use simulated samples is unavoidable. Fortunately, some fast simulation options already exist which are ready to be used and a very brief overview is given below (excluding options still in development).

In general, Gauss is a very flexible and extensible framework and if you require a large simulated sample it is useful to think about whether some change to the configuration, a reordering of steps or the event loop might net large speed-ups basically for free.

Speeding up the detector simulation

Reduced detector geometry

It is possible to turn off parts of the detector that might not be necessary for the specific analysis. Most prominently, the RICH detectors propagation of the optical photons can be deactivated which reduces the overall time required to simulate an event by 30%. More aggressively, all but the tracking system can be turned of, netting an increase in speed by about a factor of 10. However, the latter severely impacts the usability of the samples.

Particle gun

Instead of generating a full Pythia event, a single signal particle is spawned whose kinematics can be configured to follow various different distribution (with distributions as in Pythia8 available). The particle is then decayed using EvtGen to the desired final state. The overall increase in speed is about a factor of 50. However, resolutions and efficiencies are usually too good (compared to the full simulation) as the detector occupancy is much lower.

ReDecay

Allows to re-use the underlying event but generates and simulates new signal decays every time. Depending on the studied signal decay multiplicity, speed-ups by a factor between 10 and 20 are typically seen. Additionally, the same precision as in the nominal simulation is reached. However, as the underlying event and the kinematics of the signal particle remain unchanged, correlations between different events are introduced whose effect depends on the studied observables and need to be handled with care if significant.

Speeding up the generation

Split generator level cuts

Imagine you are studying the decay of a signal particle that is relatively rarely produced in Pythia 8 minimum bias events. In addition you apply tight generator level cuts which do not depend on what Pythia did (i.e. Lorentz-invariant quantities such as some invariant mass combination of some children). We have the ability to split the generator cuts: One part is applied as usual, triggering a reset of the generation phase if not passed. The other part is applied immediately after EvtGen generated the signal decay. If the generated decay fails (e.g. the invariant mass combination of two children is below a threshold), the decay products are removed and a new decay is generated until the cuts are passed. This avoids rerunning Pythia unnecessarily and can lead to substantial CPU savings when studying rare particles (e.g. $$\Lambda_b$$).

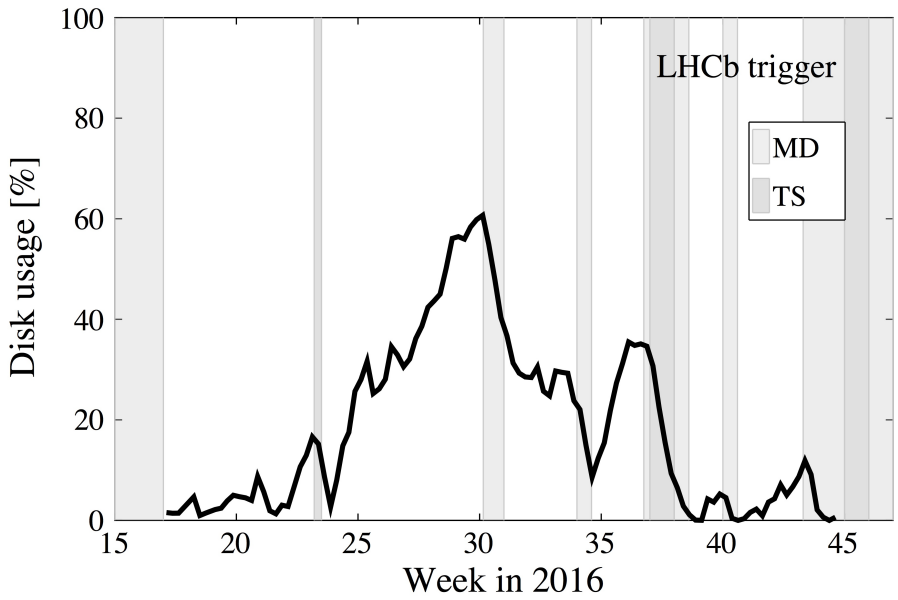
HLT intro

Learning Objectives

- Learn about the LHCb trigger.
- Learn how to run Moore from settings and from TCK.
- Getting started with writing your own trigger selection.

The LHCb trigger reduces the input rate event rate of approximately 30 MHz to 12.5 kHz of events that are written to offline tape storage. The rate reduction is achieved in three steps: L0, HLT1 and HLT2. L0 is implemented in custom FPGAs, has a fixed latency of 4 µs and a maximum output rate of 1 MHz. HLT stands for High Level Trigger.

HLT1 and HLT2 are implemented as software applications that run on the online farm; HLT1 runs in real-time and writes events to the local harddrives of the farm machines, while HLT2 uses the rest of the available CPU (100% when there is no beam) to process the events written by HLT1. The evolution of the disk buffer is shown in the figure below. Events accepted by HLT2 are sent to offline storage.



In Run I, both the reconstructions and selections used by HLT1, HLT2 and offline were very different. In Run 2 the reconstructions used in HLT2 and offline are identical, while selections might still be different. To be fast, HLT1 runs a subset of the HLT2 reconstruction, for example HLT1 provides only tracks with PT > 500 MeV and only Muon particle identification. We will see the difference in speed in the following.

A new feature in Run 2 is the so-called Turbo stream. Since the reconstruction available in HLT2 is the same as the offline reconstruction, physics analyses can be done with the candidates created in HLT2. If a line is configured to be a Turbo line, all information on the candidates that it selects is stored in the raw event. The rest of the raw event, like sub-detector raw banks, is discarded, and cannot be recovered offline. The advantage of the Turbo stream is that less data are written to tape and no CPU intensive offline processing is needed. The output of the Turbo stream is almost identical to the output of a Stripping line which goes to mdst.

The application of the software trigger is called Moore. Moore relies on the same algorithms as are used in Brunel to run the reconstruction and in DaVinci to select particle decays.

Run Moore from settings

Let's start with a simple Moore script, we call it runMoore.py:

```
from Configurables import Moore
# Define settings
Moore().ThresholdSettings = "Physics_pp_2018"
Moore().RemoveInputHltRawBanks = True
Moore().Split = ''
# A bit more output
from Gaudi.Configuration import INFO
Moore().EnableTimer = True
#Moore().OutputLevel = INFO
# Input data
from PRConfig import TestFileDB
# The following call configures input data, database tags and data type
TestFileDB.test_file_db["2017NB_L0Filt0x1707"].run(configurable=Moore())
Moore().DataType = "2018"
# Override the TCK in the ThresholdSettings to match the input data
from Configurables import HltConf
HltConf().setProp("L0TCK", '0x1707')
# Remove a line which accepts every event when run on this sample
HltConf().RemoveHlt1Lines = ["Hlt1MBNoBias"]

Moore().EvtMax = 1000

from Configurables import HltMonitoringConf
HltMonitoringConf().OutputFile = "histos.root"

print Moore()
```

Try to run it with

```
$ lb-run Moore/prod gaudirun.py runMoore.py | tee log.txt
```

The property split defines if HLT1, HLT2 or both are run. In the example above both are run. Change `Moore().Split` to `'Hlt1'` and rerun. To run HLT2 only, you have to change some settings and the input file has to be a file where HLT1 has run on:

```
...
Moore().RemoveInputHltRawBanks = False # Why?
Moore().Split = 'Hlt2'
...
TestFileDB.test_file_db["2017_Hlt1_0x11611709"].run(configurable=Moore())
HltConf().setProp("L0TCK", '0x1709')
...
```

Note: HLT2 needs to know about the decisions of trigger lines used in HLT1. The decisions are decoded from the definitions in the HLT1 TCK. Therefore, HLT2 can only read data which have been created when running Moore from TCK and not from settings.

Compare Hlt1 and Hlt2

What is reduction factor of Hlt1? (Search how many events are accepted by `Hlt1Global`.)

What is reduction factor of Hlt2? (Search how many events are accepted by `Hlt2Global`.)

What is difference in run time of Hlt1 and Hlt2?

Run Moore from TCK*

There are two ways to run Moore, from `ThresholdSettings` and from `TCK` (Trigger Configuration Key). When you develop a trigger line, it is more convenient to run from ThresholdSettings. The TCK is used when running the trigger on the online farm or in MC

productions as it uniquely defines the settings.

What is a TCK?

The Trigger Configuration Key (TCK) stores the configuration of the HLT in a database. All algorithms and their properties are defined in it. The key is usually given as a hexadecimal number. The last 4 digits define the L0 TCK. The first 4 digits define the HLT configuration. HLT1 TCKs start with 1, HLT2 TCKs start with 2.

Running from TCK has a few restrictions: 1 The L0TCK defined in the TCK and the one in data have to match. 2 The HltTCK might be incompatible with a Moore version if the properties of C++ algorithms changed.

Here is an example script to run from an Hlt1 TCK.

```
from Configurables import Moore
# Define settings
Moore().UseTCK = True
# You can check in TCKsh which TCKs exist and for which Moore versions they can be used.
Moore().InitialTCK = "0x117318A1"
Moore().Split = 'Hlt1'
Moore().RemoveInputHltRawBanks = True
# In the online farm Moore checks if the TCK in data and the configuration are the same.
# Here we disable it as we run a different TCK.
Moore().CheckOdin = False
Moore().outputFile = "TestTCK1.mdf"
# A bit more output
from Gaudi.Configuration import INFO
Moore().EnableTimer = True
#Moore().OutputLevel = INFO
# Input data
from PRConfig import TestFileDB
TestFileDB.test_file_db["2017NB_L0Filt0x18A1"].run(configurable=Moore())
Moore().DataType = "2018"
Moore().EvtMax = 1000
print Moore()
```

Run with

```
$ lb-run Moore/v28r2 gaudirun.py runMoore_hlt1_tck.py | tee log_hlt1_tck.txt
```

To run HLT2 on the output data of the first stage, use the following script:

```
from Configurables import Moore
# Define settings
Moore().UseTCK = True
Moore().InitialTCK = "0x217318A1"
Moore().DataType = "2018"
Moore().Split = 'Hlt2'
Moore().RemoveInputHltRawBanks = False
Moore().CheckOdin = False
Moore().EnableOutputStreaming = True
Moore().outputFile = "TestTCK2.mdf"
# A bit more output
from Gaudi.Configuration import INFO
Moore().EnableTimer = True
#Moore().OutputLevel = INFO
# Input data
Moore().DDDBtag = 'dddb-20150724'
Moore().CondDBtag = 'cond-20170724'
Moore().inputFiles = ["TestTCK1.mdf"]
Moore().EvtMax = 100
```

Exploring a TCK

If you are interested in how to create a TCK, you can follow the instructions given here.

To get a list of all available TCKs one can use TCKsh which is a python shell with predefined functions to explore TCKs, do

```
$ lb-run Moore/latest TCKsh
> listConfigurations()
```

Useful commands are `listHlt1Lines(<TCK>)` or `listHlt2Lines(<TCK>)` which show lists of the lines in Hlt1 or Hlt2. If you replace list with get, a list with the lines is returned.

If you want to get an overview of an Hlt line and its algorithms, you can do

```
> dump(<TCK>, lines = <line_names_regex>, file="output.txt")
```

More advanced is to search for properties of a line. One example is to search for the prescale. The prescale determines how often a line is executed, 1.0 means always, 0.0 never. Type for example:

```
> listProperties(0x214a160f,".*Hlt2DiMuonJPsi.*","AcceptFraction")
...
> listProperties(0x214a160f,".*Hlt2DiMuonJPsiPreScaler.*","AcceptFraction")
```

The regex is needed to search through all algorithms connected to a line.

Compare HLT1 lines from Run1 and Run2

Try to find out which HLT1 lines were available in Run 1 and which are now available in Run 2.

What are the names of the topological trigger lines in Run 1 and Run 2?

Write your own HLT2 trigger line or adapt an existing one

HLT2 lines are similar to stripping lines. They combine basic particles to composite objects and you apply selections to get a clean sample. The framework in which you write a trigger line looks different to a stripping line but the underlying algorithms are the same. Documentation is found here. There you also find information how to measure the efficiency or the output rate of a trigger line.

HLT2 lines are found in the Hlt gitlab project in the package Hlt2Lines, see here.

Their settings, i.e. the cut definitions, have to be defined in HltSettings package as well, see here.

As a hands on, we will change the prescale of a line with a high rate and then reduce its rate with extra cuts. First setup a Moore lb-dev project from the nightlies.

```
$ lb-dev Moore/prod
$ cd MooreDev_prod
$ git lb-use Hlt
$ git lb-checkout Hlt/2018-patches Hlt/HltSettings
$ git lb-checkout Hlt/2018-patches Hlt/Hlt2Lines
$ make
```

Go to `Hlt/HltSettings/python/HltSettings/DiMuon/DiMuon_pp_2018.py` , search for prescale and change the prescale of `Hlt2DiMuonJPsi` to 1.0. Run Moore again and see if the rate of this line has increased.

The line is defined in `Hlt/Hlt2Lines/python/Hlt2Lines/DiMuon/Lines.py` . The cut properties appear in the dictionary under `JPsi` . Add an entry with the key `MinProbNN` and set some value. If you search for `JPsi` in the file, you will find that the lines uses `JpsiFilter` from

`Stages.py` . As it is used by another line as well, you have to add the entry to `JPsiHighPT` as well. JpsiFilter simply uses muon pairs as input. Go to Stages.py and adapt the code of the `Hlt2ParticleFilter` to filter on the pid of the muons, to do that add `(MINTREE('mu-' == ABSID, PROBNNmu) > %(MinProbNN)s )` to the cut string. Run Moore again and see if the rate of this line has now decreased. If you are happy, you can do

```
$ git lb-push Hlt myNewLine
```

and create a merge request.

For more complicated developments which require changing many files or concurrent development of several people, we encourage to use a full checkout of the `Moore` and `Hlt` projects and to use vanilla git commands. A user friendly setup for this is being developed under the name trigger-dev. We encourage people to check it out and give feedback on the issues page.

Convert a stripping line to a Hlt2 line

1. Pick a stripping line and convert it to a HLT2 line.
2. Make it a Turbo line (You have to set the property Turbo to true and the name of the line has to end with Turbo).
3. Run a rate test to determine the rate of the line, instructions are found here.

TisTos DIY

Learning Objectives

- Learn what TisTos is and why it's useful
- Use an interactive python session to look at TisTos on a local DST

Once HLT1 or HLT2 has accepted an event, the candidates accepted by all trigger lines are saved to the raw event in a stripped-down form. One of the things that is saved are all the LHCbIDs of the final state particles of a decay tree.

What is an LHCbID?

Every single sub-detector element has an LHCbID which is unique across the whole detector. Physics objects, such as tracks, can be defined as sets of LHCbID objects. When a trigger decision is made, the set of LHCbID objects which comprise the triggering object is stored in the SelReports. This allows objects reconstructed later, such as in Brunel, to be compared with the objects reconstructed in the trigger.

A new feature in Run 2 is the so-called Turbo stream. Since the reconstruction available in HLT2 is the same as the offline reconstruction, physics analysis can be done with the candidates created in HLT2. If a line is configured to be a Turbo line, all information on the candidates that it selects is stored in the raw event. These candidates can be resurrected later by the Tesla application and written to a microDST. This is similar to stripping streams that go to microDST, where only candidates that are used in passing selections are available to analysts. The Turbo stream is different because information that is not saved is lost forever.

We will now have a look at some of the candidates stored by the HLT. We will use the script we used last time as a starting point, and the file `root://eoslhcb.cern.ch//eos/lhcb/user/r/raaij/Impactkit/00051318_00000509_1.turbo.mdst`. This file contains some 2016 Turbo events from run 174252. Fire up your favourite editor, open the script and save a copy to work on as `hlt_info.py`. There are a few things in the script that we don't need and can be removed, such as the `print_decay` method and the decay finder tools.

Like the stripping, the decisions of the HLT are saved in so-called DecReports. You can find them in `Hlt1/DecReports` and `Hlt2/DecReports`, have a look at what they contain.

```
evt['Hlt1/DecReports']
evt['Hlt2/DecReports']
```

An important difference with the stripping is that for the HLT, all the decisions are present, even if they are false. Copy the `advance` function to `advance_hlt`, make it work on HLT decisions and make sure it really checks the decision. This can be done using the `decision` member function of a `DecReport`. Note that all names in the reports end with `Decision`.

```
reports = evt['Hlt1/DecReports']
report = reports.decReport('Hlt1TrackAllL0Decision')
print report.decision()
```

The HLT1 selections that are most efficient for hadronic charm and beauty decays in Run 2 are called Hlt1TrackMVA and Hlt1TwoTrackMVA. Use the advance function to find an event that was accepted by either of these trigger selections.

The DecReports only contains the decisions for each line, 1 or 0. The candidates themselves are stored in the SelReports ("Hlt{1,2}/SelReports"). Get the HLT1 SelReports from the event store and retrieve the one for one of the TrackMVA selections using the

selReport function, analogously to how the DecReport was retrieved above.

The SelReports store candidates in a tree of sub-structures, which can be accessed using the `substructure` member funtion of a report. Any SelReport has at least one level of sub-structure. The sub-structure is internally stored in SmartRefs, which can be dereferenced using their "data" method. Let's have a look at a SelReport for a TrackMVA selection.

```
reports = evt['Hlt1/SelReports']
report = reports.selReport('Hlt1TrackMVADecision')
print report
report.substructure().size()
report.substructure()[0].substructure().size()
report.substructure()[0].substructure()[0]
report.substructure()[0].substructure()[0].data()
```

In addition to the LHCbIDs, some numbers are also stored, such as the momentum of the track. These are stored in the numerical info dictionary that can be retrieved using:

```
report.substructure()[0].substructure()[0].numericalInfo()
```

Plot the transverse momentum distribution

Make a plot of the total and transverse moment distributions of all candidates accepted by the Hlt1TrackMVA selection. Then add Hlt1TwoTrackMVA and consider the difference.

The LHCbIDs of the (in this case) track can be retrieved using:

```
report.substructure()[0].substructure()[0].lhcbIDs()
```

Turbo candidates

Now let's have a look at the same information that is stored for a candidate created by a Turbo line, for example Hlt2CharmHadDsp2KS0PimPipPip_KS0LLTurbo. Adapt the `advance_hlt` with an additional argument that allows specification of the location of `DecReports` it uses, then advance to an event that was selected by Hlt2CharmHadDsp2KS0PimPipPip_KS0LLTurbo, retrieve its `SelReport` and have a look at what's stored.

The LHCbIDs of the final state particles of the candidate that was created offline (in the stripping or by your script) can be compared to those saved by the HLT to find out if the offline candidate was accepted by the trigger. The classification that results from this comparison is called TisTos (Trigger independent of Signal/Trigger on Signal).

An offline candidate is considered to be Tos with respect to a trigger selection if it was accepted by that trigger selection. In more formal terms, if the LHCbIDs of each of the final state particles of the candidate accepted by the trigger selection overlap for more than 70% with the LHCbIDs of final state particles of the offline candidate.

For example, the Hlt1TrackAllL0 line accepts an event if there is at least one track with a lot of PT and a large IPCHI2. If any of the tracks accepted by the Hlt1TrackAllL0 line overlap for more than 70% with one of the tracks of the offline candidate, it is Tos with respect to Hlt1TrackAllL0. If Hlt1DiMuonHighMass is considered instead, then the LHCbIDs of both tracks that make-up the Hlt1DiMuonHighMass candidates must overlap with the LHCbIDs of two tracks that are part of the offline candidate.

To have a look at how this works, we'll use candidates from the D2hhPromptDst2D2RS selection, which can be retrieved thusly:

```
candidates = evt['AllStreams/Phys/D2hhPromptDst2D2RSLine/Particles']
candidates.size()
```

It could be that there are more than one candidates, which are unlikely to all be real. MC matching could be used to find the real one when running on simulation and on data a single candidate can be selected, either randomly or using some criterium. Dealing with multiple candidates correctly is beyond the scope of this tutorial, so just always take the first one in the container.

Let's use the TriggerTisTos tool now. In preparation for Run-II, the Hlt1 and Hlt2 DecReports and SelReports are now stored in different locations. That means two TisTos tools will be needed, each configured to pick up information from either HLT1 or HLT2. Since the tools we create are public tools, they have to be configured in the following way (before the AppMgr is instantiated):

```
from Configurables import ToolSvc, TriggerTisTos
ToolSvc().addTool(TriggerTisTos, "Hlt1TriggerTisTos")
ToolSvc().Hlt1TriggerTisTos.HltDecReportsLocation = 'Hlt1/DecReports'
ToolSvc().Hlt1TriggerTisTos.HltSelReportsLocation = 'Hlt1/SelReports'
```

Create the tools in the same way you created others during the other lesson, but use instance-specific names that correspond to the configuration we just added: `Hlt1TriggerTisTos` and `Hlt2TriggerTisTos`. The tools use `ITriggerTisTos` as an interface.

Use the advance function to find an event that has some candidates for the chosen selection and set the TisTos tools to use our candidate and trigger selection:

```
hlt1TisTosTool.setOfflineInput()
candidate = candidates[0]
hlt1TisTosTool.addToOfflineInput(candidate)
hlt1TisTosTool.setTriggerInput()
hlt1TisTosTool.addToTriggerInput("Hlt1TrackAllL0Decision")
result = hlt1TisTosTool.tisTosTobTrigger()
result.tos()
```

The `set` calls reset the internal storage of candidate or trigger information, and the `addTo` calls then add the things we are interested in.

An offline candidate is considered to be Tis with respect to a trigger selection if removing it from the event would still cause the trigger selection to accept the event, i.e. if there is another particle in the event that was also accepted by the trigger selection. In more formal terms, if the LHCbIDs of the all of the final state particles of any of the candidates accepted by the trigger selection overlap less than 1% with all of the LHCbIDs of the final state particles of the offline candidate.

```
result.tis()
```

Note that a candidate can be both Tis and Tos with respect to a trigger selection, or Tos with respect to one selection, and Tis with respect to another. To tell the tool to consider more trigger selections, use the following (regexes are also supported), and try to find some events that are both Tos and Tis:

```
hlt1TisTosTool.setTriggerInput()
hlt1TisTosTool.addToTriggerInput("Hlt1TrackAllL0Decision")
hlt1TisTosTool.addToTriggerInput("Hlt1DiMuonHighMassDecision")
result = hlt1TisTosTool.tisTosTobTrigger()
```

The (Tos) trigger efficiency of a trigger selection can be calculated as:

$$\epsilon_{\mathrm{Tos}}=N_{\mathrm{Tis}\&\mathrm{Tos}} / {N_{\mathrm{Tis}}}$$

Loop over the events in the DST and calculate the efficiency of Hlt1TrackAllL0. You can add some more Hlt1 selecitons when checking for Tis, which ones would make sense?

There is a third classification, which is called Tob. This is the case if the overlap — as defined for Tis and Tos — is between 1% and 70%.

To determine if a candidate is a combination of Tis, Tos and Tob or none of these, an LHCb software tool has been created that calculates the overlaps and classifies candidates with respect to trigger selection. This tool is called TriggerTisTos and it implements the ITriggerTisTos interface.

Scripting Ganga

We have already started using Ganga, such as when submitting jobs to the Grid and using datasets from the bookkeeping, for creating jobs; but there's a lot more you can do with it.

Part of Ganga's power comes from it being written in Python. When you run `ganga`, you're given an IPython prompt where you input Python code that's executed when you hit `<enter>`. The idea of running Python code extends outside of Ganga, where we can write scripts that Ganga will execute when starting up. This lesson will focus on writing job definition scripts, and exploring how we can define utility functions that will be available across all of our Ganga sessions.

Defining jobs with scripts

The `ganga` executable is similar to the `python` and `ipython` executables in a couple of ways. If you just run `ganga`, you are dropped into a prompt, but you can also supply the path to a Python script that will be executed. Let's start with a small script, saving it in a file called `create_job.py`:

```
greeting = 'Hello!'
print greeting
```

Run it:

```
$ ganga create_job.py

*** Welcome to Ganga ***
Version: X.Y.Z
…
Hello!
…
```

Sensible enough. Just like `python` and `ipython`, we can pass the `-i` flag before the file path to tell Ganga to give us a prompt after it's finished executing the script:

```
$ ganga -i create_job.py

*** Welcome to Ganga ***
Version: X.Y.Z
…
Hello!

Ganga In [1]: greeting
Ganga Out [1]: 'Hello!'

Ganga In [2]:
```

Notice that the variable we defined in the script, `greeting`, is available in the interactive session. The idea of doing some work in a script and then manipulating the result interactively can be quite powerful.

One workflow that you might find useful is to create a script that defines a job, because this can often take a few lines to do, and typing them out every time is boring. Let's modify `create_job.py` to do that.

```
# Note: Ganga makes objects like `Job` available in your script automagically
j = Job()
j.name = 'My job'
```

This example is quite boring, but it captures the idea. You'll want to extend this, changing the `application` property to a `GaudiExec` instance, for example, as covered in a previous lesson.

Now we can run this and interact with the job as the `j` variable:

```
$ ganga -i create_job.py

*** Welcome to Ganga ***
Version: X.Y.Z
…

Ganga In [1]: j
Ganga Out [1]:
Job (
  comment = ,
  parallel_submit = False,
  …
)

Ganga In [2]:
```

We often want to construct a set of very similar jobs that differ only by their input data, for example running the same DaVinci application over 2015 and 2016 data and for magnet up and magnet down configurations. We need to then *parameterise* our script, and one way of doing this is passing arguments to it by the command line. You can inspect arguments from a Python script by using the `argv` property on the `sys` module:

```
import sys
print sys.argv
```

Add that to your `create_job.py` script, and run `ganga` again, this time passing some arguments:

```
$ ganga -i create_job.py -v 123 --hello=world

*** Welcome to Ganga ***
Version: X.Y.Z
…

['create_job.py', '-v', '123', '--hello=world']

Ganga In [1]: j
```

Our script sees `sys.argv` as the list of the arguments that come after `ganga -i`. To parameterise our script for year and magnet polarity, we could check this list to find one of `2015` or `2016` and one of `Up` or `Down`, for example, but instead we'll opt to use the excellent `argparse` module, which comes with Python, to parse the command-line arguments for us.

```
import argparse

parser = argparse.ArgumentParser(description="Make my DaVinci job.")
parser.add_argument('year', type=int, choices=[2015, 2016],
                    help='Year of data-taking to run over')
parser.add_argument('polarity', choices=['Up', 'Down'],
                    help='Polarity of data-taking to run over')
parser.add_argument('--test', action='store_true',
                    help='Run over one file locally')
args = parser.parse_args()

year = args.year
polarity = args.polarity
test = args.test
```

Nicely, `argparse` gives us a useful `--help` argument for free:

```
$ ganga -i create_job.py --help

*** Welcome to Ganga ***
Version: X.Y.Z
…
usage: create_job.py [-h] [--test] {2015,2016} {Up,Down}

Make my DaVinci job.

positional arguments:
  {2015,2016}  Year of data-taking to run over
  {Up,Down}    Polarity of data-taking to run over

optional arguments:
  -h, --help   show this help message and exit
  --test       Run over one file locally

Ganga In [1]:
```

This help will also be printed if we don't supply all of the required arguments (the year and the magnet polarity), along with a message telling us what's missing.

Getting to grips with `argparse`

The `argparse` module can do a lot, being able to parse complex sets of arguments with much difficultly. It's a useful tool to know in general, so we recommend that you check out the documentation to learn more.

When we do supply all the necessary arguments, the values are then available in the `year`, `polarity`, and `test` variables:

```
$ ganga -i create_job.py 2015 Down

*** Welcome to Ganga ***
Version: X.Y.Z
…

Ganga In [1]: print year, polarity, test
2015 Down False
```

Once you've reached this level, a whole world of possibilities opens up! Here are a few things you might proceed to do with these parameters in your script:

- Fetch the corresponding dataset using a `BKQuery`;
- Give your `Job` object a specific name, e.g. `j.name = 'Ntuples_{0}_{1}.format(year, polarity)'`;
- Give data-specific options files to the `application` object, e.g. if you have one options file per year defining `DaVinci().DataType`.

Of course, you can add as many arguments as you think might be useful. Above we added the `--test` flag as an example: if this is `True`, you could run the application over only a single data file, and run the job locally rather than on the Grid (setting `j.backend` appropriately).

Adding helpers functions

We've seen above how giving a script to `ganga` makes the variables defined in those scripts available interactively. But what if you have, or would like to have, some set of your own custom helper methods defined in *every* session? It would be annoying to have to run `ganga my_helpers.py` every time! Luckily, the `ganga.py` file comes to the rescue.

When Ganga starts, it looks for a file in your home directory (`echo $HOME`) called `.ganga.py` (note the starting period in the filename). If it finds such a file, it executes it in the context of the Ganga session, meaning the code in the file has access to Ganga objects like `Job`, `jobs`, and so on. To demonstrate the behaviour, we can put a `print` statement on our `~/.ganga.py` file:

```
print 'Yo!'
```

Then run `ganga` (no arguments needed):

```
$ ganga

*** Welcome to Ganga ***
Version: X.Y.Z
…
Yo!
…
```

Neat. The general idea for this file is two-fold:

1. Add commands that you always want executed when Ganga starts, e.g. `print jobs.select(status='running')` ; and
2. Define functions for commonly-performed tasks.

The latter is particularly interesting. Do you often find yourself creating a file that contains all the output LFNs of your job? Write a helper!

```
def write_lfns(job, filename):
    """Write LFNs of all DiracFiles of all completed subjobs to fname."""
    # Treat a job with subjobs the same as a job with no subjobs
    jobs = j.subjobs
    if len(jobs) == 0:
        jobs = [job]

    lfns = []
    for j in jobs:
        if j.status != 'completed':
            print 'Skipping #{0}'.format(j.id)
            continue
        for df in j.outputfiles.get(DiracFile):
            lfns.append(df.lfn)

    with open(filename, 'w') as f:
        f.writelfns('\n'.join(lfns))
```

How about downloading and merging the ROOT output of a job's subjobs? Write a helper!

```
def merge_root_output(job, input_tree_name, merged_filepath):
    # Treat a job with subjobs the same as a job with no subjobs
    jobs = j.subjobs
    if len(jobs) == 0:
        jobs = [job]

    access_urls = []
    for j in jobs:
        if j.status != 'completed':
            print 'Skipping #{0}'.format(j.id)
            continue
        for df in j.outputfiles.get(DiracFile):
            access_urls.append(df.accessURL())

    tchain = ROOT.TChain(input_tree_name)
    for url in access_urls:
        tchain.Add(url)
    tchain.Merge(merged_filepath)
```

Because of the way a ROOT `TChain` works, the subjobs output won't be downloaded, so you only need enough disk space for the merged file.

Using ROOT in Ganga

By default, ROOT is not available in a Ganga session:

```
Ganga In [1]: import ROOT
ERROR    No module named ROOT
```

To remedy this, you can start Ganga inside an environment where ROOT _is_ available:

```
$ lb-run ROOT ganga
```

Once you have your helpers defined, use them in Ganga as you would any other Python function.

```
Ganga In [1]: j = jobs(123)

Ganga In [2]: write_lfns(j, '{0}.lfns'.format(j.name))
```

Here are some other common operations that you might want helpers for:

- Deleting all LFNs created by a job;
- Resetting the `backend` of all subjobs which are marked as `failed`;
- Replicating all LFNs to a specific Grid site.

What other tasks can you think of?

Managing files in Ganga

Learning Objectives

- Choose whether job output is saved locally or on the Grid
- Choose where to look for job input files
- Move files from any grid site to CERN, for analysis using EOS

Ganga allows you to define a job and have it run anywhere: on your local machine, on the batch system, or on the Grid. This is very convenient as you don't need to worry about the specifics of each platform.

Ganga treats files in a similar way to jobs, in that you only need to change the object you're using to tell Ganga to use local files, files on the Grid, or files on EOS. In this lesson, we'll see how you can efficiently manage input and output files using Ganga.

Ganga versions

It is generally advised to use the latest available version of Ganga. Functionality is not removed and there are no compatibility issues between versions (it is just python!). If you encounter problems, you should first search the archives of the `lhcb-distributed-analysis` mailing list. If you don't find an answer, you can talk to the Ganga developers directly on the GitHub issues page for Ganga, on the ~distributed-analysis mattermost channel, or by sending an email to `lhcb-distributed-analysis`.

Making a fresh start

To make sure there will be no pre-existing files from of Ganga to interfere, we will move them to a backup location.

```
$ cd ~
$ mkdir ganga-backup
# See what's in your home directory that's related to Ganga
$ ls -la | grep -i ganga
# Then move everything
$ mv gangadir .gangarc* .ganga.log .ganga.py .ipython-ganga ganga-backup
```

You can move this back after the lesson if you want to restore your old settings and data.

We'll be doing everything in Ganga, so let's start it up.

```
$ ganga
```

If it's your first time starting Ganga, you'll be asked if you want to create a default `.gangarc` file with the default settings.

> Would you like to create default config file `~/.gangarc` with standard settings ([y]/n) ?

Answer with `y`. The `.gangarc` file defines the configuration of Ganga, and the defaults are normally good enough.

You'll then be dropped in a IPython shell. We will create a job that runs a Python script that accepts a path to an input text file as an

argument, and saves a file that contains the text of the file reversed. For example, it would save a file containing '!dlrow olleH' if it was given a file containing 'Hello world!' as input.

Download the script to lxplus and set it to be executable. You can execute these commands inside Ganga, if you like, by prefixing them with a `!`.

```
$ wget https://raw.githubusercontent.com/lhcb/starterkit-lessons/master/second-analysis-steps/code/01-managing-files-with-gang
a/reverse.py
$ chmod +x reverse.py
$ ./reverse.py
Usage: reverse.py <file>
```

In Ganga, create a `Job` object with a descriptive name and take a look at it.

```
j = Job(name='Reverser')
print j
```

You'll see that Ganga has created a `Job` which will execute the `echo` command, passing the list of arguments `['Hello World']`. Each element of this list will be passed as a positional argument to the `echo` command.

We'll replace the command name and the arguments, so that our `reverse.py` script is run with a text file as input.

```
j.application.exe = File('reverse.py')
j.application.args = [File('input.txt')]
```

We haven't made `input.txt`, so let's make it by executing a couple of shell commands inside Ganga.

```
!echo -e "$(date)\nHello world!\nI am $USER!" > input.txt
!cat input.txt
```

Before submission, we just need to tell Ganga what to do with the output. The script saves the output to a file called like the input, but with `-reversed` appended before the file extension (`.txt` in this case), so we tell Ganga explicitly to move this file to the local job output directory.

```
j.outputfiles = [LocalFile('input-reversed.txt')]
```

Now we can submit the job.

```
j.submit()
```

By default, jobs run on the machine you're running Ganga on, as their `backend` property is set to an instance of the `Local` backend.

The job will finish very quickly, and we can inspect the output files.

```
j.peek()
j.peek('input-reversed.txt')
```

There are a couple of file-related things to take note of in what we just did:

1. The `File` object is used to define local files that should be available in the 'working directory' of the job (wherever it executes). We need both the script and the input text file to be in the working directory, so both of the paths to the files on our local machine are wrapped in `File`.
2. The `LocalFile` object is used to define what files in the working directory of the job should be saved in the local job output directory, in this case the file with `-reversed` in it.

Note that there are several files in the job output directory, seen with `j.peek()`, that we didn't explicitly ask for, most notably `stdout` and `stderr`. These two files are essentially the logs of the job, and Ganga always saves them in the local job output directory as they're almost always useful.

For Gaudi jobs, Ganga will also automatically download the `summary.xml` file, which contains useful information about algorithm counters.

```
df = DiracFile('input.txt', localDir='.')
df.put(uploadSE='CERN-USER')
print df.lfn
```

Couldn't upload file - This file GUID already exists

All files on the grid are required to have a unique identifier (GUID) which is normally generated from the file's content and is independent of its filename. As a result, if you try to upload a file which already exists you receive an error.

If this happens, and you have a reason to not use the pre-existing file, the simplest solution is to make the file unique in some way, in this case we add the date and time to the top line of the text file.

Grid files that are replicated at CERN are directly accessible via EOS. We can see that our file's on EOS by looking at the LFN Ganga gave us. We just need to add the prefix `/eos/lhcb/grid/user` to the LFN.

```
eos ls /eos/lhcb/grid/user//lhcb/user/a/apearce/GangaFiles_22.24_Wednesday_18_May_2016
```

Using MassStorageFile

The `MassStorageFile` object uploads job output directly to EOS. However, using `MassStorageFile` for this purpose is actively discouraged by the Ganga developers as it is highly inefficient: a file made on the Grid will first be downloaded to the machine running Ganga, and then uploaded to EOS.

Instead, always use `DiracFile` for large outputs, and then replicate them to `CERN-USER` if you want to be able to access them on EOS.

If you have any `DiracFile`, you can ask for it to be replicated to a grid site it's not currently available at.

```
df.replicate('RAL-USER')
```

Automating replication to CERN

If you have a job with subjobs, you can automate this to replicate all output files to CERN, so that you can run your analysis directly on the files on EOS.

```
j = jobs(...)
for sj in j.subjobs:
  # Get all output files which are DiracFile objects
  for df in sj.outputfiles.get(DiracFile):
    # No need to replicate if it's already at CERN
    if 'CERN-USER' not in df.locations:
      df.replicate('CERN-USER')
```

After you did this your files will go into "/eos/lhcb/grid/lhcb/{u}/{user}/"+LFN.

You could make a function from this and put it in your `.ganga.py` file, whose contents is available in any Ganga session.

You can download a `DiracFile` locally using the `get` method. If you already know an LFN, you can use this to quickly download it locally to play around with it. All you need to do is prefix the LFN with `LFN:` , and Ganga will assume that the file already exists on the Grid somewhere (whereas before it assumed the file was local).

```
df2 = DiracFile('LFN:' + df.lfn)
# The directory used for the download must exist first
!mkdir foo
dfr2.localDir = "foo"
dfr2.get()
!cat foo/input.txt
```

We can tell Ganga to upload the job output to the Grid automatically.

```
# Clone the job
j2 = Job(j)
j2.outputfiles = [DiracFile('*-reversed.txt')]
j2.submit()
```

Here we use a 'pattern' to tell Ganga that any file ending in `*-reversed.txt` should be uploaded to Grid storage. Both `DiracFile` and `LocalFile` support these patterns.

To download the output, we use `.get` as usual.

```
j2.outputfiles.get(DiracFile)[0].get()
```

Being able to manipulate files with Ganga can be very useful. Particularly for Gaudi-based jobs where:

1. We often specify large sets of `DiracFiles` as input, from the bookkeeping, but often want to download a file or two locally when testing options;
2. We want to duplicate a large number of output LFNs to `CERN-USER` so that we can use them directly with EOS and XRootD commands;
3. We want to job output to be download locally automatically when the job completes.

Defining inputfiles

The `inputfiles` attribute of a `Job` object works in a similar way to `outputfile` . In our example, the reverser script that the `Executable` application uses doesn't know how to handle things specified as `inputdata` , so we had to use `File` when defining the arguments.

For LHCb applications, you will almost always define the `inputdata` list using either `LocalFile` or `DiracFile` objects. Which one you will use just depends on where the input files are.

Analysis automation with snakemake

Learning Objectives

- Learn what analysis automation is and how it helps with analysis preservation
- Learn how to create a pipeline with Snakemake

Motivation

> "The Snakemake workflow management system is a tool to create reproducible and scalable data analyses"

- A workflow management system allows you to:
  - Keep a record of how your scripts are used and what their input dependencies are
  - Run multiple steps in sequence, parallelising where possible
  - Automatically detect if something changes and then reprocess data if needed
- Using a workflow management forces you to:
  - Keep your code and your locations in order
  - Structure your code so that it is user-independent
  - Standardise your scripts
    - Bonus: Standardised scripts can sometimes be used across analyses!

Documentation and installation

You can find full documentation for Snakemake at this link, you can also ask any questions you have on the ~reproducible channel on mattermost.

Snakemake requires Python 3, if you already have this available it can be easily installed using pip:

```
python3 -m pip install --user snakemake
# Depending on your PATH variable you may also need to use:
alias snakemake='python3 -m snakemake'
```

Installing on lxplus

Unfortunately most LHCb software only supports Python 2 and doesn't provide a Python 3 installation. When running on lxplus we recommend using the LCG Python 3 distribution and creating a function in your `.bashrc` to launch Snakemake so that it doesn't affect other LHCb applications. This can be done using this script by running:

```
curl -L http://cern.ch/go/Z8Nk | bash
source ~/.bashrc
```

You can now check if Snakemake is working by using `snakemake --help`.

Tutorial

Snakemake allows you to create a set of rules, each one defining a "step" of your analysis. The rules need to be written in a file called `Snakefile`. For each step you need to provide:

- The *input*: Data files, scripts, executables or any other files.
- The expected *output*. It's not required to list all possible outputs. Just those that you want to monitor or that are used by a subsequent step as inputs.
- A *command* to run to process the input and create the output.

The basic rule is:

```
rule myname:
    input: ['myinput1', 'myinput2']
    output: ['myoutput']
    shell: 'Some command to go from in to out'
```

An example. If you want to copy some text from a file called `input.txt` to `output.txt` you can do:

```
rule copy:
    input: 'input.txt'
    output: 'output.txt'
    shell: 'cp input.txt output.txt'
```

You can even avoid typos by substituting variables instead of typing the filenames twice:

```
rule merge_files:
    input: ['input_1.txt', 'input_1.txt']
    output: 'output.txt'
    shell: 'cat {input[0]} > {output} && cat {input[1]} >> {output}'
```

Input and output can also be parametrised using wildcards:

```
rule copy_and_echo:
    input: 'input/{filename}.txt'
    output: 'output/{filename}.txt'
    shell: 'echo {wildcards.filename} && cp {input} {output}'
```

If you then make another rule with `output/a_file.txt` and `output/another_file.txt` as inputs they will be automatically created by the `copy_and_echo` rule. This allows for rules to be reusable, for example to make a rule that can be used to process data with from different years or polarities.

Notice that:

- Inputs and outputs can be of any type
- You can provide python code after the tags. e.g. `input: glob("*.root")`
- If a single file is input or output you are allowed to omit the brackets
- Wildcards must always be present in the output of a rule (else it wouldn't be possible to know what they should be)

Write a snakefile with a single rule

To try out download:

```
wget https://github.com/lhcb/starterkit-lessons/raw/master/second-analysis-steps/code/snakemake/tutorial.tar
```

You will find one containing names and phone numbers. You can make one rule that, given a name extracts the line with the phone

of that person.

To do this in a shell you can use `grep`, which is a command that lists all lines in a file containing a certain text.

```
$ grep ciao test.txt
ciao a tutti
```

Usage and basic behaviour

And now that your `Snakefile` is done it's time to run! Just type

```
snakemake rulename_or_filename
```

This will:

1. Check that the inputs exist
   - If inputs exists → 2)
   - If inputs do not exist or have changed snakemake will check if there is an other rule that produces them → Go back to 1)
2. Run the command you defined in `rulename_or_filename` (or the rule that generates the filename that is given)
3. Check that the output was actually produced.

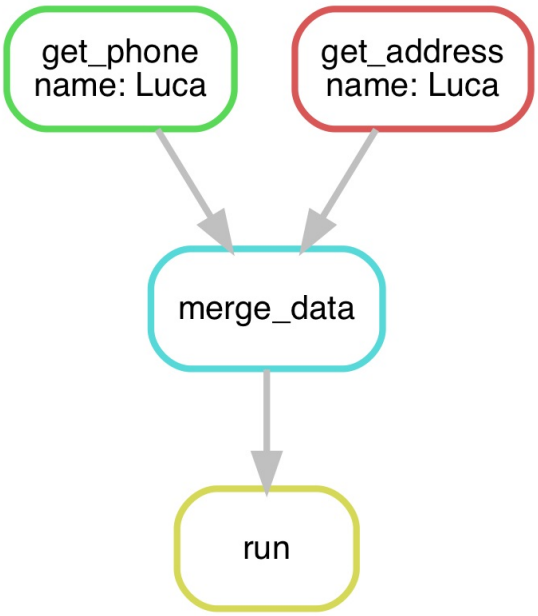
Comments, which rules are run:

- If want to run a chain of rules only up to a certain point just put the name of the rule up to which you want to run on the snakemake command.
- If you want a rule to be "standalone" just do not give its input/outputs as outputs/inputs of any other rule
- It is normal practice to put as a first rule a dummy rule that only takes as inputs all the "final" outputs you want to be created by any other rule. In this way when you run just `snakemake` with no label it will run all rules (in the correct order).

Make a snakefile with at least 3 rules connected to each other and run them in one go

In the tutorial folder you find two files containing addresses, and phone numbers. You can make rules that, given a name, `grep` the address and phone and then one other rule to merge them into your final output file.

If we do this for “Luca”, it can be represented by the following graph:



Which could be achieved using this shell script:

```
grep Luca inputs/addresses.txt > output/Luca/myaddress.txt
grep Luca inputs/phones.txt > output/Luca/myphone.txt
cat output/Luca/myaddress.txt > output/Luca/data.txt && cat output/Luca/myphone.txt >> output/Luca/data.txt
```

But it does not have to be this, any other task is fine, be creative!

Use wildcards

Following on from the previous challenge use wildcards to make it so that any name can be used, such as “Fred”

```
snakemake output/Fred/data.txt
```

Solution

See `Snakefile` in the `simple_solution` folder here.

Comments, partial running:

- If part of the input is already present and not modified present the corresponding rule will not run Note that if you put your code into the inputs snakemake will detect when your code changes and automatically rerun the corresponding rule!
- If you want to force running all rules even if part of the output is present use `snakemake --forceall`

Explore the snakemake behaviour

In the previous example try deleting one of the intermediate files, rerun snakemake and see what happens

Sub-labels

Inside the pre-defined tags you can add custom subtags as in this example.

```
rule run_some_py_script:
    input:
        exe = 'myscript.py',
        data = 'mydata.root',
        extra = 'some_extra_info.txt',
    output: ['output.txt']
    shell: 'python {input.exe} {input.data} --extra {input.extra} > {output}'
```

So this will effectively launch the command:

```
python myscript.py mydata.roo --extra some_extra_info.txt > output.txt
```

The `--extra` is not necessary. It's just to illustrate how python scripts options can be used.

Code as input

Add your python script to the inputs than make some modifications to it, rerun snakemake and see what happens.

Run and shell

You have two ways to specify commands. One is `shell` that assumes shell commands as shown before. The other is `run` that instead directly takes python code (Careful it's python3!).

For example the copy of the file as in the previous example can be done in the following way.

```
rule dosomething_py:
    input: 'myfile.txt'
    output: 'myoutput.txt'
    run:
        with open(input, 'rt') as fi:
            with open(output, 'wt') as fo:
                fo.write(fi.read())
```

And finally you can mix! Namely you can send shell commands from python code. This is useful, in particular if you have to launch the same shell command on more inputs.

```
rule dosomething_pysh:
    input:
        code = 'mycode.exe',
        data = [ 'data1.root', 'data2.root' ]
    output: ['plot1.pdf', 'plot2.pdf']
    run:
        for f in input.data:
            shell('./{input.code} %s' % f)
```

Use run instead of shell

Rewrite your previous file using a python script to run the search and use `run` to run on both phones and addresses in the same rule

Config files

Often you want to run the same rule on different sample or with different options for your scripts. This can be done in snakemake using config files written in yaml.

For example let's put the datafiles in a cfg.yaml file

```
data:
    - 'data1.root'
    - 'data2.root'
```

Now in your Snakefile you can load this config file and then its content will be available into the rules as a dictionary called "config". Yes, it seems black magic, but it works! Your Snakefile will look like this

```
configfile: '/path/to/cfg.yaml'

rule dosomething_pysh:
    input:
        code = 'mycode.exe',
        data = config['data'],
    output: ['plot1.pdf', 'plot2.pdf']
    run:
        for f in input:
            shell('./{input.code} %s' % f)
```

The config dictionary can be used anywhere, also inside the shell command or even outside a rule.

Make a config file

Put the inputs of your script into a config file

Includes

The Snakefile can quickly grow to a monster with tens of rules. For this reason it's possible to split them into more files and then include them into the Snakefile. For example you might have a `fit_rules.snake` and `efficiency_rules.snake` and then your Snakefile will look like this:

```
include: /path/to/fit_rules.snake
include: /path/to/efficiency_rules.snake
```

The order of the includes is irrelevant.

Use includes

Move your rules to other files and include them

Solution

You can find a solution in the `more_complete_solution` folder, which you can find here.

Self guided lessons

These tutorials have been written by Impactkit participants to give guidelines on how to perform a variety of generic tasks.

If you have any problems or questions on these tutorials, you can open an issue on the GitHub repository where these tutorials are developed.