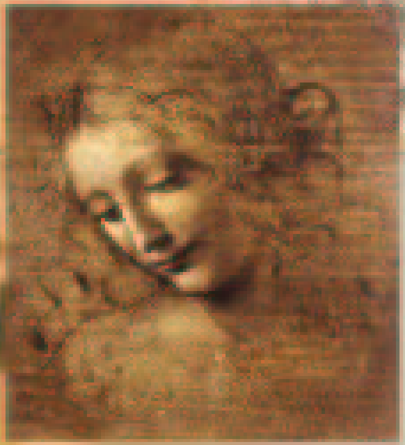
Introduction to DaVinci



- Overview
- First try
- Writing a simple algorithm
- Configuring Common Algorithms
- More about Tools
- Accessing MC truth

This session is not hands-on, but there are many examples one can try “at home”.

Patrick Koppenburg

Overview:

- Assumptions
- LHCb applications structure
- **DaVinci** structure
- Documentation sources

Assumptions

- It is assumed that you know (a little) about
  - `cmt` ...
  - **Gaudi** (some of it)
  - a few LHCb conventions
  - `C++`
- If not, have a look at the **Gaudi** tutorial (here), or at the Gaudi documentation

I assume the typical public for this tutorial are people who just did the **Gaudi** hands-on and would like to start using **DaVinci**.

I may well be wrong. . .

Assumptions

- It is assumed that you know (a little) about
  - `cmt` ...
  - **Gaudi** (some of it)
  - a few LHCb conventio
  - C++
- If not, h r at the Gaudi c

I assume the ar are people who just did the **Gaudi** h ld like to start using **DaVinci**.

I may well be wr ng. . .

Don't hesitate to interrupt and to ask questions!

Or to correct mistakes.

Conventions

Colour-coding:

- Words in Green are links to other pages
- Words in Blue are links to web pages

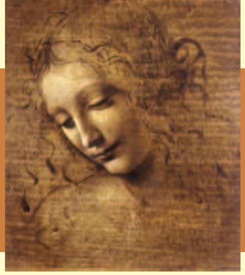
Fonts:

- `Fixed-width fonts are for code and options`
- `> echo "This is a shell command"`

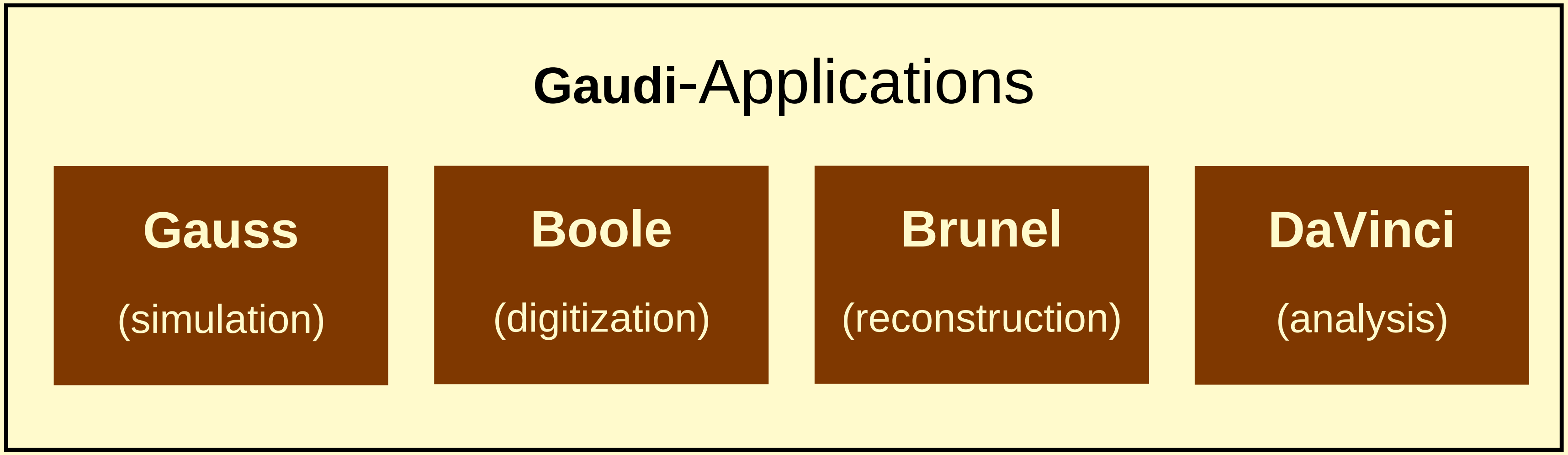
```
If it is boxed, then it is directly
copied from a *.h, *.cpp or *.opts file.
```

DaVinci Links

- **DaVinci** web page:
  http://lhcb-comp.web.cern.ch/lhcb-comp/Analysis/default.htm
  From there you'll find :
  - Some documentation
  - A "getting started" guide
  - FAQ
- Any question can be asked at the **DaVinci** mailing list:
  lhcb-davinci@cern.ch.
  - That's also the forum to propose improvements of **DaVinci**
  - You need to be registered to use it. Contact the secretariat at lhcb.secretariat@cern.ch.
- I am writing a reference guide for the "core" **DaVinci** code

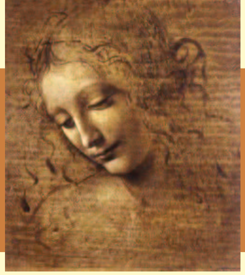


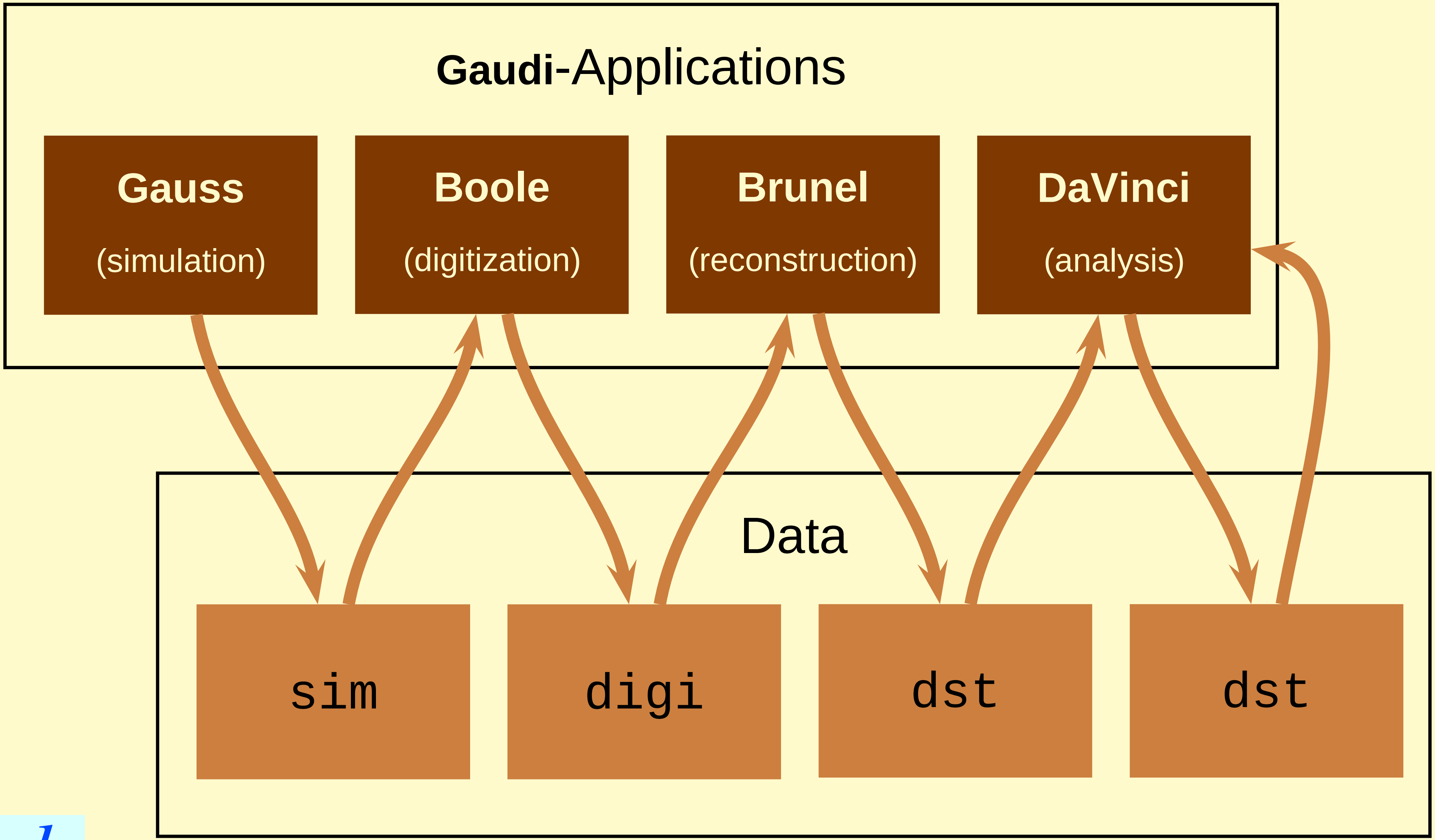
Applications



- There are four applications based on **Gaudi**
- They are actually all **Gaudi**-programs
- The only difference are the packages (shared libraries) included
- One could easily build an application that does it all (like in the old **SICB** days. . .)
- Somewhere here **Panoramix** and **Bender** are missing

Applications





Packages

DaVinci is a set of packages containing the code necessary to build a shared library and the relevant options.

They all have the sub-directories `cmt`, `src` and `options`

See the **Gaudi** tutorial for an explanation of the package structure.

- **DaVinci**-specific packages:

  `Phys/`: Physics algorithms and tools (16 packages)

  `Tools/`: Other tools (2), **LoKi** (2)

  `PhysSel/`: Specific decay channel selections (28)

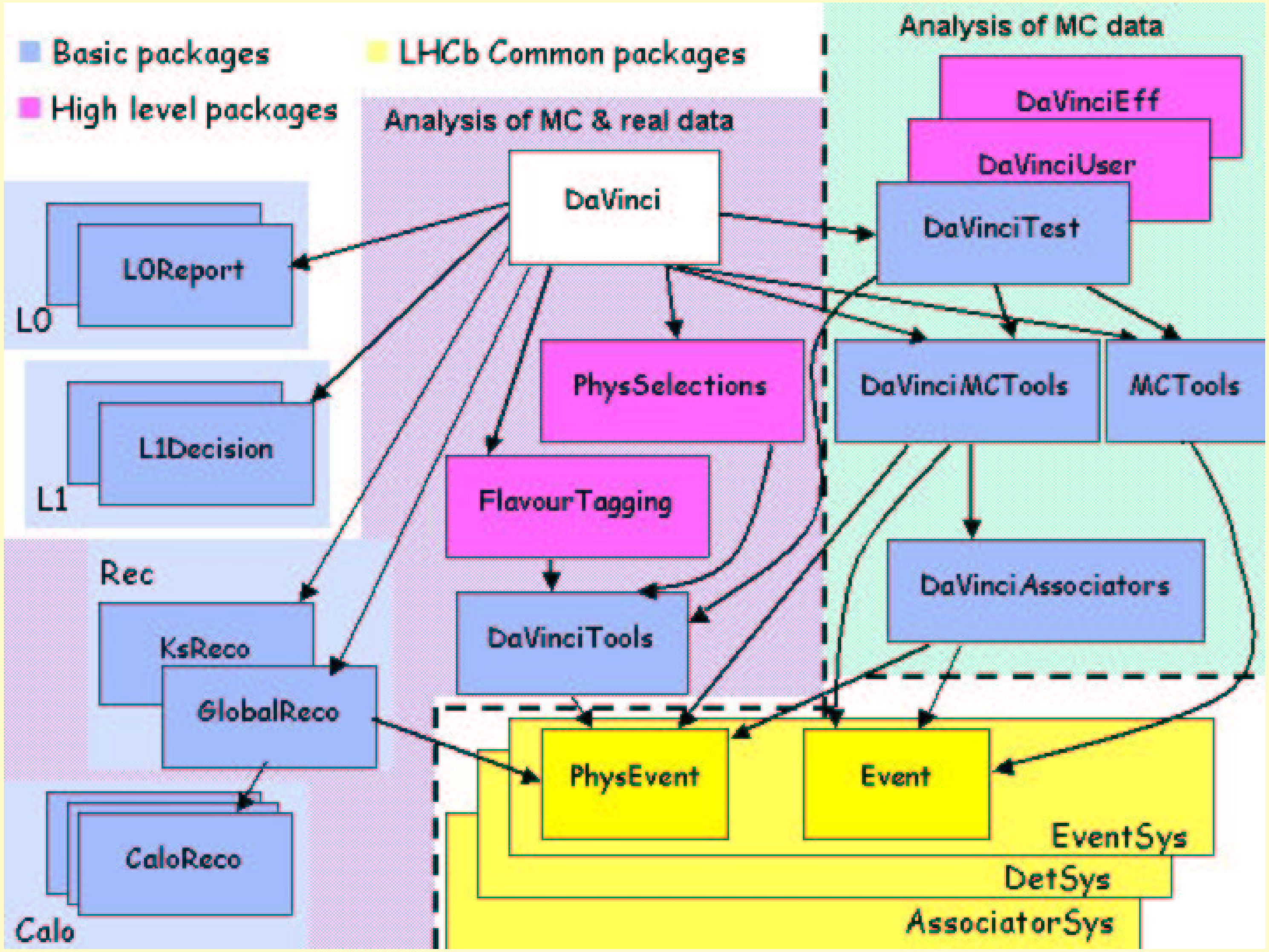
- Borrowed, to be able to redo things:

  `Calo/`, **`Muon/`**: Detector-specific PID packages (3)

  `L0/`, **`Trg/`**, **`Hlt/`**: Trigger (19)

  `Rec/`, **`Tr/`**: Reconstruction (4)

Structure (a bit old)



P. Koppenburg

Physics Packages (v12r3)

Basic components:

`Phys/DaVinci/`: Main

`Phys/DaVinciKernel/`: Base classes

`Phys/DaVinciFilter/`: Particle filters

`Phys/ParticleMaker/`: Particle makers

`Phys/VertexFit/`: Vertex fitters

`Phys/DaVinciTransporter/`: Transporters

`Phys/DaVinciTools/`: Anything else

`Tools/Utilities/`: Simple utilities

Physics analysis:

`Phys/PhysSelections/`: Generic selection algorithms

`Phys/Ks2PiPiSel/`: $K^0_S \to \pi\pi$

`Phys/CommonParticles/`: π^0

`Phys/FlavourTagging/`: Flavour tagging

`Tools/LoKi*/`: **`LoKi`**, see dedicated lesson

`Tools/Stripping/`: Stripping tools

MC-truth and test packages

`Phys/DaVinciMCTools/`: MC Tools

`Phys/DaVinciAssociators/`: Associators to MC truth

`Phys/DaVinciEff/`: Efficiency algorithms

`Phys/DaVinciTest/`: Tests

`Phys/DaVinciUser/`: Template user package

First try:

- Get it
- Compile it
- Run it
- Particles and ProtoParticles

This part is almost hands-on. Just follow the instructions on your user account after the lesson.

First try

- Set the version of **DaVinci** you want to use (always):

```
> DaVinciEnv v12r3
```

This sets the path where `cmt` will find all necessary packages.

```
> echo $CMTPATH
/afs/cern.ch/user/p/pkoppenb/cmtuser:/afs/cern.ch/lhcb/sof
ware/releases/DAVINCI/DAVINCI_v12r3:/afs/cern.ch/lhcb/soft
ware/releases/LHCB/LHCB_v16r3:/afs/cern.ch/lhcb/software/
releases/DBASE:/afs/cern.ch/lhcb/software/releases/PARAM:
/afs/cern.ch/sw/Gaudi/releases/GAUDI/GAUDI_v15r3:/afs/cern
ch/sw/lcg/app/releases/LCGCMT/LCGCMT_26_2d
```

First try

- Set the version of **DaVinci** you want to use (always):
```
> DaVinciEnv v12r3
```
- go to your working directory:
```
> cd $HOME/cmtuser
```

This sets the path where `cmt` will find all necessary packages.
```
> echo $CMTPATH
/afs/cern.ch/user/p/pkoppenb/cmtuser:/afs/cern.ch/lhcb/sof
ware/releases/DAVINCI/DAVINCI_v12r3:/afs/cern.ch/lhcb/soft
ware/releases/LHCB/LHCB_v16r3:/afs/cern.ch/lhcb/software/
releases/DBASE:/afs/cern.ch/lhcb/software/releases/PARAM:
/afs/cern.ch/sw/Gaudi/releases/GAUDI/GAUDI_v15r3:/afs/cerr
ch/sw/lcg/app/releases/LCGCMT/LCGCMT_26_2d
```

First try

- Set the version of **DaVinci** you want to use (always):
```
> DaVinciEnv v12r3
```
- go to your working directory:
```
> cd $HOME/cmtuser
```
- Get the **DaVinci** package (once):
```
> getpack Phys/DaVinci v12r3
```

The **DaVinci** “project” contains presently 75 packages. The `Phys/DaVinci` main package is just one of it.

First try

- Set the version of **DaVinci** you want to use (always):
```
> DaVinciEnv v12r3
```
- go to your working directory:
```
> cd $HOME/cmtuser
```
- Get the **DaVinci** package (once):
```
> getpack Phys/DaVinci v12r3
```
- Setup your environment (always):
```
> cd Phys/DaVinci/v12r3/cmt
> source setup.csh
```

This will set one environment variable for each of the packages needed
```
> echo $DAVINCIROOT
/afs/cern.ch/user/p/pkoppenb/cmtuser/Phys/DaVinci/v12r3/
```

First try

- Set the version of **DaVinci** you want to use (always):
```
> DaVinciEnv v12r3
```
- go to your working directory:
```
> cd $HOME/cmtuser
```
- Get the **DaVinci** package (once):
```
> getpack Phys/DaVinci v12r3
```
- Setup your environment (always):
```
> cd Phys/DaVinci/v12r3/cmt
> source setup.csh
```
- Make the executable (once):
```
> make
```

First try

- Set the version of **DaVinci** you want to use (always):
```
> DaVinciEnv v12r3
```
- go to your working directory:
```
> cd $HOME/cmtuser
```
- Get the **DaVinci** package (once):
```
> getpack Phys/DaVinci v12r3
```
- Setup your environment (always):
```
> cd Phys/DaVinci/v12r3/cmt
> source setup.csh
```
- Make the executable (once):
```
> make
```
- Execute DaVinci (whenever needed):
```
> DaVinci
```

Even simpler

- Set the version of **DaVinci** you want to use:
```
> DaVinciEnv v12r3
```
- Setup your environment:
```
> source $DaVinci_release_area/DAVINCI/
DAVINCI_v12r3/Phys/DaVinci/v12r3/cmt/setup.cs
```
- Execute DaVinci:
```
> DaVinci
```

Even simpler

- Set the version of **DaVinci** you want to use:
-
-

).cs

What did it do?

Actually not much

`DaVinci` is an alias for:

```
> which DaVinci
DaVinci:  aliased to /afs/cern.ch/user/p/pkoppenb/cmtuser/-
Phys/DaVinci/v12r3/rh73_gcc323/DaVinci.exe
```

When **DaVinci** is run with no options, it loads it's configuration from `../options/DaVinci.opts`

DaVinci.opts

`DaVinci.opts` is a dummy option file. Removing the irrelevant stuff there is:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opt
#include "$DAVINCIROOT/options/DaVinciReco.opts"
#include "$DAVINCIROOT/options/DaVinciTestData.o
ApplicationMgr.EvtMax = 1000;
```

- `DaVinciCommon.opts` is where all default settings and packages are defined. Don't touch!
- `DaVinciReco.opts` makes the `ProtoParticles` and the primary vertex.
- `DaVinciTestData.opts` provides some $\mathrm{B}\overline{\mathrm{B}}$ DST.

ProtoParticles?

`ProtoParticles`

- are the end of the reconstruction stage
- are the starting point of the physics analysis
- have all the links about how they have been reconstructed
  - Track?
  - Calo cluster?
- have a list of PID hypothesis with a probability
- contain the *kinematic* information

You need to assign them a mass and a PID to get the full 4-vector.

$\Rightarrow$ `Particles`

Particles?

- `Particle` = `ProtoParticle` + one PID *choice*
  $\longrightarrow$ one defined mass
- Physics analyses deal with `Particles`
  - You need to know the 4-vectors to compute the mass of a resonance
- The PID is your choice
  - The same `ProtoParticle` can be made as a π and as a $\mathbf{K}$ …
  - Some `ProtoParticles` can be ignored
  - All this is done by configuring the `ParticleMaker` (described later)

Select $\mathrm{B_s} \to \mathrm{J}/\psi\ \phi$:

- Design it
- Make particles
- Make J/ψ's
- Some histograms
- Add the ϕ

This part is based on the `Tutorial/Analysis` package. All can be found there.

Reminder: Algorithms

Algorithms are objects executed at each event.
The primary vertex for instance is made by an algorithm declared in `DaVinciReco.opts` by

```
ApplicationMgr.TopAlg += { "PrimVtxFinder" };
```

What **DaVinci** does is defined by the algorithms that are called. In **Gaudi**-jargon an algorithm is a class inheriting from `Algorithm`, which contains

- an `initialize()` method called at begin of run
- an `execute()` method called at each event.
- a `finalize()` method called at end of run

To make life easier **DaVinci** contains a base-class `DVAlgorithm` that provides many useful features.

Recent changes

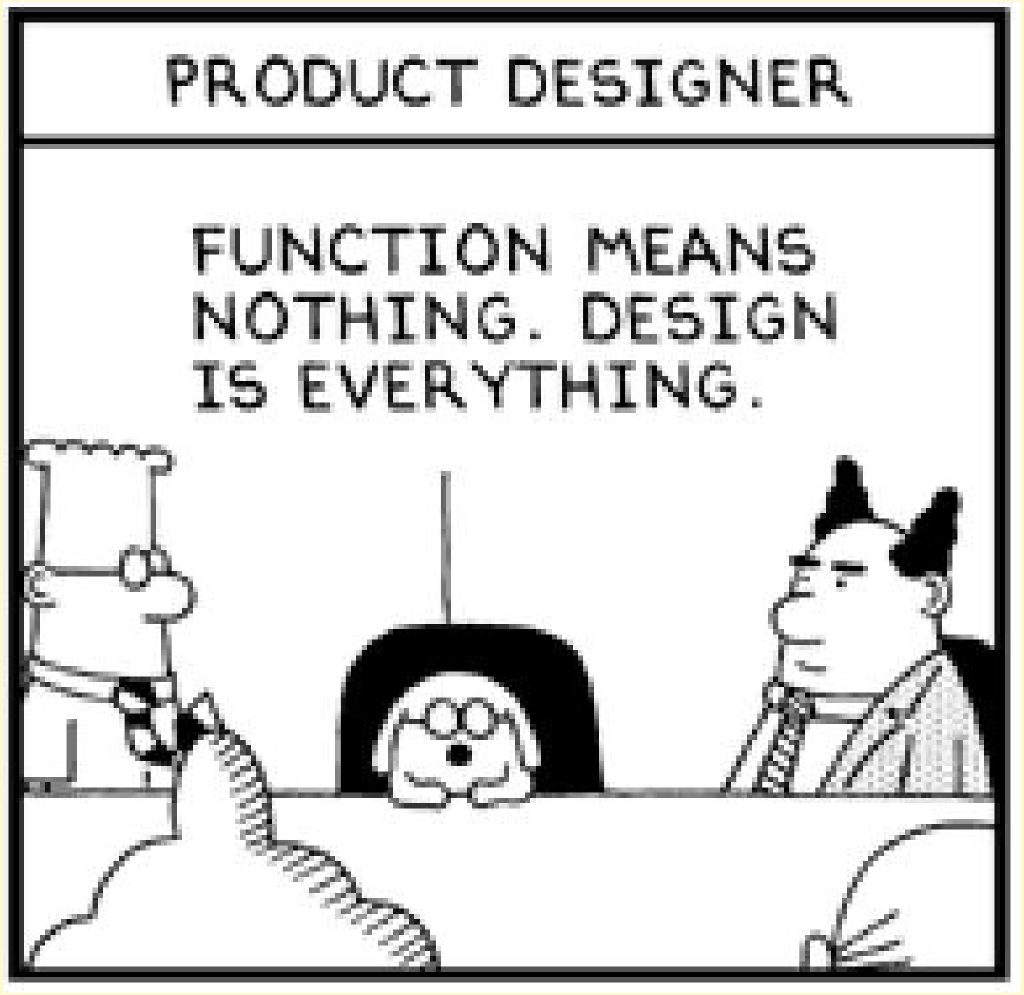
- `DVAlgorithm` now inherits from the new base-class `GaudiTupleAlg`,
- That inherits from `GaudiHistoAlg`,
- That inherits from `GaudiAlgorithm`

$\longrightarrow$ There are many new shortcuts available:

```
debug() << "Hello world" << endmsg ;
plot(twoMu.m(),"DiMu mass",2.*GeV,4.*GeV);
IDebugTool* m_debug =
   tool<IDebugTool>( "DebugTool" );
```

They succeed to much longer syntaxes that everyone had to use one year ago...

Design it

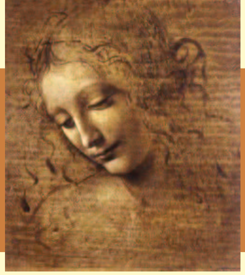


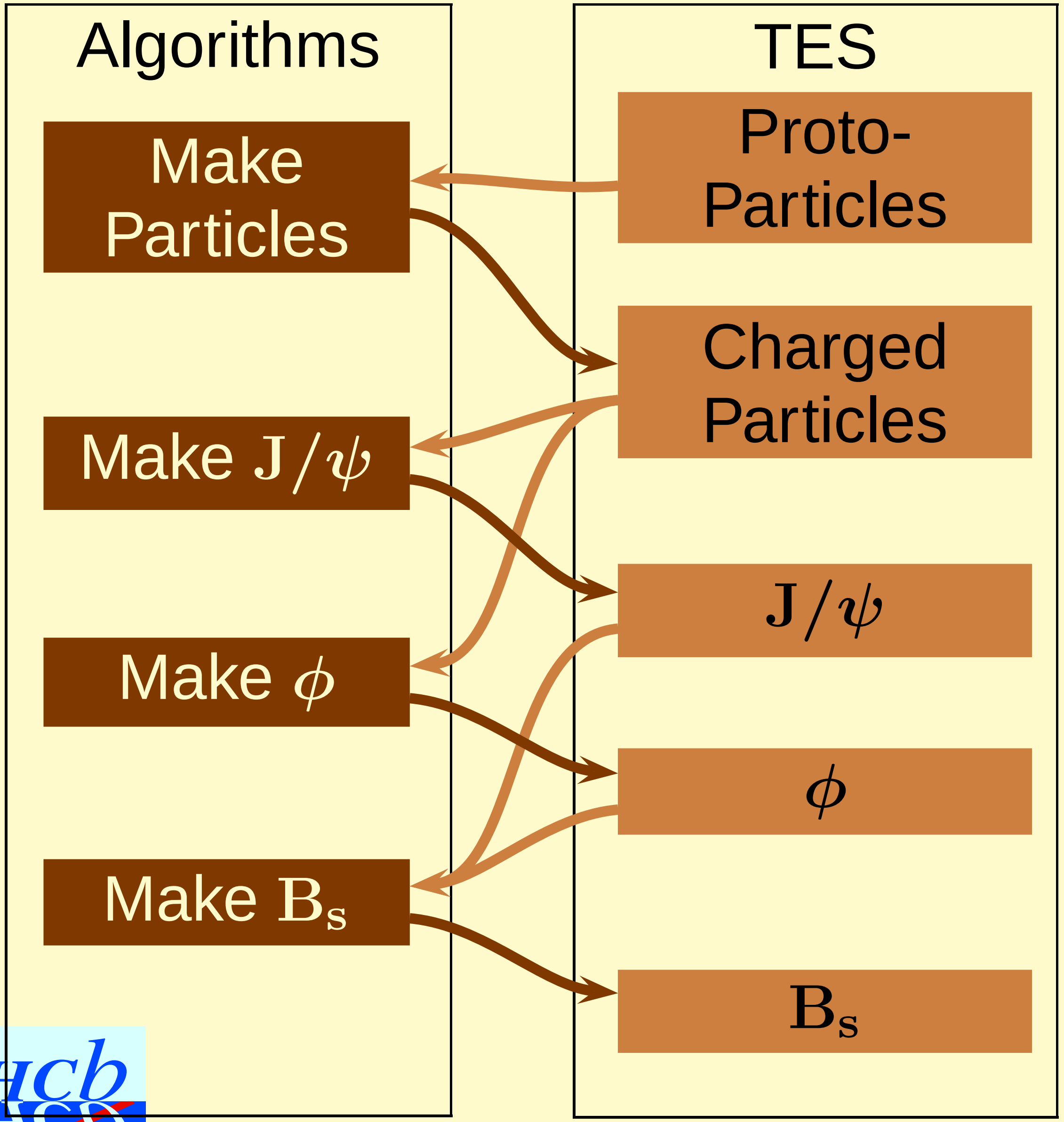
One could write a single algorithm that makes particles, combines μ into $\mathbf{J}/\psi$ and $\mathbf{K}$ into ϕ and then makes the $\mathrm{B_s}$.

This is not a good idea!

It is much better to write a simple algorithm for each task and to save the intermediate data in the transient event store (TES)

Design it



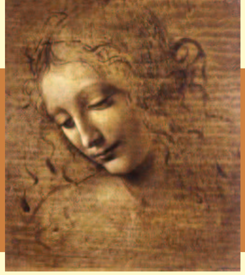


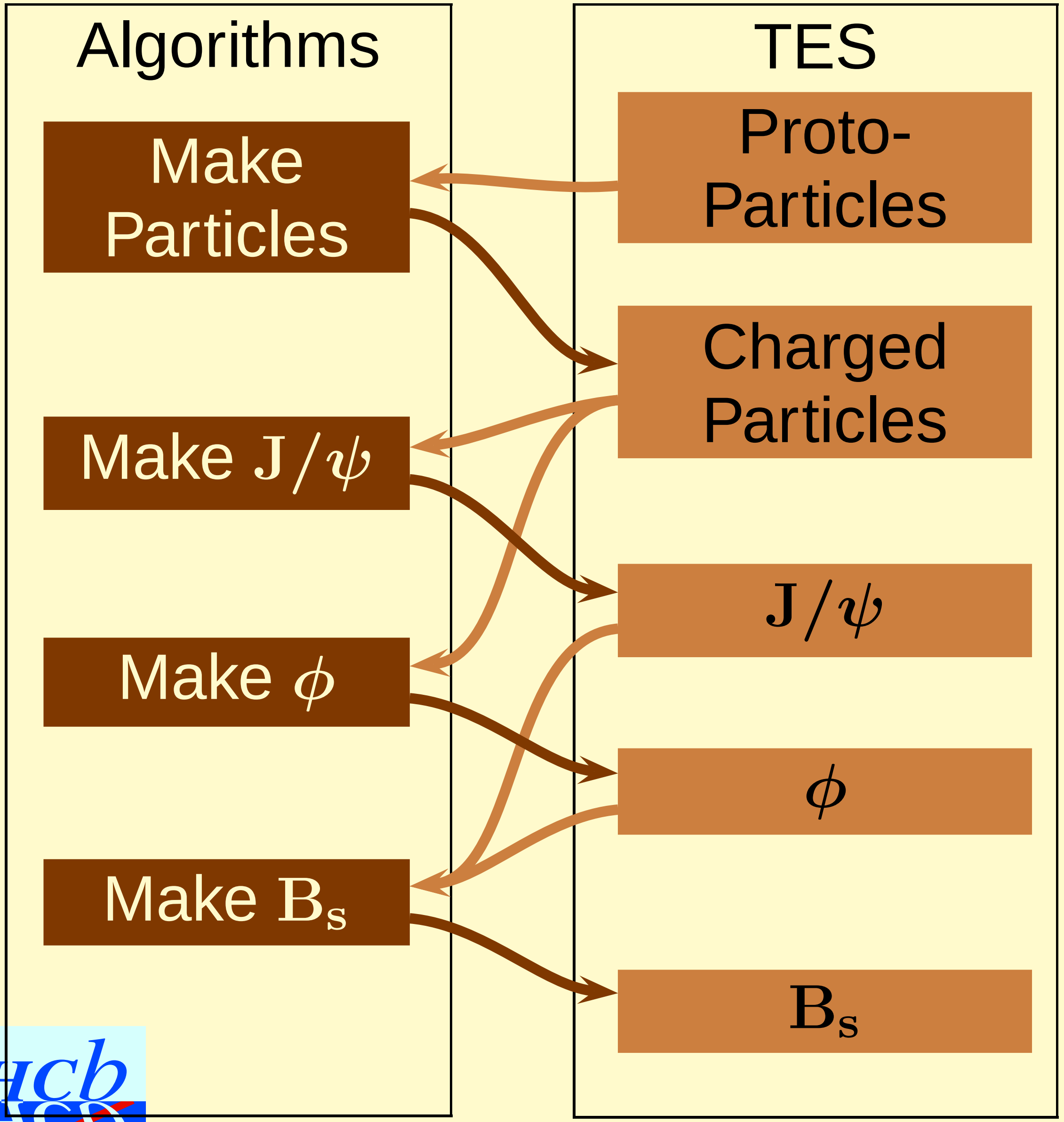
One could write a single algorithm that makes particles, combines μ into J/ψ and K into ϕ and then makes the B_s.

This is not a good idea!

It is much better to write a simple algorithm for each task and to save the intermediate data in the transient event store (TES)

Design it





- Algorithms have as many inputs as needed, but only one output
- TES locations can be read by any algorithm, but only one can write to them

Let’s start to write the chain!

Locations in the TES

The output of an algorithm called `"MyAlgo"` is saved in

- `/Event/Phys/MyAlgo/Particles` and
- `/Event/Phys/MyAlgo/Vertices`

Algorithm instance names have to be unique $\longrightarrow$ particles will be stored in different locations.

This becomes important if you want to test the correlation of your $\mathrm{B_s} \to \mathbf{J}/\psi\phi$ selection with the TDR selection of $\mathrm{B} \to \mathbf{J}/\psi\mathbf{K^0_S}$, or test the efficiency of the HLT $\mathbf{J}/\psi$ selection.

Make sure all algorithm names are unique!
It is mandatory for the stripping.

Get the Tutorial package

Get the latest version of the `Tutorial/Analysis` package.

```
  > cd $HOME/cmtuser/
  > getpack Tutorial/Analysis v4
  > cmt config
  > cmt br make
  > source setup.csh
  > echo $ANALYSISROOT
/afs/cern.ch/.../cmtuser/Tutorial/Analysis/v4
  > echo $DAVINCIROOT
/afs/cern.ch/.../cmtuser/Phys/DaVinci/v12r3
```

Or, if you don't have **DaVinci** in your area

```
/afs/cern.ch/lhcb/software/releases/DAVINCI/DAVINCI_v12r3/Phys/DaVinci/v12
```

Start to write the options

It's a good idea to start with the options. This gives the list of things to do:

```
cd $ANALYSISROOT
```
Open a file: `emacs options/DVTutorial.opts`

Start to write the options

It's a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
```

Input the common initialisation

Start to write the options

It’s a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
```

Don’t forget the DLL of the package you just added to **DaVinci**

Start to write the options

It’s a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
#include "$DAVINCIROOT/options/DaVinciReco.opts"
```

Include the reconstruction of `ProtoParticles` and primary vertices

Start to write the options

It's a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
#include "$DAVINCIROOT/options/DaVinciReco.opts"
ApplicationMgr.TopAlg += { "GaudiSequencer/Tutorial" };
```

Let's start the $\mathrm{B_s} \longrightarrow \mathrm{J}/\psi\phi$ sequence

Start to write the options

It’s a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
#include "$DAVINCIROOT/options/DaVinciReco.opts"
ApplicationMgr.TopAlg += { "GaudiSequencer/Tutorial" };
Tutorial.Members += { "PreLoadParticles" };
#include "$PARTICLEMAKERROOT/options/PreLoadParticles.opts"
```

Use the default algorithm to make particles.
We’ll have a closer look later on.

Start to write the options

It’s a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
#include "$DAVINCIROOT/options/DaVinciReco.opts"
ApplicationMgr.TopAlg += { "GaudiSequencer/Tutorial" };
Tutorial.Members += { "PreLoadParticles" };
#include "$PARTICLEMAKERROOT/options/PreLoadParticles.opts"
Tutorial.Members += { "TutorialAlgorithm" };
```

This one we’ll have to write. . .

Start to write the options

It’s a good idea to start with the options. This gives the list of things to do:

```
#include "$DAVINCIROOT/options/DaVinciCommon.opts"
ApplicationMgr.DLLs += { "Analysis" };
#include "$DAVINCIROOT/options/DaVinciReco.opts"
ApplicationMgr.TopAlg += { "GaudiSequencer/Tutorial" };
Tutorial.Members += { "PreLoadParticles" };
#include "$PARTICLEMAKERROOT/options/PreLoadParticles.opts"
Tutorial.Members += { "TutorialAlgorithm" };
EventSelector.Input = {
"DATAFILE='PFN:rfio:/castor/cern.ch/lhcb/DC04/00000543_00000017_5.ds
TYP='POOL_ROOTTREE' OPT='READ'"};
```

Add some data to read. You get it from the Bookkeeping.

Let’s write the algorithm

In `$ANALYSISROOT` type

```
> emacs src/TutorialAlgorithm.{cpp,h}
```

Emacs will ask you what you want to create. Answer (D) for `DVAlgorithm` (twice) and you will get a template for a new algorithm that compiles nicely but does nothing at all.

Before you forget it, add the following line to `src/Analysis_load.cpp`:

```
 DECLARE_ALGORITHM(TutorialAlgorithm)
```

Now go to `cmt/` and recompile the package.

A look at the header file

```
#include "DaVinciTools/DVAlgorithm.h"
class TutorialAlgorithm : public DVAlgorithm {
public:
  /// Standard constructor
  TutorialAlgorithm( const std::string& name, ISvcLocator* pSvcLocator );
  virtual ˜TutorialAlgorithm();       ///< Destructor
  virtual StatusCode initialize();    ///< Algorithm initialization
  virtual StatusCode execute   ();    ///< Algorithm execution
  virtual StatusCode finalize  ();    ///< Algorithm finalization
protected:
private:
};
```

- It inherits from `DVAlgorithm` , which provides the most frequently used tasks in a convenient way.
- The constructor allows to initialise global variables (mandatory!) and to declare options.
- The three methods `initialize()`, `execute()`, `finalize()` control your algorithm. Feel free to add more!

Edit the header file

Cuts should be defined by options, we hence need them to be data members of the algorithm. In `TutorialAlgorithm.h`:

```
private:
  double m_JPsiMassWin ; ///< Mass window
  double m_JPsiChi2 ;    ///< Max J/psi chi^2
```

We will also need the J/ψ PID, its mass and some statistics

```
  int m_JPsiID ;       ///< J/psi ID
  double m_JPsiMass ; ///< J/psi mass
  int m_nJPsis ;       ///< number of J/psis
  int m_nEvents ;      ///< number of Events
```

Constructor

All data members have to be initialised in the constructor

```
TutorialAlgorithm::TutorialAlgorithm(
   const std::string& name, ISvcLocator* pSvcLocator)
  : DVAlgorithm ( name , pSvcLocator )
    , m_JPsiID(0)
    , m_JPsiMass(0.)
    , m_nJPsis(0)
    , m_nEvents(0)
{
  declareProperty("MassWindow", m_JPsiMassWin = 10.*GeV);
  declareProperty("MaxChi2", m_JPsiChi2 = 1000.);
}
```

- Options have to be defined with `declareProperty`
- All others can be initialised to a dummy value
- You can just ignore the destructor

Initialisation

```
debug() << "==> Initialize" << endmsg;
ParticleProperty* m_psi = ppSvc()->find( "J/psi(1S)" );
m_JPsiID = m_psi->pdgID();
m_JPsiMass = m_psi->mass();
info() << "Will reconstruct " << m_psi->particle() << " (ID="
       << m_JPsiID << ") with mass " << m_JPsiMass << endreq ;
info() << "Mass window is " << m_JPsiMassWin << " MeV" << endreq ;
info() << "Max chiˆ2 is " << m_JPsiChi2 << endreq ;
```

- To initialise the J/ψ mass and PID you first need to find the particle properties of the J/ψ.
- `DVAlgorithm` provides a pointer to the Particle Property Service `ppSvc()`.
- The name of the J/ψ can be found in `$PARAMFILESROOT/data/ParticleTable.txt`.

Initialisation

```
debug() << "==> Initialize" << endmsg;
ParticleProperty* m_psi = ppSvc()->find( "J/psi(1S)" );
m_JPsiID = m_psi->pdgID();
m_JPsiMass = m_psi->mass();
info() << "Will reconstruct " << m_psi->particle() << " (ID="
       << m_JPsiID << ") with mass " << m_JPsiMass << endreq ;
info() << "Mass window is " << m_JPsiMassWin << " MeV" << endreq ;
info() << "Max chiˆ2 is " << m_JPsiChi2 << endreq ;
```

- From the `IParticlePropertySvc` class one can see in DoxyGen that there is a method

  `ParticleProperty * find (const std::string &name);`
- Then in `ParticleProperty` one locates:
  `double mass() const`
  `int pdgID() const`

DVAlgorithm base-class

A look at the DoxyGen web page shows that DVAlgorithm provides a lot of functionality (not all listed here):

```
IPhysDesktop* desktop() const;
IMassVertexFitter* massVertexFitter() const;
IVertexFitter* vertexFitter() const;
IGeomDispCalculator* geomDispCalculator() cons
IParticleFilter* particleFilter() const;
IParticlePropertySvc* ppSvc() const;
StatusCode setFilterPassed (bool);
std::string getDecayDescriptor();
```

We will use some of them.

Execute

1. Take the particles from the TES location where the particle maker algorithm has put them
2. Keep only the ones we need, i.e. muons
3. Combine them to $\mathbf{J}/\psi$'s and fit the vertex
4. Apply some cuts
5. Save the selected $\mathbf{J}/\psi$'s to the TES
6. We probably also would like to fill some histograms

For most of these tasks we have *Tools*.

Zoology of DaVinci tools

A Tool is a light weight object whose purpose is to help other components to perform their work.

- The particle filter and filter criteria are very useful tools: They allow to apply cuts steered by options.
- Vertexing tools: `UnconstVertexFitter`, `LagrangeMassVertexFitter`, `LagrangeGeomVertexFitter` . . .
- Geometrical tool
- Particle transporters
- Associators
- . . .

The PhysDesktop

The `PhysDesktop` is a tool that controls the loading and saving of the particles that are currently used.

- It collects previously maked particles
- It produces particles and saves them to the TES when needed

$\longrightarrow$ It hides the interaction with the TES

To get the particles and vertices, just do

- `const ParticleVector& parts = desktop()->particles();`
- `const VertexVector& parts = desktop()->vertices();`

Get the particles

```
// get particles. Filter muons.
const ParticleVector& parts = desktop()->particles();
ParticleVector MuPlus, MuMinus;
StatusCode sc = particleFilter()->filterNegative(parts,MuMinus);
if (sc) sc = particleFilter()->filterPositive(parts,MuPlus);
if (!sc) {
  err() << "Error while filtering" << endreq ;
  return sc ;
}
verbose() << "Filtered " << MuMinus.size() << " mu- and "
          << MuPlus.size() << " mu+" << endreq ;
```

- We get the particles from the `PhysDesktop` tool
- Then we fill them into `ParticleVector` of μ^- and μ^+ using the methods of the `ParticleFilter` (see DoxyGen)
- We'll ensure they are actually muons later on.

Combine the muons

```
// combine mu+ and mu-
ParticleVector::const_iterator imup, imum;
for ( imum = MuMinus.begin() ; imum !=  MuMinus.end() ; ++imum ){
  for ( imup = MuPlus.begin() ; imup !=  MuPlus.end() ; ++imup ){
    HepLorentzVector twoMu = (*imup)->momentum() + (*imum)->momentum();
    verbose() << "Two muon mass is " << twoMu.m()/MeV << endreq ;
    if ( fabs ( twoMu.m() - m_JPsiMass ) > m_JPsiMassWin ) continue ;
  }
}
```

- Have a look at the `Particle` class DoxyGen
- `ParticleVector` is a `typedef`
  `std::vector<Particle*>`

$\longrightarrow$ Hence the non-intuitive `(*imup)->momentum()` syntax

Vertex fit

Insert:

```
        // vertex fit
       Vertex MuMuVertex;
       sc = vertexFitter()->fitVertex(*(*imup),*(*imum),MuMuVertex);
       if (!sc){
         info() << "Failed to fit vertex" << endreq ; // no big deal
         continue ;
       }
       debug() << "Vertex fit at " << MuMuVertex.position()/cm
               << " with chi2 " << MuMuVertex.chi2() << endreq;
       // chi2 cut
       if ( MuMuVertex.chi2() > m_JPsiChi2 ) continue ;
```

- The `vertexFitter()` method returns a pointer to the unconstrained vertex fitter `UnconstVertexFitter`

Create the candidate

```
      Particle Jpsi ;
      sc = particleStuffer()->fillParticle(MuMuVertex,Jpsi,
           ParticleID(m_JPsiID));
      Particle* pJpsi = desktop()->createParticle(&Jpsi);
      info() << "Created J/psi candidate with m=" << Jpsi.mass()
             << " and chiˆ2=" << MuMuVertex.chi2() << endreq ;
      if (!pJpsi){
        err() << "Cannot save particle to desktop" << endreq ;
        return StatusCode::FAILURE;
      } else setFilterPassed(true);
```

- The `ParticleStuffer` tool makes particles from vertices. It is your job to provide the particle ID.
- Then save the new created particle to the `PhysDesktop`
- `setFilterPassed(true)` tells the algorithm that it has found what it is looking for.

Save the new particles

At the end put:

```
   sc = desktop()->saveDesktop();
   return sc;
```

This will save all *new* particles in the desktop.

The `PhysDesktop` has also methods to save a given list of particles

```
ParticleVector myPsis ;
sc = desktop()->saveTrees( myPsis );
sc = desktop()->saveTrees( m_JPsiID );
```

- All particles and vertices will be saved to `/Event/Phys/Jpsi2MuMu/Particles` and `/Event/Phys/Jpsi2MuMu/Vertices`

Particles and Vertices

The `Particle` and `Vertex` classes depend on each other

```
Vertex* Particle::endvertex() ;
SmartRefVector<Particle> & Vertex::products() ;
```

To navigate from a particle to its daughters do:

```
SmartRefVector<Particle> themus
  = Jpsi.endVertex()->products() ;
```

and use `themus` as any `std::vector` of pointers.

Note: There is no direct link between `Particles` .

Finalize

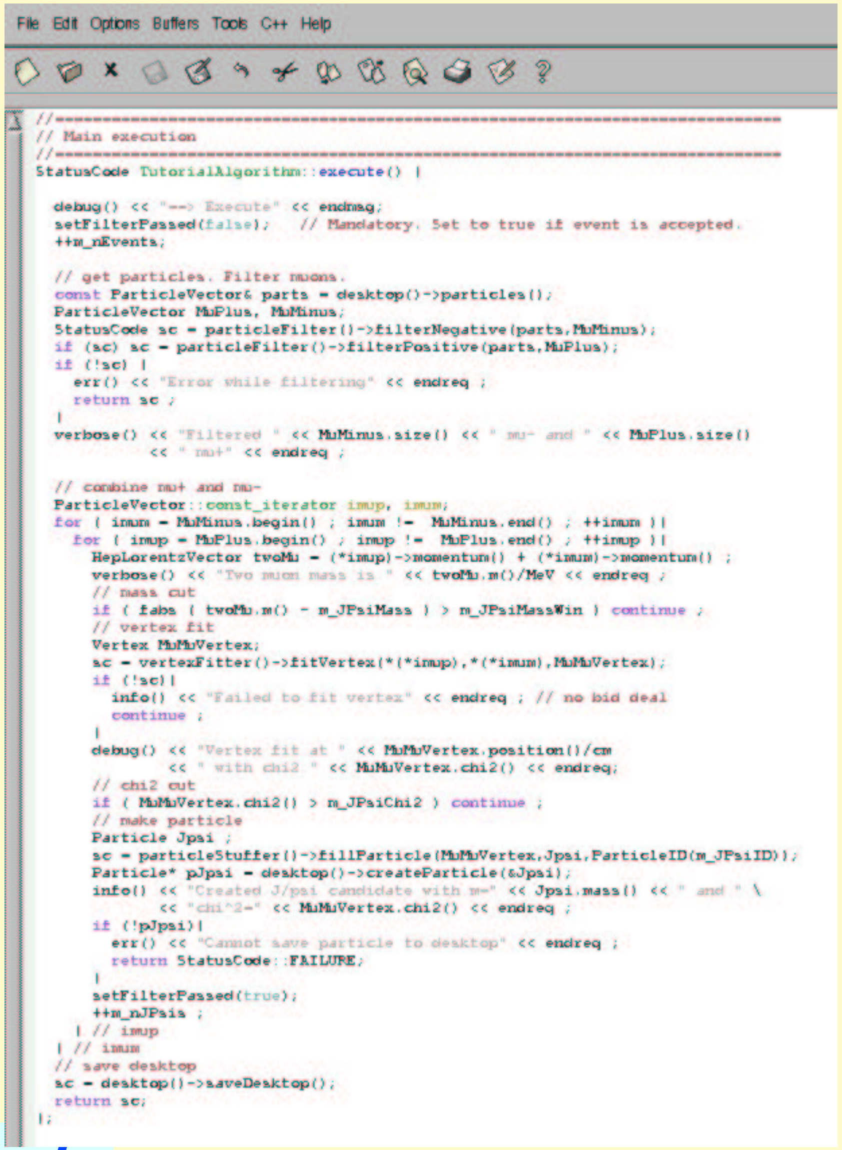
If you have incremented the counters `m_nEvents` and `m_nJpsis` you can print them at the end of the job:

```
StatusCode TutorialAlgorithm::finalize() {

  debug() << "==> Finalize" << endmsg;
  info() << "Found " << m_nJPsis << " J/psi in "
         << m_nEvents << " events" << endreq;
  return StatusCode::SUCCESS;
}
```

Note: Unlike in `GaudiAlgorithm`, don't `return GaudiAlgorithm::finalize() ;` or similar. This is done in the `sysFinalize()` method of `DVAlgorithm`.

End of C++ part



```
//==========================================================================
// Main execution
//==========================================================================
StatusCode TutorialAlgorithm::execute() {

  debug() << "==> Execute" << endmsg;
  setFilterPassed(false);   // Mandatory. Set to true if event is accepted.
  ++m_nEvents;

  // get particles. Filter muons.
  const ParticleVector& parts = desktop()->particles();
  ParticleVector MuPlus, MuMinus;
  StatusCode sc = particleFilter()->filterNegative(parts,MuMinus);
  if (sc) sc = particleFilter()->filterPositive(parts,MuPlus);
  if (!sc) {
    err() << "Error while filtering" << endreq ;
    return sc ;
  }
  verbose() << "Filtered " << MuMinus.size() << " mu- and " << MuPlus.size()
            << " mu+" << endreq ;

  // combine mu+ and mu-
  ParticleVector::const_iterator imup, imum;
  for ( imum = MuMinus.begin() ; imum !=  MuMinus.end() ; ++imum ){
    for ( imup = MuPlus.begin() ; imup !=  MuPlus.end() ; ++imup ){
      HepLorentzVector twoMu = (*imup)->momentum() + (*imum)->momentum() ;
      verbose() << "Two muon mass is " << twoMu.m()/MeV << endreq ;
      // mass cut
      if ( fabs ( twoMu.m() - m_JPsiMass ) > m_JPsiMassWin ) continue ;
      // vertex fit
      Vertex MuMuVertex;
      sc = vertexFitter()->fitVertex(*(*imup),*(*imum),MuMuVertex);
      if (!sc){
        info() << "Failed to fit vertex" << endreq ; // no bid deal
        continue ;
      }
      debug() << "Vertex fit at " << MuMuVertex.position()/cm
              << " with chi2 " << MuMuVertex.chi2() << endreq;
      // chi2 cut
      if ( MuMuVertex.chi2() > m_JPsiChi2 ) continue ;
      // make particle
      Particle Jpsi ;
      sc = particleStuffer()->fillParticle(MuMuVertex,Jpsi,ParticleID(m_JPsiID));
      Particle* pJpsi = desktop()->createParticle(&Jpsi);
      info() << "Created J/psi candidate with m=" << Jpsi.mass() << " and " \
             << "chi^2=" << MuMuVertex.chi2() << endreq ;
      if (!pJpsi){
        err() << "Cannot save particle to desktop" << endreq ;
        return StatusCode::FAILURE;
      }
      setFilterPassed(true);
      ++m_nJPsis ;
    } // imup
  } // imum
  // save desktop
  sc = desktop()->saveDesktop();
  return sc;
};
```

- We now have a complete algorithm.
- The `execute()` method still fits on a single page, but becomes a little longish to my taste
- If you'd like to split it in smaller methods, you're welcome. . .
- You can now compile it.
- The next step is to complete the options.

Options

```
Tutorial.Members += { "PreLoadParticles" };
[...]
Tutorial.Members += { "TutorialAlgorithm/Jpsi2MuMu" };
Jpsi2MuMu.PhysDesktop.InputLocations = { "Phys/PreLoadParticles" } ;
Jpsi2MuMu.MassWindow = 50*MeV ;
Jpsi2MuMu.MaxChi2 = 100 ;
Jpsi2MuMu.OutputLevel = 3 ;
```

- We already have the `PreLoadParticles` and `TutorialAlgorithm` algorithms in the `Tutorial` sequence: Let's call it `Jpsi2MuMu`.
- Configure the cuts and the verbosity level.
- Tell the `PhysDesktop` from where to take the particles.
- It automatically adds `"/Event/"` to the location if necessary.

Particle Filtering

Remember the particle filtering code:

```
ParticleVector MuPlus, MuMinus;
StatusCode sc = particleFilter()->filterNegative(parts,MuMinus);
if (sc) sc = particleFilter()->filterPositive(parts,MuPlus);
```

We want to make sure that only muons will be used:

```
Jpsi2MuMu.ParticleFilter.CriteriaNames = { "PIDFilterCriterion/Muons" } ;
Jpsi2MuMu.ParticleFilter.Muons.ParticleNames = {"mu+", "mu-"} ;
```

- The `ParticleFilter` tool accepts a list of filter criteria
- In this case we just want to filter according to PID

$\longrightarrow$ `PIDFilterCriterion`

- Simply tell it what particles you need

Particle Filtering

The `ParticleFilter` is a very powerful tool that accepts many filter criteria, all based on the same interface `IFilterCriterion`.

In the DoxyGen documentation you have the full list of criteria.

- `FlightDistanceFilterCriterion`
- `KinFilterCriterion`: P, P_T
- `LifetimeSignificanceFilterCriterion`
- `Mass(Difference)FilterCriterion`: m, Δm
- `Momentum2FlightAngleFilterCriterion`
- `PIDFilterCriterion`
- `PVIPFilterCriterion`: IP on primary vertices
- `TrackTypeFilterCriterion`
- `TrueMCFilterCriterion`: require tracks from a given decay
- `VtxFilterCriterion`: cut on the track's decay vertex
- `BooleanFilterCriterion`: allows to combine filter criteria

Run it

```
> DaVinci ../options/DVTutorial_1.opts | tee out
```

In file out we find what we did at initialization:

```
Jpsi2MuMu      INFO Will reconstruct J/psi(1S) (ID=443) with mass 3096.87
Jpsi2MuMu      INFO Mass window is 50 MeV
Jpsi2MuMu      INFO Max chiˆ2 is 100
```

In `execute()`:

```
Jpsi2MuMu      INFO Created J/psi candidate with m=3104.2 and chiˆ2=0.16634
Jpsi2MuMu      INFO Created J/psi candidate with m=3089.36 and chiˆ2=0.5617
```

In `finalize()`:

```
Jpsi2MuMu   SUCCESS Passed 176 times in 500 calls -> (35.2+/-2.13587)%, rej
Jpsi2MuMu      INFO Found 176 J/psi in 500 events
```

The first line above is printed by `DVAlgorithm` based on the number of times `execute()` issued a `setFilterPassed(true)` or `false`.

Let’s add histograms

Since `DVAlgorithm` inherits from `GaudiHistoAlg`, you can use the “on-demand” histogram booking service.

Add the following histogram at a convenient place:

```
   plot(twoMu.m(),"DiMu mass",2.*GeV,4.*GeV);
```

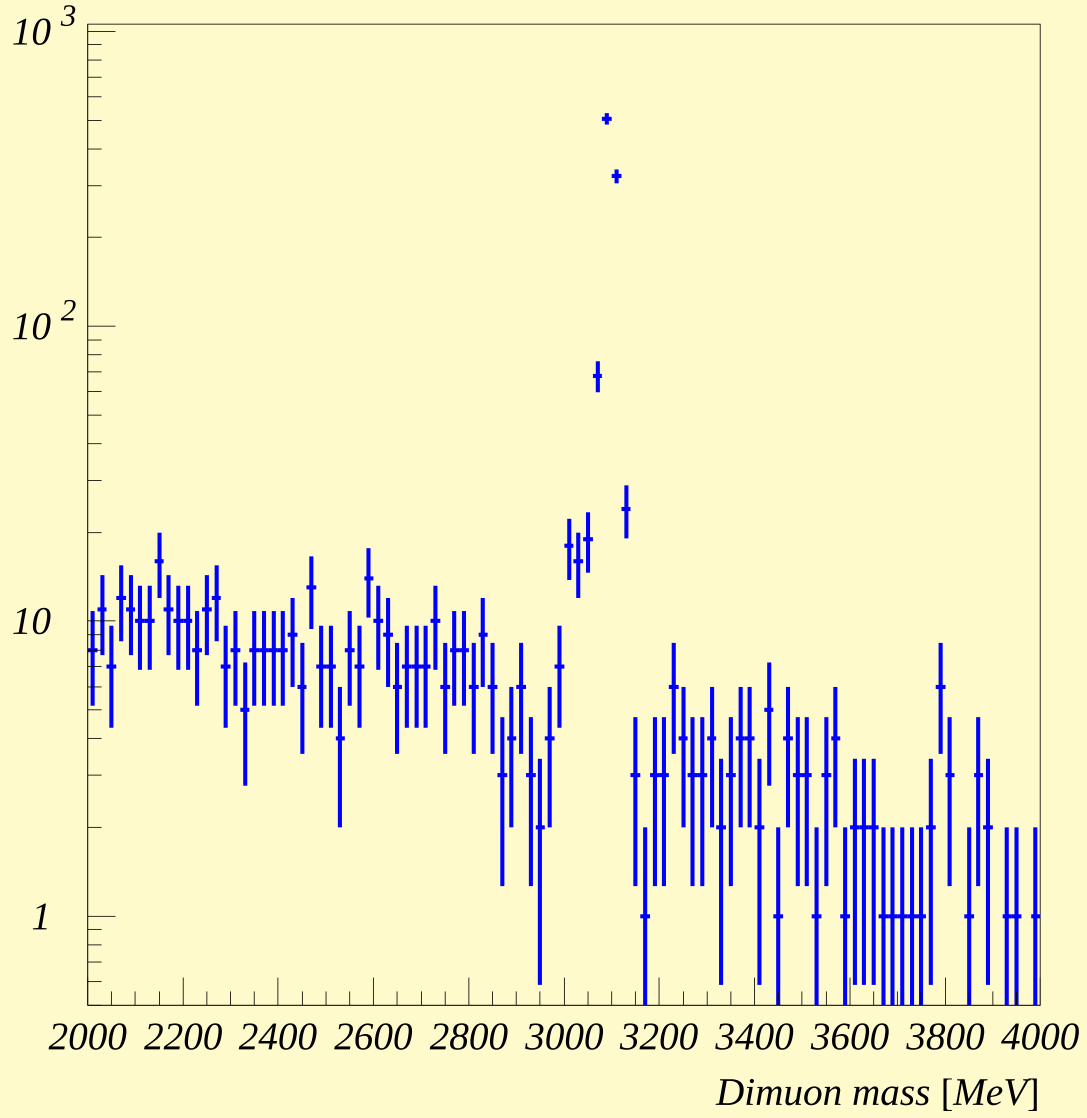
And add a persistency in the options:

```
ApplicationMgr.HistogramPersistency = "HBOOK";
HistogramPersistencySvc.OutputFile = "DVHistos.hbook";
Jpsi2MuMu.HistoProduce = true ; // default anyway
```

Feel free to use ROOT as persistency if you prefer. Hbook is probably going to dissappear someday. . .

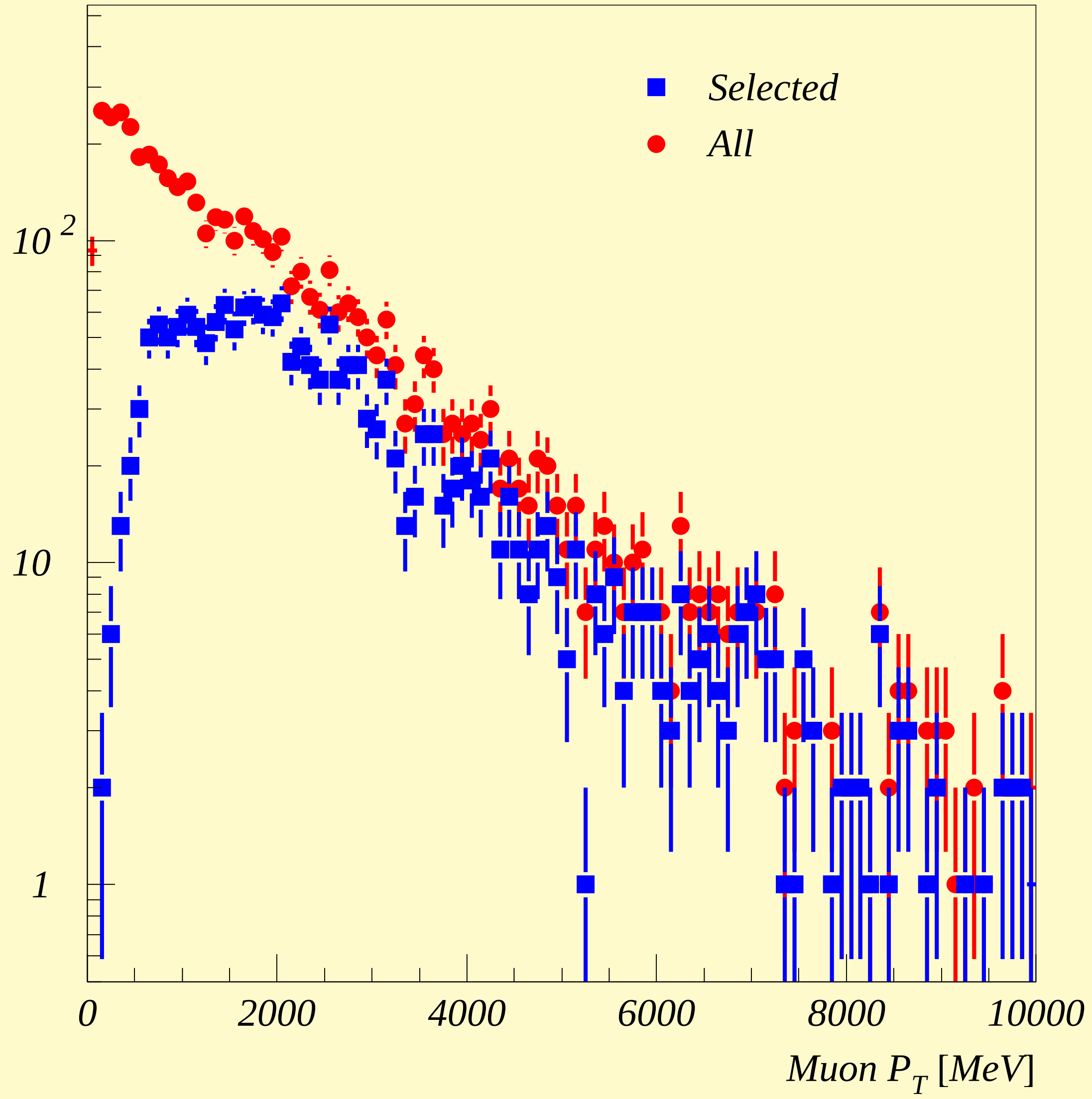
Histograms

- Here's the nice $\mathbf{J}/\psi$ peak you get



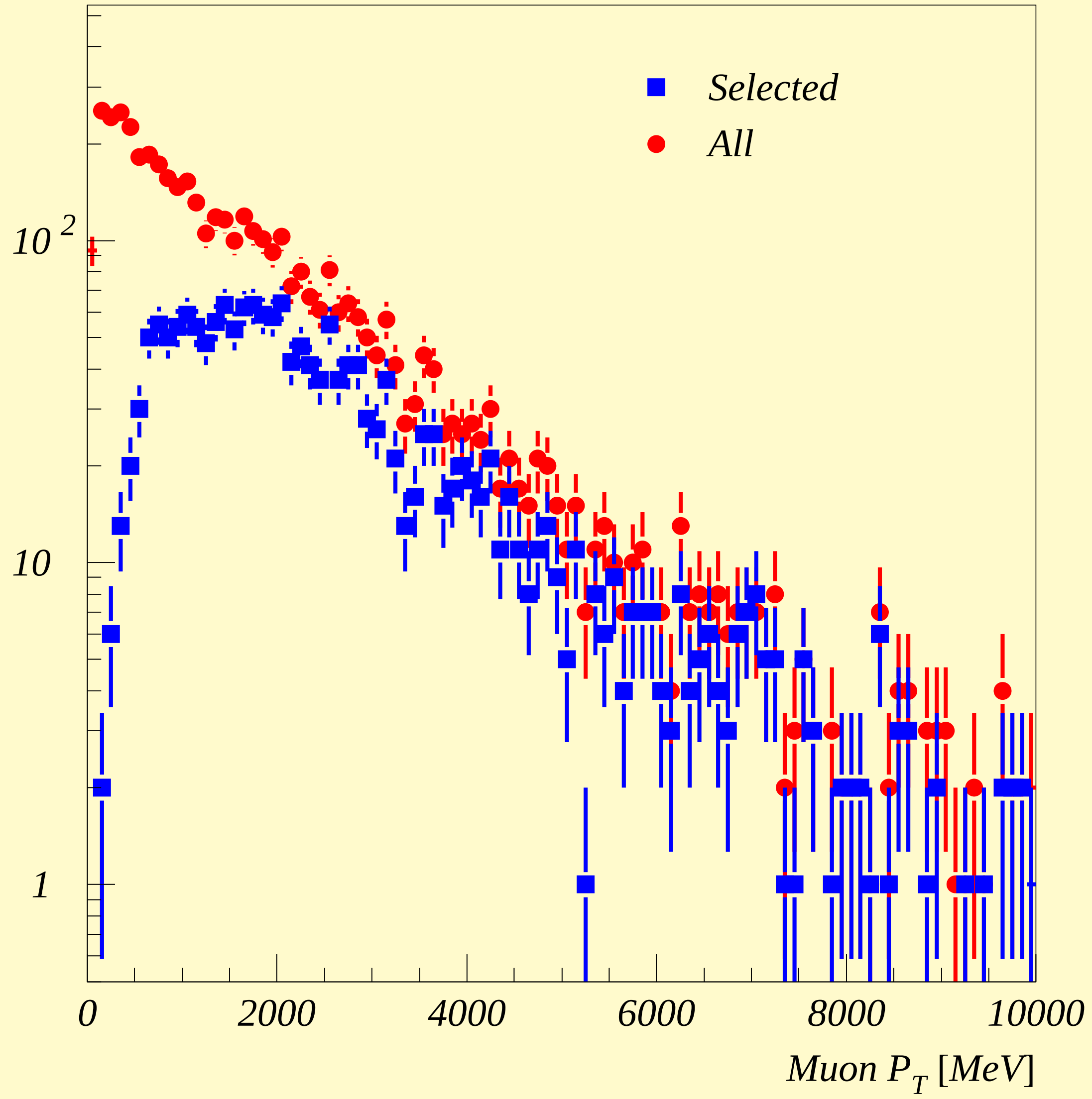
Histograms

- Here's the nice $\mathbf{J}/\psi$ peak you get
- Exercise 1: You could add two histograms of the μ's P_T, one before the $\mathbf{J}/\psi$ cuts and one after.



Histograms

- Here's the nice $\mathbf{J}/\psi$ peak you get
- Exercise 1: You could add two histograms of the μ's P_T, one before the $\mathbf{J}/\psi$ cuts and one after.
- Exercise 2: That could encourage you to add a P_T cut to your μ selection. You can do this by options only!



What we have learned so far

- To configure a simple **DaVinci** job
- To write a simple `DVAlgorithm`
- To get and save data using the `PhysDesktop`
- To use tools to perform the common tasks
- To navigate in DoxyGen to find the class definitions

One more exercise: Adapt the `TutorialAlgorithm` so that one can re-use this algorithm to also reconstruct $\phi \rightarrow \mathrm{KK}$:

```
Tutorial.Members += { "TutorialAlgorithm/Jpsi2MuMu" };
[...]
Tutorial.Members += { "TutorialAlgorithm/Phi2KK" };
Phi2KK.PhysDesktop.InputLocations = { "Phys/PreLoadParticles" } ;
Phi2KK.ParticleFilter.CriteriaNames = { "PIDFilterCriterion/Kaons" } ;
Phi2KK.ParticleFilter.Kaons.ParticleNames = {"K+", "K-"} ;
```

Use and configure standard algorithms:

- More about the `ParticleMaker`
- Make the ϕ using common tools
- `CombineParticles`
- `RefineSelection`
- Common particles
- The `SelResult` object

The ParticleMaker tools

The `IParticleMaker` interface (DoxyGen) is the base of several particle maker tools. They all make `Particles` starting from `ProtoParticles`

`CombinedParticleMaker`: makes particles from charged `ProtoParticles`

`NoPIDsParticleMaker`: make particles ignoring PID

`PhotonFromMergedParticleMaker`: makes γ from merged π^0

`(Cnv)PhotonParticleMaker`: make γ

`ParticleMakerSeq`: allow a sequence of particle makers

`MCParticleMaker`: makes particles from MC truth `MCParticles`

PreLoadParticles

A `ParticleMaker` can be declared to the `PhysDesktop`.

One could have defined a `ParticleMaker` to `Jpsi2MuMu`, but it's more transparent to use `PreLoadParticles`.

The options are:

```
Tutorial.Members += { "PreLoadParticles" };
PreLoadParticles.PhysDesktop.ParticleMakerType =
      "CombinedParticleMaker";
```

`PreLoadParticles` is a `DVAlgorithm` with one `ParticleMaker` defined that only saves the created particles.

The CombinedParticleMaker

The `CombinedParticleMaker` makes `Particles` from *charged* `ProtoParticles` *combining* the PID information of all detectors. It is documented from the **DaVinci** page.

The (main) options and default values are:

```
Particles = { "muon", "electron", "kaon", "proton", "pion" } ;
MuonSelection = "det='MUON' mu-pi='-8.0"' ;
ElectronSelection = "det='CALO' e-pi='0.0"' ;
KaonSelection = "det='RICH' k-pi='2.0' k-p='-2.0"' ;
ProtonSelection = "det='RICH' p-pi='3.0"' ;
PionSelection = "" ;
```

- Kaons for instance are made using the RICH with cuts:

$$\mathrm{DLL}(\mathrm{K}-\pi)=\ln L(\mathrm{K})-\ln L(\pi)=\ln\frac{L(\mathrm{K})}{L(\pi)}$$

P. Koppenburg

The CombinedParticleMaker

The CombinedParticleMaker makes Particles from *charged* ProtoParticles *combining* the PID information of all detectors. It is documented from the **DaVinci** page.

The (main) options and default values are:

```
Particles = { "muon", "electron", "kaon", "proton", "pion" } ;
MuonSelection = "det='MUON' mu-pi='-8.0"' ;
ElectronSelection = "det='CALO' e-pi='0.0"' ;
KaonSelection = "det='RICH' k-pi='2.0' k-p='-2.0"' ;
ProtonSelection = "det='RICH' p-pi='3.0"' ;
PionSelection = "" ;
ExclusiveSelection = true ;
```

- ExclusiveSelection means that only one Particle is made for each ProtoParticle , in the order of preference given in "Particles". This is a very dangerous option.

Back to our example options

We should have defined the cut on the muons in the particle maker rather than in the particle filter.

To make only muons and kaons:

```
Tutorial.Members += { "PreLoadParticles" };
PreLoadParticles.PhysDesktop.ParticleMakerType =
                     "CombinedParticleMaker";
PreLoadParticles.PhysDesktop.CombinedParticleMaker.Particles =
                    { "muon", "kaon" } ;
PreLoadParticles.PhysDesktop.CombinedParticleMaker.KaonSelection =
                    { "det='RICH' k-pi='2.0' k-p='-2.0'" };
PreLoadParticles.PhysDesktop.CombinedParticleMaker.MuonSelection =
                    { "det='MUON' mu-pi='-10.0'" }; // looser
PreLoadParticles.PhysDesktop.CombinedParticleMaker.ExclusiveSelection
                     = false ;
```

This is bad practice: Here `"PreLoadParticles"` has a potentially conflicting name.

Build the ϕ

To make the ϕ one can re-use the `TutorialAlgorithm` as in the suggested exercise

Or, one can use the generic `CombineParticles` algorithm.

- This algorithm reconstructs any (one-level) decay according to what is defined in the decay descriptor
- It requires one `FilterCriterion` per input or output particle.
- It's actually written using **LoKi**

You'd better learn to use this algorithm: it might become mandatory for the next stripping!

Build the ϕ

```
ApplicationMgr.DLLs   += { "PhysSelections", "LoKi" };
//
Tutorial.Members += { "CombineParticles/Phi2KK" };
Phi2KK.PhysDesktop.InputLocations = { "Phys/PreLoadParticles" } ;
Phi2KK.DecayDescriptor = "phi(1020) -> K+ K-";
Phi2KK.Selections = {"K+ : PVIPFilterCriterion",
                     "K- : PVIPFilterCriterion",
     "phi(1020) : BooleanFilterCriterion/PhiFilter"};
Phi2KK.PVIPFilterCriterion.MinIPsignif = 2 ;
Phi2KK.PhiFilter.AndList = { "MassFilterCriterion",
                             "VtxFilterCriterion" };
Phi2KK.PhiFilter.MassFilterCriterion.Window = 20*MeV ;
Phi2KK.PhiFilter.VtxFilterCriterion.MaxChi2 = 100 ;
```

- This selects ϕ in a mass window of $20\ \mathrm{MeV}$ and with a $\chi^2 > 100$,
- made from kaons with a $\mathrm{IP}/\sigma_{\mathrm{IP}} > 2$ on all reconstructed primary vertices.

Syntax of `CombineParticles`

`DecayDescriptor`: Mandatory.

- Only simple decay descriptors understood!
- Add `[...]cc` if you want both combinations.

`Selections`: vector of strings of the type

`"particle : Criterion/Name" ;`

- Use the `BooleanFilterCriterion` with no options when you don't want to filter anything
- All particles in the descriptor must be declared.
- Charge-conjugates are *never* implicit

```
CombineParticles.DecayDescriptor = "[rho(770)+ -> pi0 pi+]cc" ;
CombineParticles.Selections = { "rho(770)0 : MassFilterCriterion",
                                "pi+ : PVIPFilterCriterion",
                                "pi- : PVIPFilterCriterion", // !!!!
                                "pi0 : MassFilterCriterion" } ;
```

Build the $\mathrm{B_s}$

```
Tutorial.Members += { "CombineParticles/Bs2JpsiPhi" };
Bs2JpsiPhi.PhysDesktop.InputLocations = { "Phys/Phi2KK",
                                          "Phys/Jpsi2MuMu" } ;
Bs2JpsiPhi.DecayDescriptor = "B_s0 -> phi(1020) J/psi(1S)";
Bs2JpsiPhi.Selections = {"B_s0 : BooleanFilterCriterion/BFilter",
                         "J/psi(1S) : BooleanFilterCriterion",
                         "phi(1020) : BooleanFilterCriterion"};
Bs2JpsiPhi.BFilter.AndList = { "MassFilterCriterion"
                             , "VtxFilterCriterion"
                             , "PVIPFilterCriterion" };
Bs2JpsiPhi.BFilter.MassFilterCriterion.Window = 50*MeV ;
Bs2JpsiPhi.BFilter.VtxFilterCriterion.MaxChi2 = 100 ;
Bs2JpsiPhi.BFilter.PVIPFilterCriterion.MaxIPsignif = 5 ;
Bs2JpsiPhi.BFilter.PVIPFilterCriterion.CutBestPV = true ;
```

- This selects $\mathrm{B_s}$ in a mass window of $50\ \mathrm{MeV}$, a $\chi^2 > 100$, and $\mathrm{IP}/\sigma\mathrm{IP} < 5$ w.r.t the vertex it points to.

The end!

That’s the end of the selection!

We now have the full chain selecting $\mathrm{B_s} \to \mathrm{J}/\psi\phi$

We’ll come back to it later when we discuss MC truth and efficiencies.

RefineSelection

`RefineSelection` allows to filter particles from a given location in the TES.

Options:

`ParticleNames`: Vector of particle names.

- C.C. not implicit! (to be changed. . . ?)
- Non listed particles are not filtered, i.e. accepted!

`FilterNames`: Vector of `ParticleFilter` names.

- Note that these are `ParticleFilter` tools, not `FilterCriterion`!
- Giving a dummy filter allows to merge several TES locations to one (this is done in the stripping, but not very useful now that `CheckSelResult` exists).

- Accepted `Particles` are *cloned*

RefineSelection example

```
ApplicationMgr.DLLs   += { "PhysSelections" };
ApplicationMgr.TopAlg += { "RefineSelection" };
RefineSelection.PhysDesktop.InputLocation = { "Phys/PreLoadParticles" };
RefineSelection.ParticleNames = { "mu+", "mu-", "K+", "K-" }; // no c.c. !

RefineSelection.FilterNames = { "MuF", "MuF", "KF" , "KF" };

RefineSelection.MuF.CriteriaNames = { "KinFilterCriterion" } ;
RefineSelection.MuF.KinFilterCriterion.MinPt = 300 ;

RefineSelection.KF.CriteriaNames = { "KinFilterCriterion",
                                     "PVIPFilterCriterion" } ;
RefineSelection.KF.KinFilterCriterion.MinPt = 500 ;
RefineSelection.KF.PVIPFilterCriterion.MinIPsignif = 5.0 ;
```

This selects μ with $P_T > 300\ \mathrm{MeV}$ and K with $P_T > 500\ \mathrm{MeV}$ and $\mathrm{IP}/\sigma_{IP} > 5$.

If there are pions in `"Phys/PreLoadParticles"`, they will all pass!. . . But there's a solution.

Cut on daughters

One very nice feature of `RefineSelection` is that it allows to filter particles by cutting on its daughters:

```
HLTselBs2PhiPhi.Members += {"RefineSelection"} ;
RefineSelection.PhysDesktop.InputLocations = {"Phys/HLTPhi"}; // Phis
RefineSelection.ParticleNames = {"phi(1020)", "K+", "K-"};
RefineSelection.FilterNames = {"PhiF", "KF", "KF"};
RefineSelection.KF.CriteriaNames = {"KinFilterCriterion",
                                    "PVIPFilterCriterion"} ;


RefineSelection.KF.KinFilterCriterion.MinMomentum = 1000.; // hlt tuned
RefineSelection.KF.PVIPFilterCriterion.MinIPsignif = 1.; // hlt tuned


RefineSelection.PhiFilter.CriteriaNames = {"MassFilterCriterion"};
RefineSelection.PhiFilter.MassFilterCriterion.Window = 24*MeV; // hlt tune
```

There are actually no K in `"Phys/HLTPhi"`: The input are ϕ, the output are ϕ, but one cuts on the momentum of the$\mathrm{K}$.

CombineParticles versus RefineSel

Don’t get confused by the different syntax:

- `RefineSelection` : 1 `ParticleFilter` / particle
- `CombineParticles` : 1 `FilterCriterion` / particle

```
CombineParticles.Selections = { "phi(1020) : BooleanFilterCriterion/PhiF"
CombineParticles.PhiF.AndList = { "MassFilterCriterion",
                                       "VtxFilterCriterion" };
CombineParticles.PhiF.MassFilterCriterion.Window = 20*MeV ;
CombineParticles.PhiF.VtxFilterCriterion.MaxChi2 = 100 ;
```

But:

```
RefineSelection.Particles = { "phi(1020)" } ;
RefineSelection.FilterNames = { "PhiFilter" };
RefineSelection.PhiFilter.CriteriaNames = { "MassFilterCriterion",
                                            "VtxFilterCriterion"  } ;
RefineSelection.PhiFilter.MassFilterCriterion.Window = 20*MeV ;
RefineSelection.PhiFilter.VtxFilterCriterion.MaxChi2 = 100 ;
```

PIDFilter

`PIDFilter` selects (or rejects) particles of a given PID.

Options:

`ParticleNames`: Names of particles

`Reject = false`: Keep them or reject them?

```
ApplicationMgr.TopAlg += { "Sequencer/SeqPreselMuon" };
SeqPreselMuon.Members = {
  "PreLoadParticles/Combined",
  "PIDFilter/FilterMuon",
  "RefineSelection/PreselMuon" };

FilterMuon.PhysDesktop.InputLocations = { "Phys/Combined" } ;
FilterMuon.ParticleNames = { "mu+", "mu-" } ;
FilterMuon.Reject = false ; // default
```

`FilterMuon` just filters μ from the default `PreLoadParticles`, which is useful in the stripping.

Listing continued

```
PreselMuon.PhysDesktop.InputLocations = {"Phys/FilterMuon"};
PreselMuon.ParticleNames = { "mu+", "mu-" };
PreselMuon.FilterNames = { "MuFilter", "MuFilter" };

PreselMuon.MuFilter.CriteriaNames = { "KinFilterCriterion" } ;
PreselMuon.MuFilter.KinFilterCriterion.MinPt = 3000 * MeV ;  // from Hans
PreselMuon.MuFilter.KinFilterCriterion.MinMomentum  = 5000 * MeV ;  // fro

PreselMuon.MuFilter.CriteriaNames += { "TrackTypeFilterCriterion" } ;
PreselMuon.MuFilter.TrackTypeFilterCriterion.RequireLong  = true ; // does

PreselMuon.MuFilter.CriteriaNames += { "PVIPFilterCriterion" } ;
PreselMuon.MuFilter.PVIPFilterCriterion.MinIPsignif = 5.0 ;   // from Hans
```

This is the whole preselection for the “good muon” stream we have added to the stripping.
It starts from the standard particle maker, selects muons and applies some cuts: 0 line of C++!

Common particles

Some particles are already made for you, with options configured by the experts

π^0 are made by the package `Phys/CommonParticles`

```
ApplicationMgr.DLLs    += { "CommonParticles" };
ApplicationMgr.TopAlg += { "ResolvedPi0Alg" };
#include "$COMMONPARTICLESROOT/options/ResolvedPi0Alg.opts"
ApplicationMgr.TopAlg += { "MergedPi0Alg" };
#include "$COMMONPARTICLESROOT/options/MergedPi0Alg.opts"
```

$\mathrm{K^0_S}$ are made by the package `Phys/Ks2PiPiSel`

```
#include "$KS2PIPISELROOT/options/Ks2PiPiSel.opts"
```

For tight $\mathrm{K^0_S}$:

```
#include "$KS2PIPISELROOT/options/bestKs2PiPiSel.opts"
```

J/ψ can be found `PhysSel/Jpsi`

. . . More to come

`SelResult`

Each `DVAlgorithm` writes out a `SelResult` object containing

- the result of the `FilterPassed` output
- the decay descriptor
- the output location of the algorithm

All this is written to the TES in `SelResultLocation::Default`.

You can read the result of any algorithm from any algorithm or tool. You need:

```
#include "Event/SelResult.h"
```

The SelResult object

Some algorithms read out the SelResult object:

- CheckSelResult reads the SelResult of a given list of algorithms and allows to perform an and and or of these results. Useful if you want a sequencer to depend on an algorithm executed in another sequence.
- SelResultCorrelations prints a correlation table of efficiencies of various algorithms

```
  Algorithm              Eff.        1       2       3       4       5
-----------------------------------------------------------------------
1 AllBd2JpsiKsTracks   86.82% |  ******  98.26%  99.16%  86.82%  93.32%
2 HLTAllJpsis          87.47% |  98.99%  ****** 100.00%  87.47%  92.39%
3 HLTHighIPJpsi        82.63% |  94.37%  94.47%  ******  82.63%  88.18%
4 TDRselBd2Jpsi2MuMu  100.00% | 100.00% 100.00% 100.00%  ****** 100.00%
5 Bd2JpsiKsAndTDR      89.68% |  96.39%  94.73%  95.71%  89.68%  ******
```

Ready-to-use option files

Every option file beginning with `DV` is complete and can be used instead of `DaVinci.opts`. There are 141 available. Here are a few:

`$DAVINCIROOT/options/DVWriteMiniDst.opts`: writes a mini-DST

`$DAVINCIROOT/options/DVReadMiniDst.opts`: reads it back

`$DAVINCIROOT/options/DVTriggerFilter.opts`: writes out events that pass L0 and L1.

`PhysSel/*/*/options/DVTDRsel*.opts`: execute TDR selection

`PhysSel/*/*/options/DVPresel*.opts`: execute pre-selection

Some more Tools:

- Vertex Fitters
- The Geometrical Tool
- About the Primary Vertices
- Reminder about Tools

Vertex Fitters

`UnconstVertexFitter`: `IVertexFitter`

Performs an unconstrained vertex fit.

`LagrangeMassVertexFitter`: `IMassVertexFitter`

A kinematical constrained fit using Lagrange multipliers method with mass and geometrical constraint. If a particle has $\Gamma > 1\,\mathrm{MeV}$, its daughters are used in the fit.

`DVAlgorithm` interfaces them with `vertexFitter()` and `massVertexFitter()`:

```
Particle JPsi;
Vertex PsiVertex;
ParticleVector TheMus = ...;
StatusCode sc = vertexFitter()->fitVertex(TheMus, PsiVertex);
sc = massVertexFitter()->fitWithMass ("J/psi(1S)", TheMus,
                                       PsiVertex, JPsi) ;
```

There are also methods with 2–4 particles as input.

Geometrical Tool

- The `GeomDispCalculator` tool (`IGeomDispCalculator`) is interfaced by `geomDispCalculator()` in `DVAlgorithm`.

→ It allows to calculate distances between `Particles` and `Vertices`.

```
Particle Mu1, Mu2;
Vertex PV, JpsiVx;
double ip, dca, v2v, err;
StatusCode sc = geomDispCalculator()->calcImpactPar(Mu1, PV, ip, err);
sc = geomDispCalculator()->calcCloseAppr(Mu1, Mu2, dca, err);
sc = geomDispCalculator()->calcVertexDis( PV, JpsiVx, v2v, err ) ;
```

Primary vertex

To get the primary vertices:

```
Vertices* PV = get<Vertices>(VertexLocation::Primary));
for (iv=PV->begin();iv!=PV->end();++iv) {
  Vertex* v = *iv;
  double ip = -1 ,ipe = -1.;
  StatusCode sc = geomDispCalculator()->calcImpactPar(
      *part, *(*iv), ip, ipe);
}
```

Reminder about Tools

All this assumes that you use these tools from `DVAlgorithm` and that you need only one of each kind. If you use these tools from a simple `GaudiAlgorithm` or from a tool, or you need more than one, you will need to delare them yourself. This is very easy now:

```
#include "DaVinciTools/IGeomDispCalculator"
#include "DaVinciTools/IFilterCriterion"

IGeomDispCalculator* m_geom =
   tool<IGeomDispCalculator>("GeomDispCalculator");

std::string m_myFCname = "PVIPFilterCriterion" ;
IFilterCriterion* i_myFC =
   tool<IFilterCriterion>( m_myFCname, this );
```

Here you could pass `"PVIPFilterCriterion"` as an option.

Practical example

If you need several `ParticleFilter` tools in a `DVAlgorithm` , you need to declare some yourself

```
  declareProperty( "ParticleFilter1",
              m_MuFilterName = "MuFilter" );
  declareProperty( "ParticleFilter2",
              m_JpsiFilterName = "JPsiFilter" );

  IParticleFilter* m_MuFilter = tool<IParticleFilter>
                     ("ParticleFilter", m_MuFilterName, this);
  IParticleFilter* m_JpsiFilter = tool<IParticleFilter>
                     ("ParticleFilter", m_JPsiFilterName, this);
```

The options:

```
MyAlg.ParticleFilter1 = { "JPsiFilter" };
MyAlg.ParticleFilter2 = { "MuFilter" };
MyAlg.MuFilter.CriteriaNames = { "KinFilterCriterion" };
MyAlg.JPsiFilter.CriteriaNames = { "MassFilterCriterion" };
```

Tools

- Have a look at the new **Gaudi** basics tutorial about writing tools
- Very often a light-weight tool is the simple solution to a complicated problem.
- Please use and write `FilterCriterion` tools
- And let me know when you have a new one to be released in **DaVinci**.

MC truth:

- Efficiency algorithms
- DebugTool
- Decay Finder

- All this is based on the `DaVinciAssociators` $\longrightarrow$ see Philippe's talk

Efficiency algorithms

DaVinci contains two algorithms that allow to calculate selection efficiencies

`MCEffBuilder`: efficiency

`EffSelCheck`: selection efficiencies

As we will not be using these algorithms on background, it’s recommended to put the options in a separate file, to be put after the selection options.

```
#include "$ANALYSISROOT/options/Efficiency.opts"
```

Reconstruction efficiency

In `$ANALYSISROOT/options/Efficiency.opts`, write

```
ApplicationMgr.TopAlg += { "MCEffBuilder/EffMcTruth" };
EffMcTruth.MCDecay =  "[B_s0 -> (phi(1020) -> ^K+ ^K-)
                       (J/psi(1S) -> ^mu+ ^mu- {, gamma})]cc";
```

- It should not be in the `Tutorial` sequencer (or the efficiencies would all be 1 by construction)
- `MCEffBuilder` needs to know the decay descriptor of the decay.
- Decay descriptors are described on the web. Particles with a “ˆ” are the ones to be reconstructed.
- But it’s easier to steal them from the **EvtGen** decay file in `$LHCBRELEASES/DBASE/Gen/DecFiles/v6r3/dkfiles`

Reconstruction efficiency

```
 *************************************************************
 ***************  Output from MCEffBuilder  ***************
 *************************************************************
Decay analyzed (MC truth) [B_s0 -> (phi(1020) -> ^K+ ^K-) (J/psi(1S) -> ^m
Events processed                                          500
Decay Of Interest Generated          ( / Events )         497     0.994
DoIs Gen, Reconstructible (ALL)    ( / Generated )        103     0.207243
DoIs Gen, Reconstructed (ALL)      ( / Generated )        109     0.219316
---------------------------------------------
DoIs Gen, Rec'ble & Rec'ted (ALL)                          92
Rec. efficiency: (Rec'tible & Rec'ted)/Rec'tible (ALL):  0.893204 +- 0.030
```

- A long track is "reconstructible" if it has 3r, 3ϕ in the Velo, and 1x, 1 stereo clusters in each of the seeding stations.
- A track can be reconstructed although it is not reconstructible
- The full definitions are here

Selection efficiency

In "`$ANALYSISROOT/options/Efficiency.opts`", write

```
ApplicationMgr.TopAlg += { "EffSelCheck/EffBs2JpsiPhi" };
EffBs2JpsiPhi.Histograms = true;
EffBs2JpsiPhi.MCDecay = "[B_s0 -> (phi(1020) -> ^K+ ^K-)
                    (J/psi(1S) -> ^mu+ ^mu- {, gamma})]cc";
EffBs2JpsiPhi.SelDecay = "[B_s0 -> (phi(1020) -> K+ K-)
                         (J/psi(1S) -> mu+ mu-)]cc";
```

- The MC decay descriptor is the same as before.
- The selection decay is what we actually reconstruct. There are no "ˆ" needed.
- `EffSelCheck` produces a histograms of m, P, P_T, z, r, z_{PV}, r_{PV}, decay distance and flight time for all initial and intermediate particles and for MC truth, selected and associated.

Selection efficiency

```
*******************    Sub-tree head ... B_s0    *******************
Mass window for this sub-tree head ... 5.3696 +- 0.05 (GeV/c2)
**************************************************************************

DoIs Selected                    ( / Reconstructed )       44     0.40367
DoIs Selected, in Mass Window                              44     0.40367
--------------------------------------------------------------------------
DoIs Sel, Associated (Comp.OR.Chi2)         ( / Selected ) 44     1
DoIs Sel, Assoc (Comp.OR.Chi2), in Mass Window             44     1
--------------------------------------------------------------------------
Efficiency: (Sel and Assoc(.OR.))/Reconstructed           0.40367 +- 0.046
Efficiency: (Sel and Assoc(.OR.) and Mass)/Reconstructed  0.40367 +- 0.046
Purity:    (Selected and Associated(.OR.))/Selected       1 +- 0
```

- Looking at the $\mathrm{B_s}$, we have 44 selected
- all being associated to truth
- There are similar tables for J/ψ

The DebugTool

- The debug tool provides a human-readable dump of the event
- It works both with MC truth and with reconstructed particles
- It looks like this:

```
<---------------------------------------- Particle --------------
       Name         E         M         P        Px        Py        Pz
                  GeV       GeV       GeV       GeV       GeV       GeV
B_s0          255.062     8.686   254.915   -20.824    -0.062   254.063
+-->J/psi(1S) 202.675     3.127   202.651   -19.344    -1.318   201.721
|+-->mu+       91.705     0.106    91.705    -7.480    -1.478    91.388
|+-->mu-      110.970     0.106   110.970   -11.865     0.160   110.334
+-->phi(1020)  52.387     1.030    52.377    -1.479     1.256    52.341
 +-->K-        21.810     0.494    21.804    -0.498     0.523    21.792
 +-->K+        30.577     0.494    30.573    -0.981     0.733    30.549
```

Debug algorithms

There are provided algorithms that call the debug tool:

DumpEvent: No options. Dumps the whole MC event.

PrintTree: Prints the reconstructed tree

```
Tutorial.Members += { "PrintTree/PrintFoundBs" };
PrintFoundBs.DebugTool.Informations = "Name E M P Px Py Pz Pt phi Vz"
PrintFoundBs.PhysDesktop.InputLocations = { "Phys/Bs2JpsiPhi" } ;
PrintFoundBs.OutputLevel = 3 ;
```

PrintMCTree: Prints the MC decay tree of particles of a given ID

```
Tutorial.Members += { "PrintMCTree/PrintTrueBs" };
PrintTrueBs.DebugTool.Informations = "Name E M P Px Py Pz Pt phi Vz" ;
PrintTrueB.ParticleNames = {  "B_s0", "B_s~0" } ;
PrintTrueBs.OutputLevel = 3 ;
PrintTrueB.Depth = 2;  // down to the K and mu
```

Using the debug tool

The `DebugTool` can be used directly from an algorithm, for instance to print only when something goes wrong. It is not already present in `DVAlgorithm`.

- Declare it:

```
#include "DaVinciMCTools/IDebugTool.h"
```

- Use it:

```
IDebugTool* m_debug =  tool<IDebugTool>( "DebugTool" );
m_debug->printTree(part [, depth]);
m_debug->printAncestor(mcpart);
```

- Configure it:

```
Jpsi2MuMu.DebugTool.Informations = "Name E M P Px Py Pz Pt phi Vz" ;
Jpsi2MuMu.DebugTool.PrintDepth = 3 ;
```

- There are other methods and options. Have a look at DoxyGen.

Decay Finder

- The decay finder allows to find any decay in the event
- It works both on MC and reconstructed particles
- It uses a decay descriptor string. Look at the DOC.

Practical example:

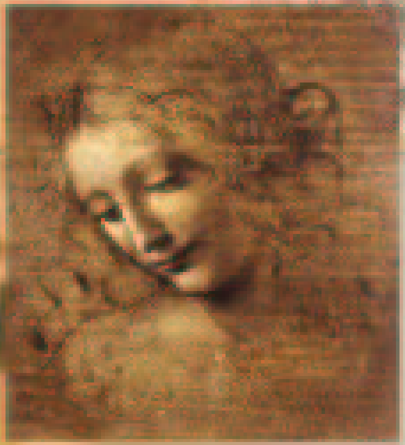
```
#include "DaVinciMCTools/I(MC)DecayFinder.h"
I(MC)DecayFinder* m_finder = tool<I(MC)DecayFinder>("(MC)DecayFinder") ;

const (MC)Particle *result = NULL;
while ( m_finder->findDecay( (mc)parts.result() ){
// the decay has been found
   m_debug->printTree( result ) ;
}
```

Or just test if a decay is here:

```
   bool found = m_debug->hasDecay( (mc)parts ) ;
```

Conclusion



- During the last year **DaVinci** evolved from a framework for writing selection code in C++ to a set of algorithms and tools that allow to perform many tasks with very little private code.
- If you feel something is missing. Please write something generic and add it to **DaVinci**!
- The evolution of **DaVinci** is now driven by the HLT.
  - Encourages the development of generic code
  - Forces common components to handle both on- and offline particles
  - Sets up a framework that can also be used for the stripping