Table of Contents

Towards a Common Ntuple (and microDST) for J/psi phi

The goal of this page is to produce a common set of python options that writes an ntuple containing all necessary variables for a full J/psi phi analysis. In order of priority, the aims are to:

- Include every variable to be used for cut optimisation, tagging studies, and the final fit
- Ensure the variables are consistent with LoKi Hybrid filters so that the variables we cut on are the variables in the ntuple
- Maintain a consistent and sensible naming convention for all variables across the group
- Be able to run anywhere (Grid, ECDF, Condor, Local)
- Serve as a beginner's guide to running python options and the latest LHCb software

Eventually, we should have a single common ntuple usable by all the group that won't need to be re-run privately except in special circumstances.

In addition, the python options now write out a microDST. The advantage of this is that if we ever forget to put anything into the nTuple, it should be possible to create a new nTuple with the additional quantities by just running over the microDST. Actually, we should be able to do away with nTuples altogether by just performing all processing within GaudiPython (using pyROOT for all of the ROOT/RooFit stuff). The microDST currently contains:

- Rec Header
- All LHCb::MCParticles that come from [B_s0 -> (J/psi(1S) -> mu+ mu- {,gamma} {,gamma}) (phi(1020) -> K+ K-)]cc
- All selected LHCb::Particles
- All LHCb::Particle to PV relations
- All LHCb::MCParticles that are related to the LHCb::Particles

Location of the code

The latest version of the code can be found in CVS:

```
$ cvs co Analysis/Bs2Jpsiphi/options
```

Configuration and setup

In order to run the options file, you'll need at least DaVinci v20r0:

```
$ setupLHCb
$ setenvDaVinci v20r0
$ getpack Phys/DaVinci v20r0
$ getpack MicroDST/MicroDSTAlgorithm v3r3p1
$ cvs -d :kserver:isscvs.cern.ch:/local/reps/lhcb co -d Phys/DaVinci/v20r2 -r jpalac_20080925 Phy
```

This sets the LHCb software environment, supplements it with the environment necessary to run DaVinci v20r0, and gets the CMT package for DaVinci and the MicroDST (along with a patch that is required to run the MicroDST code).

```
$ cp -r ~s0127440/public/EventCount Phys/
$ cp -r ~s0127440/public/OriginFinder Phys/
```

This copies the EventCount and OriginFinder algorithms cmt package into your current DaVinci cmt area. The EventCount algo is used to work out how many events you processed and how many of those passed your preselection when running distributed jobs. The OriginFinder algorithm prints the true MC association of each final state particle in order to determine peaking backgrounds.

```
$ cd Phys/EventCount/v1r0/cmt/
$ cmt setup
$ cmt br gmake clean
$ cmt br gmake
```

This configures the EventCount package, cleans out any cruft from previous compilations and compiles it for the current architecture. Do the same for OriginFinder if you intend to use it.

Add the following lines to the DaVinci requirements file:

```
use EventCounter        v*      Phys
use OriginFinder        v*      Phys
use MicroDSTAlgorithm   v3r3p1  MicroDST
```

and run through the usual CMT configuration steps.

Lastly, copy the two python options files attached to a directory of your choice. These tell DaVinci and Ganga what exactly they should be doing, a step-by step explanation follows.

_Presel_Options.py

This is the steering file for DaVinci, telling it what particles to look for and make, what cuts to apply to them once made, and what variables we want to save to our ntuple:

```
#Bs2JpsPhi loose preselection python options
# Conor Fitzpatrick, Sept. 2008

from os import environ
import GaudiKernel.SystemOfUnits as Units
from Gaudi.Configuration import *

#import necessary configurables
from Configurables import PhysDesktop, DecayTreeTuple, CombineParticles, EventSelector, EventCoun
```

The above loads all the necessary functions, constant definitions, etc from other sources. For those familiar with C/C++ this can be considered as the header information.

```
#LOAD COMMON OPTS (not yet converted to python)
importOptions( "$DAVINCIROOT/options/DaVinciCommon.opts" )
importOptions( "$DAVINCIROOT/options/DaVinciMainSeqFixes.opts" )
importOptions( "$HLTSYSROOT/options/L0.opts" )
importOptions( "$HLTSYSROOT/options/Hlt.opts" )
importOptions( "$FLAVOURTAGGINGOPTS/BTagging.opts"
```

These are old (deprecated) .opts files that are in the process of being converted to python options. Most of them should be converted in v20r1 of DaVinci. Until then, python can parse these options files using the importOptions() command. These load the tagging tools, the HLT/L0 tools and the common DaVinci options for all jobs.

```
#GLOBAL OUTPUT LEVEL
outputLevel = INFO
```

```
#APPLICATION MANAGER INITIALISATION
appMgrConf = ApplicationMgr()
appMgrConf.HistogramPersistency = 'ROOT'
appMgrConf.EvtMax = -1
appMgrConf.OutputLevel = outputLevel
```

The application manager oversees the whole execution of the program. Anything you run must be appended to the application manager, otherwise it is ignored. EvtMax = -1 means that all the events you feed the program will be processed. You can set this to any number of events, useful for running a small testjob. We define a global output level so that we can limit the amount of message spam sent to stdout. If it usually advisable to leave this at INFO, and change individual algorithms to VERBOSE in order to find out where problems are.

```
#COUNTERS-      Adds a histogram entry to DVHistos indicating
#               events processed and preslected.  You'll need to add my CMT package to use this
inputCount = EventCount("nEventsTotal")
inputCount.addTool( PhysDesktop() )
inputCount.PhysDesktop.InputLocations = ["Phys/StdLooseMuons","Phys/StdLooseKaons"]
inputCount.OutputLevel = outputLevel

outputCount = EventCount("nEventsPreselected")
outputCount.addTool( PhysDesktop() )
outputCount.PhysDesktop.InputLocations = ["Phys/Bs2JpsiPhi"]
outputCount.OutputLevel = outputLevel
```

These are the reason why we compiled in the EventCount package. In the output file DVHistos.root will be 2 histograms, nEventsTotal and nEventsPreselected. The former tells you the total number of events over which you ran the code, the latter how many made it past your choice of preselection cuts. Here we also introduce the PhysDesktop. It stores the output particles from each algorithm under the algorithm name. You can think of it as a filesystem where algorithms get their particles from one directory, work on them, and write the results to another directory.

The final state particles in this case are muons and kaons. The preselection takes these and combines them to form the J/psi and phi, which are then combined to form the Bs. The first locations are therefore StdLooseMuons and StdLooseKaons, and the last location is Bs2JpsiPhi. The counters therefore count the number of events at the start, and the number of events that end up in Bs2JpsiPhi.

```
#COMBINEPARTICLES- makes the jpsi, phi, and finally the B_s

#J/psi -> mu+ mu-
Jpsi2mumu = CombineParticles("Jpsi2mumu")
Jpsi2mumu.DecayDescriptor = "J/psi(1S) -> mu+ mu-"
Jpsi2mumu.addTool( PhysDesktop() )
Jpsi2mumu.PhysDesktop.InputLocations = ["Phys/StdLooseMuons"]
Jpsi2mumu.CombinationCut = "(ADAMASS('J/psi(1S)')<200*MeV)"
Jpsi2mumu.MotherCut = "(ADMASS('J/psi(1S)')<100*MeV)"
Jpsi2mumu.DaughtersCuts = {"mu+"        :       "(PT>500*MeV)",
                           "mu-"        :       "(PT>500*MeV)"}
Jpsi2mumu.OutputLevel = outputLevel

#phi -> K+ K-
Phi2KK = CombineParticles("Phi2KK")
Phi2KK.DecayDescriptor = "phi(1020) -> K+ K-"
Phi2KK.addTool( PhysDesktop() )
Phi2KK.PhysDesktop.InputLocations = ["Phys/StdLooseKaons"]
Phi2KK.CombinationCut = "(ADAMASS('phi(1020)')<100*MeV)"
Phi2KK.MotherCut = "(ADMASS('phi(1020)')<40*MeV)"
Phi2KK.DaughtersCuts = {"K+"        :       "(PT>500*MeV)",
                        "K-"        :       "(PT>500*MeV)"}
Phi2KK.OutputLevel = outputLevel

#Bs -> J/psi phi
Bs2JpsiPhi = CombineParticles("Bs2JpsiPhi")
```

```
Bs2JpsiPhi.DecayDescriptor = "B_s0 -> J/psi(1S) phi(1020)"
Bs2JpsiPhi.addTool( PhysDesktop() )
Bs2JpsiPhi.PhysDesktop.InputLocations = ["Phys/Phi2KK","Phys/Jpsi2mumu"]
Bs2JpsiPhi.CombinationCut = "(ADAMASS('B_s0')<2*GeV)"
Bs2JpsiPhi.MotherCut = "(ADMASS('B_s0')<1*GeV)"
Bs2JpsiPhi.DaughtersCuts = {    "phi(1020)"     :       "(PT>500*MeV)",
                                "J/psi(1S)"     :       "(PT>500*MeV)"}
Bs2JpsiPhi.OutputLevel = outputLevel
```

Here we make the phi, Jpsi and then finally the Bs using the CombineParticles algorithm. The Jpsi and phi take muons and kaons from the PhysDeskop, apply DaughtersCuts to the final states seperately, then a CombinationCut to the pair, and finally a MotherCut to the result. The syntax, naming conventions and brief mode of operation of these cuts can be found on the LoKiHybridFilters page. At the end of this we have our preselected Bs candidates sitting in the Bs2JpsiPhi PhysDesktop location.

```
#DECAYTREETUPLES - Decaytreetuple writes out a few generic properties of the particles
#NOTE: These need to be changed to add/remove the variables you want for the studies

Tuple = DecayTreeTuple("Bs2JpsiPhiTuple")
Tuple.addTool( PhysDesktop() )
Tuple.PhysDesktop.InputLocations = ["Phys/Bs2JpsiPhi"]
Tuple.Decay = "[B_s0 -> (^J/psi(1S) => ^mu+ ^mu-) (^phi(1020) => ^K+ ^K-)]cc";
```

Now we use the DecayTreeTuple to write our results out to the Ntuple. The decay tree specified contains charets denoting which particles we want to write out- in this case all of them. We want to write out different properties depending on the particle type, so we do this by specifying branches:

```
#breaks the particles into branches indicated by a "^". This allows each particle to have differe
Tuple.Branches = {
"muonplus"        : "[B_s0 -> (J/psi(1S) => ^mu+ mu-) (phi(1020) => K+ K-)]cc"
,"muonminus"        : "[B_s0 -> (J/psi(1S) => mu+ ^mu-) (phi(1020) => K+ K-)]cc"
,"kaonplus"       : "[B_s0 -> (J/psi(1S) => mu+ mu-) (phi(1020) => ^K+ K-)]cc"
,"kaonminus"       : "[B_s0 -> (J/psi(1S) => mu+ mu-) (phi(1020) => K+ ^K-)]cc"
,"phi"        : "[B_s0 -> (J/psi(1S) => mu+ mu-) (^phi(1020) => K+ K-)]cc"
,"Jpsi"       : "[B_s0 -> (^J/psi(1S) => mu+ mu-) (phi(1020) => K+ K-)]cc"
,"B_s"       : "[B_s0]cc :[B_s0 -> (J/psi(1S) => mu+ mu-) (phi(1020) => K+ K-)]cc"
}

Tuple.addTool(TupleToolDecay, name = 'Jpsi')
Tuple.addTool(TupleToolDecay, name = 'phi')
Tuple.addTool(TupleToolDecay, name = 'muonplus')
Tuple.addTool(TupleToolDecay, name = 'muonminus')
Tuple.addTool(TupleToolDecay, name = 'kaonplus')
Tuple.addTool(TupleToolDecay, name = 'kaonminus')
Tuple.addTool(TupleToolDecay, name = 'B_s')
```

Again, the charets denote which particle is going to be written out in each branch. For the final states we want to produce the combined delta-log-likelihood particle ID:

```
kaonplusLoKiTool = LoKi__Hybrid__TupleTool( 'kaonplusLoKiTool')
kaonplusLoKiTool.Variables = {
"LOKI_PIDK_PIDpi" : "PIDK-PIDpi",
"LOKI_PIDK_PIDe" : "PIDK-PIDe",
"LOKI_PIDK_PIDmu" : "PIDK-PIDmu",
"LOKI_PIDK_PIDp" : "PIDK-PIDp",
}
```

The first string in quotations denotes the name of the column in the ntuple, the second is the variable that this column will be filled with. We therefore are writing out the 4 dll variables for this final state. Similarly for the other 3 final states. All LoKi variables begin with a "LOKI_" to make it clear that these are variables that we cut on.

```
#Jpsi specific
JpsiLoKiTool = LoKi__Hybrid__TupleTool( 'JpsiLoKiTool')
JpsiLoKiTool.Variables = {
"LOKI_VCHI2" : "VFASPF(VCHI2)",
"LOKI_BPVVDCHI2" : "BPVVDCHI2",
"LOKI_MIPCHI2_PRIMARY" : "MIPCHI2DV(PRIMARY)"
}
```

For the Jpsi and phi we want the Vertex chi2, Impact parameter chi2, and primary vertex separation chi2. Again for an explanation of the variable names, see LoKiHybridFilters

```
#B_s specific
BsLoKiTool =  LoKi__Hybrid__TupleTool( 'BsLoKiTool')
BsLoKiTool.Variables = {
"LOKI_VCHI2" : "VFASPF(VCHI2)",
"LOKI_BPVVDCHI2" : "BPVVDCHI2",
"LOKI_DIRA" : "BPVDIRA",
"LOKI_MIPCHI2_PRIMARY" : "MIPCHI2DV(PRIMARY)",
"LOKI_BPVLTIME" : "BPVLTIME()",
"LOKI_BPVLTFITCHI2" : "BPVLTFITCHI2()",
"LOKI_BPVLTCHI2" : "BPVLTCHI2()"
}


TupleToolP2VV = TupleToolP2VV( 'TupleToolP2VV')
TupleToolP2VV.Calculator = 'Bs2JpsiPhiAngleCalculator'

Tuple.B_s.addTool(BsLoKiTool, name='BsLoKiTool')
Tuple.B_s.ToolList = [
"LoKi::Hybrid::TupleTool/BsLoKiTool"
, "TupleToolPropertime"
,"TupleToolP2VV"
,"TupleToolTagging"
]
```

The Bs gets a few more variables for the propertime, as well as a generic TupleTool that writes out the tag informaton for you.

```
#Common to all particles
LoKiTool = LoKi__Hybrid__TupleTool( 'LoKiTool')
LoKiTool.Variables = {
"LOKI_MM" : "MM",
"LOKI_M" : "M",
"LOKI_P" : "P",
"LOKI_PT" : "PT",
"LOKI_TRCHI2" : "TRCHI2",
"LOKI_ABSID" : "ABSID",
"LOKI_ID" : "ID",
"LOKI_IPCHI2" : "BPVIPCHI2()"
}

Tuple.addTool(LoKiTool, name = 'LoKiTool')
Tuple.ToolList = [
"LoKi::Hybrid::TupleTool/LoKiTool"
,"TupleToolEventInfo"
,"TupleToolGeometry"
,"TupleToolKinematic"
,"TupleToolMCBackgroundInfo"
,"TupleToolMCHierarchy"
,"TupleToolMCTruth"
,"TupleToolPid"
,"TupleToolPrimaries"
,"TupleToolTrackInfo"
,"TupleToolTrigger"
```

```
,"TupleToolTISTOS"
,"TupleToolVtxIsoln"
]

Tuple.addTool(TupleToolEventInfo)
Tuple.TupleToolEventInfo.OutputLevel = outputLevel

Tuple.addTool(TupleToolGeometry)
Tuple.TupleToolGeometry.OutputLevel = outputLevel

Tuple.addTool(TupleToolKinematic)
Tuple.TupleToolKinematic.OutputLevel = outputLevel

Tuple.addTool(TupleToolMCBackgroundInfo)
Tuple.TupleToolMCBackgroundInfo.OutputLevel = outputLevel

Tuple.addTool(TupleToolMCHierarchy)
Tuple.TupleToolMCHierarchy.OutputLevel = outputLevel

Tuple.addTool(TupleToolMCTruth)
Tuple.TupleToolMCTruth.OutputLevel = outputLevel


#TupleToolP2VV.OutputLevel = DEBUG
Tuple.B_s.addTool(TupleToolP2VV)

Tuple.addTool(TupleToolPid)
Tuple.TupleToolPid.OutputLevel = outputLevel

Tuple.addTool(TupleToolPrimaries)
Tuple.TupleToolPrimaries.OutputLevel = outputLevel

Tuple.B_s.addTool(TupleToolPropertime)
Tuple.B_s.TupleToolPropertime.OutputLevel = outputLevel

Tuple.B_s.addTool(TupleToolTagging)
Tuple.B_s.TupleToolTagging.OutputLevel = outputLevel

Tuple.addTool(TupleToolTISTOS)
Tuple.TupleToolTISTOS.OutputLevel = outputLevel

Tuple.addTool(TupleToolTrackInfo)
Tuple.TupleToolTrackInfo.OutputLevel = outputLevel

Tuple.addTool(TupleToolTrigger)
Tuple.TupleToolTrigger.OutputLevel = outputLevel
Tuple.TupleToolTrigger.FillL0 = 1
Tuple.TupleToolTrigger.FillHlt = 1
Tuple.TupleToolTrigger.VerboseL0 = 1
Tuple.TupleToolTrigger.VerboseHlt1 = 1
Tuple.TupleToolTrigger.VerboseHlt2 = 1

Tuple.addTool(TupleToolVtxIsoln)
Tuple.TupleToolVtxIsoln.OutputLevel = outputLevel
```

Lastly, we add all of the common variables we want to write out for each particle, and some more tupletools. Each particle will get P and Pt, Mass (measured and naively calculated), Track Chi2 and IP chi2 values. The tupletools write out complimentary information such as event ID, MC information, etc. It is here that we will also add the Helicity/Transversity angles when the code is completed. Note that these variables are by no means final. Please update this page and the steering file with the variables you want added.

```
#ORIGIN FINDERS - My origin finder code. You'll need to add my OriginFinder cmt package to use th
# Prints the MC association for all 4 final state particles and locates their specific background
# This will only be useful when we have enough b-inclusive to find peaks.
```

```
#Origin = OriginFinder("Origin")
#Origin.addTool( PhysDesktop() )
#Origin.PhysDesktop.InputLocations = ["Phys/Bs2JpsiPhi"]
#Origin.pidToCheck = 531
#Origin.OutputLevel = 3
```

The origin finder is optional, and prints where each final state really came from by way of MC association. It is not recommended to run this on signal data at preselection level as you will increase your logfile size by ~100 lines per event. This is useful for locating peaking backgrounds at selection level (assuming we ever get enough bb-inclusive to run it).

```
#INPUT DATA - This is a signal dataset. Ganga overwrites this, but it is good to have as a test j
EventSelector(Input=["DATAFILE=' dcap://pool1.epcc.ed.ac.uk:22125/pnfs/epcc.ed.ac.uk/data/lhcb/pr
EventSelector().PrintFreq= 10000
EventSelector().FirstEvent = 1
EventSelector.OutputLevel = outputLevel
```

Here we add a single dataset to allow us to test the code locally.

```
#SEQUENCER - As per the old .opts files.
Bs2JpsiPhiSeq = GaudiSequencer("Bs2JpsiPhiSeq")

Bs2JpsiPhiSeq.Members.append(inputCount)
Bs2JpsiPhiSeq.Members.append(Phi2KK)
Bs2JpsiPhiSeq.Members.append(Jpsi2mumu)
Bs2JpsiPhiSeq.Members.append(Bs2JpsiPhi)
Bs2JpsiPhiSeq.Members.append(Tuple)
#Bs2JpsiPhiSeq.Members.append(Origin)
Bs2JpsiPhiSeq.Members.append(outputCount)

appMgrConf.TopAlg.append( Bs2JpsiPhiSeq)
```

And now we put all the different algorithms together in a sequence, which we add to the application manager. Note that the order is important- we want to count the input events, create the resonances, merge them to a Bs, write out the result, (optionally) print the origin, and finally count the events passing the preselection. Changing the order would be meaningless as we can't create the Bs without first creating the resonances.

```
#OUTPUT
HistogramPersistencySvc().OutputFile = "DVHistos.root"
NTupleSvc().Output=["FILE1 DATAFILE='DVNtuple.root' TYP='ROOT' OPT='NEW'"]
```

And lastly, we specify the output filenames and types.

_Presel_Ganga.py

Running a local test

```
$ setupLHCb
$ SetupProject DaVinci v20r0
$ gaudirun.py Bs2JpsiPhi_Presel_Options.py
```

Running via ganga

```
$ setupLHCb
$ GangaEnv latest
$ ganga Bs2JpsiPhi_Presel_Ganga.py
$ ganga
$ gu.summary(jobs[-1])
```

-- ConorFitzpatrick - 20 Sep 2008

- Bs2JPsiPhi_Presel_Ganga.py.txt: Ganga job steering file
- Bs2JPsiPhi_Presel_Options.py.txt: DaVinci Preselection steering file

---

This topic: Main > LHCbEdinburghGroupCommonJpsiPhiNtupleOptions
Topic revision: r4 - 2008-10-06 - GreigCowan