LCFIVertexPackage Reference Manual

Generated by Doxygen 1.3.5

Mon Jan 12 10:23:47 2009

Contents

1 The LCFI Vertex Package

The LCFI Vertex Package provides the vertex finder ZVTOP, originally developed for SLD by D. Jackson [1], flavour tagging as well as vertex charge determination for b- and c-jets. By default, the flavour tag provided is obtained from the algorithm by R. Hawkings [2]. It is based on a neural net approach, combining track and vertex information to distinguish b, c- and light jets. The algorithm to determine vertex charge follows the SLD-approach [3], with modifications for b-jets developed by S. Hillert [4].

In addition to the algorithms, the package provides an object-oriented framework, in which the default approach can easily be modified and extended. Care was taken to make all main parameters of the code accessible to the user as steering parameters. The code was interfaced to the MarlinReco analysis framework and uses LCIO for input and output, permitting it to be used in conjunction with algorithms from other reconstruction frameworks.

The code was implemented by Ben Jeffery (ZVTOP, LCIO/Marlin interface, working classes design and testing), Erik Devetak (Flavour tag inputs calculation and testing, MC Jet flavour, Vertex Charge Processor), Mark Grimes (Flavour tag procedure, Vertex fitter), Dave Bailey (neural network code), Victoria Martin (AIDA Plot Processor), Tomas Lastovicka (Kalman filter for vertex fitting), Kristian Harder (use GEAR-geometry for suppression of hadronic interactions, Purity - Efficiency macro) and Sonja Hillert (coordination, system test). The authors thank the LCFI physics group for help and advice during the development phase, in particular D. Jackson (advice on ZVTOP), K. Harder(testing, Mokka/Gear interface) V. Martin (test of vertex charge procedure for c-jets), T. Lastovicka (testing), R. Walsh (testing), Clare Lynch (testing).

We would also like to thank F. Gaede, T. Behnke and N. Graf for fruitful discussions, D. Martsch for producing a test sample on the GRID and A. Raspereza for advice and for extending the track cheater functionality to provide the input required by the Vertex Package.

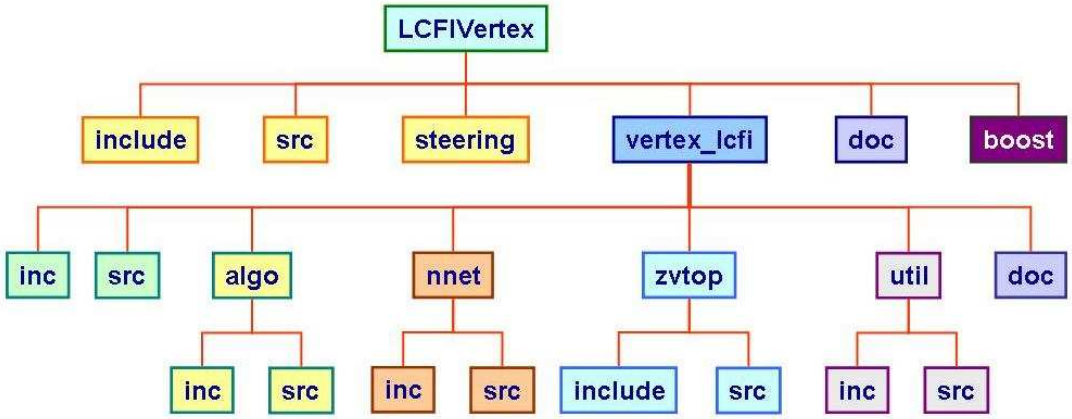

Figure 1: Vertex Package directory structure

Directories of the package are organised as follows: the top-level directories contain the Marlin processors (src) and the include- and steering files they require, as well as macros (macro) that can be run on output created by some of the processors. For example, a root macro is provided to make comparison plots of purity vs efficiency obtained from two subsequent runs of the PlotProcessor. (Note, that the package provides root output only if compiled with root option). The top-level directories provide an interface to the main part of the code, which is located in the directory vertex_lcfi. The Marlin processors access a set of algorithm classes for ZVTOP, flavour tag and vertex charge calculation which can be found in the subdirectory algo. These algorithm classes all inherit from a simple interface Algo, providing parameters and the method "calculateFor", returning the output of the algorithm. Input to the algorithm classes are objects like jets or events. The implementation of these object classes can be found in the directories vertex_lcfi/inc and vertex_lcfi/src. Working classes specific to the vertex finder ZVTOP, providing functionality like vertex finding, vertex resolving and vertex fitting, are located in the directory vertex_lcfi/zvtop. The neural network code is kept in the directory vertex_lcfi/nnet.

The following Marlin processors are provided:

- TrueAngularJetFlavourProcessor: provides the true jet flavour using MC information
- PerEventIPFitterProcessor: determines the event vertex (IP)
- RPCutProcessor: flexible processor for applying various track selection cuts
- ZVTOPZVRESProcessor: find vertices running the ZVRES branch of ZVTOP
- ZVTOPZVKINProcessor: find vertices running the ZVKIN branch of ZVTOP (ghost track algorithm)
- FlavourTagInputsProcessor: calculate input variables for the flavour tag neural net and the vertex charge
- NeuralNetTrainerProcessor: train neural networks for flavour tag
- FlavourTagProcessor: use pretrained neural nets to obtain flavour tag
- VertexChargeProcessor: calculate vertex charge for b- and/or c-jets
- PlotProcessor: calculate purity and efficiency and produce performance plot (if compiled with root)

- LCFIAIDAPlotProcessor: diagnostic plots and tables for flavour tag inputs and outputs

The example steering files provided show how the package could be run in a typical analysis. The order in which the steering files would be called is as follows:

- `cheattracks+jetfind.xml` - note that hit collection names in this file are geometry specific, the default detector geometry assumed in this example file is LDC01_05Sc.
- `truejetflavour.xml`
- `ipfit.xml`
- `zvres.xml`
- `fti.xml`
- `trainNeuralNets.xml` (optional, only needed for special training run to obain new flavour tag neural nets)
- `ft.xml`
- `Bvertexcharge.xml`
- `Cvertexcharge.xml`
- `ftplot.xml`

The first of these steering files calls event reconstruction processors from MarlinReco that are not part of the package but need to be run in order to obtain the collections required. Running Marlin with this steering file creates an input LCIO file for the package, by default called cheatout.slcio. It contains collections with MC particles, jets and the ReconstructedParticles within the jets. The execution flow diagram shows how these collections are used by the LCFIVertex package, as well as processors used and collections created if using the example steering files above. (The training of new neural nets is not covered in the diagram, as this would be done in a dedicated training run; the typical application uses networks that have already been trained). Unless otherwise indicated, all collections shown in the diagram are of type ReconstructedParticle.

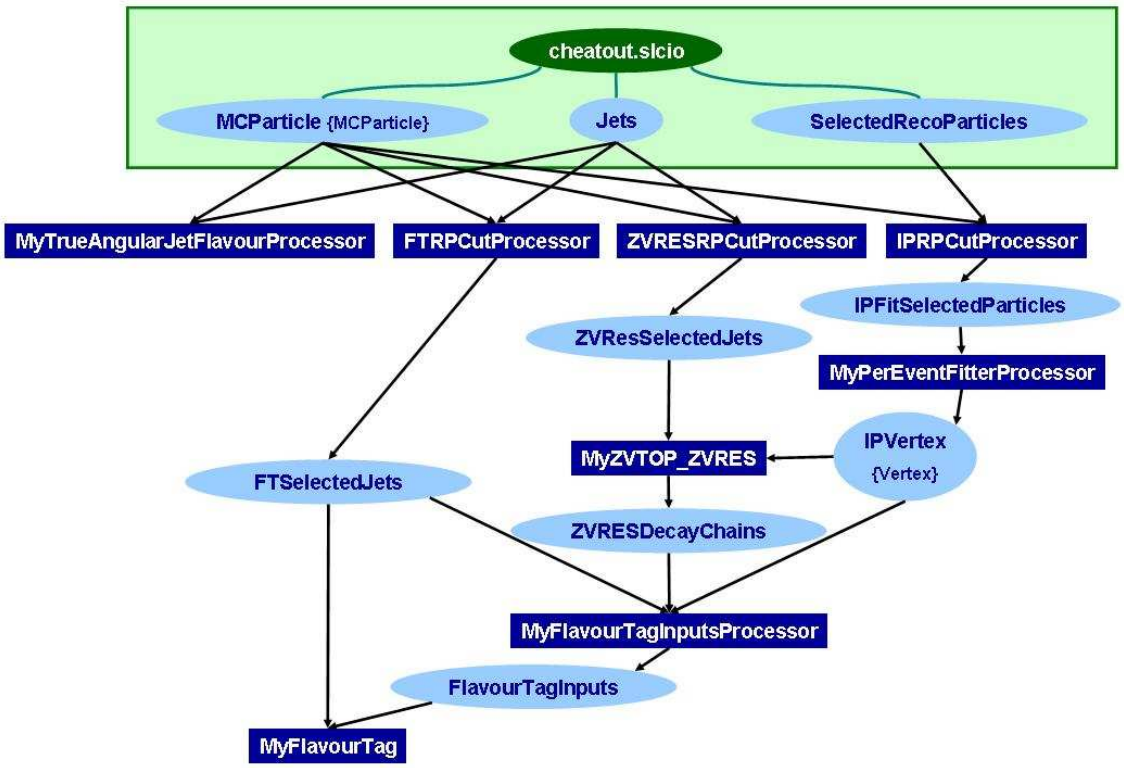

Figure 2: Vertex Package execution flow obtained from the example steering files

Some processors depend on others to have run before, e.g. ZVTOP and the flavour tag inputs processor each require their own track selection implemented by the RPCutProcessor. Further details are provided in the documentation for each of the processors (see above) and in the tutorial section. Information on the internal ZVTOP classes can be found in the ZVTOP documentation. Scope and usage of the neural network code (which can also be used for purposes other than flavour tagging) is described in the neural net documentation.

In addition to code and example steering files, the package provides a set of pre-trained networks for the Hawkings default flavour tag. These have been trained using the fast MC SGV and are located in a new repository **tagnet** in the `Zeuthen CVS repository` (Use 'tagnet' as project name when checking out the directory). **We strongly recommend submitting any new networks that users train with different boundary conditions to this directory along with a detailed description of training conditions.** A form for providing training information can be found in the same directory in order to make ILC physics studies more transparent and ease comparisons of analyses performed within different groups, frameworks or detector concepts.

Summary of changes in release versions:

- ReleaseNotesv00-02-01 "v00-02-01"

[1] D. Jackson, NIM A 388 (1997) 247

[2] R. Hawkings, LC-PHSM-2000-021

[3] J. Thom, SLAC-R-585 (2002), T. Wright, SLAC-R-602 (2002)

[4] S. Hillert, proceedings LCWS 2005

In case of comments or questions **not answered by the documentation** please contact the development and maintenance team:

Erik Devetak (mailto:E.Devetak1@physics.ox.ac.uk)

Mark Grimes (mailto:Mark.Grimes@bristol.ac.uk)

Kristian Harder (mailto:K.Harder@rl.ac.uk)

Sonja Hillert (mailto:S.Hillert1@physics.ox.ac.uk)

Talini Pinto Jayawardena (mailto:T.S.Pinto.Jayawardena@rl.ac.uk)

Ben Jeffery (mailto:B.Jeffery1@physics.ox.ac.uk)

Tomas Lastovicka (mailto:T.Lastovicka1@physics.ox.ac.uk)

Clare Lynch (mailto:Clare.Lynch@bristol.ac.uk)

Victoria Martin (mailto:victoria.martin@ed.ac.uk)

Roberval Walsh (mailto:r.walsh@ed.ac.uk)

2 LCFIVertexPackage Namespace Index

2.1 LCFIVertexPackage Namespace List

Here is a list of all documented namespaces with brief descriptions:

3 LCFIVertexPackage Class Index

3.1 LCFIVertexPackage Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 LCFIVertexPackage Page Index

4.1 LCFIVertexPackage Related Pages

Here is a list of all related documentation pages:

5 LCFIVertexPackage Namespace Documentation

5.1 vertex_lcfi::nnet Namespace Reference

Neural Net namespace.

5.1.1 Detailed Description

Neural Net namespace.

6 LCFIVertexPackage Class Documentation

6.1 DSTPlotProcessor Class Reference

Creates some sample plots from the data calculated by the LCFI vertex package.

`#include <DSTPlotProcessor.h>`

Protected Member Functions

- bool _passesEventCuts (lcio::LCEvent *pEvent)

  A function that contains all the event cuts - returns true if the event passes all of the cuts, false otherwise.

- bool _passesJetCuts (lcio::ReconstructedParticle *pJet)

  A function that contains all the jet cuts - returns true if the event passes all of the cuts, false otherwise.

6.1.1 Detailed Description

Creates some sample plots from the data calculated by the LCFI vertex package.

An example of getting the flavour tag results from the LCIO file and plotting an efficiency purity graph with them. Also plots a graph of jet energies for good measure.

The processor checks the specifed LCFloatVec collections for the flavour tag values "BTag", "CTag" and "BCTag" which are the names that FlavourTagProcessor stores its b tag, c tag and c tag (only b background) values in respectively.

These values are checked against the true jet flavour (from the TrueJetFlavour LCIntVec) and efficiency-purity values calculated for a range of cuts.

The jet energy is taken from the energy of the reconstructed particle used to represent the jet.

Getting Root output To output to a Root file instead of CSV files the processor has to be compiled with the USEROOT preprocessor flag defined. You could add "#define USEROOT" to the code, or more easily add the line

USERINCLUDES += -D USEROOT

to the userlib.gmk file that is in the Marlin directory. If Marlin is not already set up to use Root then you will also need to add the following lines (this assumes a fully working root installation):

USERINCLUDES += ‘root-config –cflags‘

USERLIBS += ‘root-config –libs‘

Input From the LCIO file, flavour tag variable values of:

"BTag" "CTag" "BCTag"

And

"JetType"

Output If the USEROOT preprocessor flag was defined when this processor was compiled, then the output will be a root file with the filename specified in the steering file. Otherwise, the efficiency-purity values will be output as comma separated values to the file +".csv", and the jet energies to +"-JetEnergies.csv".

Parameters:

JetCollectionName Name of the ReconstructedParticle collection that represents jets.

FlavourTagCollections Names of the LCFloatVec collections holding the Flavour tags, all tags in this list will be produced in one file for comparison

TrueJetFlavourCollection LCIntVec that contains the MC Jet flavour (from TrueJetFlavourProcessor)

OutputFilename The name of the file that will hold the output.

The documentation for this class was generated from the following file:

- DSTPlotProcessor.h

6.2 FlavourTagInputsProcessor Class Reference

Calculates the Flavour tag input variables for flavour tagging.

`#include <FlavourTagInputsProcessor.h>`

6.2.1 Detailed Description

Calculates the Flavour tag input variables for flavour tagging.

The aim of the processor is to calculate a series of highly discriminating tagging variables. At present the default variables are the one defined in the R. Hawking LC note LC-PHSM-2000-021. All the variables are calculated inside independent classes that inherit from the vertex_lcfi::Algo template and not in the main processor file. This makes the processor file extremely flexible and new variables easy to add. Similary it is also very simple to remove undesired variables. The following variables are presently calculated (variables depending on the vertex_lcfi::TrackAttach procedure are marked by $*$, variables depending on vertex_lcfi::TwoTrackPid are marked by $\hat{}$)

D0Significance1 - calculated in vertex_lcfi::ParameterSignificance

D0Significance2 - calculated in vertex_lcfi::ParameterSignificance

Z0Significance1 - calculated in vertex_lcfi::ParameterSignificance

Z0Significance2 - calculated in vertex_lcfi::ParameterSignificance

Momentum1 - calculated in vertex_lcfi::ParameterSignificance

Momentum2 - calculated in vertex_lcfi::ParameterSignificance

JointProbRPhi - $\hat{}$ calculated in vertex_lcfi::JointProb

JointProbZ - $\hat{}$ calculated in vertex_lcfi::JointProb

DecayLengthSignificance - calculated in vertex_lcfi::VertexDecaySignificance

DecayLength - calculated in vertex_lcfi::VertexDecaySignificance

PTCorrectedMass - $*$ calculated in vertex_lcfi::VertexMass

RawMomentum - $*$ calculated in vertex_lcfi::VertexMomentum

NumTracksInVertices - calculated in vertex_lcfi::VertexMultiplicity

SecondaryVertexProbability - $*$ calculated in vertex_lcfi::SecVertexProb

For more information about the algorithms themselves please consult the specific algorithm documentation pages. The processor also uses the following algorithms:

vertex_lcfi::TwoTrackPid - algorithm that calulates the id of two charged tracks by using mass considerations. This algorithm removes tracks consistent with the hypothesis that they have been generated from Ks and gamma. This algorithm is not used in ZVTOPZVRESProcessor or ZVTOPZVKINProcessor

vertex_lcfi::TrackAttach - algorithm that adds tracks close to the seed vertex.

Input

- A collection of ReconstructedParticles that represents the jets in the event (obtained from a jet finder, say SatoruJetFinderProcessor).
- A collection of vertices that contains the per event primary vertices; one for each event. (optional) This collection is filled in the vertex_lcfi::PerEventIPFitter processor.
- A collection of decay chains as filled by the the ZVTOPZVRESProcessor or ZVTOPZVKINProcessor.

Output The processor writes into the selected lcio output file. All the values calculated by the processor are saved in the same LCFloatVec collection. The default name of the output collection is FlavourTagInputs.For more details see the interface documentation.

Parameters:

JetRPCollection Name of the ReconstructedParticle collection that represents jets.

IPVertexCollection Name of the Vertex collection that contains the primary vertices (optional)

FlavourTagInputsCollection Name of the LCFloatVec Collection that will be created to contain the flavour tag inputs

The following parameters are parameters for the algorithms used by the processor.These parameters are all optional.

Parameters:

LayersHit Momentum cuts will be applied on number of LayersHit and LayersHit minus one, used by vertex_lcfi::ParameterSignificance

AllLayersMomentumCut Cut on the minimum momentum if track hits LayersHit, used by vertex_lcfi::ParameterSignificance

AllbutOneLayersMomentumCut Cut on the minimum momentum if track hits LayersHit minus one, used by vertex_lcfi::ParameterSignificance

JProbMaxD0Significance Maximum d0 significance of tracks used to calculate the joint probability, used in vertex_lcfi::JointProb

JProbMaxD0andZ0 Maximum d0 and z0 of tracks used to calculate the joint probability, used in vertex_lcfi::JointProb

PIDChi2Cut Cut on the Chi squared of two tracks being in the same vertex, used by vertex_lcfi::TwoTrackPid

PIDMaxGammaMass Cut on the upper limit of the photon candidate mass, used by vertex_lcfi::TwoTrackPid

PIDMaxKsMass Cut on the upper limit of the Ks candidate mass, used by vertex_lcfi::TwoTrackPid

PIDMinKsMass Cut on the lower limit of the Ks candidate mass, used by vertex_lcfi::TwoTrackPid

PIDRPhiCut Cut on the maximum RPhi of the Ks/gamma decay vertex candidate, used by vertex_lcfi::TwoTrackPid

PIDSignificanceCut Cut on the minimum RPhi significance of the tracks, used by vertex_lcfi::TwoTrackPid

SecondVertexNtrackscut Cut on the minimum number of tracks in the seed vertex, used by vertex_lcfi::SecVertexProb

SecondVertexProbChisquarecut Cut on the Chi Squared of the seed vertex, used by vertex_lcfi::SecVertexProb

TrackAttachAllSecondaryTracks include or exclude tracks in the inner vertices for the track attachment.

TrackAttachCloseapproachCut upper cut on track distance of closest approach to the seed axis used by vertex_lcfi::TrackAttach (when used for ∗ flagged variables)

TrackAttachLoDCutmax Cut determining the maximum L/D for the track attachment, used by vertex_lcfi::TrackAttach (when used for ∗ flagged variables)

TrackAttachLoDCutmin Cut determining the minimum L/D for the track attachment, used by vertex_lcfi::TrackAttach (when used for ∗ flagged variables)

VertexMassMaxKinematicCorrectionSigma Maximum Sigma (based on error matrix) by which the vertex axis can move when kinematic correction is applied, used by vertex_lcfi::VertexMass

VertexMassMaxMomentumAngleCut Upper cut on angle between momentum of vertex and the vertex axis, used by vertex_lcfi::VertexMass

VertexMassMaxMomentumCorrection Maximum factor, by which vertex mass can be corrected, used by vertex_lcfi::VertexMass

As a final remark one should notice that two additional values are stored in the Output LC Collection. These are:

NumVertices - number of vertices in the jet; used to determine what variables to use in the following flavour tag processor. Calculated in the processor.

DecayLength(SeedToIP)- distance from the vertex seed in the trackattach processor and IP. This veriable can be used for further analysis, but it is not used in flavour tagging. Calculated in the processor.

Author:
Erik Devetak(erik.devetak1@physics.ox.ac.uk),
interface by Ben Jeffery (ben.jeffery1@physics.ox.ac.uk)

The documentation for this class was generated from the following file:

- FlavourTagInputsProcessor.h

6.3 FlavourTagProcessor Class Reference

Performs a neural net based flavour tag using data calculated by the LCFI vertex package.

`#include <FlavourTag.h>`

6.3.1 Detailed Description

Performs a neural net based flavour tag using data calculated by the LCFI vertex package.

Loads in previously trained neural networks from the filenames provided in the steering file, and performs a flavour tag with them using the data previously stored in the file by the FlavourTagInputsProcessor. The networks can be trained using the NeuralNetTrainerProcessor.

This processor requires 9 neural networks, which are 3 for each of the 1 vertex, 2 vertices and 3 or more vertices cases. These 3 are a b jet tagging network, a c jet tagging network and a c jet with only b background tagging network. If any of these saved neural network files are not present the processor will throw a lcio::Exception. The networks can be in either text or XML format; the processor checks to see if the file starts with "<?xml" and decides whether to load as text or XML.

N.B. The code that loads the XML networks is currently a little shaky. **If the XML is not properly formed then you may get a segmentation fault or runaway memory allocation leading to Marlin crashing.** This is still being looked into.

For more information on the tagging variables used as input, have a look at the documentation for FlavourTagInputsProcessor. The flavour tag result will be in the range 0 to 1; so to select tagged jets apply a cut on this value (e.g. the b-tag value to tag b-jets). If anything goes wrong (that doesn't produce an exception) then a -1 will be stored instead.

Input

- 9 previously trained neural networks (trained by NeuralNetTrainerProcessor).
- A collection of LCFloatVec that hold the flavour tag variables (put in the lcio file by FlavourTagInputsProcessor).
- A collection of ReconstructedParticles that represents the jets in the event (put in by your jet finder, say SatoruJetFinderProcessor).

Output

- A collection of LCFloatVec that contains the 3 tag results for each jet (b tag, c tag and c only b background tag). The collection will have a float vector for each jet in the same order as the jets; so for example, the tags for "pJetCollection → getElementAt(3)" will be in "pFlavourTagCollection → getElementAt(3)". For more details see the interface documentation

Parameters:

JetCollectionName The name of the collection of ReconstructedParticles representing the jets.

FlavourTagInputsCollection The name of the collection of LCFloatVec that is the flavour tag inputs.

FlavourTagCollection The name of the collection of the flavour tag results that will be created.

Filename-b_net-1vtx Filename for the 1 vertex b tag network.

Filename-c_net-1vtx Filename for the 1 vertex c tag network.

Filename-bc_net-1vtx Filename for the 1 vertex c tag (only b background) network.

Filename-b_net-2vtx Filename for the 2 vertex b tag network.

Filename-c_net-2vtx Filename for the 2 vertex c tag network.

Filename-bc_net-2vtx Filename for the 2 vertex c (only b background) tag network.

Filename-b_net-3plusvtx Filename for the 3 or more vertices b tag network.

Filename-c_net-3plusvtx Filename for the 3 or more vertices c tag network.

Filename-bc_net-3plusvtx Filename for the 3 or more vertices c tag (only b background) network.

Author:

Mark Grimes (mark.grimes@bristol.ac.uk)

The documentation for this class was generated from the following file:

- FlavourTag.h

6.4 LCFIAIDAPlotProcessor Class Reference

LCFIAIDAPlotProcessor Class - make plots of the LCFI flavour tag and vertex charge code.

`#include <LCFIAIDAPlotProcessor.h>`

Protected Member Functions

- int FindTrueJetType (LCEvent *pEvent, unsigned int jetNumber)

  Finds the true flavour of a jet (uses TrueJetFlavourCollection).

- float FindTrueJetHadronCharge (LCEvent *pEvent, unsigned int jetNumber)

  Finds the true charge of the hadron producing a jet (uses TrueJetFlavourCollection).

- int FindTrueJetPDGCode (LCEvent *pEvent, unsigned int jetNumber)

  Finds the PDG code of the hadron producing a jet (uses TrueJetFlavourCollection).

- float FindTrueJetPartonCharge (LCEvent *pEvent, unsigned int jetNumber)

  Finds the true charge of the parton producing a jet (uses TrueJetFlavourCollection).

- int FindTrueJetFlavour (LCEvent *pEvent, unsigned int jetNumber)

  Finds the true flavour of the jet (uses TrueJetFlavourCollection).

- void FindTrueJetDecayLength (LCEvent *pEvent, unsigned int jetNumber, std::vector< double > &decaylengthvector, std::vector< double > &bjetdecaylengthvector, std::vector< double > &cjet-decaylengthvector)

  Finds the true decay length of the longest b- or c- hadron in a jet.

- int FindNumVertex (LCEvent *pEvent, unsigned int jetNumber, unsigned int iInputsCollection)

  Finds the number of vertices in an event (from the flavour tag inputs).

- int FindCQVtx (LCEvent *pEvent, unsigned int jetNumber)

  Finds the vertex charge of the jet - using cuts tuned to find vertex charge for C-jets (from CVertexCharge-Collection).

- int FindBQVtx (LCEvent *pEvent, unsigned int jetNumber)

  Finds the vertex charge of the jet - using cuts tuned to find vertex charge for B-jets (from BVertexCharge-Collection).

- AIDA::IDataPointSet * CreateEfficiencyPlot (const AIDA::IHistogram1D *pSignal, AIDA::IData-PointSet *pDataPointSet)

  Makes a DataPointSet of the tag efficiency e.g number of B-jets passing a given B-tag NN cut, as a function of NN.

- AIDA::IDataPointSet * CreateEfficiencyPlot2 (const AIDA::IHistogram1D *pAllEvents, const AIDA::IHistogram1D *pPassEvents, AIDA::IDataPointSet *pDataPointSet)

  Makes a DataPointSet of histogram 1 divide by histogram 2 - this is an IDataPointSet as a histrogram gives the wrong errors.

- AIDA::IDataPointSet * CreateIntegralPlot (const AIDA::IHistogram1D *pNN, AIDA::IDataPoint-Set *pIntegral)

  Makes a DataPointSet integrating a histogram from the first bin to the last bin – NOT USED.

- AIDA::IDataPointSet * CreatePurityPlot (const AIDA::IHistogram1D *pSignal, const AIDA::IHistogram1D *pBackground, AIDA::IDataPointSet *pDataPointSet)

  Makes a DataPointSet of the tag purity e.g. N(B-jets passing NN cut)/N(all-jets passing NN cut) for a given B-tag NN cut, as a function of NN.

- AIDA::IDataPointSet * CreateLeakageRatePlot (const AIDA::IHistogram1D *pBackground, AIDA::IDataPointSet *pDataPointSet)

  Makes a DataPointSet showing the tagging leakage e.g. the number of non-B-jets passing a given B-tag NN cut, as a function of NN.

- AIDA::IDataPointSet * CreateXYPlot (const AIDA::IDataPointSet *pDataPointSet0, const AIDA::IDataPointSet *pDataPointSet1, AIDA::IDataPointSet *xyPointSet, const int dim0=0, const int dim1=0)

  Plots two DataPointSets against each other.

- AIDA::IHistogram1D * CreateIntegralHistogram (const AIDA::IHistogram1D *pNN, AIDA::IHistogram1D *pIntegral)

  Makes a histogram integrating a histogram from the first bin to the last bin - THE ERRORS RETURNED ARE WRONG!

- void CreateVertexChargeLeakagePlot (AIDA::IDataPointSet *pBJetVtxChargeDPS, AIDA::IDataPointSet *pCJetVtxChargeDPS)

  Makes DataPointSets for the number of.

Protected Attributes

- std::vector< std::string > _FlavourTagCollectionNames

  required input collections

- double _CosThetaJetMax

  cuts on all jets

- double _CosThetaJetMin

  cuts on all jets

- double _PJetMin

  cuts on all jets

- double _PJetMax

  cuts on all jets

- double _BTagNNCut

  Cut on the NN output variables - applied in vertex charge plots.

- double _CTagNNCut

  Cut on the NN output variables - applied in vertex charge plots.

- bool _PrintTrackVertexOutput

  optional parameters to make an ntuple of the neural net inputs; and print out the tagging ouputs (useful for scripts)

- AIDA::IHistogram2D * _pBJetCharge2D

  True B-jets - vertex charge vs true charge.

- AIDA::IHistogram2D * _pCJetCharge2D

  True C-jets - vertex charge vs true charge.

- AIDA::IHistogram1D * _pBJetLeakageRate

  True B-jets - vertex charge leakage rate.

- AIDA::IHistogram1D * _pCJetLeakageRate

  True C-jets - vertex charge leakage rate.

- AIDA::IHistogram1D * _pBJetVertexCharge

  True B-jets - vertex charge.

- AIDA::IHistogram1D * _pCJetVertexCharge

  True C-jets - vertex charge.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _inputsHistogramsBJets

  Histograms of the neural net inputs for true B-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _inputsHistogramsCJets

  Histograms of the neural net inputs for true C-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _inputsHistogramsUDSJets

  Histograms of the neural net inputs for light B-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _zoomedInputsHistogramsBJets

  Zoomed-in histograms of some of the neural net inputs for true B-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _zoomedInputsHistogramsCJets

  Zoomed-in histograms of some of the neural net inputs for true C-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _zoomedInputsHistogramsUDSJets

  Zoomed-in histograms of some of the neural net inputs for true light-jets.

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pLightJetBTag

  Histograms of the neural net B-tag outputs for true light-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pLightJetCTag

  Histograms of the neural net C-tag outputs for true light-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBJetBTag

  Histograms of the neural net B-tag outputs for true B-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBJetCTag

  Histograms of the neural net C-tag outputs for true B-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pCJetBTag

  Histograms of the neural net B-tag outputs for true C-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pCJetCTag

  Histograms of the neural net C-tag outputs for true C-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBJetBCTag

  Histograms of the neural net BC-tag outputs for true B-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pCJetBCTag

  Histograms of the neural net BC-tag outputs for true C-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pLightJetBCTag

  Histograms of the neural net BC-tag outputs for true light-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBTagBackgroundValues

  Histograms of the neural net B-tag outputs for non B-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pCTagBackgroundValues

  Histograms of the neural net C-tag outputs for non C-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBCTagBackgroundValues

  Histograms of the neural net BC-tag outputs for non C-jets - seperately for different number of vertices in the jets, 1, 2, >=3, any (sum of previous).

- std::vector< std::map< std::string, AIDA::IHistogram1D * > > _pBJetBTagIntegral

  Histograms of the neural net tags - number of events that pass a given cut: jet NN value > given NN value for the three tags - B-tag, C-tag, BC-tag - separately for true B jets, true C jets & true light jets and different number of vertices in the jets, 1, 2 or >=3 & any (sum of previous three) See comments above.

- int _numberOfPoints

  Number of bins used for neural nets plots.

- AIDA::ITuple * _pMyTuple

  Tuple of the input variables - only filled for one input collection - selected with UseFlavourTagCollectionForVertexCharge.

- int _cJet_truePlus2

  numbers of true C-jets with true charge ++

- int _cJet_truePlus

  numbers of true C-jets with true charge +

- int _cJet_trueNeut

numbers of true C-jets with true charge 0

- int _cJet_trueMinus
  numbers of true C-jets with true charge -
- int _cJet_trueMinus2
  numbers of true C-jets with true charge –
- int _cJet_truePlus2_recoPlus
  numbers of true C-jets with true charge ++; reconstructed vertex charge >0
- int _cJet_truePlus2_recoNeut
  numbers of true C-jets with true charge ++; reconstructed vertex charge =0
- int _cJet_truePlus2_recoMinus
  numbers of true C-jets with true charge ++; reconstructed vertex charge <0
- int _cJet_truePlus_recoPlus
  numbers of true C-jets with true charge +; reconstructed vertex charge >0
- int _cJet_truePlus_recoNeut
  numbers of true C-jets with true charge +; reconstructed vertex charge =0
- int _cJet_truePlus_recoMinus
  numbers of true C-jets with true charge +; reconstructed vertex charge <0
- int _cJet_trueNeut_recoPlus
  numbers of true C-jets with true charge 0; reconstructed vertex charge >0
- int _cJet_trueNeut_recoNeut
  numbers of true C-jets with true charge 0; reconstructed vertex charge =0
- int _cJet_trueNeut_recoMinus
  numbers of true C-jets with true charge 0; reconstructed vertex charge <0
- int _cJet_trueMinus_recoPlus
  numbers of true C-jets with true charge -; reconstructed vertex charge >0
- int _cJet_trueMinus_recoNeut
  numbers of true C-jets with true charge -; reconstructed vertex charge =0
- int _cJet_trueMinus_recoMinus
  numbers of true C-jets with true charge -; reconstructed vertex charge <0
- int _cJet_trueMinus2_recoPlus
  numbers of true C-jets with true charge –; reconstructed vertex charge >0
- int _cJet_trueMinus2_recoNeut
  numbers of true C-jets with true charge –; reconstructed vertex charge =0

- int _cJet_trueMinus2_recoMinus

  numbers of true C-jets with true charge –; reconstructed vertex charge <0

- int _bJet_truePlus2

  numbers of true B-jets with true charge ++

- int _bJet_truePlus

  numbers of true B-jets with true charge +

- int _bJet_trueNeut

  numbers of true B-jets with true charge 0

- int _bJet_trueMinus

  numbers of true B-jets with true charge -

- int _bJet_trueMinus2

  numbers of true B-jets with true charge –

- int _bJet_truePlus2_recoPlus

  numbers of true B-jets with true charge ++; reconstructed vertex charge >0

- int _bJet_truePlus2_recoNeut

  numbers of true B-jets with true charge ++; reconstructed vertex charge =0

- int _bJet_truePlus2_recoMinus

  numbers of true B-jets with true charge ++; reconstructed vertex charge <0

- int _bJet_truePlus_recoPlus

  numbers of true B-jets with true charge +; reconstructed vertex charge >0

- int _bJet_truePlus_recoNeut

  numbers of true B-jets with true charge +; reconstructed vertex charge =0

- int _bJet_truePlus_recoMinus

  numbers of true B-jets with true charge +; reconstructed vertex charge <0

- int _bJet_trueNeut_recoPlus

  numbers of true B-jets with true charge 0; reconstructed vertex charge >0

- int _bJet_trueNeut_recoNeut

  numbers of true B-jets with true charge 0; reconstructed vertex charge =0

- int _bJet_trueNeut_recoMinus

  numbers of true B-jets with true charge 0; reconstructed vertex charge <0

- int _bJet_trueMinus_recoPlus

  numbers of true B-jets with true charge -; reconstructed vertex charge >0

- int _bJet_trueMinus_recoNeut

  numbers of true B-jets with true charge -; reconstructed vertex charge =0

- int _bJet_trueMinus_recoMinus
  numbers of true B-jets with true charge -; reconstructed vertex charge <0
- int _bJet_trueMinus2_recoPlus
  numbers of true B-jets with true charge –; reconstructed vertex charge >0
- int _bJet_trueMinus2_recoNeut
  numbers of true B-jets with true charge –; reconstructed vertex charge =0
- int _bJet_trueMinus2_recoMinus
  numbers of true B-jets with true charge –; reconstructed vertex charge <0
- std::vector< unsigned int > _cJet_truePlus2_angle
  Vector of numbers of true C-jets with true charge ++ See above for details.
- int _nb_twoVertex_bTrack_Primary
  Numbers for purity if reconstructed track-vertex association.

Static Protected Attributes

- const unsigned int N_VERTEX_CATEGORIES = 3
  number of different vertex categories we want to look at: 1 vertex, 2 vertices, >=3 vertices
- const int N_JETANGLE_BINS = 10
  number of bins used in vertex charge leakage plots

6.4.1 Detailed Description

LCFIAIDAPlotProcessor Class - make plots of the LCFI flavour tag and vertex charge code.

Please note that LCFIAIDAPlotProcessor will not compile with RAIDA v01-03 To use LCFIAIDAPlotProcessor please use AIDAJNI for the implementation of AIDA run "cmake -DBUILD_WITH="ROOT;AIDAJNI" -DAIDAJNI_HOME=${AIDAJNI_HOME}"

This sorry states of affairs comes about becayse not all AIDA functions are defined in RAIDA v01-03

In order to make a histogram file, LCFIAIDAPlotProcessor must be run with AIDAProcessor.

LCFIAIDAPlotProcessor reads in one (or more) FlavourTagCollections, e.g. from FlavourTag and one (or more) TagInputCollections. Histograms/plots are made of the neural net outputs, the purity and leakage rate of the flavour tag. These are split into sub-samples based on the number of vertices found in the jets. Plots are also made of the inputs to the FlavourTagCollections - split into sub-samples based on the true (MC) flavour of the jet.

Options are given to make a tuple of the flavour tag inputs and to print out a text file of the different flavour tag neural net outputs.

(When providing more than one FlavourTagCollection and/or TagInputCollection plots for each collection will be made in different directories.)

In addition LCFIAIDAPlotProcessor also requires a jet collection, and the following collections, which should refer to the *same* jet collection.

BVertexChargeCollection – calculated in VertexChargeProcessor

CVertexChargeCollection – calculated in VertexChargeProcessor

TrueJetFlavourCollection – calculated in TrueAngularJetFlavourProcessor

Input The following collections must be available:

Parameters:

FlavourTagCollections StringVec of LCFloatVec names representing the flavour tag inputs collections - may be more than one collection.

TagInputsCollections StringVec of LCFloatVec names the flavour tag input collections - may be more than one collection.

JetCollectionName Name of ReconstructedParticleCollection representing the jets.

VertexCollectionName Name of VertexCollection representing the Vertex collection of the jets.

BVertexChargeCollection Name of LCFloatVector of the vertex charge of the jet collection, assuming the jets are b-jets (calculated in VertexChargeProcessor)

CVertexChargeCollection Name of LCFloatVector of the vertex charge of the jet collection, assuming the jets are c-jets (calculated in VertexChargeProcessor)

TrueJetFlavourCollection Name of LCFloatVector of the true (MC) flavour of the jets (calculated in TrueAngularJetFlavourProcessor)

VertexCollectionName Name of VertexCollection representing the vertices.

BTagNNCut Double reprsenting the lower cut on the b-tag NN value for some of the plots.

CTagNNCut Double reprsenting the lower cut on the c-tag NN value for some of the plots.

CosThetaJetMax Double representing upper cut on cos(theta) of the jets for the plots.

CosThetaJetMin Double representing lower cut on cos(theta) of the jets for the plots.

PJetMax Double representing upper cut on momentum of the jet for the plots.

PJetMin Double representing lower cut on momentum of the jet for the plots.

MakeTuple Bool set true if you want to make a tuple of the TagInputCollection variables.

NeuralNetOutputFile String representing name of text file of neural net values to. Only used if PrintNeuralNetOutput parameter is true. If left blank, output will be directed to standard out

PrintNeuralNetOutput Bool set true if you want to make a text file of the neural net values (useful for some scripts).

UseFlavourTagCollectionForVertexCharge For vertex charge plots we demand the cTag>CTagNNCut and bTag>BTagNNCut. This integer is used if there is more than one tag collection, to determine which of the collections should be used to apply this cut.

Output

- An aida (or root??) file containing the histograms, plots and tuples.
- (Optionally) a text file containing some of the neural net tagging output

Author:

Victoria Martin (victoria.martin@ed.ac.uk)

The documentation for this class was generated from the following file:

- LCFIAIDAPlotProcessor.h

6.5 NeuralNetTrainerProcessor Class Reference

Trains neural networks to be used for jet flavour tagging.

`#include <NeuralNetTrainer.h>`

Protected Member Functions

- bool _passesCuts (lcio::LCEvent *pEvent)

  All the code for the cuts should be put in here; returns false if the event fails any of the cuts.

6.5.1 Detailed Description

Trains neural networks to be used for jet flavour tagging.

Trains flavour tagging networks using the BackPropagationCGAlgorithm (see the Neural Net Package page) with 500 epochs. The networks are trained on the following data:

If only 1 vertex is found (i.e. only the interaction point)

$$tanh\left(\frac{D0Significance1}{100}\right)$$

$$tanh\left(\frac{D0Significance2}{100}\right)$$

$$tanh\left(\frac{Z0Significance1}{100}\right)$$

$$tanh\left(\frac{Z0Significance2}{100}\right)$$

$$JointProbRPhi$$

$$JointProbZ$$

$$tanh\left(\frac{3 \times Momentum1}{E}\right)$$

$$tanh\left(\frac{3 \times Momentum2}{E}\right)$$

If 2 or more vertices are found

$$tanh\left(\frac{DecayLengthSignificance}{6 \times E}\right)$$

$$tanh\left(\frac{DecayLength}{10}\right)$$

$$tanh\left(\frac{PTCorrectedMass}{5}\right)$$

$$tanh\left(\frac{RawMomentum}{E}\right)$$

$$JointProbRPhi$$

$$JointProbZ$$

$$tanh\left(\frac{NumTracksInVertices}{10}\right)$$
$$SecondaryVertexProbability$$

Where E is the jet energy, everything else is the data calculated by FlavourTagInputsProcessor.

Note that **the processor applies it's own hard coded cuts**. These are documented under the full description of NeuralNetTrainerProcessor::_passesCuts().

Input

- A collection of ReconstructedParticles that represents the jets in the event (put in by your jet finder, say SatoruJetFinderProcessor).
- A LCFloatVec collection that holds the true jet flavours, in the same order as the jets (put in by TrueAngularJetFlavourProcessor).
- A LCFloatVec collection that holds the flavour tag inputs (put in by FlavourTagInputsProcessor)
- Between 1 and 9 filenames for the generated networks. If an output filename is left blank then that network is not trained, but if none are supplied then a lcio::Exception is thrown.

Output Trained neural networks to the filenames supplied, in the format requested. The LCIO file is not modified at all.

Parameters:

JetCollectionName Name of the ReconstructedParticle collection that represents jets.

FlavourTagInputsCollection Name of the LCFloatVec collection that holds the flavour tag inputs.

TrueJetFlavourCollection Name of the LCIntVec Collection that contains the true jet flavours.

Filename-b_net-1vtx Output filename for the trained 1 vertex b-tag net.

Filename-c_net-1vtx Output filename for the trained 1 vertex c-tag net.

Filename-bc_net-1vtx Output filename for the trained 1 vertex c-tag (with only b background) net.

Filename-b_net-2vtx Output filename for the trained 2 vertex b-tag net.

Filename-c_net-2vtx Output filename for the trained 2 vertex c-tag net.

Filename-bc_net-2vtx Output filename for the trained 2 vertex c-tag (with only b background) net.

Filename-b_net-3plusvtx Output filename for the trained 3 or more vertices b-tag net.

Filename-c_net-3plusvtx Output filename for the trained 3 or more vertices c-tag net.

Filename-bc_net-3plusvtx Output filename for the trained 3 or more vertices c-tag (with only b background) net.

Author:

Mark Grimes (mark.grimes@bristol.ac.uk)

The documentation for this class was generated from the following file:

- NeuralNetTrainer.h

6.6 PerEventIPFitterProcessor Class Reference

Determine IP position and error from the tracks in an event by simple fit.

`#include <PerEventIPFitter.h>`

6.6.1 Detailed Description

Determine IP position and error from the tracks in an event by simple fit.

Inputs

Name	Type	Represents
InputRPCollectionName	ReconstructedParticle	Tracks to be fit

Outputs

Name	Type	Represents
VertexCollectionName	Vertex	Fitted IP

Description This processor fits a set of LCIO ReconstructedParticles (which must have an LCIO Track attached to be used) to a common point. This is performed by iterative fitting, with removal of the track with highest chi-squared at each iteration until the fit reaches the probability threshold. If only one track remains then the default IP position and error are used. The result is stored as an LCIO Vertex.

This processor is highly unoptimised and untuned, and may take a long time to execute on a large set of tracks.

Currently uses VertexFitterLSM (from ZVTOP) to perform fitting.

Parameters:

InputRPCollection Name of the ReconstructedParticle collection to be fit

VertexCollectionName Name of the output Vertex collection

DefaultIPPos Length 3 Float Vector of position (x,y,z) returned (as LCIO Vertex) if no fit is found

DefaultIPErr Length 6 Float Vector of covariance (lower symmetric) returned (as LCIO Vertex) if no fit is found

ProbabilityThreshold Once the vertex is above this probability it is returned

Author:

Ben Jeffery (b.jeffery1@physics.ox.ac.uk)

The documentation for this class was generated from the following file:

- PerEventIPFitter.h

6.7 PlotProcessor Class Reference

Creates some sample plots from the data calculated by the LCFI vertex package.

`#include <PlotProcessor.h>`

Protected Member Functions

- bool _passesEventCuts (lcio::LCEvent *pEvent)

  A function that contains all the event cuts - returns true if the event passes all of the cuts, false otherwise.

- bool _passesJetCuts (lcio::ReconstructedParticle *pJet)

  A function that contains all the jet cuts - returns true if the event passes all of the cuts, false otherwise.

6.7.1 Detailed Description

Creates some sample plots from the data calculated by the LCFI vertex package.

An example of getting the flavour tag results from the LCIO file and plotting an efficiency purity graph with them. Also plots a graph of jet energies for good measure.

The processor checks the specifed LCFloatVec collections for the flavour tag values "BTag", "CTag" and "BCTag" which are the names that FlavourTagProcessor stores its b tag, c tag and c tag (only b background) values in respectively.

These values are checked against the true jet flavour (from the TrueJetFlavour LCIntVec) and efficiency-purity values calculated for a range of cuts.

The jet energy is taken from the energy of the reconstructed particle used to represent the jet.

Getting Root output To output to a Root file instead of CSV files the processor has to be compiled with the USEROOT preprocessor flag defined. You could add "#define USEROOT" to the code, or more easily add the line

USERINCLUDES += -D USEROOT

to the userlib.gmk file that is in the Marlin directory. If Marlin is not already set up to use Root then you will also need to add the following lines (this assumes a fully working root installation):

USERINCLUDES += ‘root-config –cflags‘

USERLIBS += ‘root-config –libs‘

Input From the LCIO file, flavour tag variable values of:

"BTag" "CTag" "BCTag"

And

"JetType"

Output If the USEROOT preprocessor flag was defined when this processor was compiled, then the output will be a root file with the filename specified in the steering file. Otherwise, the efficiency-purity values will be output as comma separated values to the file +".csv", and the jet energies to +"-JetEnergies.csv".

Parameters:

JetCollectionName Name of the ReconstructedParticle collection that represents jets.

FlavourTagCollections Names of the LCFloatVec collections holding the Flavour tags, all tags in this list will be produced in one file for comparison

TrueJetFlavourCollection LCIntVec that contains the MC Jet flavour (from TrueJetFlavourProcessor)

OutputFilename The name of the file that will hold the output.

The documentation for this class was generated from the following file:

- PlotProcessor.h

6.8 RPCutProcessor Class Reference

Cuts ReconstuctedParticles(RPs) from a collection (or from a list of RPs held by another RP) based on several cut criteria.

`#include <RPCutProcessor.h>`

6.8.1 Detailed Description

Cuts ReconstuctedParticles(RPs) from a collection (or from a list of RPs held by another RP) based on several cut criteria.

Input

Name	Type	Represents
InputRCPCollection	ReconstructedParticle	Collection to be cut

Output

Name	Type	Represents
OutputRCPCollection	ReconstructedParticle	If WriteNewCollection=true contains the RPs that passed the cuts.

Description Based on several criteria this processor removes RPs from a collection, or if SubParticleLists = true then it removes RPs held by RPs in a collection.

Depending on WriteNewCollection, the Output is either the original collection with the RPs removed, or a new collection with the input collection remaining untouched.

NOTE - A track is cut if its ReconstructedParticle has no Track objects attached.

Most cuts follow a standard set of parameters:

a1_{CutName}Enable	If true the cut is enabled
a2_{CutName}CutLowerThan	If true RPs with a value lower than the cut value are cut, if false those higher than the cut value are cut.
a3_{CutName}CutValue	The value of the cut

(The letter and number index prefixed to each parameter are to ensure they stay together in the output of Marlin -x) The cuts that follow this scheme are:

Name	Description
Chi2OverDOF	Chi squared over degrees of freedom (Track::gethi2())
D0	Track D0 (Track::getD0())
D0Err	D0 Covariance (Track::getCovMatrix()[0])
Z0	Track Z0 (Track::getZ0())
Z0Err	Z0 Covariance (Track::getCovMatrix()[9])
PT	Track Pt (rPhi projection of Track::getMomentum())

Subdetector hits The cut on subdetector hits relies on information in Track::getSubdetectorHitNumbers() this is an array. The processor is told what order the detectors are in this array by parameter "g2_SubDetectorNames" which is the sting names of the detectors in the same order. The other parameters then use these names.

MC PID of Parents This cut uses MC information provided by the MCParticleRelation collection to cut particles whose parents have a PID in the list provided by parameter "h2_CutPIDS"

Bad parameters cut If enabled by "i1_BadParametersEnable" this cut removes tracks with nan covariances and parameters.

MC Vertex cut Experimental MC Cut - most likely removed in next release

Author:
Ben Jeffery (b.jeffery1@physics.ox.ac.uk)

The documentation for this class was generated from the following file:

- RPCutProcessor.h

6.9 TrueAngularJetFlavourProcessor Class Reference

Determine MC Jet Flavour by angular matching of heavy quarks to jets, also determine hadronic and partonic charge of jet.

`#include <TrueAngularJetFlavourProcessor.h>`

6.9.1 Detailed Description

Determine MC Jet Flavour by angular matching of heavy quarks to jets, also determine hadronic and partonic charge of jet.

The processor looks at all the PDG codes of all MC particles and recognises all particles containing b- and c-quarks. It then looks at the momentum of the heavy MC particles and at the momentum of the jets. The association is done by matching heavy flavour hadrons to the jet that is closest in agle. More than one heavy particle can therefore be associated with the same jet. If this happens the jet flavour is the flavour of the first particle in the parent-daughter chain associated with the jet. The pdg code of particle is subsequently used to determine the hadronic charge of the jet and the partonic charge of the heavy particle.

Input - Prerequisites Needs the collection of MCParticles. Needs the collection of Reconstructed Particles that represent the jets.

Output It writes to lcio the calculated flavours of the jets. This is stored in a collection of LCIntVec. By default the collection is called TrueJetFlavour.Writes also the PDG of the used particle and the hadronic and the partonic charge. By definition these collections are called: TrueJetPDGCode, TrueJetHadronCharge and TrueJetPartonCharge.

Parameters:

MCParticleColName Name of the MCParticle collection.

JetRPColName Name of the ReconstructedParticle collection that represents jets.

TrueJetFlavourCollection Name of the output collection where the jet flavours will be stored.

TrueJetPDGCodeCollection Name of the output collection where the PDG of the heavy particle associated to the jet is stored.

TrueJetHadronChargeCollection Name of the output collection where the hadronic charge is stroed.

TrueJetPartonChargeCollection Name of the output collection where the parton charge (charmness, bottomnes) is stored.

MaximumAngle Maximum value allowed between MCParticle and jet momentum expressed in degrees. If the closest jet is at a wider angle than MaximumAngle the MC particle does not get assigned.

Author:
Erik Devetak (e.devetak1@physics.ox.ac.uk),
interface by Ben Jeffery (b.jeffery1@physics.ox.ac.uk)

The documentation for this class was generated from the following file:

- TrueAngularJetFlavourProcessor.h

6.10 VertexChargeProcessor Class Reference

Calculates the Vertex Charge.

`#include <VertexChargeProcessor.h>`

6.10.1 Detailed Description

Calculates the Vertex Charge.

The processor calculated the vertex charge of a Decay Chain by using the tracks and the verteces present in the chain. Two logically slightly different algorithms are used depending on the hypothesis of a B or C quark Vertex. In the B hypothesis we include the inner verteces, in the C we do not include them. This choice is controlled by the parameter ChargeAllSecondaryTracks.

Input

- A collection of ReconstructedParticles that represents the jets in the event (obtained from a jet finder, say SatoruJetFinderProcessor, although in order not to break the reconstruction chain we suggest you run this after the flavour tagging. In this way the LCFI chain remains intact).
- A collection of vertices that contains the per event primary vertices; one for each event. (optional) This collection is filled in the vertex_lcfi::PerEventIPFitter processor.
- A collection of decay chains as filled by the the ZVTOPZVRESProcessor or ZVTOPZVKINProcessor.

Output The processor writes into the selected lcio output file. All the values calculated by the processor are saved in the same LCFloatVec collection. The default name of the output collection is Charge.Although this is changed in the steering files to something more appropriate, depending on B or C calculation. For more details see the interface documentation.

Parameters:

VertexChargeCollection collection where results will be stored.

ChargeAllSecondaryTracks include or exclude tracks in the inner vertices for the track attachment.

ChargeCloseapproachCut upper cut on track distance of closest approach to the seed axis in the calculation of the vertex charge variable, used by vertex_lcfi::TrackAttach.

ChargeLoDCutmax Cut determining the maximum L/D for the Charge, used by vertex_lcfi::TrackAttach (when calculating C-Charge)

ChargeLoDCutmin Cut determining the minimum L/D for the Charge, used by vertex_lcfi::TrackAttach (when calculating C-Charge)

Author:
Erik Devetak(erik.devetak1@physics.ox.ac.uk)

The documentation for this class was generated from the following file:

- VertexChargeProcessor.h

6.11 ZVTOPZVKINProcessor Class Reference

Find vertices in a jet using kinematic ZVTOP-ZVKIN algorithm.

`#include <ZVTOPZVKINProcessor.h>`

6.11.1 Detailed Description

Find vertices in a jet using kinematic ZVTOP-ZVKIN algorithm.

Input

Name	Type	Represents
JetRPCollectionName	ReconstructedParticle	Jets to be Vertexed (eg from SatoruJetFinderProcessor
IPVertexCollectionName	Vertex	Event Interaction Point (eg from PerEventIPFitterProcessor) - optional can be manually specified

Output

Name	Type	Represents
DecayChainCollectionName	ReconstructedParticle	Decay Chains (set of found vertices)
VertexCollection	Vertex	Found vertices
DecayChainRPTracks-CollectionName	ReconstructedParticle	Tracks used in Decay Chains and found vertices

Description This processor finds vertices in a set of LCIO ReconstructedParticles (usually a jet) using the algorithm ZVTOP(ZVKIN) Also see (INSERT LINK TO ZVTOP DOC) To be used each ReconstructedParticle must have an attached LCIO Track. Note it is imperative that the tracks have well formed and preferably accurate covariance matrices in d0 and z0. If the covariances are too small fake or no vertices may be found. Too large and vertices will be combined.

The algorithm also requires an interaction point in the form of an LCIO Vertex or a manually set position and covariance. - NOTE Only an ip at the origin (0,0,0) is supported as the ghosttrack has that origin, this will hopefully be upgraded in a future release.

The set of vertices forming a decay chain as output as set of LCIO Vertices and LCIO ReconstructedParticles for details see the interface documentation

For more details on algorihmic parameters see the ZVKIN paper "zvkin.ps" in the doc directory.

Author:
Ben Jeffery (b.jeffery1@physics.ox.ac.uk)

Parameters:

JetRPCollection Name of the ReconstructedParticle collection that represents jets

IPVertexCollection Name of the Vertex collection that contains the primary vertex (Optional)

DecayChainRPTracksCollectionName Name of the ReconstructedParticle collection that represents tracks in output decay chains

VertexCollection Name of the Vertex collection that contains found vertices

DecayChainCollectionName Name of the ReconstructedParticle collection that holds RPs representing output decay chains

ManualIPVertex If false then the primary vertex from VertexCollection is used

ManualIPVertexPosition Manually set position of the primary vertex (cm) - non origin IP not yet fully supported

ManualIPVertexError Manually set error matrix of the primary vertex (cm) (lower symmetric)

MinimumProbability If a vertex candidate has a probability below this it will not be considered - lower value results in more merging and lower vertex multiplicity

InitialGhostWidth Width in cm of the ghost inital ghosttrack also the smallest width it is allowed to have

MaxChi2Allowed The ghost track is widened until all forward jet tracks have a chi squared lower than this value

OutputTrackChi2 If true the chi squared contributions of tracks to vertices is written to LCIO

The documentation for this class was generated from the following file:

- ZVTOPZVKINProcessor.h

6.12 ZVTOPZVRESProcessor Class Reference

Find vertices in a jet using topological ZVTOP-ZVRES algorithm.

`#include <ZVTOPZVRESProcessor.h>`

6.12.1 Detailed Description

Find vertices in a jet using topological ZVTOP-ZVRES algorithm.

Input

Name	Type	Represents
JetRPCollectionName	ReconstructedParticle	Jets to be Vertexed (eg from SatoruJetFinderProcessor
IPVertexCollectionName	Vertex	Event Interaction Point (eg from PerEventIPFitterProcessor) - optional can be manually specified

Output

Name	Type	Represents
DecayChainCollectionName	ReconstructedParticle	Decay Chains (set of found vertices)
VertexCollection	Vertex	Found vertices
DecayChainRPTracks-CollectionName	ReconstructedParticle	Tracks used in Decay Chains and found vertices

Description This processor finds vertices in a set of LCIO ReconstructedParticles (usually a jet) using the algorithm ZVTOP(ZVRES) described in D. Jackson, NIM A388:247-253, 1997 Also see (INSERT LINK TO ZVTOP DOC) To be used each ReconstructedParticle must have an attached LCIO Track. Note it is imperative that the tracks have well formed and preferably accurate covariance matrices in d0 and z0. If the covariances are too small fake or no vertices may be found. Too large and vertices will be combined.

The algorithm also requires an interaction point in the form of an LCIO Vertex or a manually set position and covariance.

The set of vertices forming a decay chain as output as set of LCIO Vertices and LCIO ReconstructedParticles for details see the interface documentation

For more details on algorihmic parameters see the ZVTOP paper.

Author:
Ben Jeffery (b.jeffery1@physics.ox.ac.uk)

Parameters:

JetRPCollectionName Name of the ReconstructedParticle collection that represents jets

IPVertexCollectionName Name of the Vertex collection that contains the primary vertex (Optional)

DecayChainRPTracksCollectionName Name of the ReconstructedParticle collection that represents tracks in output decay chains

VertexCollection Name of the Vertex collection that contains found vertices

DecayChainCollectionName Name of the ReconstructedParticle collection that holds RPs representing output decay chains

ManualIPVertex If false then the primary vertex from VertexCollection is used

ManualIPVertexPosition Manually set position of the primary vertex (cm)

ManualIPVertexError Manually set error matrix of the primary vertex (cm) (lower symmetric)

IPWeighting Weight of the IP in the Vertex Function

JetWeightingEnergyScaling Scaling factor for Weight of the jet direction in the Vertex Function. Kalpha = Scaling $*$ JetEnergy

TwoTrackCut Chi Squared cut for making initial track pairs - chi squared of either track NOT sum

TrackTrimCut Chi Squared cut for final trimming of tracks from vertices

ResolverCut Cut to determine if two vertices are resolved

OutputTrackChi2 If true the chi squared contributions of tracks to vertices is written to LCIO

The documentation for this class was generated from the following file:

- ZVTOPZVRESProcessor.h

7 LCFIVertexPackage Page Documentation

7.1 LCIO Interface

Description of this package's use of LCIO to store results of processors.

7.1.1 Storage of Vertexing Result

The processors ZVTOPZVKINProcessor and ZVTOPZVRESProcessor

both store their results in the same way. They provide information on the decay vertices found, stored in a collection of LCIO Vertices and a collection of ReconstructedParticles, representing decaying particles that give rise to these vertices, as described in Frank Gaedeâ128153s Frank Gaede's forum post

In LCIO, Vertices and ReconstructedParticles, are connected as follows:

- Each decay vertex found has a corresponding LCIO::Vertex.
- Each decay vertex found also has a corresponding LCIOReconstructedParticle which respresents the decaying particle and holds kinematic information.
- This accompanying decaying ReconstructedParticle is accessed though Vertex::getAssociated-Particle()
- The descendant tracks which are produced by the particle are attached to the decaying Reconstructed-Particle and accessed through ReconstructedParticle::getParticles()
- Each ReconstructedParticle points to its start and end vertex (if any) through Reconstructed-Particle::getStartVertex() and ReconstructedParticle::getEndVertex()

In essence we end up with three types of objects with links between them; Vertices, Decaying ReconstructedParticles, Stable ReconstructedParticles.

Note that the information stored in the ReconstructedParticles created using ZVTOP information differs from the one of those created by the track reconstruction code. The getStartVertex and getEndVertex methods permit building up a representation of a heavy flavour decay chain. In the current ZVTOP version, this is reconstructed as follows: vertices are sorted by increasing radius from the IP, the start vertex of the accompanying ReconstructedParticle is set to point to the previous vertex in the collection in this order. Note that this is only an approximation of the decay chain in an actual physics event, which may differ or be more complex (e.g. one decay vertex may give rise to two instable particles. If correctly reconstructed, their ReconstructedParticles would point to the same start vertex. Of the corresponding ZVTOP vertices, only the one closer to the IP will point to its correct start vertex, while the further one will point to the nearer one instead.

As each RP points to its vertices, to store more than one vertexing result it is necessary to take a copy of the set of RPs that are in each vertex. Each copy points to the original unique RP through Reconstructed-Particle::getParticles()[0].

A master ReconstructedParticle object is created that points to all decaying and stable Reconstructed-Particles in the decay chain though ReconstructedParticle::getParticles(). The ReconstructedParticle::get-StartVertex() is set to the first Vertex in the decay chain (usually the IP). This is the main object for accessing the decay chains as it allows one to iterate over all the ReconstructedParticles contained within.

These objects are then stored in three collections:

- DecayChainRPTracksCollectionName: default name: ZV***DecayChainRPTracks, stores decaying RPs and copies of input RPs for all decay chains.
- VertexCollection: default name: ZV***Vertices, stores the Vertices for all decay chains.
- DecayChainCollectionName: default name: ZV***DecayChains, stores the master Reconstructed-Particle DecayChainCollectionName, in the same order as the Jets that were input to the algoithm in JetRPCollection

Example If you wanted to print out the d0 of each LCIO::Track in each Vertex of a decay chain found for the second jet in the jet collection:

```
//Get the decay chain master RP collection and get the second decay chain
LCCollection* DecayChainRPCol = evt->getCollection( _DecayChainRPColName );
ReconstructedParticle* DecayChainRP = dynamic_cast<ReconstructedParticle*>(DecayChainRPCol
//Make a list of vertices
vector<lcio::Vertex*> LCIOVertices;
//Add the primary first
LCIOVertices.push_back(DecayChainRP->getStartVertex());
//Loop over RPs to find all the other vertices
vector<ReconstructedParticle*> RPs = DecayChainRP->getParticles();
for (vector<ReconstructedParticle*>::const_iterator iRP = RPs.begin();iRP < RPs.end();++iR
{
         lcio::Vertex* MyVertex = (*iRP)->getStartVertex();
         if(MyVertex)
         {
                 vector<lcio::Vertex*>::const_iterator it = find(LCIOVertices.begin(),LCIO
                 if (it == LCIOVertices.end())
                 {
                         LCIOVertices.push_back(MyVertex);
                 }
         }
}
//Loop over the vertices
for (size_t i = 0; i < LCIOVertices.size(); ++i)
{
        std::cout << "Vertex " << i << "has d0's:" << std::endl;

        //Get the Vertex RPs from the Vertices decaying RP
        std::vector<ReconstuctedParticle*> VertexRPs = LCIOVertices[i]->getAssociatedParti
        for (size_t j = 0; j < VertexRPs.size(); ++i)
        {
                //Get the Track (remember the Vertex RP is a copy which points to the orig
                std::cout << VertexRPs[j]->getParticles()[0]->getTracks()[0]->getD0() << "
        }
        std::cout << std::endl;
}
```

Storage of track chi squareds If OutputTrackChi2 is set to true the vertexing processors will output the chi squared each track contributes to its vertex.

This is stored in a collection named: TrackRPCollectionName+"TrackChiSquareds" ie. The name of the first collection appended with "TrackChiSquareds"

The chi squareds are stored as a collection of LCFloatVecs in the same order as the tracks in DecayChainRPTracksCollection. Currently the LCFloatVec contains one value - the chi squared in the start vertex, but the end vertex could be supported if needed as a second value.

7.1.2 Storage of Flavour Tag Inputs and Flavour Tag Result

The lists of variables produced by the Flavour Tag and Inputs are stored in collections of LCFloatVecs, with one LCFloatVec per jet. Within the LCFloatVec, the names and order of the variables are stored in the parameter of the run header as a string vector stored under the name of the LCFloatVec collection. Note that the same LCFloatVec is used for the flavour tag inputs, the vertex charge and further additional variables.

Example Get the secondary vertex probability for the second jet:

```
//Get the variable names in the FlavourTagInputsCollection
```

```
std::vector<std::string> VarNames;
(pRun->parameters()).getStringVals(_FlavourTagInputsCollectionName,VarNames);
//Convert this to a convenient map
std::map<std::string,unsigned int> IndexOf;
for (size_t i = 0;i < VarNames.size();++i)
{
        IndexOf[VarNames[i]] = i;
}
//Get the inputs for the second jet
lcio::LCCollection* pInputs=pEvent->getCollection( _FlavourTagInputsCollectionName );
LCFloatVec* FTInputs = dynamic_cast<lcio::LCFloatVec*>( pInputs->getElementAt(1) );
</PRE>
Get the Secondary Probability by indexing the LCFloatVec
<pre>
double SecProb = (*FTInputs)[IndexOf["SecondaryVertexProbability"]];
```

Ben Jeffery - b.jeffery1@physics.ox.ac.uk

7.2 Description of Track Selection Cuts

The track selection cuts are mainly from LC note LC-PHSM-2000-021 with minor changes.

These have not yet been optimised for full reconstruction.

7.2.1 IP Fitting Cuts

Details to follow - for now see steering file ipfit.xml

7.2.2 ZVTOP Cuts

Details to follow - for now see steering file zvres.xml

7.2.3 FlavourTagInputs Cuts

Details to follow - for now see steering file fti.xml

7.3 Usage tutorials

7.3.1 Event reconstruction required

The vertex finder and flavour tagging software expects a set of tracks - usually the tracks belonging to one jet in an event - as input. Initially, it is therefore necessary to

- obtain digitized hits
- reconstruct tracks
- run the jet finder
- reconstruct the event vertex / IP, if running the flavour tagging code
- determine the MC true jet flavour if studying flavour tag purity, efficiency
- determine the MC true quark charge to study performance of quark charge reconstruction

The performance presented at the ILC software workshop, Orsay, May 2007 was obtained with the digitization and track cheating code developed by Alexei Raspereza. For the jet finding the Durham algorithm with a y- cut of 0.04 was used, as implemented in the Satoru jet finder within MarlinReco.

An example of the steering for the first three of the above event reconstruction steps, used to obtain the shown results, is given in the steering file cheattracks+jetfind.xml. **This example assumes that the detector geometry LDC01_05Sc is used - hit collection names will need to be changed in the steering file when running with a different geometry.**

The event vertex or IP is needed for calculating the default flavour tag inputs. Since no code to do this was yet available in MarlinReco, the LCFI group implemented a procedure similar to the one used in the SGV fast MC . This should only be considered as spaceholder for a future improved procedure in which for the vertex position in the x-y-plane one would average over tracks from a number of consecutive events. The current algorithm is implemented in the PerEventIPFitterProcessor and run by calling Marlin with the steering file ipfit.xml.

The true flavour of a jet is based on the MC record in the event. It searches the event for the leading hadron, and if this is a heavy flavour particle determines which of the jets in the event is closest in angle. For heavy flavour jets, based on the leading hadron MC information also the quark charge of the heavy flavour hadron is determined. Further details can be found in the documentation of the TrueAngularJetFlavourProcessor. This part of the reconstruction is performed by running Marlin with the truejetflavour.xml steering file.

7.3.2 How to run the vertex finder ZVTOP

Make sure the necessary event reconstruction steps have been run in advance. The vertex finder ZVTOP has two branches: the standard branch ZVRES, based purely on topological information, and the more specialized branch ZVKIN, which uses kinematic information from heavy flavour decay chains in addition. Flavour tagging for ILC physics simulation so far has been performed using ZVRES only. The use of ZVKIN for this purpose is yet to be explored. It should also be noted that ZVKIN parameters have not yet been tuned for ILC conditions. This vertexing algorithm is generally less tested and optimised in terms of runtime than the ZVRES branch.

For each of the two branches, a dedicated steering file is provided, named zvres.xml and zvkin.xml, respectively. The output of ZVTOP consists of one collection storing the vertices that were found, one collection holding the corresponding ReconstructedParticles decaying at these vertex positions and one collection containing one ReconstructedParticle per jet which gives access to the full decay chain. Further details on the storage of the output can be found in the LCIO Interface.

7.3.3 How to flavour tag jets

Before running the flavour tag, make sure the necessary event reconstruction processors and ZVTOP have been run. In the default flavour tagging algorithm information from the vertex finder is both directly used as input for the tagging neural networks and to determine, which set of neural networks is used. You can either begin by training new neural nets or use pre-trained nets. One set of nets, trained with input from the fast MC SGV, is provided with the Vertex Package. These nets are available from the new repository tagnet for flavour tag neural nets in the Zeuthen CVS repository .

Flavour tag inputs are calculated by running Marlin with the steering file fti.xml. As input it requires the LCIO file with information from ZVTOP and the IP fit processor (default filename zvresout.slcio). The flavour tag inputs are written out into collections of LCFloatVec's as described in more detail in the LCIO Interface.

The neural network output values are obtained from an independent Marlin processor; the corresponding example steering file is ft.xml. It requires the LCIO file written out in the Marlin run with fti.xml,

which is by default called ftiout.slcio. The default algorithm is based on nine neural networks, three for each of the three classes of jets, namely those with 1, 2 and 3 or more vertices found by ZVTOP. It thus provides three output values for each jet: a b-tag value, a c-tag value for arbitrary jet sample composition and a c-tag value assuming the background is known to consist of b-jets only (yielding improved c-tag purity). These are stored as components "BTag", "CTag" and "BCTag" of an LCFloatVec collection. The collection name - FlavourTag by default - can be specified in the steering file. This collection, along with the information from the input-LCIO file, is written into an LCIO output file, named flavourtagout.slcio in the example.

7.3.4 How to evaluate and plot flavour tag purity vs efficiency

Flavour tag purity as function of efficiency is a measure of how well the flavour tag performs for a certain mix of jet flavours. The Vertex Package provides two processors to calculate purity and efficiency: the PlotProcessor and the LCFIAIDAPlotProcessor. An example how to call these processors is given in the steering file `ftplot.xml`.

The PlotProcessor writes out a table with the resulting efficiency and purity values as well as the cut values on the neural net outputs these correspond to, into a comma separated value file. Additionally, if the root library is linked in at compile time by defining the preprocessor flag USEROOT, the corresponding graphs are written out into a root-file. These graphs can be plotted using the root-macro MakePurityVsEfficiency-RootPlot.C provided in the macro directory. This macro also allows to plot the resulting graphs from two different runs onto the same canvas to compare performance.

The LCFIAIDAPlotProcessor provides further diagnostic tools for the flavour tag. Since different neural networks are used for the cases that 1-, 2- or at least 3 vertices are found, purity and efficiency are calculated separately for these cases. Graphs in AIDA format are created for purity vs efficiency and for the flavour leakage (i.e. the "efficiency" of tagging the wrong flavour) as function of efficiency - e.g. the leakage of usd-jets into the c-tagged sample. Also, distributions of all flavour tag input variables, the vertex charge and of the neural net output variables are created separately for the 1-, 2- and 3 or more vertex case. Optionally information can be written out into an AIDA tuple and in text format, for further details see the LCFIAIDAPlotProcessor documentation. Output from the LCFIAIDAPlotProcessor can be plotted using the python script FlavourTagInputsOverlay.py from the macro directory.

7.3.5 How to train new neural networks for flavour tagging

The Vertex Package is very flexible, so it is straightforward to entirely change the flavour tagging procedure. This is just an overview of changes possible, pointing out where to find further details.

In the simplest case that requires retraining, the tagging procedure itself is not changed. For example, one might want to retrain the networks after tuning ZVTOP, or changing other boundary conditions, such as track selection or composition of the input sample (as may happen when studying a specific physics channel). For this purpose, an example steering file `trainNeuralNets.xml` showing how to run the network training processor, is provided. You may choose to retrain only some of the networks - each can be enabled in the steering file independently of the others.

Changes to the tagging algorithm will require writing new processors and recompiling Marlin. A simple example would be the change of the network architecture, such as number or type of nodes and / or internal layers. Please refer to the neural network documentation for details on how networks can be defined. As long as the number and choice of input variables remains unchanged, only the training processor NeuralNetTrainerProcessor will have to be modified (or a new one added).

More complex modifications of processors are necessary when changes of the input variables are involved. This will require changes to at least three processors: the FlavourTagInputsProcessor calculating the inputs, the NeuralNetTrainerProcessor for training and the FlavourTagProcessor for obtaining the network outputs in the subsequent analysis. If further variables are to be added, this might additionally require some famil-

iarity with the internal structure of ZVTOP. The current way of writing out the inputs into an LCFloatVec collection permits further variables to be added in a straightforward way - make sure to also update the section of the processor called at the start of the Marlin run, where the variable names are defined.

Changes to the input variables used in the training processor obviously require that the corresponding changes also be made in the processor obtaining the network outputs. In order to keep track of networks used and to allow shared use of networks within the community, a new `cvs repository tagnet` has been set up in the `Zeuthen cvs area` . **We strongly recommend submission of networks used for your studies to this repository**, along with a description of the boundary conditions for training - make sure these descriptions are as complete as possible,including details on training sample and any changes to the defaults made (track selection, ZVTOP settings, addition of input variables, if possible with a reference to the code, with which these have been obtained). Providing this information will save time when it comes to comparisons of analyses made within different frameworks / detector concepts etc.

7.3.6 How to determine the vertex charge

From version v00-02-02 onwards, the vertex charge is reconstructed in a dedicated processor, the VertexChargeProcessor, and stored in its own LCFloatVec collections. Two vertex charge variables are calculated, one assuming the jet is a b-jet, the other assuming it is a c-jet. Two steering files are provided: `Bvertexcharge.xml`, to be run first, producing an output LCFloatVec collection which is by default named Bcharge, and `Cvertexcharge.xml`, to be run subsequently and producing an output LCFloatVec collection with default name Ccharge.

7.3.7 How to apply track selection cuts

The various track selection criteria used in the code - which differ between IP determination (cf IP Fitting Cuts), ZVTOP (cf ZVTOP Cuts) and the calculation of the flavour tag inputs (cf FlavourTagInputs Cuts) - are implemented by a flexible RPCutProcessor that runs on reconstructed particles, containing the tracks in question.

7.4 Neural Net Package

7.4.1 Acknowledgements

All code in this neural net package was written by David Bailey of the University of Manchester.

7.4.2 Assumed knowledge

- Standard Template Library (STL) vectors.

7.4.3 Remarks

- All neural net classes are in the namespace `nnet`.
- This is by no means a complete guide to every feature available, at present at least.

7.4.4 Basic principles of an artificial neural network

This is a very basic introduction to the principles of a neural network (geared specifically at the way this package works). If you have any experience with neural networks you can safely skip this section.

Neurons are created to accept an arbitrary number of inputs, and based on these provide a single output value. The output is given by the neurons $thresholdfunction$, which can be any given function of the neurons $activationvalue$ (see the Neuron Descriptions for the functions actually provided with this package).

The activation value is given by multiplying each input by a pre calculated $weight$ depending on how important that input is, and summing these results. Each neuron can also be given a bias, depending on how important that neuron is to the network, but more on that later.

Calculating these weights is the important part, and is what differentiates a well performing network from a bad one. This process is known as $training$, and is performed by a training algorithm (see Training the network for the algorithms provided here). Basically, you provide the training algorithm with a set of data that you know the answers to (the result you would want the network to give you), and it changes the weights to give the best possible results for all the elements in the data set.

As a basic example, imagine a network composed of a single neuron that tells you if a food is bad for you or not. Say it is set up with three inputs, fibre content, fat content and colour. For simplicity, lets give the neuron a linear threshold function, so just a function that multiplies the activation value by a set constant, say k. The output of the network would be

$$output = f(activationvalue) = k \times (fibrecontent \times weight_{fibre} + fatcontent \times weight_{fat} + colour \times weight_{colour})$$

The network is useless until the values of the weights are adjusted so that they give an accurate output. To do this, a large database of foods is required where the properties of colour, fibre and fat content are known, as well as some reliable value as to how healthy the food is. The training algorithm then modifies the weights to try and get the best match of the output to the expected value for each food in the database. When somebody comes along with a new food, its properties can be put into the network and a (hopefully) reliable value as to how healthy it is pops out the other end.

Ideally, once trained, the weight given to colour will be zero since that is completely irrelevant (ignoring artificial colourings). However, if the training sample has just a few blue foods, which just happen to be bad for you, then the training algorithm will wrongly ascribe a high weight.to the colour input. Also, if the training sample foods have pretty similar fat and fibre contents, but are radically differently healthy (say, maybe due to salt content), then the training algorithm will probably be unable to make any sense of the sample, and give useless weights. This emphasises the need to select a large and varied training sample (as well as setting up the network with meaningful inputs in the first place).

Realistically, a network will be composed of many neurons so that all 'cross effects' between the inputs are taken into account (where a weighting for one input needs to depend on other inputs as well). Here, the network would be built up with layers of neurons where the input for each neuron in a layer is the output from each neuron in the layer before. The final layer would have just one neuron, so that you get just one output for the network.

7.4.5 Creating and training a new neural net

The method used to create a new network varies slightly depending on the algorithm used to train it. Sections Building the neuron layers and Creating the network describe how to setup the network ready for training, which is common to all training algorithms. The `BatchBackPropagationAlgorithm`, `BackPropagationCGAlgorithm` and `GeneticAlgorithm` algorithms require the training data to be pre-stored in a `nnet::NeuralNetDataSet` class (section Building the training sample), and will train themselves over the whole data set. `BackPropagationAlgorithm` on the other hand performs one training step at a time to provide more control over each training step.

Descriptions of the algorithms are given in Training the network.

7.4.5.1 Building the neuron layers Only simple nets can be built, where each neuron takes the outputs of all of the neurons in the previous layer as its inputs. Details about the neurons behaviour are given in Neuron Descriptions.

There are two methods, one where neurons can have different types, and a simpler one where all of the neurons have the same type.

All neurons of the same type Building the neuron layers simply consists of creating an STL vector of integers with the number of neurons in each layer, including the output layer but excluding the input layer. The type of all of the neurons is set later when the network is built. So if a network takes 3 inputs, has two hidden layers with 6 neurons and 4 neurons respectively, and 2 outputs the layers would be set like this:

```
std::vector<int> neuronsInLayer;
neuronsInLayer.push_back( 6 );
neuronsInLayer.push_back( 4 );
neuronsInLayer.push_back( 2 );
// The number of inputs is set later.
```

Neurons with different types These are set in a similar way, but instead of integers specifying the number of neurons in each layer, another STL vector of strings specifying the name of each neuron type is used, with the number of neurons set by the size of the vector. Currently available types (descriptions are given in Neuron Descriptions) are:

```
LinearNeuron
SigmoidNeuron
TanSigmoidNeuron
```

So if a network as in the previous example is to be built (with arbitrary neuron types):

```
std::vector<std::string> layer1;
layer1.push_back( "LinearNeuron" );
layer1.push_back( "SigmoidNeuron" );
// ...and so on until there are six names in the vector -> six neurons in the layer

std::vector<std::string> layer2;
layer2.push_back( "TanSigmoidNeuron" );
// ... and so on another three times

std::vector<std::string> outputlayer;
outputlayer.push_back( "LinearNeuron" );
outputlayer.push_back( "LinearNeuron" );

std::vector< std::vector<std::string> > neuronNames;
neuronNames.push_back( layer1 );
neuronNames.push_back( layer2 );
neuronNames.push_back( outputlayer );
// The number of inputs is set later.
```

7.4.5.2 Creating the network Once the layer structure has been set up, the network can be created as follows, depending on which layer specification method was used.

All neurons of the same type The type of the neurons is set by creating a neuron builder and passing its address to the network constructor. The names of available builders are the same as for the neurons, but with "`Builder`" on the end, for example "`nnet::LinearNeuronBuilder`" will build "`Linear-Neuron`"s.

```
int numberOfInputs=3; // number of inputs set when the network is created
nnet::SigmoidNeuronBuilder myNeuronBuilder; // the type of ALL the neurons is set here

// now create the network using the neuronsInLayer vector from before
nnet::NeuralNet sameNeuronsNet( numberOfInputs, neuronsInLayer, &myNeuronBuilder );
```

Neurons with different types All that is needed here is the STL vectors of neuron names previously initialised and the number of inputs.

```
int numberOfInputs=3; // number of inputs set when the network is created

// now create the network using the neuronNames vector from before
nnet::NeuralNet differentNeuronsNet( numberOfInputs, neuronNames );
```

Random number generation In each case, you can optionally use a random seed for the random number generator that sets the initial neuron weights by adding a boolean parameter at the end of the constructor arguments. Default is to use a random seed.

```
//don't use a random seed
nnet::NeuralNet sameNeuronsNet( numberOfInputs, neuronsInLayer, &myNeuronBuilder, false );

//use a random seed (default)
nnet::NeuralNet differentNeuronsNet( numberOfInputs, neuronNames, true );
```

Note that currently **the default implementation uses `rand()` for random numbers**, and the random seed is taken from the current system time. If you require something more sophisticated modify the "`RandomNumberUtils.h`" file.

7.4.5.3 Building the training sample A network can be trained without setting out the data sample into a `nnet::NeuralNetDataSet` using the `BackPropagationAlgorithm`, but large scale training is easiest using the other algorithms so this will be covered here.

Data is added to the `nnet::NeuralNetDataSet` by calls to `addDataItem`, with a vector of inputs and a vector of the expected outputs as the arguments. All items in the data set must have the same number of inputs and outputs; the first item you add sets these sizes for the whole data set. If you try and add an item where the input or output vectors are not the correct size, then an error will be printed to standard error and the item will be ignored.

For example:

```
// for a network to calculate the probability a given animal is a donkey
// with inputs, in order, of "number of legs", "height" and "length of tail"
nnet::NeuralNetDataSet \label{donkeyNetDataSet}animalSample;
std::vector<double> inputs;
std::vector<double> output;

// for donkey 1
output.push_back( 1 ); // I know for certain this animal is a donkey
inputs.push_back( 4 ); // it has four legs
inputs.push_back( 1.45 ); // it is 1.45m tall
inputs.push_back( 0.32 ); // it has a tail 32cm long

// This call sets the sample to demand 3 inputs and 1 output
animalSample.addDataItem( inputs, output ); // Add this animal to the training sample

inputs.clear(); // Clear all the data so that the vectors can be reused
output.clear();
```

```
//for Geoff
output.push_back( 0 ); // I know for certain Geoff isn't a donkey
inputs.push_back( 2 ); // he has 2 legs
inputs.push_back( 1.82 ); // he is 1.82m tall

// This will not be added, because there are not enough inputs!
animalSample.addDataItem( inputs, output ); // Add Geoff to the training sample

// To add Geoff to the training sample we need to match the number of inputs
inputs.push_back( 0 ); // Geoff's tail is 0cm long
animalSample.addDataItem( inputs, output ); // This will now work
```

7.4.5.4 Training the network To train the network, a training algorithm is created with the network to be trained as the constructor argument, and a call to train is made with the number of training epochs and the training data. Currently available training algorithms are:

```
nnet::BackPropagationAlgorithm
nnet::BackPropagationCGAlgorithm
nnet::BatchBackPropagationAlgorithm
nnet::GeneticAlgorithm
```

Training with BackPropagationAlgorithm The Back Propagation Algorithm uses the back propagation method for determining the gradient of the error, and then gradient descent to modify the weights to minimise the error. It is very similar to the BatchBackPropagationAlgorithm except that it only performs one training step at a time to give more control over the training parameters at each step.

The algorithm class is constructed by giving it the network to be trained, and optionally values for $learningRate$ and $momentumConstant$ (defaults are 0.5 for both). The $learningRate$ parameter is just a multiplier applied to the calculated change required for each weight, larger values will mean the weights will change more rapidly with each step. The previous steps' calculated change is also added to the current steps', but multiplied by the $momentumConstant$ value. A value greater than or equal to one for this would stop the algorithm settling on a maximum because (at least) the full previous change is added as well.

The `train` method is used to perform one training run, and returns the error. It takes a vector of the inputs and a vector of the required outputs, so if the first data item in the previous example is used for the step:

```
std::vector<double> inputs;
std::vector<double> output;

// for donkey 1
output.push_back( 1 ); // I know for certain this animal is a donkey
inputs.push_back( 4 ); // it has four legs
inputs.push_back( 1.45 ); // it is 1.45m tall
inputs.push_back( 0.32 ); // it has a tail 32cm long

//set learningRate to 0.6 and momentumConstant to 0.4
nnet::BackPropagationAlgorithm myTrainer( myPreviouslyCreatedNetwork, 0.6, 0.4 );

double errorForThisStep=myTrainer.train( inputs, output );
```

Training with BatchBackPropagationAlgorithm This is essentially the same as `BackPropagationAlgorithm`, except it is supplied with a training sample which it will loop over itself. It can also be set do so repeatedly by specifying the number of epochs to run when calling the `train` method. The error for the most recent epoch is returned by `train`, and the errors from previous epochs can be retrieved as a vector with the `getTrainingErrorValuesPerEpoch` method.

```
//created in the same way as for the single step version
```

```
nnet::BatchBackPropagationAlgorithm myTrainer( myPreviouslyCreatedNetwork ) //use default learningRate and

//train using the sample in the previous example and 50 epochs
double finalError=myTrainer.train( 50, animalSample );

//get the errors from previous epochs to see how things are converging
std::vector<double> errors=myTrainer.getTrainingErrorValuesPerEpoch();
```

Training with BackPropagationCGAlgorithm This algorithm is similar to `BatchBackPropagationAlgorithm` except that it uses the conjugate gradient method to minimise the error instead of gradient descent. It offers three types of function to calculate the β coefficient (see any detailed description of conjugate gradients) selected using the `setBetaFunction` method. These are "FletcherReves", "PolakRibiere", and "ConjugateGradient", used as an enumeration as quoted. The default is FletcherReves.

```
//created in the same way as for the single step version
nnet::BackPropagationCGAlgorithm myTrainer( myPreviouslyCreatedNetwork )

//set the beta function to Polak-Ribiere
myTrainer.setBetaFunction( nnet::BackPropagationCGAlgorithm::PolakRibiere );

//train using the sample in the previous example and 500 epochs
double finalError=myTrainer.train( 500, animalSample );
```

7.4.6 Obtaining results

To get results from the neural network, the output method takes the inputs as an STL vector of doubles, and provides the results as an STL vector of doubles. So to determine if some animal is a donkey using a network trained from data of form of the data set in the previous example:

```
// Using the previously trained net "donkeyNet" to
// find out if Fido is a donkey...
std::vector<double> inputs;
inputs.push_back( 4 ); // Fido has four legs
inputs.push_back( 0.54 ); // he is 54cm tall
inputs.push_back( 0.25 ); // his tail is 25cm long

std::vector<double> output=donkeyNet( inputs );
// The network was set to have only one output, if there were any
// more then they would be the higher elements in the vector.
std::cout << "Likelihood Fido is a donkey= " << output[0] << std::endl;
```

7.4.7 Saving a neural net to disk

Neural nets can either be saved as plain text or XML files, with the default being XML. To choose between the two make a call to `NeuralNet::setSerialisationMode` with either `nnet::NeuralNet::PlainText` or `nnet::NeuralNet::XML`.

The network can then be saved to disk by passing a C++ stream to serialise. For example:

```
myNeuralNet.setSerialisationMode( nnet::NeuralNet::XML );
std::ofstream outputFile( "/home/me/myNeuralNet.xml" );
myNeuralNet.serialise( outputFile );
```

The network can also of course be printed to standard output by calling `serialise( std::cout )`.

7.4.8 Loading a neural net from disk

A network can be loaded from disk by simply passing the filename and the serialisation mode as the constructor arguments. If the serialisation mode is not specified then XML is assumed. For example:

```
nnet::NeuralNet myXMLNet( "/home/me/myNeuralNet.xml", nnet::NeuralNet::XML );
nnet::NeuralNet anotherXMLNet( "/home/me/myOtherNeuralNet.xml" );//XML is the default
nnet::NeuralNet myTextNet( "/home/me/myNeuralNet.txt", nnet::NeuralNet::PlainText );
```

Note that there is currently no error checking when loading XML nets, **if you try and load a plain text net as XML, or the file is not properly structured you will get a segmentation fault or runaway memory allocation**. This is still being looked into.

7.4.9 Neuron Descriptions

The output from a neuron is given by its *threshold function* which is unique to each type of neuron. This is a function of the neurons *activation value*, which is calculated the same way for each type.

The activation value a for a neuron with N inputs, i_n , each with weights w_n is given by

$$a = \sum_{n=1}^{N} i_n \times w_n + b \times w_b$$

Where b is a bias that can be assigned to a particular neuron (and w_b the bias' weight). The weights are initially random, and are then fine tuned by the training algorithms to try and give the desired output. The bias is set when the neuron is created but that process is done internally by the neuron builders. All current neuron builders set the bias to -1.

Some of the neurons have methods to change their behaviour. To get the neuron pointer to call these methods use "`NeuralNet::layer(layerNumber)->neuron(neuronNumber)`", where the numbers of available layers and neurons per layer can be found with "`NeuralNet::numberOfLayers()`" and "`NeuralNet::layer(layerNumber)->numberOfNeurons()`" respectively.

7.4.9.1 Linear Neuron Linear neurons give, as the name suggests, a linear output between -1 and +1 with a gradient of $1/slopeEnd$. The value of *slopeEnd* can be set using the `LinearNeuron::setSlopeEnd(newValue)` method. If the output is greater than +slopeEnd, then the output is limited to +1; any less than -slopeEnd and the output is limited to -1. Anywhere in between gives the expected linear output of $activationvalue/slopeEnd$.

7.4.9.2 Sigmoid Neuron The sigmoid neuron gives sigmoid (sort of resembles a slanted "S") output, o, of between 0 and 1 from the function

$$o = \frac{1}{1 + e^{-a/r}}$$

Where r, the "response", can be set with the `SigmoidNeuron::setResponse(newValue)` method. The default is 1.

7.4.9.3 Tan Sigmoid Neuron This neuron gives a similar looking output to the sigmoid neuron, but between -1 and 1. The value is given by

$$o = tanh(s \times a)$$

Where the value of s (the “scale”) can be set with the `TanSigmoidNeuron::setScale(newValue)` method. The default is 1.

Author:
Mark Grimes (mark.grimes@bristol.ac.uk)

The main algorithm for ZVRES is VertexFinderClassic and for ZVKIN VertexFinderGhost

"classic" denotes implementation as in Dave Jacksons original ZVTOP paper - Nuc. Inst. Meth. A 388 (1997) 247-253

Please see the accompianing file zvtop.pdf

Index