Abstract

The exploitation of hadronic final states played a key role in the successes of all recent HEP collider experiments, and the ability to use the hadronic final state will continue to be one of the decisive issues during the LHC era. Jet finders play a key role in this endeavour, and strong activities in this area have been progressing since many years.

We use the advantages of the abstractions enabled by object technology, in order to extend the flexibility of the jet finding possibilities during analysis. The jet finder library described provides a comprehensive framework for jet-finding. It starts from any kind of consistent collection of reconstruction objects or particles, and results in a collection of user-defined jet class instances.

We provide the jet-finding framework for CMS, and use this opportunity to exercise industrial software engineering processes to establish the proof of concept of physics object reconstruction for hadronic final states.
1 Noun List
In order to go by the book, we base our design and analysis phase on the noun list in the application domain. The nouns we find are

- Jet finder
- Jet algorithm
- Jet variable
- Jet collection
- Jet
- Cluster, track, particle, cell

2 Requirements
User requirements on jets:

JET-1 A jet shall be able to construct from various inputs.
JET-2 A jet shall be able to compute jet energy.
JET-3 A jet shall be able to compute jet mass.
JET-4 A jet shall be able to compute jet rapidity.
JET-5 A jet shall be able to compute jet azimuthal angle.
JET-6 A jet shall be able to compute jet polar angle.
JET-7 A jet shall be able to provide access to its constituents.

All other requests, like b-tag or estimators for \( \tau \) identification, seem specific to a given analysis, and the design is required to allow for extendibility of the system such that the jet-finder can be used together with any user-defined jet class.

User requirements on jet-finders:

JETF-1 A jet finder shall allow for input filters.
JETF-2 A jet finder shall provide the possibility of using different jet-algorithms during one job.
JETF-3 A jet finder shall be able to run on any consistent set of inputs (cells, tracks, clusters, particles, tracks and clusters, etc.)
JETF-4 A jet finder shall be able to provide access to any type of jets.

3 Responsibilities and mapping to design elements
3.1 Responsibilities
The above list of requirements served as starting point for designing the functionality in the abstractions of the system. We assign the following responsibilities to the main classes in the system.

Jet finder
The jet finder is the entity that steers the jet finding and manages the message passing between the different classes in the system.

Cluster, track, particle, cell, etc.
Cluster, track, particle, or cell are examples of the input classes to the jet-finder.
**Jet algorithm**
The Jet algorithm is a class that knows to build jets from a concrete set of input class instances.

**Jet**
The Jet in general is a user-designed class. Its design might depend very much on the objective of the analysis (like QCD, B-physics, $H \rightarrow WWjj$, etc.). The Jet class is responsible to guarantee access to the elementary properties of the jets and their constituents.

**Jet collection**
The Jet collection is able to hold a list of jets, and to allow access to these jets.

### 3.2 Mapping of user requirements on design elements

The requirements can be mapped to design elements as described in the following. Note that in general more than one possibility is considered before a decision is taken, but that the set of possibilities considered is not meant to be complete.

#### 3.2.1 Requirements on Jet finder

A jet finder shall allow for input filters:
- The jet finder can be templated with the filter. This allows for exactly one filter.
- The jet finder can have a pointer to a filter base class that defines the protocol for the filter. This enables the possibility to have a cascade of filters. A registration mechanism, along with a clearing mechanism is necessary. It also implies a coherent interface for the inputs on which the filter can run. A base class filter with a well defined interface is needed, which implies the need for a JetableObject class.
- An user defined input generator class the interface of which is defined by a base class can provide this functionality.

A jet finder shall provide the possibility of using different jet-algorithms during one job:
- To separate the jet-algorithm from the jet-finder and use a concrete jet-algorithm in the jet-finder through a jet-algorithm base class interface, can provide this functionality.

A jet finder shall be able to run on any consistent set of inputs:
- Templating with the input solves the problem, if there is only one type of inputs in one job, and the input has by convention a well defined interface.
- A JetableObject class that can construct from the various inputs is an alternative. In this case, the JetFinder still either needs a loading mechanism for all possible inputs, or a separate class takes this responsibility.
- A third possibility is to template the jet finder with a user-definable JetableObject class that obeys an interface given by a base class, and hence can allow polymorphic access to the constituents while keeping performance.

A jet finder shall be able to provide access to jets:
- Returning an iterator over jets will do this job. Since the Jet will be user defined, the JetFinder could be templated with the Jet.

#### 3.2.2 Requirements on Jet

A Jet base-class that defines the interface to the required functionality will satisfy all requirements.

### 4 Object Oriented Design

#### 4.1 Base-line design

The design choices made are to template the JetFinder with the concrete jet-type, i.e. to write generic algorithms for jet-finding, and hence de-couple the algorithm from the concrete analysis.
The input is handled via an input generator base class, providing maximum flexibility for the user to do selections and cuts, and decoupling the jet-finding algorithm from the concrete input data types.

The jet-finding is done internally on a JetableObject class that can construct from a number of different input types. This choice restricts the accessibility of constituents to the basic properties of the constituents, and might be challenged in due course. It provides the user with a clean and simple interface for finding jets.

The interface to the application framework is done via a facade class, following the component approach, and allowing to easily follow changes in application framework functionality.

For more information on the design, please see figure1.

5 Concrete Algorithms

Several concrete jet-finders as commonly used in $e^+e^-$ and hadron colliders have been implemented to demonstrate the flexibility of the system. They are described below in some detail.

5.1 Simple Cone Algorithm

A trivial cone-based algorithm is implemented in the DummyJetAlgorithm class. The algorithm searches the maximum energy object and throws an $\eta - \phi$ cone around its direction. Any object within that cone will be merged to form a jet. The constituents are removed from the list of objects, and the procedure is repeated until no objects are left in the list.

5.2 Iterative Cone Algorithm

An iterative, cone-based algorithm[1] is implemented in the IterativeConeAlgorithm class. The algorithm again searches the maximum energy object and throws an $\eta - \phi$ cone around its direction. Any object within that cone will be merged to form a proto-jet. The proto-jet direction is calculated from the energy weighted directions of the constituents, and a cone in $\eta - \phi$ is thrown around the new direction to form a new proto-jet. The procedure is repeated until the proto-jet does not change significantly between two iterations, which is that the jet energy change is smaller than a tunable value (1% by default) and $\sqrt{\Delta \eta^2 + \Delta \phi^2}$ is below a tunable value (0.01 by default). The constituents are removed from the list of objects, and the procedure is repeated until no objects are left in the list.

5.3 Successive Combination Algorithm

In the successive combination[2] algorithm of S.D. Ellis and E. Soper, the test criterion is defined as

$$d_{ij} = \min(E_{T,i}^2, E_{T,j}^2) [(\eta_i - \eta_j)^2 + (\phi_i - \phi_j)^2]/R^2.$$ 

Here $R$ is a free parameter of the algorithm that should be of order 1, $E_{T,i}$ is the transverse energy of an object under test, and $i, j$ refers to a pair of candidate objects.

Any object in the input is considered a jet candidate object. The minimum $d_{min}$ of all $d_{ij}$ is compared to the minimum $E_{T,min}^2$ of all $E_{T,i}^2$. Is $d_{min}$ smaller than $E_{T,min}^2$, the corresponding pair of jet candidates is merged into a single jet candidate. The new direction is calculated from the $E_T$ weighted directions of the constituents. Otherwise, the candidate jet corresponding to $E_{T,min}^2$ is considered a jet and is removed from the iteration loop. The iteration continues until no jet candidates are left.

The algorithm is implemented in the SuccessiveCombinationJetAlgo class.

5.4 $k_t$ algorithm for hadron hadron colliders

In the $k_t$[6] algorithm for hadron hadron colliders, the test-criterion used is

$$y_{ij} = \frac{2\min(E_{T,i}^2, E_{T,j}^2)}{E_T^2} (1 - \cos \theta_{ij}).$$

In contrast to other jet-algorithms there is a pre-clustering stage that aims at identifying the beam-jets. During this stage, macro jets are formed by comparing for each pair of hadronic objects the values of $y_{ij}$ with $y_{ip}$, the
Figure 1: Baseline design of the jet finder package. Framework, basic algorithm, and input generator classes are shown. Detailed designs are omitted for better readability.
equivalent quantity with respect to beam direction $p$:

$$y_{ip} = \frac{2E_i^2}{E_t^2} (1 - \cos \theta_{ip})$$

If the minimum $y$ is one of the $y_{ip}$, the associated hadronic object is moved into the corresponding beam jet. Otherwise the particles $i, j$ are merged into a new hadronic object. The procedure is repeated on the new set of hadronic objects, until no $y_{ij}$ or $y_{ip}$ is smaller than 1. The beam-jets are identified, and the other hadronic objects are now considered to be macro-jets with sub-structure.

In a second step, the jet-structure in the macro-jets is resolved by successively merging minimum $y_{ij}$ objects into jet prototypes, which themselves are subject to the iteration. The iteration is done separately for each macro-jet, and is stopped once the $y_{ij}$ within each macro-jet are above a tunable cut-off value.

The algorithm is implemented in the `HadronHadronK_T_Alg` class.

### 5.5 Inverse colour dipole model

The inverse colour-dipole model[7] is an intriguing algorithm in so far, as it generates jets in such a manner that one cannot associate one constituent to exactly one jet. The reason lies in the usage of three cluster combinations and will become apparent from the description. For the practical reasons of performance and the desire to use and recalibrate constituents, the algorithm is considered an exercise of the framework.

The algorithm considers all three particle combinations as candidates for colour dipoles that have radiated. The scale $p_t^2$ for the emission in a colour dipole is

$$p_t^2 = S_{dip} \left(1 - x_1 + \frac{m_1^2 - (m_2 + m_3)^2}{S_{dip}} \right) \left(1 - x_3 + \frac{m_3^2 - (m_2 + m_1)^2}{S_{dip}} \right).$$

Here $x_1$ and $x_3$ are the energy fractions in the final state of the two particles forming the emitting dipole, $x_2$ is the energy fraction of the emitted particle, and $S_{dip}$ is the dipole strength. Note that $p_t^2$ is a Lorentz-invariant quantity.

The final state particle configuration is identified for a given set of three particles by minimising $p_t^2$. In the commonly used implementation of the colour dipole model, the relation between the relative angles of the three particles in the the center of mass system of the radiating colour dipole is

$$\theta = \frac{x_3^2}{x_1^2 + x_3^2} (\pi - \psi),$$

Here $\theta$ is the angle between the initial colour dipole axis and the direction of particle 1, and $\psi$ is the angle between the two radiating particles in the final state. We use this convention in our implementation.

The algorithm takes the minimum $p_t^2$ configuration of all 3-particle combinations, and merges the radiated particle into the two radiating particles. The 3-particle system is then replaced by two mass-less particles along the dipole axis with opposite directions, each carrying half the dipole strength in energy.

The procedure is iterated, until the minimum $p_t$ of all 3-particle combinations is above a cut-off, and the remaining particles are considered to be jets.

### 5.6 JADE Algorithm

In the Jade[3] clustering algorithm pairs of objects are formed, based on the ordering in the test-variable $y_{ij}$, where

$$y_{ij} = \frac{2E_iE_j}{E_{vis}^2} (1 - \cos \theta).$$

Here $E_i$ and $E_j$ denote energies of the candidate constituent objects, and $E_{vis}$ is the total energy in the event. $\theta$ is the opening angle between the two candidates. The pair of objects with the smallest value of $y_{ij}$ is found, and combined into a new candidate constituent object in case $y_{ij} < y_{cut}$. $y_{cut}$ is a free parameter of the algorithm, and defines the structural resolution of the jet algorithm. The new direction is calculated from the $E_T$ weighted directions of the constituents. The iteration stops, when the minimum value of $y_{ij}$ is found to be larger than $y_{cut}$. The JADE algorithm is implemented in the `JADEAlgorithm` class.
5.7 Durham Algorithm

The logic of the Durham[4] clustering algorithm is analogous to the JADE algorithm. The test criterion is changed to

\[ y_{ij} = \frac{2\min(E_i^2, E_j^2)}{E_{\text{vis}}^2} (1 - \cos \theta). \]

The algorithm is implemented in the DurhamAlgorithm class.

5.8 Cambridge Algorithm

In the Cambridge[5] algorithm the test-criterion is identical to that of the Durham algorithm, but the test-criterion is only used to terminate the iteration loop. The ordering criteria differs from the test criterion, and is defined as

\[ v_{ij} = 2(1 - \cos \theta). \]

The logic is the following. First select the pair of candidate jets with the smallest value of \( v_{ij} \). If the corresponding value for \( y_{ij} \) is below cut-off, combine the candidates to form a new candidate jet, where the new direction is calculated from the \( E_T \) weighted directions of the constituents. If the corresponding \( y_{ij} \) is above a tunable cut-off, the candidate jet that has the lowest energy in the pair is considered a jet. The iteration is continued, until only one candidate jet is left, which then is considered to be a jet.

This algorithm is implemented in the CambridgeAlgorithm class.

6 Usage of the Package

This package provides the framework for jet-finding, a set on input generators, a set of jet algorithms, and one concrete jet class with the possibility to user-define other jet-classes.

6.1 How to build an application

The elementary building blocks of a concrete instance of the JetFinder package are the input generator, the concrete algorithm you would like to use, the framework, and the concrete jet type you want to reconstruct.

The jet-finder is designed to be able to run on any input. Currently provided are mechanisms to run on calorimeter digits (JetFinderDigiInput), Higher Lever Trigger towers (JetFinderCaloTowerInput), tracks (JetFinderTrackInput), and generator particles (JetFinderGeneratorInput). Any filtering for the moment is meant to be done in the input generator class. The VJetFinderInputFilter class is currently a placeholder for future functionality. The possibility to add user-defined input generators will be added in the next iteration.

The concrete algorithms provided are described in the previous section. Flexibility that allows for user-defined algorithms will be released in the next iteration.

For use with the CARF application framework, we provide the JetFinderCARFFacade class. It is made to be registered with CARF in the Reconstructor::buildDict() method in your analysis program. It allows to register a jet algorithm through the JetFinderCARFFacade::setAlgorithm(...) method, and to register the input generator of your choice through the JetFinderCARFFacade::setInputGenerator(...) method. Only one algorithm and input generator are allowed at a time. Note that the JetableObject class used internally for jet-finding can construct from many different types. This allows in a first iteration to write input generators for mixed input. A superior design is in preparation.

The concrete jet type to build is a user defined class, and the JetFinder package is templated with the jet-type. This allows to use the jet-finder package in different analysis, where the services the jet-type has to provide might be entirely different (b-jets, or QCD jets, etc.), but the algorithm and input generator for finding the jets can still be the same. The following methods are currently required in a jet-type for use with the JetFinders package.

```cpp
ConcreteJet(HepLorentzVector & aLorentzVector);
double getEta();
double getPhi();
double getTheta();
```
Here an example program snippet.

```cpp
JetFinderCARFFacade<ConcreteJet> * aFacade = new JetFinderCARFFacade<ConcreteJet>();
VJetAlgorithm<ConcreteJet> * anAlgo;
anAlgo = new IterativeConeAlgorithm<ConcreteJet>;
JetFinderDigiInput * anInput = new JetFinderDigiInput;
aFacade->setAlgorithm(anAlgo);
aFacade->setInputGenerator(anInput);
addDefault(aFacade);
```

7 Conclusions

One of the goals of this note is the demonstration of physics object reconstruction using object oriented programming and the exploration of this new technology in this context.

This was done by achieving decoupling of algorithms, jet types, and analysis, so the CMS jet-finders can be used in different contexts without duplication of basic functionality, and without dependence on data-structures that are outside the scope of jet reconstruction. OO techniques have been explored and exercised to provide a variety of jet-finding algorithms and input generators.

The basic concepts inherent to the C++ language employed to achieve decoupling of algorithms, output functionality, and analysis are:

- Function overloading, to allow for a variety of inputs, and to decouple the algorithms from their inputs.
- Polymorphism, to allow user defined input generators and filters that are switchable at run-time, as to make the algorithms independent from data structures outside the scope of jet-finding.
- Generic programming to allow for decoupling of the algorithms from the functionality of a concrete Jet-type necessary for a given analysis.

Note that none of the above techniques are available in procedural languages like C or FORTRAN.

A brief field-test in the context of the higher level trigger studies and \( \tau \) reconstruction showed that the built-in flexibility is sufficient to allow re-use of the package in the different contexts. The prove of concept of physics object reconstruction in the context of the reconstruction of hadronic final states was successful.

References