FastJet 3.0.0 user manual

Matteo Cacciari,[1,2] Gavin P. Salam[3,4,1] and Gregory Soyez[5]

[1]LPTHE, UPMC Univ. Paris 6 and CNRS UMR 7589, Paris, France

[2]Université Paris Diderot, Paris, France

[3]CERN, Physics Department, Theory Unit, Geneva, Switzerland

[4]Department of Physics, Princeton University, Princeton, NJ 08544,USA

[5]Institut de Physique Théorique, CEA Saclay, France

Abstract

`FastJet` is a `C++` package that provides access to a broad range of jet finding and analysis tools. It includes efficient native implementations of all widely-used $2 \to 1$ sequential recombination jet algorithms for pp and e^+e^- collisions, as well as access to 3rd party jet algorithms through a plugin mechanism, including all currently-used cone algorithms. `FastJet` also provides means to facilitate the manipulation of jet subtructure, including some common boosted heavy-object taggers, as well as tools for estimation of pileup and underlying event noise levels, determination of jet areas and subtraction or suppression of noise in jets.

Contents

1 Introduction

Jets are the collimated sprays of hadrons that result from the fragmentation of a high-energy quark or gluon. Jet finders identify these sprays of particles, grouping particles into "jets" according to some well-defined algorithm. Jet finding is applied not only to hadronic final states, but also to partonic final states that come out of perturbative QCD calculations. Jets are useful both as observables that stand in for the naive concept of partons and as tools for comparing experimental and theoretical results.

There are two broad groups of jet algorithms: sequential recombination algorithms usually carry out successive mergings of pairs of particles until some stopping condition is reached; cone algorithms define jets based on cones in phase space containing the bulk of an event's energy flow.

FastJet is a C++ package that provides efficient implementations of many sequential recombination algorithms, through the computational strategies of [1]. It also provides a common interface to many legacy and current cone algorithms, in large part through suitably wrapped versions of 3rd party code ("plugins").

In addition to its basic jet-finding capabilities, FastJet provides access to a number of more advanced functionalities. These include tools for the determination of jet areas, facilities to estimate the level of noise in a given event due to underlying events and pileup, and the subtraction of that noise from individual jets. Another set of tools helps analyse the substructure of jets, including a number of widely used boosted heavy-object taggers and substructure-based noise-suppression classes.

2 Quick-start guide

For the impatient, the `FastJet` package can be set up and run as follows.

- Download the code and the unpack it

  ```
  wget http://www.lpthe.jussieu.fr/~salam/fastjet/repo/fastjet-X.Y.Z.tar.gz
  tar zxvf fastjet-X.Y.Z.tar.gz
  cd fastjet-X.Y.Z/
  ```

  (replacing `X.Y.Z` with the appropriate version number).

- Compile and install (choose your own preferred prefix), and when you're done go back to the original directory

  ```
  ./configure --prefix=`pwd`/../fastjet-install
  make
  make check
  make install
  cd ..
  ```

  If you copy and paste the above lines from one very widespread PDF viewer, you should note that the first line contains *backquotes* not forward quotes but that your PDF viewer may nevertheless paste forward quotes, causing problems down the line (the issue arises again below).

- Now paste the following piece of code into a file called `short-example.cc`

  ```
  #include "fastjet/ClusterSequence.hh"
  #include <iostream>
  using namespace fastjet;
  using namespace std;

  int main () {
    vector<PseudoJet> particles;
    // an event with three particles:   px    py  pz      E
    particles.push_back( PseudoJet(   99.0,  0.1,  0, 100.0) );
    particles.push_back( PseudoJet(    4.0, -0.1,  0,   5.0) );
    particles.push_back( PseudoJet(  -99.0,    0,  0,  99.0) );

    // choose a jet definition
    double R = 0.7;
    JetDefinition jet_def(antikt_algorithm, R);

    // run the clustering, extract the jets
    ClusterSequence cs(particles, jet_def);
    vector<PseudoJet> jets = sorted_by_pt(cs.inclusive_jets());

    // print out some infos
    cout << "Clustering with " << jet_def.description() << endl;

    // print the jets
    cout <<   "        pt y phi" << endl;
    for (unsigned i = 0; i < jets.size(); i++) {
      cout << "jet " << i << ": "<< jets[i].perp() << " "
                   << jets[i].rap() << " " << jets[i].phi() << endl;
      vector<PseudoJet> constituents = jets[i].constituents();
      for (unsigned j = 0; j < constituents.size(); j++) {
        cout << "    constituent " << j << "'s pt: "<< constituents[j].perp() << endl;
      }
    }
  }
  ```

- Then compile and run it with

  ```
   g++ short-example.cc -o short-example \
       `fastjet-install/bin/fastjet-config --cxxflags --libs --plugins`
  ```

  (watch out, once again, for the backquotes if you cut and paste from the PDF).

The output will consist of a banner, followed by the lines

```
Clustering with Longitudinally invariant anti-kt algorithm with R = 0.7
and E scheme recombination
        pt y phi
jet 0: 103 0 0
```

```
    constituent 0's pt: 99.0001
    constituent 1's pt: 4.00125
jet 1: 99 0 3.14159
    constituent 0's pt: 99
```

More evolved example programs, illustrating many of the capabilites of `FastJet`, are available in the `example/` subdirectory of the `FastJet` distribution.

3 Jet-finding interface

All classes are contained in the `fastjet` namespace. For brevity this namespace will usually not be explicitly written below, with the possible exception of the first appearance of a `FastJet` class, and code excerpts will assume that a `using namespace fastjet;` is present in the user code. For basic usage, the user is exposed to three main classes:

```
class fastjet::PseudoJet;
class fastjet::JetDefinition;
class fastjet::ClusterSequence;
```

`PseudoJet` provides a jet object with a four-momentum and some internal indices to situate it in the context of a jet-clustering sequence. `ClusterSequence` is the class that carries out jet-clustering and provides access to the final jets. The class `JetDefinition` contains a specification of how jet clustering is to be performed.

3.1 fastjet::PseudoJet

All jets, as well as input particles to the clustering (optionally) are `PseudoJet` objects. They can be created using one of the following constructors

```
PseudoJet (double px, double py, double pz, double  E);
template<class T> PseudoJet (const T & some_lorentz_vector);
```

where the second form allows the initialisation to be obtained from any class `T` that allows subscripting to return the components of the momentum (running from $0 \ldots 3$ in the order p_x, p_y, p_z, E), for example the `CLHEP HepLorentzVector` class.[1] The default constructor for a `PseudoJet` sets the momentum components to zero.

The `PseudoJet` class includes the following member functions for accessing the components

```
double E()         const ; // returns the energy component
double e()         const ; // returns the energy component
double px()        const ; // returns the x momentum component
double py()        const ; // returns the y momentum component
double pz()        const ; // returns the z momentum component
double phi()       const ; // returns the azimuthal angle in range 0...2π
double phi_std()   const ; // returns the azimuthal angle in range −π...π
double rap()       const ; // returns the rapidity
```

[1] `PseudoJet` is the analogue of `KtJet`'s `KtLorentzVector`. A significant difference is that it is not derived from `HepLorentzVector` (so as to allow compilation even without `CLHEP`).

```
double rapidity() const ; // returns the rapidity
double pseudorapidity() const ; // returns the pseudo-rapidity
double eta()      const ; // returns the pseudo-rapidity
double kt2()      const ; // returns the squared transverse momentum
double perp2()    const ; // returns the squared transverse momentum
double perp()     const ; // returns the transverse momentum
double m2()       const ; // returns squared invariant mass
double m()        const ; // returns invariant mass (-√-m² if m² < 0)
double mperp2()   const ; // returns the squared transverse mass = k_t² + m²
double mperp()    const ; // returns the transverse mass
double operator[] (int i) const; // returns component i
double operator() (int i) const; // returns component i

/// return a valarray containing the four-momentum (components 0--2
/// are 3-momentum, component 3 is energy).
valarray<double> four_mom() const;
```

There are two ways of associating user information with a `PseudoJet`. The simpler method is through an integer called the user index

```
/// set the user_index, intended to allow the user to label the object (default is -1)
void set_user_index(const int index);

/// return the user_index
int user_index() const ;
```

A more powerful method, new in `FastJet` 3, involves passing a pointer to a derived class of `PseudoJet::UserInfoBase`. The two essential calls are

```
/// set the pointer to user information (the PseudoJet will then own it)
void set_user_info(UserInfoBase * user_info);
/// retrieve a reference to a dynamic cast of type L of the user info
template<class L> const L & user_info() const;
```

Further details are given in appendix A.

A `PseudoJet` can be reset with

```
/// Reset the 4-momentum according to the supplied components, put the user
/// and history indices and user info back to their default values (-1, unset)
inline void reset(double px, double py, double pz, double E);
/// Reset just the 4-momentum according to the supplied components,
/// all other information is left unchanged
inline void reset_momentum(double px, double py, double pz, double E);
```

and similarly taking as argument a templated `some_lorentz_vector` or a `PseudoJet` (in the latter case, or when `some_lorentz_vector` is of a type derived from `PseudoJet`, `reset` also copies the user and internal indices and user-info).

Additionally, the `+`, `-`, `*` and `/` operators are defined, with `+`, `-` acting on pairs of `PseudoJet`s and `*`, `/` acting on a `PseudoJet` and a `double` coefficient. The analogous `+=`, etc., operators, are also defined.

There are also equality testing operators: `(jet1 == jet2)` returns true if the two jets have identical 4-momenta, structural information and user information; the `(jet == 0.0)` test returns true if all the components of the 4-momentum are zero. The `!=` operator works analogously.

`E_scheme`
`pt_scheme`
`pt2_scheme`
`Et_scheme`
`Et2_scheme`
`BIpt_scheme`
`BIpt2_scheme`

Table 1: Members of the `RecombinationScheme` enum; the last two refer to boost-invariant version of the p_t and p_t^2 schemes (as defined in section 3.4).

Finally, we also provide routines for taking an unsorted vector of `PseudoJet`s and returning a sorted vector,

```
/// return a vector of jets sorted into decreasing transverse momentum
vector<PseudoJet> sorted_by_pt(const vector<PseudoJet> & jets);

/// return a vector of jets sorted into increasing rapidity
vector<PseudoJet> sorted_by_rapidity(const vector<PseudoJet> & jets);

/// return a vector of jets sorted into decreasing energy
vector<PseudoJet> sorted_by_E(const vector<PseudoJet> & jets);
```

These will typically be used on the jets returned by `ClusterSequence`.

3.2 fastjet::JetDefinition

The class `JetDefinition` contains a full specification of how to carry out the clustering. According to the Les Houches convention detailed in [2], a 'jet definition' should include the jet algorithm name, its parameters (often the radius R) and the recombination scheme. Its constructor is[2]

```
JetDefinition(fastjet::JetAlgorithm jet_algorithm,
              double R,
              fastjet::RecombinationScheme recomb_scheme = E_scheme,
              fastjet::Strategy strategy = Best);
```

The jet algorithm is one of the entries of the `JetAlgorithm` enum[3]:

```
enum JetAlgorithm {kt_algorithm, cambridge_algorithm,
                   antikt_algorithm, genkt_algorithm,
                   ee_kt_algorithm, ee_genkt_algorithm, ...};
```

where the ... represent additional values that are present for internal or testing purposes. `R` specifies the value of R that appears in eqs. (2,3,4,5,6).

[2]The v. 2.0 constructor, without the recombination scheme argument, still remains valid.

[3]As of v2.3, the `JetAlgorithm` name replaces the old `JetFinder` one, in keeping with the Les Houches convention. Backward compatibility is assured at the user level by a typedef and a doubling of the methods names. Backward compatibility (with versions < 2.3) is however broken for user-written derived classes of `ClusterSequence`, as the protected variables `_default_jet_finder` and `_jet_finder` have been replaced by `_default_jet_algorithm` and `_jet_algorithm`.

`N2Plain`	a plain N^2 algorithm (fastest for $N \lesssim 50$)
`N2Tiled`	a tiled N^2 algorithm (fastest for $50 \lesssim N \lesssim 400$)
`N2MinHeapTiled`	a tiled N^2 algorithm with a heap for tracking the minimum of d_{ij} (fastest for $400 \lesssim N \lesssim 15000$)
`NlnN`	the Voronoi-based $N \ln N$ algorithm (fastest for $N \gtrsim 15000$)
`NlnNCam`	based on Chan's $N \ln N$ closest pairs algorithm (fastest for $N \gtrsim 6000$), suitable only for the Cambridge jet algorithm
`Best`	automatic selection of the best of these based on N and R

Table 2: The more interesting of the various algorithmic strategies for clustering. Other strategies are given `JetDefinition.hh` — note however that strategies not listed in the above table may disappear in future releases. For jet algorithms with spherical distance measures (those whose name starts with "`ee_`"), only the `N2Plain` strategy is available.

For one algorihm, `ee_kt_algorithm`, there is no R parameter, so the constructor is to be called without the `R` argument:

```
JetDefinition(JetAlgorithm jet_algorithm,
              RecombinationScheme recomb_scheme = E_scheme,
              Strategy strategy = Best);
```

For the generalised k_t algorithm and its e^+e^- version, one requires R and an extra parameter p, and the following constructor should then be used

```
JetDefinition(JetAlgorithm jet_algorithm,
              double R,
              double p,
              RecombinationScheme recomb_scheme = E_scheme,
              Strategy strategy = Best);
```

If the user calls a constructor with the incorrect number of arguments for the requested jet algorithm, a `fastjet::Error()` exception will be thrown with an explanatory message.

The default constructor for `JetDefinition` sets the jet algorithm to `undefined_jet_algorithm`.

The recombination scheme is set by an `enum` of type `RecombinationScheme`, and it is related to the choice of how to recombine the 4-momenta of `PseudoJet`s during the clustering procedure. The default in `FastJet` is the E-scheme, where the four components of two 4-vectors are simply added. This scheme is used when no explicit choice is made in the constructor. The list of available recombination schemes is given in table 1, and more details are given in section 3.4.

The strategy selects the algorithmic strategy to use while clustering and is an `enum` of type `Strategy` with potentially interesting values listed in table 2. Nearly all strategies are based on the factorisation of energy and geometrical distance components of the d_{ij} measure [1]. In particular they involve the dynamic maintenance of a nearest-neighbour graph for the geometrical distances. They apply equally well to any of the internally implemented jet algorithms. The one exception is `NlnNCam`, which is based on a computational geometry algorithm for dynamic maintenance of closest pairs [3] (rather than the more involved nearest neighbour graph), and is suitable only for the Cambridge algorithm whose distance measure is purely geometrical.

The `N2Plain` strategy uses a "nearest-neighbour heuristic" [4] approach to maintaining the geometrical nearest-neighbour graph; `N2Tiled` tiles the $y - \phi$ cylinder to limit the set of points over which nearest-neighbours are searched for,[4] and `N2MinHeapTiled` differs only in that it uses an $N \ln N$ (rather than N^2) data structure for maintaining in order the subset of the d_{ij} that involves nearest neighbours. The `NlnN` strategy uses CGAL's Delaunay Triangulation [6] for the maintenance of the nearest-neighbour graph. Note that $N \ln N$ performance of is an *expected* result, and it holds in practice for the k_t and Cambridge algorithms, while for anti-k_t and generalised-k_t with $p < 0$, hub-and-spoke (bicycle-wheel!) type configurations emerge dynamically during the clustering and these break the conditions needed for the expected result to hold (this however has a significant impact only for $N \gtrsim 10^5$).

If `strategy` is omitted then the `Best` option is set. Note that the N ranges quoted above for which a given strategy is optimal hold for $R = 1$; the general R dependence can be significant (and non-trivial), for example for the Cambridge/Aachen jet algorithm with $R = 0.4$, `NlnNCam` beats the `N2MinHeapTiled` strategy only for $N \gtrsim 37000$. While some attempt has been made to account for the R-dependence in the choice of the strategy with the "`Best`" option, there may exist specific regions of N and R in which a manual choice of strategy can give faster execution. Furthermore the `NlnNCam` strategy's timings may depend strongly on the size of the cache, and the transitions that have been adopted are based on a cache size of 2 MB. Finally for a given N and R, the optimal strategy may also depend on the event structure.

A textual description of the jet definition can be obtained by a call to the member function

```
std::string description();
```

3.3 fastjet::ClusterSequence

To run the jet clustering, create a `ClusterSequence` object,[5] through the following constructor

```
template<class L> ClusterSequence(const std::vector<L> & input_particles,
                                  const JetDefinition & jet_def);
```

where `input_particles` is the vector of initial particles of any type (`PseudoJet`, `HepLorentzVector`, etc.) that can be used to initialise a `PseudoJet` and `jet_def` contains the full specification of the clustering (see Section 3.2).

If the user wishes to access inclusive jets, the following member function should be used

```
/// return a vector of all jets (in the sense of the inclusive
/// algorithm) with pt >= ptmin.
vector<PseudoJet> inclusive_jets (const double & ptmin = 0.0) const;
```

where `ptmin` may be omitted (then implicitly taking value 0).

There are two ways of accessing exclusive jets,[6] one where one specifies d_{cut}, the other where one specifies that the clustering is taken to be stopped once it reaches the specified number of jets.

```
/// return a vector of all jets (in the sense of the exclusive
/// algorithm) that would be obtained when running the algorithm
/// with the given dcut.
```

[4]Tiling is a textbook approach in computational geometry, where it is often referred to as bucketing. It has been used also in certain cone jet algorithms, notably at trigger level and in [5].

[5]The analogue of `KtJet`'s `KtEvent`.

[6]In contrast to `KtJet` the class constructor is the same for the inclusive and exclusive cases. This choice has been made because the clustering sequence is identical in the two cases.

```
  vector<PseudoJet> exclusive_jets (const double & dcut) const;

  /// return a vector of all jets when the event is clustered (in the
  /// exclusive sense) to exactly njets. Throws an error if the event
  /// has fewer than njets particles.
  vector<PseudoJet> exclusive_jets (const int & njets) const;

  /// return a vector of all jets when the event is clustered (in the
  /// exclusive sense) to exactly njets. If the event has fewer than
  /// njets particles, it returns all available particles.
  vector<PseudoJet> exclusive_jets_up_to (const int & njets) const;
```

The PseudoJet vectors returned by the above routines can all be sorted with the routines described at the end of section 3.1.

The user may also wish just to obtain information about the number of jets in the exclusive algorithm:

```
  /// return the number of jets (in the sense of the exclusive
  /// algorithm) that would be obtained when running the algorithm
  /// with the given dcut.
  int n_exclusive_jets (const double & dcut) const;
```

Another common query is to establish the $d_{\min}$ value associated with merging from $n+1$ to n jets. Two member functions are available for determining this:

```
  /// return the dmin corresponding to the recombination that went from
  /// n+1 to n jets (sometimes known as d_{n n+1}).
  double exclusive_dmerge (const int & njets) const;

  /// return the maximum of the dmin encountered during all recombinations
  /// up to the one that led to an n-jet final state; identical to
  /// exclusive_dmerge, except in cases where the dmin do not increase
  /// monotonically.
  double exclusive_dmerge_max (const int & njets) const;
```

The first returns the $d_{\min}$ in going from $n+1$ to n jets. Occasionally however the $d_{\min}$ value does not increase monotonically during successive mergings and using a d_{cut} smaller than the return value from exclusive_dmerge does not guarantee an event with more than njets jets. For this reason the second function exclusive_dmerge_max is provided — using a d_{cut} below its return value is guaranteed to provide a final state with more than n jets, while using a larger value will return a final state with n or fewer jets.

For e^+e^- collisions, where it is usual to refer to $y_{ij} = d_{ij}/Q^2$ (Q is the total (visible) energy) FastJet provides the following methods:

```
  double exclusive_ymerge (int njets);
  double exclusive_ymerge_max (int njets);
  int n_exclusive_jets_ycut (double ycut);
  std::vector<PseudoJet> exclusive_jets_ycut (double ycut);
```

which are relevant for use with the e^+e^- k_t algorithm and with the Jade plugin (section 5.4.2).

Unclustered particles. User-supplied plugin jet algorithms (see section 5) may have the property that not all particles are clustered into jets. In such a case it can be useful to obtain the list of unclustered particles. This can be done as follows:

```
vector<PseudoJet> unclustered = clust_seq.unclustered_particles();
```

Copying and transforming a ClusterSequence. A standard copy constructor is available for `ClusterSequence`s. Additionally it is possible to copy the clustering history of a `ClusterSequence` while modifying the momenta of the jets, with the `ClusterSequence` member function

```
void transfer_from_sequence(const ClusterSequence & original_cs,
                            const FunctionOfPseudoJet<PseudoJet> * action_on_jets = 0);
```

`fastjet::FunctionOfPseudoJet<PseudoJet>` is an abstract base class whose interface provides a `PseudoJet operator()(const PseudoJet & jet)` function, i.e. a function of a `PseudoJet` that returns a `PseudoJet`, as discussed in appendix C. As the clustering history is copied to the target `ClusterSequence`, each `PseudoJet` in the target `ClusterSequence` is given by the result of applying this function to the corresponding `PseudoJet` in the original sequence. One use case for this is if one wishes to obtain a Lorentz-boosted copy of a `ClusterSequence`, which can be achieved as follows

```
#include "fastjet/tools/Boost.hh"
// ...
ClusterSequence original_cs(...);
ClusterSequence boosted_cs;
Boost boost(some_jet_to_describe_the_boost);
boosted_cs.transfer_from_sequence(cs, &boost);
```

3.4 Recombination schemes

When merging particles (i.e. `PseudoJet`s) during the clustering procedure, one must specify how to combine the momenta. The simplest procedure (E-scheme) simply adds the four-vectors. This has been advocated as a standard in [7], and is the default option in `FastJet`. Other choices are listed in table 1, and are described below.

Other schemes for *pp* collisions. Other schemes provided by earlier k_t-clustering implementations [8] are the p_t, p_t^2, E_t and E_t^2 schemes. They all incorporate a 'preprocessing' stage to make the initial momenta massless (rescaling the energy to be equal to the 3-momentum for the p_t and p_t^2 schemes, rescaling to the 3-momentum to be equal to the energy in the E_t and E_t^2 schemes). Then for all schemes the recombination p_r of p_i and p_j is a massless 4-vector satisfying

$$p_{t,r} = p_{t,i} + p_{t,j}\,, \tag{1a}$$

$$\phi_r = (w_i\phi_i + w_j\phi_j)/(w_i + w_j)\,, \tag{1b}$$

$$y_r = (w_i y_i + w_j y_j)/(w_i + w_j)\,, \tag{1c}$$

where w_i is $p_{t,i}$ for the p_t and E_t schemes, and is $p_{t,i}^2$ for the p_t^2 and E_t^2 schemes.

Note that for massive particles the schemes defined in the previous paragraph are not invariant under longitudinal boosts. We therefore also introduce boost-invariant p_t and p_t^2 schemes, which are identical to the normal p_t and p_t^2 schemes, except that they omit the preprocessing stage.

Other schemes for e^+e^- collisions. On request, we may in the future provide dedicated schemes for e^+e^- collisions.

User-defined schemes. The user may define their own, custom recombination schemes, as described in Appendix D.1.

3.5 Constituents and substructure of jets

For any `PseudoJet` that results from a clustering, the user can obtain information about its constituents, internal substructure, etc., through member functions of the `PseudoJet` class. [7]

Jet constituents For example the constituents of a give `PseudoJet jet` can be obtained as

```
vector<PseudoJet> constituents = jet.constituents();
```

Note that if the user wishes to identify these constituents with the original particles provided to `ClusterSequence`, she or he should have set a unique index for each of the original particles with the `PseudoJet::set_user_index` function. Alternatively more detailed information can also be set through the `user_info` facilities of `PseudoJet`, as discussed in appendix A.

Subjet properties. To obtain the set of subjets at a specific $d_{\rm cut}$ scale inside a given jet, one may use the following `PseudoJet` member function:

```
/// return a vector of all subjets of the current jet (in the sense
/// of the exclusive algorithm) that would be obtained when running
/// the algorithm with the given dcut.
std::vector<PseudoJet> exclusive_subjets (const double & dcut) const;
```

If m jets are found, this takes a time $\mathcal{O}(m \ln m)$ (owing to the internal use of a priority queue). Alternatively one may obtain the jet's constituents, cluster them separately and then carry out an `exclusive_jets` analysis on the resulting `ClusterSequence`. The results should be identical. This second method is mandatory if one wishes to identify subjets with an algorithm that differs from the one used to find the original jets.

In analogy with the `exclusive_jets(...)` functions of `ClusterSequence`, `PseudoJet` also has `exclusive_subjets(int nsub)` and `exclusive_subjets_up_to(int nsub)` functions.

One can also make use of the following methods, which allow one to follow the merging sequence (and walk back through it):

```
/// if the jet has parents in the clustering, it returns true
/// and sets parent1 and parent2 equal to them.
///
/// if it has no parents it returns false and sets parent1 and
/// parent2 to zero
bool has_parents(PseudoJet & parent1, PseudoJet & parent2) const;
```

[7] This is a new development in version 3 of `FastJet`. In earlier versions, access to information about a jet's contents had to be made through its `ClusterSequence`. Those forms of access, though not documented here, will be retained throughout the 3.X series.

```
/// if the jet has a child then return true and set the child jet
/// otherwise return false and set the child to zero
bool has_child(PseudoJet & child) const;

/// if this jet has a child (and so a partner) return true
/// and give the partner, otherwise return false and set the
/// partner to zero
bool has_partner(PseudoJet & partner) const;
```

If any of the above functions are used with a `PseudoJet` that is not associated with a `ClusterSequence`, an error will be thrown. Since the information about a jet's constituents is actually stored in the `ClusterSequence` and not in the jet itself, these methods will also throw an error if the `ClusterSequence` associated with the jet has gone out of scope, been deleted, or in any other way become invalid. One can establish the status of a `PseudoJet`'s associated cluster sequence with the following enquiry functions:

```
// returns true if this PseudoJet has an associated (and valid) ClusterSequence.
bool has_associated_cluster_sequence() const;

// get a (const) pointer to the parent ClusterSequence (NULL if inexistent)
const ClusterSequence* associated_cluster_sequence() const;
```

There are contexts in which, within some function, you might create a `ClusterSequence`, obtain a jet from it and then return that jet from the function. For the user to be able to access the information about the jet's internal structure, the `ClusterSequence` must not have gone out of scope and/or been deleted. To aid with potential memory management issues in this case, as long as you have created the `ClusterSequence` via a `new` operation, then you can tell the `ClusterSequence` that it should be automatically deleted after the last external object (jet, etc.) associated with it has disappeared. The call to do this is `ClusterSequence::delete_self_when_unused()`. There must be at least one external object already associated with the `ClusterSequence` at the time of the call.

3.6 Composite jets, general considerations on jet structure

There are a number of cases where it is useful to be able to take two separate jets and create a single object that is the sum of the two, not just from the point of view of its 4-momentum, but also as concerns its structure. For example, in a search for a dijet resonance, some user code may identify two jets, `jet1` and `jet2`, that are thought to come from a resonance decay and then wish to return a single object that combines both `jet1` and `jet2`. This can be accomplished with the function `join`:

```
PseudoJet resonance = join(jet1,jet2);
```

The 4-momenta are added,[8] and in addition the `resonance` remembers that it came from `jet1` and `jet2`. So, for example, a call to `resonance.constituents()` will return the constituents of both `jet1` and `jet2`. It is possible to `join` 1, 2, 3 or 4 jets or a `vector` of jets. If the jets being joined had areas (section 7) then the joined jet will also have an area.

For a jet formed with `join`, one can find out which pieces it has been composed from with the function

[8] This corresponds to E-scheme recombination. If the user wishes to have the jets joined with a different recombination scheme he/she can pass `JetDefinition::Recombiner` (cf. section D.1) as the last argument to `join(...)`.

```
vector<PseudoJet> pieces = resonance.pieces();
```

In the above example, this would simply return a vector of size 2 containing `jet1` and `jet2`. The `pieces()` function also works for jets that come from a `ClusterSequence`, returning two pieces if the jet has parents, zero otherwise.

Enquiries as to available structural information. Whether or not a given jet has constituents, recursive substructure or pieces depends on how it was formed. Generally a user will know how a given jet was formed, and so know if it makes sense, say, to call `pieces()`. However if a jet is returned from some third-party code, it may not always be clear what kinds of structural information it has. Accordingly a number of enquiry functions are available:

```
bool has_structure();          // true if it has some kind of structural info
bool has_constituents();       // true if it has constituents
bool has_exclusive_subjets(); // true if there is cluster-sequence style subjet info
bool has_pieces();             // true if the jet can be broken up into pieces
bool has_area();               // true if it has jet-area informaion
```

Asking (say) for the `pieces()` of a jet for which `has_pieces()` returns false will cause an error to be thrown. The structural information available for different kinds of jets is summarised in appendix B.

3.7 Version information

Information on the version of `FastJet` that is being run can be obtained by making a call to

```
std::string fastjet_version_string();
```

(defined in `fastjet/JetDefinition.hh`). In line with recommendations for other programs in high-energy physics, the user may wish to include this information in publications and plots so as to facilitate reproducibility of the jet-finding.[9] We recommend also that the main elements of the `jet_def.description()` be provided, together with citations to the original article that defines the algorithm, as well as to the `FastJet` paper [1].

4 FastJet native jet algorithms

4.1 k_t jet algorithm

The longitudinally invariant k_t jet algorithm [9, 10] comes in inclusive and exclusive variants. The inclusive variant (corresponding to [10], modulo small changes of notation) is formulated as follows:

1. For each pair of particles i, j work out the k_t distance
$$d_{ij} = \min(k_{ti}^2, k_{tj}^2)\, \Delta R_{ij}^2/R^2 \tag{2}$$
with $\Delta R_{ij}^2 = (y_i - y_j)^2 + (\phi_i - \phi_j)^2$, where k_{ti}, y_i and ϕ_i are the transverse momentum, rapidity and azimuth of particle i and R is a jet-radius parameter usually taken of order 1; for each parton i also work out the beam distance $d_{iB} = k_{ti}^2$.

[9]While we devote significant effort to ensuring that all versions of `FastJet` give identical, correct results, we are obviously not able to completely guarantee the absence of bugs that might have an effect on the jet finding.

2. Find the minimum $d_{\min}$ of all the d_{ij}, d_{iB}. If $d_{\min}$ is a d_{ij} merge particles i and j into a single particle, summing their four-momenta (this is E-scheme recombination); if it is a d_{iB} then declare particle i to be a final jet and remove it from the list.

3. Repeat from step 1 until no particles are left.

The exclusive variant of the longitudinally invariant k_t jet algorithm [9] is similar, except that (a) when a d_{iB} is the smallest value, that particle is considered to become part of the beam jet (i.e. is discarded) and (b) clustering is stopped when all d_{ij} and d_{iB} are above some d_{cut}. In the exclusive mode R is commonly set to 1.

The inclusive and exclusive variants are both obtained through

```
JetDefinition jet_def(kt_algorithm, R);
ClusterSequence cs(particles, jet_def);
```

The clustering sequence is identical in the inclusive and exclusive cases and the jets can then be obtained as follows:

```
vector<PseudoJet> inclusive_kt_jets = cs.inclusive_jets();
vector<PseudoJet> exclusive_kt_jets = cs.exclusive_jets(dcut);
```

4.2 Cambridge/Aachen jet algorithm

Currently the *pp* Cambridge/Aachen (C/A) jet algorithm [11, 12] is provided only in an inclusive version [12], whose formulation is identical to that of the k_t jet algorithm, except as regards the distance measures, which are:

$$d_{ij} = \Delta R_{ij}^2 / R^2 \,, \tag{3a}$$

$$d_{iB} = 1 \,. \tag{3b}$$

To use this algorithm, define

```
JetDefinition jet_def(cambridge_algorithm, R);
```

and then extract inclusive jets from the cluster sequence.

Attempting to extract exclusive jets from the Cambridge/Aachen algorithm with a d_{cut} parameter simply provides the set of jets obtained up to the point where all $d_{ij}, d_{iB} > d_{cut}$. Having clustered with some given R, this can actually be an effective way of viewing the event at a smaller radius, $R_{eff} = \sqrt{d_{cut}}R$, thus allowing a single event to be viewed at a continuous range of R_{eff} within a single clustering.

We note that the true exclusive formulation of the Cambridge algorithm [11] (in e^+e^-) instead makes use an auxiliary (k_t) distance measure and 'freezes' pseudojets whose recombination would involve too large a value of the auxiliary distance measure. Details are given in section 5.4.1.

4.3 Anti-k_t jet algorithm

This new algorithm, introduced and studied in [13], is defined exactly like the standard k_t algorithm, except for the distance measures which are now given by

$$d_{ij} = \min(1/k_{ti}^2, 1/k_{tj}^2)\, \Delta R_{ij}^2 / R^2 \,, \tag{4a}$$

$$d_{iB} = 1/k_{ti}^2 \,. \tag{4b}$$

While being a sequential recombination algorithm like k_t and Cambridge/Aachen, the anti-k_t algorithm behaves in some sense like a 'perfect' cone algorithm, in that its hard jets are exactly circular on the y-ϕ cylinder [13]. To use this algorithm, define

```
  JetDefinition jet_def(antikt_algorithm, R);
```

and then extract inclusive jets from the cluster sequence.

4.4 Generalised k_t jet algorithm

The "generalised k_t" algorithm again follows the same procedure, but depends on an additional continuous parameter p, with has the following distance measure:

$$d_{ij} = \min(k_{ti}^{2p}, k_{tj}^{2p})\, \Delta R_{ij}^2 / R^2\,, \tag{5a}$$

$$d_{iB} = k_{ti}^{2p}\,. \tag{5b}$$

For specific values of p, it reduces to one or other of the algorithms list above, k_t ($p = 1$), Cambridge/Aachen ($p = 0$) and anti-k_t ($p = -1$). To use this algorithm, define

```
 JetDefinition jet_def(genkt_algorithm, R, p);
```

and then extract inclusive jets from the cluster sequence.

4.5 Generalised k_t algorithm for e^+e^- collisions

`FastJet` also provides native implementations of clustering algorithms in spherical coordinates (specifically for e^+e^- collisions) along the lines of the original k_t algorithms [14], but extended in analogy with the generalised pp algorithm of [13] and section 4.4. We define the two following distances:

$$d_{ij} = \min(E_i^{2p}, E_j^{2p}) \frac{(1-\cos\theta_{ij})}{(1-\cos R)}\,, \tag{6a}$$

$$d_{iB} = E_i^{2p}\,, \tag{6b}$$

for a general value of p and R. At a given stage of the clustering sequence, if a d_{ij} is smallest then i and j are recombined, while if a d_{iB} is smallest then i is called an "inclusive jet".

For values of $R \leq \pi$ in eq. (6), the generalised e^+e^- k_t algorithm behaves in analogy with the pp algorithms: when an object is at an angle $\theta_{iX} > R$ from all other objects X then it forms an inclusive jet. With the choice $p = -1$ this provides a simple, infrared and collinear safe way of obtaining a cone-like algorithm for e^+e^- collisions, since hard well-separated jets have a circular profile on the 3D sphere, with opening half-angle R. To use this form of the algorithm, define

```
  JetDefinition jet_def(ee_genkt_algorithm, R, p);
```

and then extract inclusive jets from the cluster sequence.

If one imagines a (complex) value of R such that $(1-\cos R) > 2$, then the d_{iB} will be smallest only if the event consists of a single particle, and thus with the additional choice of $p = 1$ the clustering sequence will correspond to that of the e^+e^- k_t algorithm [14], often referred to also as the Durham algorithm, which has a single distance:

$$d_{ij} = 2\min(E_i^{2p}, E_j^{2p})(1-\cos\theta_{ij})\,. \tag{7}$$

Note the difference in normalisation between the d_{ij} in eqs. (6) and (7), and the fact that in neither case have we normalised to the total energy Q in the event, contrary to the convention adopted originally in [14] (where the distance measure was called y_{ij}). To use this form of the algorithm, define

```
JetDefinition jet_def(ee_kt_algorithm);
```

and then extract exclusive jets from the cluster sequence.

5 Plugin jet algorithms

It can be useful to have a common interface for a range of jet algorithms beyond the native (k_t, anti-k_t and Cambridge/Aachen) algorithms, and it can also be useful to use the area-measurement tools for these other jet algorithms. In order to facilitate this, the `FastJet` package provides a *plugin* facility, allowing almost any other jet algorithm[10] to be used within the same framework.

5.1 Generic plugin use

Plugins are classes derived from the abstract base class `fastjet::JetDefinition::Plugin`. A `JetDefinition` can be constructed by providing a pointer to a `JetDefinition::Plugin`; the resulting `JetDefinition` object can then be used identically to the normal `JetDefinition` objects used in the previous sections:

```
// have some plugin class derived from the Plugin base class
class CDFMidPointPlugin : public fastjet::JetDefinition::Plugin {...};

// create an instance of the CDFMidPointPlugin class
CDFMidPointPlugin cdf_midpoint( [... options ...] );
// create the jet definition by passing a pointer to the plugin
fastjet::JetDefinition jet_def = fastjet::JetDefinition( & cdf_midpoint);

// then create a ClusterSequence with the input particles and jet_def,
// and use it to extract jets as usual
// ...
```

In cases where the plugin has been created with a `new` statement and the user does not wish to manage the deletion of the corresponding memory when the `JetDefinition` (and any copies) using the plugin goes out of scope, then the user may wish to call the `JetDefinition`'s `delete_plugin_when_unused()` function, which tells the `JetDefinition` to acquire ownership of the pointer to the plugin and delete it when it is no longer needed.

A number of plugins are provided with `FastJet`, providing implementations of most of the jet algorithms used experimentally since the 1990's. They are described in sections 5.2–5.4.1. The code for them is to be found in `FastJet`'s `plugins/` directory.

New user-defined plugins can also be implemented, as described in section D.2.

[10]Except those that perform $3 \rightarrow 2$ clusterings for which there is no unique mapping of particles into jets (some particles are effectively shared among more than one jet).

5.2 SISCone Plugin

SISCone [15] is an implementation of a cone type jet algorithm. As with most modern cone algorithms, it is divided into two parts: first it searches for stable cones; then, because a particle can appear in more than one stable cone, a 'split–merge' procedure is applied, which ensures that no particle ends up in more than one jet. The stable cones are identified using an $\mathcal{O}\left(N^2 \ln N\right)$ seedless approach. This (and some care in the the 'split–merge' procedure) ensures that the jets it produces are insensitive to additional soft particles and collinear splittings (i.e. the jets are infrared and collinear safe).

The plugin library and include files are to be be found in the `plugins/SISCone` directory, and the main header file is `SISConePlugin.hh`. The `SISConePlugin` class has a constructor with the following structure

```
SISConePlugin (double cone_radius,
               double overlap_threshold = 0.5,
               int    n_pass_max = 0,
               double protojet_ptmin = 0.0,
               bool   caching = false,
               SISConePlugin::SplitMergeScale
                             split_merge_scale = SISConePlugin::SM_pttilde);
```

A cone centered at y_c, ϕ_c is stable if the sum of momenta of all particles i satisfying $\Delta y_{ic}^2 + \Delta\phi_{ic}^2 <$ `cone_radius`^{2} has rapidity y_c, ϕ_c. The `overlap_threshold` is the fraction of overlapping momentum above which two protojets are merged in a Tevatron Run II type [7] split–merge procedure.[11] The radius and overlap parameters are a common feature of most modern cone algorithms. Because some event particles are not to be found in any stable cone [16], SISCone can carry out multiple stable-cone search passes (as advocated in [17]): in each pass one searches for stable cones using just the subset of particles not present in any stable cone in earlier passes. Up to `n_pass_max` passes are carried out, and the algorithm automatically stops at the highest pass that gives no new stable cones. The default of `n_pass_max` $= 0$ is equivalent to setting it to ∞. Since concern has been expressed that an excessive number of stable cones may complicate cone jets in events with high noise [7], the `protojet_ptmin` parameter allows one to use only protojets with $p_t \geq$ `protojet_ptmin` in the split–merge phase (all others are thrown away).[12]

In many cases SISCone's most time-consuming step is the search for stable cones. If one has multiple `SISConePlugin`-based jet definitions, each with `caching=true`, a check will be carried out whether the previously clustered event had the same set of particles and the same cone radius and number of passes. If it did, the stable cones are simply reused from the previous event, rather than being recalculated, and only the split–merge step is repeated, often leading to considerable speed gains.

A final comment concerns the `split_merge_scale` parameter. This controls both the scale used for ordering the protojets during the split–merge step during the split–merge step, and also the scale used to measure the degree of overlap between protojets. While various options have been implemented,

```
enum SplitMergeScale {SM_pt, SM_Et, SM_mt, SM_pttilde };
```

[11] Though its default value is 0.5 (retained for backwards compatibility of the interface) we strongly recommend using a higher value, e.g. 0.75, especially in high-noise environments, in order to disfavour the production of monster jets through repeated merge operations.

[12] Early experience indicates that `protojet_ptmin` is actually perfectly adequate and that potential problems of massively agglomerated jets that can occur in high-noise environments (for a wide range of cone algorithms) can be addressed with a slightly larger value of the `overlap_threshold`, $\gtrsim 0.6$.

we recommend using only the last of them $\tilde{p}_t = \sum_{i\in\text{jet}} |p_{t,i}|$, which is also the default scale. The other scales are included only for historical comparison purposes: p_t (used in several other codes) is IR unsafe for events whose hadronic component conserves momentum, E_t (advocated in [7]) is not boost-invariant, and $m_t = \sqrt{m^2 + p_t^2}$ is IR unsafe for events whose hadronic component conserves momentum and stems from the decay of two identical particles.

An example of the use of the SISCone plugin would be as follows:

```
// define a SISCone plugin pointer
fastjet::SISConePlugin * plugin;

// allocate a new plugin for SISCone
double cone_radius = 0.7;
double overlap_threshold = 0.5;
plugin = new fastjet::SISConePlugin (cone_radius, overlap_threshold);

// create a jet-definition based on the plugin
fastjet::JetDefinition jet_def(plugin);

// prepare the set of particles
vector<fastjet::PseudoJet> particles;
read_input_particles(cin, particles); // or whatever you want here

// run the jet algorithm and look at the jets
fastjet::ClusterSequence clust_seq(particles, jet_def);
vector<fastjet::PseudoJet> inclusive_jets = clust_seq.inclusive_jets();
// then analyse the jets as for native FastJet jets

// only when you will no longer be using the jet definition, or
// ClusterSequence objects that involve it, may you delete the
// plugin
delete plugin;
```

Note that the it makes no sense to ask for exclusive jets from a SISCone based `ClusterSequence`.

Some extra output information is appropriate for a cone algorithm that is not of relevance in clustering algorithms, through the `extras` resource,

```
const fastjet::SISConeExtras * extras =
          dynamic_cast<const fastjet::SISConeExtras *>(clust_seq.extras());
```

To determine the pass at which a given jet was found, one does the following

```
int pass = extras->pass(jet);
```

The user may also obtain a list of the positions of original stable cones as follows:

```
vector<PseudoJet> stable_cones(extras->stable_cones());
```

The stable cones are represented as four-momenta, for which only the rapidity and azimuth are meaningful. The `user_index()` indicates the pass at which a given stable cone was found.

In the current version of SISCone, the `user_index()` of a jet also corresponds to the pass at which it was found, however this manner of accessing the pass for a jet is *deprecated* (for reasons related to the internal representation of jets, it fails for single-particle jets). It is retained in version 2.4 for backwards compatibility, but will be removed at some stage in the future.

SISCone uses E-scheme recombination internally and also for constructing the final jets from the list of constituents. For the latter task, the user may instead instruct SISCone to use the jet-definition's own recombiner, with the command

```
  plugin->set_use_jet_def_recombiner(true);
```

In this case the `user_index()` no longer contains the information about the pass.

Since SISCone is infrared safe, it may meaningfully be used also with the `ClusterSequenceArea` class. Note however that in that case ones loses the cone-specific information from the jets, because of the way `FastJet` filters out the information relating to ghosts in the clustering. If the user needs both areas and cone-specific information, she/he may use the `ClusterSequenceActiveAreaExplicitGhosts` class (for usage information, see the corresponding `.hh` file).

A final remark concerns speed and memory requirements: as mentioned above, SISCone takes $\mathcal{O}(N^2 \ln N)$ time to find jets, and the memory use is $\mathcal{O}(N^2)$; taking $N = 10^3$ as a reference point, it runs in a few tenths of a second, making it about 100 times slower than native `FastJet` algorithms. These are 'expected' results, i.e. valid for a suitably random set of particles. In area determinations, the ghost particles are anything but random, and run times and memory usage are, in practice, somewhat larger than for a normal QCD event with the same number of particles. We therefore recommend running with not too small a `ghost_area` (e.g. ~ 0.05) and using `grid_scatter` $= 1$, which helps to reduce the number of stable cones (and correspondingly, the time and memory usage of the subsequent split–merge step). An alternative, which has been found to be acceptable in most situations, is to use a passive area, since this is relatively fast to calculate with SISCone.

5.3 Other plugins for *pp*

Not all plugins are enabled by default in `FastJet`. At configuration time `./configure --help` will indicate which ones get enabled by default. To enable all plugins, run `configure` with the argument `--enable-allplugins`, while to enable all but PxCone (which requires fortran, and can introduce link-time issues) use `--enable-allcxxplugins`.

All plugins are in the `fastjet` namespace. Below we show the file that needs to be included and the constructor for each plugin.

Except where stated, the usual way to access jets from these plugins is through `ClusterSequence::inclusive_jets()`.

Most of the algorithms listed below are either infrared (IR) or collinear unsafe. The details are indicated for each algorithm as follows: IR$_{n+1}$ means that the hard jets may be modified if, to an ensemble of n hard particles in a common neighbourhood, one adds a single soft particle; Coll$_{n+1}$ means that for n hard particles in a common neighbourhood, the collinear splitting of one of them may modify the hard jets. The `FastJet` authors (and numerous theory-experiment accords) advise against the use IR and collinear safe jet algorithms. Interfaces to these algorithms have been provided mainly for legacy comparison purposes.

As of `FastJet` version 2.4, this section is partially incomplete (in particular it misses many references). This will hopefully evolve for future versions.

5.3.1 CDF Midpoint.

One of the two algorithms used by CDF in Run II of the Tevatron, based on [7]. It is a midpoint-type iterative cone with a split–merge step.

```
#include ``fastjet/CDFCones.hh''
// ...
CDFMidPointPlugin(double R,
                  double overlap_threshold,
                  double seed_threshold = 1.0,
                  double cone_area_fraction = 1.0);
```

The overlap threshold (f) used by CDF is usually 0.5, the seed threshold is 1 GeV and in most measurements the cone area fraction is 1. With an area fraction < 1 this becomes the searchcone algorithm of [16].

Further control over the plugin can be obtained by consulting the header file.

The underlying code for this algorithm was taken from a webpage [18] provided by Joey Huston (with minor modifications to ensure reasonable run times with optimising compilers for 32-bit intel processors — these modifications do not affect the final jets).

Note: this algorithm is IR_{3+1} unsafe (in the limit of zero seed threshold [15]; with `cone_area_fraction`$\neq 1$ it becomes IR_{2+1} unsafe [17]). It is to be deprecated for new experimental or theoretical analyses.

5.3.2 CDF JetClu

The other algorithm used by CDF during Run II, as well as their main algorithm during Run I [19].

```
#include ``fastjet/CDFCones.hh''
// ...
CDFJetCluPlugin (double   cone_radius,
                 double   overlap_threshold,
                 double   seed_threshold = 1.0,
                 int      iratch = 1);
```

This is an iterative cone with split-merge and optional "ratcheting" if `iratch == 1` (particles that appear in one iteration of a cone are retained in future iterations). The overlap threshold is usually set to 0.75 in CDF analyses.

Further control over the plugin can be obtained by consulting the header file.

The underlying code for this algorithm was taken from a webpage provided by Joey Huston. [18]

Note: this algorithm is IR_{2+1} unsafe (and some IR unsafety persists with non-zero seed threshold). It is to be deprecated for new experimental or theoretical analyses. Note also that the underlying implementation groups particles ogether into calorimeter towers (with CDF-type geometry) before running the jet algorithm.

5.3.3 DØ Run I cone

The main algorithm used by DØ in Run I of the Tevatron [20], which is an iterative cone algorithm with a split-merge. It comes in two versions

```
#include ‘‘fastjet/D0RunIpre96ConePlugin.hh’’
// ...
D0RunIpre96ConePlugin (double R,
                       double min_jet_Et,
                       double split_ratio = 0.5);
```

and

```
#include ‘‘fastjet/D0RunIConePlugin.hh’’
// ...
D0RunIConePlugin (double R,
                  double min_jet_Et,
                  double split_ratio = 0.5);
```

corresponding to versions of the algorithm used respectively before and after 1996. They differ only in the way the jet momenta are calculated, as described in [20].

Instead of the seed threshold used in the CDF cones, the algorithm places a cut on the minimum E_t of the cones during iteration (related to `min_jet_Et`). The `split_ratio` is the same as the overlap threshold in other split-merge based algorithms (DØ usually use 0.5). It is the `FastJet` authors' understanding that the value used for `min_jet_Et` was 8 GeV.

The underlying code for this algorithm was provided by Lars Sonnenschein.

Note: this algorithm is IR_{2+1} unsafe. It is recommended that it be used only for the purpose of comparison with Run I data from DØ.

5.3.4 DØ Run II cone

The main algorithm used by DØ in Run II of the Tevatron, which is a midpoint type iterative cone with split-merge.

```
#include ‘‘fastjet/D0RunIIConePlugin.hh’’
// ...
D0RunIIConePlugin (double R,
                   double min_jet_Et,
                   double split_ratio = 0.5);
```

The parameters have the same meaning as in the DØ Run I cone and, once again, instead of a seed threshold there is a cut on the minimum E_t of the cones during iteration (related to `min_jet_Et`). It is the `FastJet` authors' understanding that two values have been used for `min_jet_Et`, 8 GeV (in earlier publications) and 6 GeV (in more recent publications).

The underlying code for this algorithm was provided by Lars Sonnenschein.

Note: this algorithm is IR_{3+1} unsafe (IR_{2+1} for jets with energy too close to `min_jet_Et`). It is to be deprecated for new experimental or theoretical analyses.

5.3.5 ATLAS iterative cone

The (iterative) cone (with split-merge) algorithm used by ATLAS during the preparation for the LHC.

```
#include ‘‘fastjet/AtlasConePlugin.hh’’
// ...
```

```
ATLASConePlugin (double R,
                 double seedPt = 2.0,
                 double f = 0.5);
```

f is the overlap threshold

The underlying code for this algorithm was extracted from SpartyJet [21]. Since version 3.0 of FastJet it is a slightly modified version that we distribute, where an internal sort function has been replaced with a stable_sort, to ensure reproducibility of results across compilers and architectures (results are unchanged when the results of the sort are unambiguous).

Note: this algorithm is IR_{2+1} unsafe (in the limit of zero seed threshold). It is to be deprecated for new experimental or theoretical analyses.

5.3.6 CMS iterative cone

The (iterative) cone (with progressive removal) algorithm used by CMS during the preparation for the LHC.

```
#include ‘‘fastjet/CMSIterativeConePlugin.hh’’
// ...
CMSIterativeConePlugin (double ConeRadius, double SeedThreshold=0.0);
```

The underlying code for this algorithm was extracted from the CMSSW web site, with certain small service routines having been rewritten by the FastJet authors. The resulting code was validated by clustering 1000 events with the original version of the CMS software and comparing the output to the clustering performed with the FastJet plugin. The jet contents were identical in all cases. However the jet momenta differed at a relative precision level of 10^{-7}, related to the use of single-precision arithmetic at some internal stage of the CMS software (while the FastJet version is in double precision).

Note: this algorithm is Coll_{3+1} unsafe [13]. It is to be deprecated for new experimental or theoretical analyses.

5.3.7 PxCone

A fortran plugin for the PxCone algorithm, which is an iterative cone with midpoints and a split-drop procedure

```
#include ‘‘fastjet/PxConePlugin.hh’’
// ...
PxConePlugin (double  cone_radius      ,
              double  min_jet_energy = 5.0  ,
              double  overlap_threshold = 0.5,
              bool    E_scheme_jets = false);
```

with a threshold on the minimum cone transverse energy if it is to be included in the split-drop stage. If E_scheme_jets is true then the plugin applies an E-scheme recombination to provide the momenta of the final jets (by default an E_t type recombination scheme is used).

The base code for this plugin is written in Fortran and, on some systems, problems have been reported at the link stage due to mixing Fortran and C++. The Fortran code has been modified by the FastJet authors to provide the same jets regardless of the order of the input particles. This involved a small modification of the midpoint procedure, which can have a minor effect on the final jets and should make the algorithm correspond to the description of [22].

The underlying code for this algorithm was taken from a google search for PxCone! [23]. It is also described in ref. [22].

Note: this algorithm is IR_{3+1} unsafe. It is to be deprecated for new experimental or theoretical analyses.

5.3.8 TrackJet

This algorithm has been used at the Tevatron for identifying jets from charged-particle tracks in underlying-event studies (a citation is needed here!).

```
#include ‘‘fastjet/TrackJetPlugin.hh’’
// ...
TrackJetPlugin (double radius,
                RecombinationScheme jet_recombination_scheme=pt_scheme,
                RecombinationScheme track_recombination_scheme=pt_scheme);
```

Two recombination schemes are involved: the first one indicates how momenta are recombined to provide the final jets (once particle-jet assignments are known), the second one indicates how momenta are combined in the procedure that constructs the jets.

The underlying code for this algorithm was written by the `FastJet` authors, based on code extracts from the Rivet implementation, written by Andy Buckley with input from Manuel Bahr and Rick Field. Since version 3.0 of `FastJet` it is a slightly modified version that we distribute, where an internal `sort` function has been replaced with a `stable_sort`, to ensure reproducibility of results across compilers and architectures (results are unchanged when the results of the sort are unambiguous).

Note: this algorithm is believed to be Coll_{3+1} unsafe. It is to be deprecated for new experimental or theoretical analyses.

5.3.9 GridJet

GridJet allows you to define a grid and then cluster particles such that all particles in a common grid cell combine to form a jet. Its main interest is in providing fast clustering for high multiplicities (the clustering time scales linearly with the number of particles). The jets that it forms are not always physically meaningful: for example, a genuine physical jet may lie at the corner of 4 grid cells and so be split up somewhat arbitrarily into 4 pieces. However for some purposes (such as background estimation) this drawback is offset by the greater uniformity of the area of the jets. Its interface is as follows

```
#include ‘‘fastjet/GridJetPlugin.hh’’
// ...
GridJetPlugin (double ymax, double requested_grid_spacing);
```

creating a grid that covers $|y| <$`ymax` with a grid spacing close to the `requested_grid_spacing`: the spacings chosen in ϕ and y are those that are closest to the requested spacing while also giving an integer number of grid cells that fit exactly into the rapidity and $0 < \phi < 2\pi$ ranges.

Note that for background estimation purposes the `GridMedianBackgroundEstimator` is much faster than using the `GridJetPlugin` with ghosts and a `JetMedianBackgroundEstimator`.

5.4 Other plugins for e^+e^-

5.4.1 Cambridge algorithm

The original e^+e^- cambridge [11] algorithm is provided as a plugin:

```
#include ‘‘fastjet/EECambridgePlugin.hh’’
// ...
EECambridgePlugin (double ycut);
```

This algorithms performs sequential recombination of the pair of particles that is closest in angle, except when $y_{ij} = \frac{2\min(E_i^2,E_j^2)}{Q^2}(1-\cos\theta) > y_{cut}$, in which case the less energetic of i and j is labelled a jet, and the other member of the pair remains free to cluster.

To access the jets, the user should use the `inclusive_jets()`, *i.e.* as they would for the majority of the pp algorithms.

The underlying code for this algorithm was written by the `FastJet` authors.

5.4.2 Jade algorithm

The JADE algorithm [24, 25], a sequential recombination algorithm with distance measure $d_{ij} = 2E_iE_j(1-\cos\theta)$, is available through

```
#include ‘‘fastjet/JadePlugin.hh’’
// ...
JadePlugin ();
```

To access the jets at a given $y_{cut} = d_{cut}/Q^2$, the user should call `ClusterSequence::exclusive_jets_ycut(double ycut)`.

Note: the JADE algorithm has been used with various recombination schemes. The current plugin will use whatever recombination scheme the user specifies with for the jet definition. The default E-scheme is what was used in the original JADE publication [24]. To modify the recombination scheme, the user may first construct the jet definition and then use either of

```
void JetDefinition::set_recombination_scheme(RecombinationScheme recomb_scheme);
void JetDefinition::set_recombiner(const Recombiner * recomb)
```

(cf. sections 3.4,D.1) to modify the recombination scheme.

The underlying code for this algorithm was written by the `FastJet` authors.

6 Selectors

Analyses often place constraints (cuts) on jets' transverse momenta, rapidity, maybe consider only some N hardest jets, etc. There are situations in which it is convenient to be able to define a set of cuts in one part of a program and then have it used elsewhere. To allow for this, we provide a `fastjet::Selector` class, available through

```
#include "fastjet/Selector.hh"
```

6.1 Essential usage

As an example of how `Selector`s are used, suppose that we have a vector of jets, `jets`, and wish to select those that have rapidities $|y| < 2.5$ and transverse momenta above 20 GeV. We might write the following:

```
Selector select_rapidity = SelectorAbsRapMax(2.5);
Selector select_pt       = SelectorPtMin(20.0);
Selector select_both     = select_pt && select_rapidity;

vector<PseudoJet> selected_jets = select_both(jets);
```

Here, `Selector` is a class, while `SelectorAbsRapMax` and `SelectorPtMin` are functions that return an instance of the `Selector` class containing the internal information needed to carry out the selection. `Selector::operator(const vector<PseudoJet> & jets)` takes the jets given as input and returns a vector containing those that pass the selection cuts. The logical operations `&&`, `||` and `!` enable different selectors to be combined.

Nearly all selectors, like those above, apply jet by jet (they return `Selector::applies_jet_by_jet()` true). For these, one can query whether a single jet passes the selection with the help of the function `bool Selector::pass(const PseudoJet &)`.

There are also selectors that only make sense applied to an ensemble of jets. This is the case specifically for `SelectorNHardest(unsigned int n)`, which, acting on an ensemble of jets, selects the n with largest transverse momenta. If there are fewer than n jets, then all jets pass.

When a selector is applied to an ensemble of jets one can also use

```
Selector::sift(vector<PseudoJet> & jets,
               vector<PseudoJet> & jets_that_pass,
               vector<PseudoJet> & jets_that_fail)
```

to obtain the vectors of `PseudoJet`s that pass or fail the selection.

For selectors that apply jet-by-jet, the selectors on either side of the logical operators `&&` and `||` naturally commute. For operators that act only on the ensemble of jets the behaviour needs specifying. The design choice that we have made is that

```
SelectorNHardest(2)    && SelectorAbsRapMax(2.5)
SelectorAbsRapMax(2.5) && SelectorNHardest(2)
```

give identical results: in logical combinations of selectors, each constituent selector is applied independently to the ensemble of jets, and then a decision whether a jet passes is determined from the corresponding logical combination of each of the selectors' results. Thus, here only jets that are among the 2 hardest of the whole ensemble and that have $|y| < 2.5$ will be selected. If one wishes to first apply a rapidity cut, and *then* find the 2 hardest among those jets that pass the rapidity cut, then one should instead use the `*` operator:

```
SelectorNHardest(2)  *  SelectorAbsRapMax(2.5)
```

In this combined selector, the right-hand selector is applied first, and then the left-hand selector is applied to the results of the right-hand selection.

A complementary selector can also be defined using the negation operator. For instance

```
Selector sel_allbut2hardest = !SelectorNHardest(2);
```

Note that, if directly applying (as opposed to first defining) a similar negated selector to a collection of jets, one should write

```
vector<PseudoJet> allbut2hardest = (!SelectorNHardest(2))(jets);
```

with the brackets around the selector definition being now necessary due to `()` having higher precedence in C++ than boolean operators.

A user can obtain information on what a given `Selector` does by calling its `description()` member function. This behaves sensibly also for compound selectors.

New selectors can be implemented as described in section D.3.

6.1.1 Other information about selectors

Selectors contain a certain amount of additional information that can provide useful hints to the functions using them.

One such piece of information is a selector's rapidity extent, accessible through a `get_rapidity_extent(rapmin,rapmax)` call, which is useful in the context of background estimation (section 8). If it is not sensible to talk about a rapidity extent for a given selector (e.g. for `SelectorPtMin`) the rapidity limits that are returned are the largest (negative and positive) numbers that can be represented as doubles. The function `is_geometric()` returns true if the selector places constraints only on rapidity and azimuth.

Selectors may also have areas associated with them (in analogy with jet areas, section 7). The `has_finite_area()` member function returns true if a selector has a meaningful finite area; The `area()` function returns this area. In some cases the area may be computed using ghosts (by default with ghosts of area 0.01; the user can specify a different ghost area as an argument to the `area` function).

6.2 Available selectors

6.2.1 Absolute kinematical cuts

A number of selectors have been implemented following the naming convention `Selector`{*Var*}{*LimitType*}. The {*Var*} indicates which variable is being cut on, and can be one of

```
    pt, Et, E, Mass, Rap, AbsRap, Eta, AbsEta
```

The {*LimitType*} indicates whether one places a lower-limit on the variable, an upper limit or a range, corresponding to the choices

```
    Min, Max, Range
```

A couple of examples are

```
SelectorPtMin(25.0)        // Selects pt > 25 (units are user's default for momenta)
SelectorRapRange(1.9,4.9)  // Selects 1.9 < y < 4.9
```

Following a similar naming convention, there are also `SelectorPhiRange(`$\phi_{\min}, \phi_{\max}$`)` and `SelectorRapPhiRange(`$y_{\min}, y_{\max}, \phi_{\min}, \phi_{\max}$`)`.

6.2.2 Relative kinematical cuts

Some selectors take a *reference jet*. They have been developed because it is often useful for a selector to make its decision based on information about some other jet. For example one might wish to select all jets within some distance of a given reference jet; or all jets whose transverse momentum is at least some fraction of a reference jet's. That reference jet may change from event to event, or even from one invocation of the Selector to the next, even though the Selector is fundamentally performing the same underlying type of action.

The available selectors of this kind are:

```
SelectorCircle(R)                    // a circle of radius R around the reference jet
SelectorDoughnut(R_in, R_out)        // a doughnut between R_in and R_out
SelectorStrip(half_width)            // a rapidity strip 2*half_width large
SelectorRectangle(half_rap_width, half_phi_width) // a rectangle in rapidity and phi
SelectorPtFractionMin(f)             // p_t larger than f p_t^ref
```

One example of selectors taking a reference jet is the following. First, one constructs the selector,

```
Selector sel = SelectorCircle(1.0);
```

Then if one is interested in the subset of `jets` near `jet1`, and then those near `jet2`, one performs the following operations:

```
sel.set_reference(jet1);
vector<PseudoJet> jets_near_jet1 = sel(jets);

sel.set_reference(jet2);
vector<PseudoJet> jets_near_jet2 = sel(jets);
```

If one uses a selector that takes a reference without the reference having been actually set, an exception will be thrown. If one sets a reference for a compound selector, the reference is automatically set for all components that take a reference. One can verify whether a given selector takes a reference by calling the member function

```
bool Selector::takes_reference() const;
```

Attempting to set a reference for a Selector that returns `false` here will cause an exception to be thrown.

6.2.3 Other selectors

The following selectors are also available:

```
SelectorNHardest(n)      //  selects the n hardest jets
SelectorIsPureGhost()    //  selects jets that are made exclusively of ghost particles
SelectorIsZero()         //  selects jets with zero momentum
SelectorIdentity()       //  selects everything. Included for completeness
```

7 Jet areas

Since a jet is made up of only a finite number of particles, one needs a specific definition in order to make its *area* (i.e. the surface in the y-ϕ plane over which it extends) an unambiguous concept. Three definitions of area have been proposed in [26] and they are implemented in `FastJet`:

- Active areas add a uniform background of extremely soft massless 'ghost' particles to the event and allow them to participate in the clustering. The area of a given jet is proportional to the number of ghosts it contains. Because the ghosts are extremely soft (and sensible jet algorithms are infrared safe), the presence of the ghosts does not affect the set of user particles that end up in a given jet.

- Passive areas are defined as follows. One adds a single randomly placed ghost at a time to the event. One examines which jet (if any) the ghost ends up in. One repeats the procedure many times and the passive area of a jet is then proportional to the probability of it containing the ghost.

- The Voronoi area of a jet is the sum of the Voronoi areas of its constituent particles. The Voronoi area of a particle is obtained by determining the Voronoi diagram for the event as a whole, and intersecting the Voronoi cell of the particle with a circle of radius R centred on the particle. Note that for the k_t algorithm (but not for Cambridge/Aachen or anti-k_t, nor in general for any other algorithm) the Voronoi area of a jet coincides with its passive area.

The area can be calculated either as a scalar, or as a 4-vector. Essentially the scalar case corresponds to counting the number of ghosts in the jet, while the 4-vector case corresponds to summing their 4-vectors (normalised such that for a narrow jet, the transverse component of the 4-vector is equal to the scalar area).

Jet areas are obtained by clustering with the class `ClusterSequenceArea` (rather than `ClusterSequence`, from which it is derived). Its constructor takes an `AreaDefinition` argument in addition to the list of particles and the `JetDefinition`.

It is worth noting that in the limit of very densely populated events, all area definitions tend to the same value [26]. It might therefore be advantageous to select a Voronoi area type, rather than an active one, when using areas for phenomenological tasks like pileup subtraction [27], as it generally requires less CPU time to calculate.

To summarise, in order to access the areas of the jets the user is exposed to two main classes:

```
class fastjet::AreaDefinition;
class fastjet::ClusterSequenceArea;
```

If jet areas are to be used to study the level of a diffuse noise which might be present in the event (like underlying event particles or pileup) and maybe subtract it from jets, two further classes are useful:

```
class fastjet::JetMedianBackgroundEstimator
class fastjet::Subtractor
```

These classes are described in detail below and an example program is given in `example/07-subtraction.cc`.

7.1 fastjet::AreaDefinition

Area definitions are contained in `AreaDefinition` class. Its two main constructors are:

```
AreaDefinition(fastjet::AreaType area_type,
               fastjet::GhostedAreaSpec ghost_spec);
```

for the various active and passive areas (which all involve ghosts) and

```
AreaDefinition(fastjet::VoronoiAreaSpec voronoi_spec);
```

for the Voronoi area. A default constructor exists, and provides an active area with a `ghost_spec` that is suitable for a majority of area measurements with clustering algorithms and typical Tevatron and LHC rapidity coverage.

Information about the current `AreaDefinition` can be retrieved as follows:

```
/// return a description of the current area definition
std::string description() const ;

/// return info about the type of area being used by this defn
AreaType area_type() const ;

/// return a reference to the ghosted area spec (where relevant)
const GhostedAreaSpec  & ghost_spec()  const ;

/// return a reference to the voronoi area spec (where relevant)
const VoronoiAreaSpec & voronoi_spec() const ;
```

7.1.1 Ghosted Areas (active and passive)

There are two variants each of the active and passive areas, as defined by the `AreaType enum`:

```
enum AreaType{ [...],
               active_area,
               active_area_explicit_ghosts,
               one_ghost_passive_area,
               passive_area,
               [...]};
```

The two active variants give identical results. The second one explicitly includes the ghosts when the user requests the constituents of a jet. The first of the passive variants explicitly runs through the procedure mentioned above, *i.e.* it clusters the events with one ghost at a time, and repeats this for very many ghosts. This can be quite slow, so we also provide the `passive_area` option, which makes use of information specific to the jet algorithm in order to speed up the passive-area determination.[13]

In order to carry out a clustering with a ghosted area determination, the user should also create an object that specifies how to distribute the ghosts.[14] This is done via the class `fastjet::GhostedAreaSpec` whose constructor is

```
GhostedAreaSpec(double ghost_maxrap,
                int    repeat        = 1,
                double ghost_area    = 0.01,
```

[13]This ability is provided for k_t, Cambridge/Aachen, anti-k_t and the SISCone plugin. In the case of k_t it is actually a Voronoi area that is used, since this can be shown to be equivalent to the passive area [26]. For other algorithms it defaults back to the `one_ghost_passive_area` approach.

[14]Or accept a default — which uses the default values listed in the explicit constructor and `ghost_maxrap` = 6

```
                 double grid_scatter  = 1.0,
                 double kt_scatter    = 0.1,
                 double mean_ghost_kt = 1e-100);
```

The ghosts are distributed on a uniform grid in y and ϕ, with small random fluctuations to avoid degeneracies. The `ghost_maxrap` defines the maximum rapidity up to which ghosts are generated — typically jet areas will be reliable for jets up to rapidity $|y| \simeq$ `ghost_maxrap` $- R$. The `ghost_area` is the area associated with a single ghost. The number of ghosts is inversely proportional to the ghost area, and so a smaller area leads to a longer CPU-time for clustering. However small ghost areas give more accurate results. We have found the default ghost area given above to be suitable for most applications.

For sparse events, the set of ghost particles that end up in a given jet is not unique, and depends on the degeneracy-breaking random shifts added to the ghost positions, as compared to a perfect grid distribution. To obtain a reliable area one may then repeat the area determination several times, the number of times being specified by the `repeat` variable. For hadron-level events a value of 5 is sufficient to give jet areas that are determined to within a few percent. In practice it is usually satisfactory even to set `repeat` $= 1$ and this is the default since it runs faster. For events with a dense distribution of true particles, there is no degeneracy in the ghost clustering and there is no need at all to use `repeat` > 1. If `repeat` > 1, a statistical uncertainty on the area, given by $\sigma/\sqrt{\texttt{repeat} - 1}$, is provided for each jet. Note that the `repeat` value is ignored (*i.e.* taken to be 1) for `active_area_explicit_ghosts` and meaningless for the passive area in the k_t algorithm, which just calculates the Voronoi area discussed below (since they are identical).

Other variables that the user may wish to set are: `grid_scatter` and `kt_scatter`, which are fractional random fluctuations of the position of the ghosts on the y-ϕ grid and of their transverse momentum; and `mean_ghost_kt` which is the average transverse momentum of the ghosts.

Even after the initialisation, the parameters can be modified by

```
void set_ghost_area    (double ) ;
void set_ghost_etamax  (double ) ;
void set_ghost_maxrap  (double ) ;
void set_grid_scatter  (double ) ;
void set_kt_scatter    (double ) ;
void set_mean_ghost_kt (double ) ;
void set_repeat        (int    ) ;
```

and information about the `GhostedAreaSpec` in use can be retrieved as follows:

```
/// for a summary
std::string description() const;

double ghost_etamax  () const ;
double ghost_maxrap  () const ;
double ghost_area    () const ;
double grid_scatter  () const ;
double kt_scatter    () const ;
double mean_ghost_kt () const ;
int    repeat        () const ;
```

An alternative constructor

```
GhostedAreaSpec(const Selector & selector,
                int    repeat        = 1,
                double ghost_area    = 0.01,
                double grid_scatter  = 1.0,
                double kt_scatter    = 0.1,
                double mean_ghost_kt = 1e-100);
```

allows the user to have ghosts placed in the region specified by the `selector`, which must be purely geometrical and have finite rapidity extent. This option is useful, for example, if one wishes to place ghosts in a manner that reflects a detector's acceptance for particles. Though this will not give the "true" geometrical area of jets near the boundary of the acceptance, it will ensure that the area measures sensitivity to contamination only in the region where contamination by particles can effectively take place. This is often more useful information than the geometrical area.

7.1.2 Voronoi Areas

The Voronoi areas of jets are evaluated by summing the corresponding Voronoi areas of the jets' constituents. The latter are obtained by considering the intersection between the Voronoi cell of each particle and a circle of radius R centred on the particle's position in the rapidity-azimuth plane.

The jets' Voronoi areas can be obtained from `ClusterSequenceArea` by passing the proper `VoronoiAreaSpec` specification to `AreaDefinition`. Its constuctors are

```
/// default constructor (effective_Rfact = 1)
VoronoiAreaSpec() ;

/// constructor that allows you to set effective_Rfact
VoronoiAreaSpec(double effective_Rfact) ;
```

The second constructor allows one to modify (by a multiplicative factor `effective_Rfact`) the radius of the circle which is intersected with the Voronoi cells. With `effective_Rfact` $= 1$, for the k_t algorithm, the Voronoi area is equivalent to the passive area.

Information about the specification in use is returned by

```
/// return the value of effective_Rfact
double effective_Rfact() const ;

/// return a textual description of the area definition.
std::string description() const ;
```

The Voronoi areas are calculated with the help of Fortune's ($N \ln N$) Voronoi diagram generator for planar static point sets [28].

One use for the Voronoi area is in background determination with the k_t algorithm (below, section 8): with the choice `effective_Rfact` $\simeq 0.9$ it provides an acceptable approximation to the k_t algorithm's active area that is often significantly faster to compute than the active area.

7.2 fastjet::ClusterSequenceArea

This is the main class[15] to which the user is exposed for accessing cluster sequences that include information about jet areas. It is derived from `fastjet::ClusterSequenceAreaBase` (itself derived from `ClusterSequence`) and includes the methods

```
/// return a reference to the area definition
virtual const fastjet::AreaDefinition & area_def() const ;

/// return the area associated with the given jet
virtual double area (const PseudoJet & jet) const ;

/// return the error (uncertainty) associated with the determination
/// of the area of the jet; returns 0 when the repeat value = 1, and
/// also for the active_area_explicit_ghosts and certain passive areas
virtual double area_error (const PseudoJet & jet) const ;

/// return a PseudoJet whose 4-vector is defined by the following integral
///
///       ∫ dydφ PseudoJet(y,φ,pt=1) * Θ("y,φ inside jet boundary")
///
/// where PseudoJet(y,φ,pt=1) is a 4-vector with the given
/// rapidity (y), azimuth (φ) and pt=1, while Θ("y,φ inside jet boundary")
/// is a function that is 1 when y,φ define a direction inside the
/// jet boundary and 0 otherwise.
///
virtual PseudoJet area_4vector(const PseudoJet & jet) const ;
```

When the `AreaType` is `active_area_explicit_ghosts`, one may additionally use the following function

```
/// true if a jet is made exclusively of ghosts
virtual bool is_pure_ghost(const PseudoJet & jet) const;
```

to determine whether a jet is made purely of ghosts. Its argument can also be one of the constituents of a jet, in which case it will return `true` if that constituent is a ghost.

8 Background estimation and subtraction

Events with hard jets are often accompanied by a more diffuse "background" of relatively soft particles, for example from the underlying event (in *pp* or PbPb collisions) or from pileup (in *pp* collisions). For many physical applications, it is often useful to be able to estimate characteristics of the background on an event-by-event basis, for example the p_t per unit area (ρ), or fluctuations from point to point (σ). One use of this information is to correct the hard jets for the soft contamination, as discussed below in section 8.5.

[15] `ClusterSequenceArea` makes use of one among `ClusterSequenceActiveAreaExplicitGhosts`, `ClusterSequenceActiveArea`, `ClusterSequencePassiveArea`, `ClusterSequence1GhostPassiveArea` or `ClusterSequenceVoronoiArea` (all of them in the `fastjet` namespace of course), according to the choice made with `AreaDefinition`. The user might of course also use these classes directly.

One of the issues in characterising the background is that it is difficult to introduce a robust criterion to distinguish "background" jets from hard jets. The main method that is available in `FastJet` involves the determination of the distribution of p_t/A for the jets in a given event (or region of the event) and then taking the median of the distribution as an estimate of ρ, as proposed in [27] and studied in detail also in [29]. This is largely insensitive to the presence of a handful of hard jets, and avoids any need for introducing a p_t scale to distinguish hard and background jets.

The original form of this method used the k_t or Cambridge/Aachen jet algorithms to find the jets. These algorithms have the advantage that the resulting jets tend to have reasonably uniform areas (whereas anti-k_t and SISCone suffer from jets with near zero areas or sometimes huge, "monster" jets, biasing the ρ determination and are not recommended). In the meantime a variant of the approach that has emerged is to cluster the particles into rectangular grid cells in y and ϕ and determine their median p_t/A. This has the advantage of simplicity and much greater speed. One may worry that a hard jet will sometimes lie at a corner of multiple grid cells, inducing larger biases in the median than with a normal jet finder jets, however we have (preliminarly) found this not to be an issue in practice, and further tests are planned in order to quantify these effects more precisely.

A choice that needs to be made in both the jet-based and grid-based variants of the median method is that of the jet radius or grid spacing. For the jet-based case we have found that a choice in the range $R = 0.4 - 0.6$ is generally adequate [27, 29] with a preference for larger values if the events are relatively sparse and smaller values if you expect your events to have many hard jets. The grid-based case has been less extensively explored, but initial studies suggest that grid spacings in the range $\delta y = \delta\phi \simeq 0.5 - 0.7$ are appropriate.

8.1 General Usage

The simplest workflow for background estimation is first, outside the event loop, to create a background estimator. For the jet-based method, one creates a `fastjet::JetMedianBackgroundEstimator`,

```
#include "fastjet/tools/JetMedianBackgroundEstimator.hh"
// ...
JetMedianBackgroundEstimator bge(const Selector & selector,
                                 const JetDefinition & jet_def,
                                 const AreaDefinition & area_def);
```

where the selector is used to indicate which jets are used for background estimation (for simple use cases, one just specifies a rapidity range, e.g. `SelectorAbsRapMax(4.5)` to use all jets with $|y| < 4.5$), together with a jet definition (typically, the k_t or Cambridge/Aachen jet algorithm with $R = 0.4-0.6$) and an area definition (typically, an active area with explicit ghosts is recommended[16]). For the grid based method one creates a `fastjet::GridMedianBackgroundEstimator`,

```
#include "fastjet/tools/GridMedianBackgroundEstimator.hh"
// ...
GridMedianBackgroundEstimator bge(double max_rapidity,
                                  double requested_grid_spacing);
```

[16]With the k_t algorithm one may also use a Voronoi area (`effective_Rfact = 0.9` is recommended), which has the advantage of being deterministic and faster than ghosted areas. In this case however one must use a selector which is geometrical and selects only jets well within the range of event particles. When using ghosts, instead, the selector can go up right to the edge of the acceptance, if the ghosts also only go right up to the edge.

where, as already mentioned above, one can use a grid spacing $\delta y = \delta\phi \simeq 0.5 - 0.7$. Both of these background estimators derive from a `fastjet::BackgroundEstimatorBase` class.

Then, for each event, one tells the background estimator about the event particles,

```
bge.set_particles(event_particles);
```

where `event_particles` is a vector of `PseudoJet`, and then extracts the background density and a measure of its fluctuations with the two following calls

```
// the median of (p_t/area) for grid cells, or for jets that pass the selection cut,
// making use also of information on empty area in the event (in the jets case)
rho = bge.rho();

// an estimate of the fluctuations in the p_t density per unit sqrt(A),
// which is obtained from the 1-sigma half-width of the distribution of pt/A.
// To be precise it is defined such that a fraction (1-0.6827)/2 of the jets
// (including empty jets) have p_t/A < rho - sigma sqrt(<A>)
sigma = bge.sigma();
```

Note that ρ and σ determinations count empty area within the relevant region as consisting of jets of zero p_t. Thus (roughly speaking), if more that half of the area covered by the jets selector or grid rapidity range is empty, the median estimate for ρ will be zero, as expected and appropriate for quiet events.

8.2 Positional dependence of background

One drawback of a ρ estimate based on all jets or grid cells from (say) $|y| < 4.5$ is that the background density in *pp* and heavy-ion collisions usually has some non-negligible dependence on rapidity (and sometimes azimuth) and this information is not correctly accounted for. Two techniques can help alleviate this problem. The first, available for now only in the case of the jet-based estimator, involves the use of a more local range for the determination of ρ, with the help of a `Selector` that is able to take a reference jet, e.g. `SelectorStrip(`Δy`)`, a strip of half-width Δy (which might be of order 1) centred on whichever jet is set as its reference. With this kind of selector, when the user calls either of the `JetMedianBackgroundEstimator` member functions

```
double rho  (const PseudoJet & jet); // p_t density per unit area A near jet
double sigma(const PseudoJet & jet); // fluctuations in the p_t density near jet
```

a `selector.set_reference(jet)` call is made to centre the selector on the specified jet. Then only the jets that pass the cut specified by this newly positioned `selector` are used to estimate ρ or σ.[17] This method is adequate if the number of jets that pass the selector is much larger than the number of hard jets in the range (otherwise the median p_t/A will be noticeably biased by the hard jets). It therefore tends to be suitable for dijet events in *pp* or PbPb collisions, but may fare less well in event samples such as hadronically decaying $t\bar{t}$ which may have many central hard jets. One can attempt to remove some given number of hard jets before carrying out the median estimation, e.g. with a `selector` such as

```
selector = SelectorStrip(Δy) * (!SelectorNHardest(2))
```

[17]If the selector does not take a reference jet, then these calls give identical results to the plain `rho()` and `sigma()` calls, unless a manual rapidity rescaling is in effect.

which removes the 2 hardest jets globally and then, of the remainder, takes the ones within the strip. This is however not always very effective, because one may not know how many hard jets to remove.[18]

In unpublished studies for the case of *pp* pileup events (with Pythia 8.145, tune 4C for the pileup [30]), we have found that an alternative method that is adequate for handling the rapidity dependence is to parametrise the average shape of the rapidity dependence from some number of pileup events and then carry out an event-by-event global ρ determination taking into account that average shape. This method is available for both grid and jet-based methods. One first creates an object that encodes the shape, e.g.

```
// gives rescaling(y) = 1.16 + 0 · y − 0.023 · y^2 + 0 · y^3 + 0.000041 · y^4
fastjet::BackgroundRescalingYPolynomial rescaling(1.16, 0, -0.023, 0, 0.000041);
```

(for other shapes, e.g. parametrization of elliptic flow in heavy ion collisions, with both rapidity and azimuth dependence, derive a class from `FunctionOfPseudoJet<double>` — see appendix C) and then one tells the background estimator (whether jet or grid based) about the rescaling with the call

```
// tell the JetMedianBackgroundEstimator or
// GridMedianBackgroundEstimator about the rescaling
bge.set_rescaling_class(&rescaling);
```

Subsequent calls will then take the median of the distribution $p_t/A/$`rescaling`(y) (rather than p_t/A) and any calls to `rho(jet)` and `sigma(jet)` will include an additional factor of `rescaling`$(y_{\texttt{jet}})$. Note that any overall factor in the rescaling function cancels out for `rho(jet)` and `sigma(jet)`, but not for calls to `rho()` and `sigma()` (which are in any case less meaningful when a rapidity dependence is being assumed for the background).

8.3 Other facilities

In the case of the jet-based estimator, a number of enquiry functions are available to obtain information used internally within the median ρ and σ determination.

```
// Returns the mean area of the jets used to actually compute the background properties,
// including empty area and jets (available also in grid-based estimator)
double mean_area() const;

// returns the number of jets used to actually compute the background properties
// (including empty jets)
unsigned int n_jets_used() const;

// Returns the estimate of the area (within the range defined by the selector) that
// is not occupied by jets.
double empty_area() const;

// Returns the number of empty jets used when computing the background properties.
double n_empty_jets() const;
```

For area definitions with explicit ghosts the last two functions return 0. For active areas without explicit ghosts the results are calculated based on the observed number of internally recorded pure

[18] If you use non-geometric selectors in determining ρ, the area must have explicit ghosts in order to simplify the determination of the empty area. If it does not, an error will be thrown.

ghost jets (and unclustered ghosts) that pass the selector; for Voronoi and passive areas, they are calculated using the difference between the total range area and the area of the jets contained in the range, with the number of empty jets then being calculated based on the average jet area for ghost jets ($0.55\pi R^2$ [26]). All four function above return a result corresponding to the last call to `rho` or `sigma` (as long as the particles, cluster sequence or selector have not changed in the meantime).

To allow flexibility in the user's workflow, alternative constructors to `JetMedianBackgroundEstimator` are provided. These can come in useful if, for example, the user wishes to carry out multiple background estimations with the same particles but different selectors, or wishes to take care of the jet clustering themselves, e.g. because the results of that same jet clustering will be used in multiple contexts and it is more efficient to perform it just once. These contructors are:

```
// create an estimator that uses the inclusive jets from the supplied cluster sequence
JetMedianBackgroundEstimator(const Selector & rho_range,
                             const ClusterSequenceAreaBase & csa);
// a default constructor that requires all information to be set later
JetMedianBackgroundEstimator();
```

In the first case, the background estimator already has all the information it needs. Instead, if the default constructor has been used, one can then employ

```
// (re)set the selector to be used for future calls to rho() etc.
void set_selector(const Selector & rho_range_selector);
// (re)set the cluster sequence to be used by future calls to rho() etc.
// (as with the cluster-sequence based constructor, its inclusive jets are used)
void set_cluster_sequence(const ClusterSequenceAreaBase & csa);
```

to set the rest of the necessary information. If a list of jets is already available, they can be submitted to the background estimator in place os a cluster sequence:

```
// (re)set the jets to be used by future calls to rho() etc.
void set_jets(const std::vector<PseudoJet> & jets);
```

Note that the jets passed via the `set_jets()` call above must all originate from a common `ClusterSequenceAreaBase` type class.

8.4 Backwards compatibility

The `JetMedianBackgroundEstimator` and `GridMedianBackgroundEstimator` classes are new to `FastJet` 3. In `FastJet` versions 2.3 and 2.4, the background estimation tools were instead integrated into the `ClusterSequenceAreaBase` class. Rather than using Selectors to specify the jets used in the background estimation, they used the `RangeDefinition` class. For the purpose of backwards compatibility, these facilities will remain present in all 3.0.x versions. Note that `ClusterSequenceAreaBase` now actually uses a Selector in its background estimation interface, and that a `RangeDefinition` is automatically converted to a Selector.

An explicit argument in ρ-determination calls in `FastJet` 2.4 concerned the choice between the use of scalar areas and the transverse component of the 4-vector area in the denominator of p_t/A. The transverse component gives the more accurate ρ determination and that is now the default in `JetMedianBackgroundEstimator`. The behaviour can be changed with a call to

```
void set_use_area_4vector(bool use_it = true);
```

Finally, the calculation of σ in FastJet 2.x incorrectly handled the limit of a small number of jets. This is now fixed in FastJet 3, but a call to `set_provide_fj2_sigma(true)` causes `JetMedianBackgroundEstimator` to reproduce that behaviour.

FastJet 2.x also placed the ghosts differently, resulting in different event-by-event rho estimates, and possibly a small systematic offset (scaling as the square-root of the ghost area) when ghosts and particles both covered identical (small) regions. This offset is no longer present with the FastJet 3 ghost placement. If the old behaviour is needed, a call to a specific `GhostedAreaSpec`'s `set_fj2_placement(true)` function causes ghosts to placed as in the 2.x series.

8.5 Background subtraction

A common use of an estimation of the background is to subtract its contamination from the transverse momentum of hard jets, in the form

$$p_{t,jet}^{sub} = p_{t,jet}^{raw} - \rho A_{jet} \tag{8}$$

or its 4-vector version

$$p_{\mu,jet}^{sub} = p_{\mu,jet}^{raw} - \rho A_{\mu,jet}\,, \tag{9}$$

as first advocated in [27].

To this end, the `Subtractor` class is defined in `include/tools/Subtractor.hh`. Its constructor takes a pointer to a background estimator:

```
JetMedianBackgroundEstimator bge(....); // or a grid-based estimator
Subtractor subtractor(&bge);
```

(it is also possible to construct the Subtractor with a fixed value for ρ). The subtractor can then be used as follows:

```
PseudoJet jet;
vector<PseudoJet> jets;
// ...
PseudoJet subtracted_jet = subtractor(jet);
vector<PseudoJet> subtracted_jets = subtractor(jets);
```

The subtractor normally returns `jet - bge.rho(jet)*jet.area_4vector()`. If `jet.perp() < bge.rho(jet)*jet.area_4vector().perp()`, then the subtractor instead returns a jet with zero 4-momentum (so that `(subtracted_jet==0)` returns `true`). In both cases, the returned jet retains the user and structural information of the original jet.

An example program is given in `example/07-subtraction.cc`.

Note that `Subtractor` derives from the `Transformer` class (see section 9) and hence from `FunctionOfPseudoJet<PseudoJet>` (cf. appendix C).

9 Jet transformers (substructure, taggers, etc...)

Performing post-clustering actions on jets has in recent years become quite widespread: for example, numerous techniques have been introduced to tag boosted hadronically objects, and various methods also exist for removing the underlying event and pileup from jets. FastJet 3 provides a common interface for such tools, intended to help simplify their usage and to guide authors of new ones. Below, we first discuss generic considerations about these tools, which we call `fastjet::Transformer`s. We then describe some that have already been implemented. New user-defined transformers can be implemented as described in section D.4.

A transformer derived from `Transformer`, e.g. the class `MyTransformer`, will generally be used as follows:

```
MyTransformer transformer;
PseudoJet transformed_jet = transformer(jet);
```

Often, transformers provide new structural information that is to be associated with the returned result. For a given transformer, say `MyTransformer`, the new information that is not already directly accessible from `PseudoJet` (like its `constituents`, `pieces` or `area` when they are available), can be accessed through

```
transformed_jet.structure_of<MyTransformer>()
```

which would give direct access to the structural information produced by `MyTransformer`. This is illustrated below on a case-by-case basis for each of the transformers that we discuss. Using the boolean function `transformed_jet.has_structure_of<MyTransformer>()` it is possible to check if `transformed_jet` is compatible with the structure provided by `MyTransformer`.

A number of the transformers that we discuss below are "taggers" for boosted objects. In some cases they will determine that a given jet does not satisfy the tagging conditions (e.g., for a top tagger, because it seems not to be a top jet). We will adopt the convention that in such cases the result of the transformer is a jet whose 4-momentum is zero, i.e. one that satisfies `jet == 0`. Such a jet may still have structural information however (such information might, for example, indicate why the jet was not tagged).

9.1 Noise-removal transformers

The `Subtrator` transformer defined in 8.5 belongs to this category. Others are described in the following.

9.1.1 Jet Filtering and Trimming using `Filter`

Filtering was first introduced in [31] to reduce the sensitivity of the boosted Higgs jet to the underlying event. Originally, the idea was, starting from a jet obtained with a Cambridge-Aachen clustering, to uncluster it down to a smaller jet radius R_{filt} and only keep the 3 hardest of these subjets as the pieces of the final filtered jet, rejecting the others. Similarly, trimming [32] also unclusters a jet into subjets but selects the subjets to be kept based on a p_t cut.

The use of filtering and trimming have been advocated in number of contexts, beyond just the realm of boosted object reconstruction. The `fastjet::Filter` class derives from `Transformer`, and one can construct, using a `JetDefinition`, a `Selector` and (optionally) a value for the background density,

```
#include "fastjet/tools/Filter.hh"
// ...
Filter filter(subjet_def, selector, rho);
```

a filter/trimmer that reclusters the jet's constituents with the jet definition `subjet_def`[19] and then applies `selector` on the `inclusive_jets` resulting from the clustering to decide which of these (sub)jets have to be kept. If `rho` is non-zero, each of the subjets will be subtracted (using the specified value for the background density) prior to the selection of the kept subjets. Alternatively, the user can set a `Subtractor` (see section 8.5), e.g.

```
GridMedianBackgroundEstimator bge(...);
Subtractor sub(&bge);
filter.set_subtractor(sub);
```

When this is done, the subtraction operation will be performed using the `Subtractor`, independently of the value of `rho`.

If the jet definition to be used to recluster the jet's constituents is the Cambridge/Aachen algorithm, two additional constructors are available:

```
Filter(double Rfilt, Selector selector, double rho = 0.0);
Filter(FunctionOfPseudoJet<double> * Rfilt_dyn, Selector selector, double rho = 0.0);
```

In the first one, only the radius parameter is specified instead of the full subjet definition. In the second, one has to provide a (pointer to) a class derived from `FunctionOfPseudoJet<double>` which dynamically computes the filtering radius as a function of the jet being filtered (as was originally used in [31] where $R_{\text{filt}} = \min(0.3, R_{b\bar{b}/2})$, with $R_{b\bar{b}}$ the distance between the parents of the jet).

As an example, a simple filter, giving the subjets obtained clustering with the Cambridge/Aachen algorithm with radius R_{filt} and keeping the n_{filt} hardest subjets found, can be set up and applied using

```
Filter filter(Rfilt, SelectorNHardest(nfilt));
PseudoJet filtered_jet = filter(jet);
```

The `pieces()` of the resulting filtered/trimmed jet correspond to the subjets that were kept:

```
vector<PseudoJet> kept = filtered_jet.pieces();
```

Additional structural information is available as follows:

```
// the subjets (on the scale Rfilt) not kept by the filtering
vector<PseudoJet> rejected = filtered_jet.structure_of<Filter>().rejected();
```

Trimming, which keeps the subjets with a p_t larger than a fixed fraction of the input jet, can be obtained defining

```
Filter trimmer(Rfilt, SelectorPtFractionMin(pt_fraction_min));
```

and then applying `trimmer` similarly to `filter` above.

Note that the jet being filtered must have constituents. Furthermore, if `rho` is non-zero or if a `Subtractor` is set, the input jet must come from a cluster sequence with area support and explicit ghosts. If any of these requirements fail, an exception will be thrown.

9.1.2 Jet pruning

Pruning was introduced in [33]. Its aim is similar in part to that of filtering, namely reducing the contamination of soft noise in a jet while retaining the bulk of the hard perturbative radiation,

[19] When the input jet was obtained with the Cambridge/Aachen algorithm and the subjet definition also involves the Cambridge/Aachen algorithm, the `Filter` uses the exclusive subjets of the input jet to avoid having to recluster its constituents.

though it is worth noting that, when used together with a mass cut, it can effectively also play the role of a tagger. The noise removal is achieved in pruning in a way which differs from filtering: instead of unclustering a jet and selecting only part of its subjets, pruning works by reclustering its constituents and vetoing soft and large-angle recombinations between pseudojets i and j. In practice, recombinations are vetoed in the clustering sequence when neither of the following criteria are met:

1. the geometric distance between i and j is smaller than a parameter `Rcut`, with `Rcut` = `Rcut_factor`$\times 2m/p_t$, with m and p_t taken from the jet as a whole;

2. $p_t^i, p_t^j >$ `zcut`$\times p_t^{i+j}$.

When both these criteria fail, i and j are not recombined, the harder of i and j is kept, and the softer is rejected. `Rcut_factor` and `zcut` are parameters of the pruning algorithm, and m and p_t are the mass and the transverse momentum of the jet being pruned.

The `fastjet::Pruner` class, derived from `Transformer`, can be used as follows, using a `JetAlgorithm` and two `double` parameters:

```
#include "fastjet/tools/Pruner.hh"
// ...
Pruner pruner(jet_algorithm, zcut, Rcut_factor);
// ...
PseudoJet pruned_jet = pruner(jet);
```

The `pruned_jet` will have a valid associated cluster sequence, so that one can, for instance, ask for its constituents with `pruned_jet.constituents()`. In addition, the subjets that have been rejected by the pruning algorithm (i.e. have been 'pruned away') can be obtained with

```
vector<PseudoJet> rejected_subjets = pruned_jet.structure_of<Pruner>().rejected();
```

and each of these subjets will also have a valid associated clustering sequence.

When using the constructor given above, the jet radius used by the pruning clustering sequence is set internally to the functional equivalent of infinity. Alternatively, a pruner transformer can be constructed with a `JetDefinition` instead of just a `JetAlgorithm`:

```
JetDefinition pruner_jetdef(jet_algorithm, Rpruner);
Pruner pruner(pruner_jetdef, zcut, Rcut_factor);
```

In this situation, the jet definition `pruner_jetdef` should normally have a radius `Rpruner` large enough to ensure that all the constituents of the jet being pruned are reclustered into a single jet. If this is not the case, pruning is applied to the entire reclustering and it is the hardest resulting pruned jet that is returned; the others can be retrieved using

```
vector<PseudoJet> extra_jets = pruned_jet.structure_of<Pruner>().extra_jets();
```

Finally, note that a third constructor for `Pruner` exists, that allows one to construct the pruner using functions that dynamycally compute `zcut` and `Rcut` for the jet being pruned:

```
Pruner (const JetDefinition &jet_def,
        FunctionOfPseudoJet< double > *zcut_dyn,
        FunctionOfPseudoJet< double > *Rcut_dyn);
```

9.2 Boosted-object taggers

A number of the taggers developed to distinguish 2- or 3-pronged decays of massive objects from plain QCD jets (see the review [34]) naturally fall into the category of transformers. Typically they search for one or more hard branchings within the jet and then return the part of the jet that has been identified as associated with those hard branchings. They share the convention that if they were not able to identify suitable substructure, they return a `jet` that has the property `jet == 0`.

At the moment, we have implemented only a small set of taggers. These include one main two-body tagger, the `fastjet::MassDropTagger` introduced in [31] and one main boosted top tagger, `fastjet::JHTopTagger` from [35] (Note that `JHTopTagger` derives from the `fastjet::TopTaggerBase` class, expressely included to provide a common framework for all top taggers capable of also returning a W). In addition, to help provide a more complete set of examples of coding methods to which users may refer when writing their own taggers, we have also included the `fastjet::CASubJetTagger` introduced in [36], which illustrates the use of a `WrappedStructure` (cf. section D.4) and the rest-frame `fastjet::RestFrameNSubjettinessTagger` from Ref. [37], which makes use of facilities to boost a cluster sequence.

We refer the reader to the original papers for a more extensive description of the physics use of these taggers.

More taggers may be provided in the future, either through native implementations or, potentially, through a "contrib" type area. Users are invited to contact the `FastJet` authors for further information in this regard.

9.2.1 The mass-drop tagger

Introduced in [31] for the purpose of identifying a boosted Higgs decaying into a $b\bar{b}$ pair, this is a general 2-pronged tagger. It starts with a fat jet obtained with a Cambridge/Aachen algorithm (originally, $R = 1.2$ was suggested for boosted Higgs tagging). Tagging then proceeds as follows:

1. the last step of the clustering is undone: $j \to j_1, j_2$, with $m_{j_1} > m_{j_2}$;
2. if there is a significant mass drop, $\mu \equiv m_{j_1}/m_j < \mu_{\rm cut}$, and the splitting is sufficiently symmetric, $y \equiv \min(p_{tj_1}^2, p_{tj_2}^2)\Delta R_{j_1 j_2}^2/m_j^2 > y_{\rm cut}$, then j is the resulting heavy particle candidate with j_1 and j_2 its subjets;
3. otherwise, redefine j to be equal to j_1 and go back to step 1.

The tagger can be constructed with

```
#include "fastjet/tools/MassDropTagger.hh"
// ...
MassDropTagger mdtagger(double μ, double y_cut);
```

and applied using

```
PseudoJet tagged_jet = mdtagger(jet);
```

This tagger will run with any jet that comes from a `ClusterSequence`. A warning will be issued if the `ClusterSequence` is not based on the C/A algorithm. If the `JetDefinition` used in the `ClusterSequence` involved a non-default recombiner, that same recombiner will be used when joining the final two prongs to form the boosted particle candidate.

For a jet that is returned by the tagger and has the property that `tagged_jet != 0`, two enquiry functions can be used to return the actual value of μ and y for the clustering that corresponds to the tagged structure:

```
tagged_jet.structure_of<MassDropTagger>.mu();
tagged_jet.structure_of<MassDropTagger>.y();
```

Note that in [31] the mass-drop element of the tagging was followed by a filtering stage using $\min(0.3, R_{jj}/2)$ as the reclustering radius and selecting the three hardest subjects. That can be achieved with

```
vector<PseudoJet> tagged_pieces = tagged_jet.pieces();
double Rfilt = min(0.3, 0.5 * pieces[0].delta_R(pieces[1]));
PseudoJet filtered_tagged_jet = Filter(Rfilt, SelectorNHardest(3))(tagged_jet);
```

(It is also possible to use the `Rfilt_dyn` option to the filter discussed in section 9.1.1).

9.2.2 The Johns-Hopkins top tagger

The Johns Hopkins top tagger [35] is a 3-pronged tagger specifically designed to identify top quarks. It recursively breaks a jet into pieces, finding up to 3 or 4 subjets and then looking for a W candidate among them. The parameters used to identify the relevant subjets include a momentum fraction cut and a minimal separation in Manhattan distance ($|\Delta y| + |\Delta\phi|$) between subjets obtained from a declustering.

The tagger will run with any jet that comes from a `ClusterSequence`, however to conform with the original formulation of [35], the `ClusterSequence` should be based on the C/A algorithm. A warning will be issued if this is not the case. If the `JetDefinition` used in the `ClusterSequence` involves a non-default recombiner, that same recombiner will be used when joining the final two prongs to form the boosted particle candidate. The tagger can be used as follows:

```
#include "fastjet/tools/JHTopTagger.hh"
// ...
double delta_p = 0.10; // subjets must carry at least this fraction of original jet's pt
double delta_r = 0.19; // subjets must be separated by at least this Manhattan distance
double cos_theta_W_max = 0.7; // the maximal allowed value of the W helicity angle
JHTopTagger top_tagger(delta_p, delta_r, cos_theta_W_max);
// indicate the acceptable range of top, W masses
top_tagger.set_top_selector(SelectorMassRange(150,200));
top_tagger.set_W_selector  (SelectorMassRange( 65, 95));
// now try and tag a jet
PseudoJet top_candidate = top_tagger(jet); // jet should come from a C/A clustering
if (top_candidate != 0) { // successful tagging
  double top_mass = top_candidate.m();
  double W_mass   = top_candidate.structure_of<JHTopTagger>().W().m();
}
```

Other information available through the `structure_of<JHTopTagger>()` call includes: `W1()` and `W2()`, the harder and softer of the two W subjets; `non_W()`, the part of the top that has not been identified with a W (i.e. the candidate for the b); and `cos_theta_W()`. The `top_candidate.pieces()` call will return 2 pieces, where the first is the W candidate (identical to `structure_of<JHTopTagger>().W()`), while the second is the remainder of the top jet (i.e. `non_W`).

Note the above calls to `set_top_selector()` and `set_W_selector()`. If these calls are not made, then the tagger places no cuts on the top or W candidate masses and it is then the user's responsibility to verify that they are in a suitable range.

Note further that `JHTopTagger` does not derive directly from `Transformer`, but from the `fastjet::TopTaggerBase` class instead. This class (which itself derives from `Transformer`) has been included to provide a proposed common interface for all the top taggers. In particular, `TopTaggerBase` provides (via the associated structure)

```
top_candidate.structure_of<TopTaggerBase>().W()
top_candidate.structure_of<TopTaggerBase>().non_W()
```

and standardizes the fact that the resulting top candidate is a `PseudoJet` made of these two pieces.

The benefits of the base class for top taggers will of course be more evident once more than a single top tagger has been implemented.

9.2.3 The Cambridge/Aachen subjet tagger

The Cambridge/Aachen subjet tagger [36], originally implemented in a 3-pronged context, is really a generic 2-body tagger, which can also be used in a nested fashion to obtained multi-pronged tagging. It can be obtained through the include

```
#include "fastjet/tools/CASubjetTagger.hh"
```

Its description will follow in a later version of this manual.

9.2.4 The rest-frame N-subjettiness tagger

The rest-frame N-subjettiness tagger [37], meant to identify a highly boosted color singlet particle decaying to 2 partons, can be obtained through the include

```
#include "fastjet/tools/RestFrameNSubjettinessTagger.hh"
```

Its description will follow in a later version of this manual.

10 Compilation notes

Compilation and installation make use of the standard

```
% ./configure
% make
% make check
% make install
```

procedure. Explanations of available options are given in the `INSTALL` file in the top directory, and a list can also be obtained running `./configure --help`.

In order to access the `NlnN` strategy for the k_t algorithm the library needs to be compiled with the Computational Geometry Algorithms Library `CGAL` [6].[20] At configure time the `--enable-cgal` option may be used to specify that CGAL support should be included.

[20]This same strategy gives $N \ln N$ performance for Cambridge/Aachen and $N^{3/2}$ performance for anti-k_t (whose sequence for jet clustering triggers a worst-case scenario for the underlying computational geometry methods.)

CGAL may be obtained in source form from http://www.cgal.org/. Under linux, with CGAL versions 3.2 and 3.3, after compilation and installation, the user will be encouraged to set an environment variable CGAL_MAKEFILE, which points to the Makefile generated by CGAL at install time, which contains various definitions of locations of include files. The user may specify the location of this file to FastJet either through the above environment variable, or with the --with-cgalmakefile=... configuration option. For CGAL 3.4 the user should instead specify --with-cgaldir=... unless the CGAL files are installed in a standard location.

The NlnNCam strategy does not require CGAL, since it is based on a considerably simpler computational-geometry structure [3].

Acknowledgements

Many people have provided bug reports, suggestions for development and in some cases explicit code for plugin algorithms. We would in particular like to thank Vanya Belyaev, Andy Buckley, Timothy Chan, Pierre-Antoine Delsart, Olivier Devillers, Joey Huston, Sue Ann Koay, Andreas Oehler, Sal Rappoccio, Juan Rojo, Sebastian Sapeta, Mike Seymour, Jessie Shelton, Lars Sonnenschein, Hartmut Stadie, Chris Vermilion, Markus Wobisch.

Since its inception, this project has been supported in part by grants ANR-05-JCJC-0046-01, ANR-09-BLAN-0060 and ANR-10-CEXC-009-01 from the French Agence Nationale de la Recherche, PITN-GA-2010-264564 from the European Commission and DE-AC02-98CH10886 from the U.S. Department of Energy.

We would also like to thank the numerous institutes that have hosted us for shorter or longer stays while FastJet was being developed, including the GGI in Florence, KITP at Santa Barbara, Rutgers University and Brookhaven National Laboratory.

A User Info in PseudoJets

One method for associating extra user information with a PseudoJet is via its user index (section 3.1). This is adequate for encoding simple information (for example an input particle's barcode in a HepMC event), but can quickly show its limitations (for example, when simulating pileup one might have several HepMC events and it is then useful for each particle to additionally store information about which HepMC event it comes from).

A second method for supplementing a PseudoJet with extra user information is for the user to derive a class from PseudoJet::UserInfoBase and associate the PseudoJet with a pointer to an instance of that class:

```
void set_user_info(UserInfoBase * user_info);
const UserInfoBase* user_info_ptr() const;
```

It is important to be aware that the function set_user_info(...) transfers ownership of the pointer to the PseudoJet. This is achieved with the help of a shared pointer. Copies of the PseudoJet will point to the same user_info. When the PseudoJet and all its copies go out of scope the user_info will be deleted. Since nearly all practical uses of user_info will require it to be cast to the relevant derived class of UserInfoBase, we also provide the following member function for convenience:

```
template<class L> const L & user_info() const;
```

which casts the extra info to type `L`. If the cast fails, or the user info has not been set, an error will be thrown.[21]

The user may wonder why we have used shared pointers internally (i.e. have ownership transferred to the `PseudoJet`) rather than normal pointers. An example use case where the difference is important is if, for example, one wishes to write a `Recombiner` that sets the `user_info` in the recombined `PseudoJet`. Since this is likely to be new information, the `Recombiner` will have to allocate some memory for it. With a normal pointer, there is then no easy way to clean up that memory when the `PseudoJet` is no longer relevant (e.g. because the `ClusterSequence` that contains it has gone out of scope). In contrast, with a shared pointer the memory is handled automatically.[22]

The shared pointer type in `FastJet` is a template class called `SharedPtr`, available through

```
#include "fastjet/SharedPtr.hh"
```

It behaves almost identically to the `C++0x shared_ptr`.[23] The end-user should not usually need to manipulate the `SharedPtr`, though the `SharedPtr` to `user_info` is accessible through `PseudoJet`'s `user_info_shared_ptr()` member.

An example of the usage might be the following. First you define a class `MyInfo`, derived from `PseudoJet::UserInfo`,

```
class MyInfo: public PseudoJet::UserInfoBase {
   MyInfo(int id) : _pdg_id(id);
   int pdg_id() const {return _pdg_id;}
   int _pdg_id;
};
```

Then you might set the info as follows

```
PseudoJet particle(...);
particle.set_user_info(new MyInfo(its_pdg_id));
```

and later access the PDG id through the function

```
particle.user_info<MyInfo>().pdg_id();
```

B Structural information for various kinds of `PseudoJet`

Starting with `FastJet` version 3.0, a `PseudoJet` can access information about its structure, for example its constituents if it came from a `ClusterSequence`, or its pieces if it was the result of a `join(...)` operation. In this appendix, we summarise what the various structural access methods will return for different types of `PseudoJets`: input particles, jets resulting from a clustering, etc. Table 3 provides the information for the most commonly-used methods.

[21]For clustering with explicit ghosts, even if the particles being clustered have user information, the ghosts will not. The user should take care therefore not to ask for user information about the ghosts, e.g. using the `PseudoJet::is_pure_ghost()` or `PseudoJet::has_user_info<L>()` calls. The `SelectorIsPureGhost()` can also be used for this purpose.

[22] The user may also wonder why we didn't simply write a templated version of `PseudoJet` in order to contain extra information. The answer here is that to introduce a templated `PseudoJet` would imply that every other class in `FastJet` should then also be templated.

[23] Internally it has been designed somewhat differently, in order to limit the memory footprint of the `PseudoJet` that contains it. One consequence of this is that dynamic casts of `SharedPtr`'s are not supported.

	particle	jet	jet (no CS)	constituent	`join`(j_1, j_2)	`join`(p_1, p_2)
`has_associated_cs()`	false	true	true	true	false	false
`associated_cs()`	NULL	CS	NULL	CS	NULL	NULL
`has_validated_cs()`	false	true	false	true	false	false
`validated_cs()`	*throws*	CS	*throws*	CS	*throws*	*throws*
`has_constituents()`	false	true	true	true	true	true
`constituents()`	*throws*	from CS	*throws*	itself	recurse	pieces
`has_pieces()`	false	true	*throws*	false	true	true
`pieces()`	*throws*	parents	*throws*	empty	pieces	pieces
`has_parents(...)`	*throws*	from CS	*throws*	from CS	*throws*	*throws*
`has_child(...)`	*throws*	from CS	*throws*	from CS	*throws*	*throws*
`contains(...)`	*throws*	from CS	*throws*	from CS	*throws*	*throws*

Table 3: summary of the behaviour obtained when requesting structural information from different kinds of `PseudoJet`. A particle (also p_1, p_2) is a `PseudoJet` constructed by the user, without structural information; a "jet" (also j_1, j_2) is the output from a `ClusterSequence`; "from CS" means that the information is obtained from the associated `ClusterSequence`. A "jet (no CS)" is one whose `ClusterSequence` has gone out of scope. All other entries should be self-explanatory.

Additionally, all the methods that access information related to the clustering (`has_partner()`, `is_inside()`, `has_exclusive_subjets()`, `exclusive_subjets()`, `n_exclusive_subjets()`, `exclusive_subdmerge()`, and `exclusive_subdmerge_max`) require the presence of an associated cluster sequence and throw an error if none is available (except for `has_exclusive_subjets()` which just returns `false`).

For area-related calls, `has_area()` will be `false` unless the jet is obtained from a `ClusterSequenceAreaBase` or is a composite jet made from such jets. All other area calls (`validated_csab()`, `area()`, `area_error()`, `area_4vector()`) will return the information from the `ClusterSequence` (or the pieces in case of a composite jet) and throw an error if the jet is not associated with a `ClusterSequenceAreaBase`.

Internal storage of structural information. The means by which information about a jet's a structure is stored is generally transparent to the user. The main exception that arises is when the user wishes to create jets with a new kind of structure, for example when writing boosted-object taggers. Here, we simply outline the approach adopted. For concrete usage examples one can consult section 9 and appendix D.4, where we discuss transformers and taggers.

To be able to efficiently access structural information, each `PseudoJet` has a shared pointer to a class of type `fastjet::PseudoJetStructureBase`. For plain `PseudoJet`s the pointer is null. For `PseudoJet`s obtained from a `ClusterSequence` the pointer is to a class `fastjet::ClusterSequenceStructure`, which derives from `PseudoJetStructureBase`. For `PseudoJet`s obtained from a `join(...)` operation, the pointer is to a class `fastjet::CompositeJetStructure`, again derived from `PseudoJetStructureBase`. It is these classes that are responsible for answering structural queries about the jet, such as returning its constituents, or indicating whether it `has_pieces()`. Several calls are available for direct access to the internal structure storage, among them

```
const PseudoJetStructureBase* structure_ptr() const;
PseudoJetStructureBase*       structure_non_const_ptr();
template<typename StructureType> const StructureType & structure() const;
```

where the first two return simply the structure pointer, while the last one casts the pointer to the desired derived structure type.

C Functions of a `PseudoJet`

A concept that is new to `FastJet` 3 is that of a `fastjet::FunctionOfPseudoJet`. Functions of `PseudoJet`s arise in many contexts: many boosted-object taggers take a jet and return a modified version of a jet; background subtraction does the same; so does a simple Lorentz boost. Another class of functions returns a floating-point number associated with the jet: for example jet shapes; but also the rescaling functions used to provide local background estimates in section 8.2.

To help provide a uniform interface for functions of a `PseudoJet`, `FastJet` provides the following template base class:

```
// a generic function of a PseudoJet
template<typename TOut> class FunctionOfPseudoJet{
  // the action of the function (this _has_ to be overloaded in derived classes)
  virtual TOut result(const PseudoJet &pj) const = 0;
};
```

Derived classes should implement the `result(...)` function. In addition it is good practice to overload the `description()` member,

```
virtual std::string description() const{ return "";}
```

Usage of a `FunctionOfPseudoJet` is simplest through the `operator(...)` member functions

```
TOut operator()(const PseudoJet & pj) const;
vector<TOut> operator()(const vector<PseudoJet> & pjs) const;
```

which just call `result(...)` either on the single jet, or separately on each of the elements of the vector of `PseudoJet`s.[24].

This definition allows one for example to pass functions of `PseudoJet`s as arguments. This is *e.g.* used for the background rescalings in section 8.2 which are just derived from `FunctionOfPseudoJet<double>`. It is also used for the `Transformer`s of section 9, which all derive from `FunctionOfPseudoJet<PseudoJet>`. The use of a class for these purposes, rather than a pointer to a function, provides the advantage that the class can be initialised with additional arguments.

[24]Having `result(...)` and `operator(...)` doing the same thing may seem redundant, however, it allows one to redefine only `result` in derived classes. If we had had a virtual `operator(...)` instead, both the `PseudoJet` and `vector<PseudoJet>` versions would have had to be overloaded.

D User-defined extensions of FastJet

D.1 External Recombination Schemes

If the user wishes to introduce a new recombination scheme, she may do so writing a class derived from `JetDefinition::Recombiner`:

```
class JetDefinition::Recombiner {
public:
  /// return a textual description of the recombination scheme
  /// implemented here
  virtual std::string description() const = 0;

  /// recombine pa and pb and put result into pab
  virtual void recombine(const PseudoJet & pa, const PseudoJet & pb,
                         PseudoJet & pab) const = 0;

  /// routine called to preprocess each input jet (to make all input
  /// jets compatible with the scheme requirements (e.g. massless).
  virtual void preprocess(PseudoJet & p) const {};

  /// a destructor to be replaced if necessary in derived classes...
  virtual ~Recombiner() {};
};
```

A jet definition can then be constructed by providing a pointer to an object derived from `JetDefinition::Recombiner` instead of the `RecombinationScheme` index:

```
JetDefinition(JetAlgorithm jet_algorithm,
              double R,
              const JetDefinition::Recombiner * recombiner,
              Strategy strategy = Best);
```

The derived class `JetDefinition::DefaultRecombiner` is what is used internally to implement the various recombination schemes if an external `Recombiner` is not provided. It provides a useful example of how to implement a new `Recombiner` class.

The recombiner can also be set with a `set_recombiner(...)` call. If the recombiner has been created with a `new` statement and the user does not wish to manage the deletion of the corresponding memory when the `JetDefinition` (and any copies) using the recombiner goes out of scope, then the user may wish to call the `delete_recombiner_when_unused()` function, which tells the `JetDefinition` to acquire ownership of the pointer to the recombiner and delete it when it is no longer needed.

D.2 Implementation of a plugin jet algorithm

The base class from which plugins derive has the following structure:

```
class JetDefinition::Plugin{
public:
  /// return a textual description of the jet-definition implemented
  /// in this plugin
```

```
    virtual std::string description() const = 0;

    /// given a ClusterSequence that has been filled up with initial
    /// particles, the following function should fill up the rest of the
    /// ClusterSequence, using the following member functions of
    /// ClusterSequence:
    ///   - plugin_do_ij_recombination(...)
    ///   - plugin_do_iB_recombination(...)
    virtual void run_clustering(ClusterSequence &) const = 0;

    /// a destructor to be replaced if necessary in derived classes...
    virtual ~Plugin() {};

    //------- ignore what follows for simple usage! ---------
    /// return true if there is passive areas can be efficiently determined by
    /// (a) setting the ghost_separation scale (see below)
    /// (b) clustering with many ghosts with pt << ghost_separation_scale
    /// (c) counting how many ghosts end up in a given jet
    virtual bool supports_ghosted_passive_areas() const {return false;}

    /// set the ghost separation scale for passive area determinations
    /// in future runs (NB: const, so should set internal mutable var)
    virtual void set_ghost_separation_scale(double scale) const;
    virtual double ghost_separation_scale() const;

  };
```

Any plugin class must define the `description` and `run_clustering` member functions. The former just returns a textual description of the jet algorithm and its options (e.g. radius, etc.), while the latter does the hard work of running the user's own jet algorithm and transferring the information to the `ClusterSequence` class. This is best illustrated with an example:

```
using namespace fastjet;

void CDFMidPointPlugin::run_clustering(ClusterSequence & clust_seq) {

  // when run_clustering is called, the clust_seq has already been
  // filled with the initial particles, which are available through the
  // jets() array
  const vector<PseudoJet> & initial_particles = clust_seq.jets();

  // it is up to the user to do their own clustering on these initial
  // particles

  // ...
```

Once the plugin has run its own clustering it must transfer the information back to the `clust_seq`. This is done by recording mergings between pairs of particles or between a particle and the beam. The new momenta are stored in the `clust_seq.jets()` vector, after the initial particles. Note though that the plugin is not allowed to modify `clust_seq.jets()` itself. Instead it must tell `clust_seq` what recombinations have occurred, via the following (`ClusterSequence` member) functions

```
/// record the fact that there has been a recombination between
/// jets()[jet_i] and jets()[jet_k], with the specified dij, and
/// return the index (newjet_k) allocated to the new jet, whose
/// momentum is assumed to be the 4-vector sum of that of jet_i and
/// jet_j
void plugin_record_ij_recombination(int jet_i, int jet_j, double dij,
                                    int & newjet_k);

/// as for the simpler variant of plugin_record_ij_recombination,
/// except that the new jet is attributed the momentum and
/// user_index of newjet
void plugin_record_ij_recombination(int jet_i, int jet_j, double dij,
                                    const PseudoJet & newjet,
                                    int & newjet_k);

/// record the fact that there has been a recombination between
/// jets()[jet_i] and the beam, with the specified diB; when looking
/// for inclusive jets, any iB recombination will returned to the user
/// as a jet.
void plugin_record_iB_recombination(int jet_i, double diB);
```

These `dij` recombination functions return the index `newjet_k` of the newly formed pseudojet. The plugin may need to keep track of this index in order to specify subsequent recombinations.

Certain (cone) jet algorithms do not perform pairwise clustering — in these cases the plugin must invent a ficititious series of pairwise recombinations that leads to the same final jets. Such jet algorithms may also produce extra information that cannot be encoded in this way (for example a list of stable cones), but to which one may still want access. For this purpose, during `run_clustering(...)`, the plugin may call the `ClusterSequence` member function:

```
inline void plugin_associate_extras(std::auto_ptr<ClusterSequence::Extras> extras);
```

where `ClusterSequence::Extras` is a dummy class which the plugin should extend so as to provide the relevant information:

```
class ClusterSequence::Extras {
public:
  virtual ~Extras() {}
  virtual std::string description() const;
};
```

A method of `ClusterSequence` then provides the user with access to the extra information:

```
/// returns a pointer to the extras object (may be null) const
Extras * extras() const;
```

The user should carry out a dynamic cast so as to convert the extras back to the specific plugin extras class, as illustrated for SISCone in section 5.2.

D.2.1 Building new sequential recombination algorithms

To enable users to more easily build plugins for new sequential recombination algorithms, `FastJet` also provides a class `NNH`, which provides users with access to an implementation of the nearest-neighbour heuristic for establishing and maintaining information about the closest pair of objects in a dynamic set of objects (see [38] for an introduction to this and other generic algorithms). In good cases this allows one to construct clustering that runs in N^2 time, though its worst case can be as bad as N^3. It is a templated class and the template argument should be a class that stores the minimal information for each jet so as to be able to calculate interjet distances. It underlies the implementations of the Jade and e^+e^- Cambridge plugins. The interested user should consult those codes for more information, as well as the header for the `NNH` class.

D.3 Implementing new selectors

Technically a `Selector` contains a shared pointer to a `SelectorWorker`. Classes derived from `SelectorWorker` actually do the work. So, for example, the call to the function `SelectorAbsRapMax(2.5)` first causes a new instance of the internal `SW_AbsRapMax` class to be constructed with the information that the limit on $|y|$ is 2.5 (`SW_AbsRapMax` derives from `SelectorWorker`). Then a `Selector` is constructed with a pointer to the `SW_AbsRapMax` object, and it is this `Selector` that is returned to the user:

```
Selector SelectorAbsRapMax(double absrapmax) {
  return Selector(new SW_AbsRapMax(absrapmax));
}
```

Since `Selector` is really nothing more than a shared pointer to the `SW_AbsRapMax` object, it is a lightweight object. The fact that it's a shared pointer also means that it looks after the memory management issues associated with the `SW_AbsRapMax` object.

If a user wishes to implement a new selector, they should write a class derived from `SelectorWorker`. The base is defined with sensible defaults, so for simple usage, only two `SelectorWorker` functions need to be overloaded:

```
/// returns true if a given object passes the selection criterion.
pass(const PseudoJet & jet) const = 0;

/// returns a description of the worker
virtual std::string description() const {return "missing description";}
```

For information on how to implement more advanced workers (for example workers that do not apply jet-by-jet, or that take a reference), users may wish to examine the extensive in-code documentation of `SelectorWorker`, the implementation of the existing workers and/or consult the authors. A point to be aware of in the case of constructors that take a reference is the need to implement the `SelectorWorker::copy()` function.

D.4 User-defined transformers

All transformers are derived from the `Transformer` base class, declared in the `fastjet/tools/Transformer.hh` header:

```
class Transformer : public FunctionOfPseudoJet<PseudoJet> {
public:
  // the result of the Transformer acting on the PseudoJet.
  // this has to be overloaded in derived classes
```

```
    virtual PseudoJet result(const PseudoJet & original) const = 0;

    // should be overloaded to return a description of the Transformer
    virtual std::string description() const = 0;

    // information about the associated structure type
    typedef PseudoJetStructureBase StructureType;

    // destructor is virtual so that it can be safely overloaded
    virtual ~Transformer(){}
  };
```

Relative to the `FunctionOfPseudoJet<PseudoJet>` (cf. appendix C) from which it derives, the `Transformer`'s main additional feature is that the jets resulting from the transformation are generally expected to have standard structural information, e.g. constituents, and will often have supplemental structural information, which the `StructureType typedef` helps access. As for a `FunctionOfPseudoJet<PseudoJet>`, the action of a `Transformer` is to be implemented in the `result(...)` member function, though typically it will be used through the `operator()` function:

```
  /// just wraps result()
  PseudoJet operator()(const PseudoJet &pj) const {return result(pj);}
  /// returns a vector of PseudoJet containing result(...)
  /// applied to each of input PseudoJets,
  vector<PseudoJet> operator()(const vector<PseudoJet> &pjs) const;
```

To help understand how to create user-defined transformers, it is perhaps easiest to consider the example of a filtering/trimming class. The simplest form of such a class is the following:[25]

```
  /// a simple class to carry out filtering and/or trimming
  class SimpleFilter: public Transformer {
  public:
    SimpleFilter(const JetDefinition & subjet_def, const Selector & selector) :
                             _subjet_def(subjet_def), _selector(selector) {}

    virtual std::string description() const {
      return "Filter that finds subjets with " + _subjet_def.description()
             + ", using a (" + _selector.description() + ") selector" ;}

    virtual PseudoJet result(const PseudoJet & jet) const;

    // CompositeJetStructure is the structural type associated with the
    // join operation that we use shall use to create the returned jet below
    typedef CompositeJetStructure StructureType;

  private:
    JetDefinition _subjet_def;
    Selector      _selector;
  };
```

[25]The actual `Filter` class is somewhat more elaborate than this, since it also handles areas, pileup subtraction and avoids reclustering when the jet and subjet definitions are C/A based.

The function that does the work in this class is `result(...)`:

```
PseudoJet SimpleFilter::result(const PseudoJet & jet) const {
  // get the subjets
  ClusterSequence * cs = new ClusterSequence(jet.constituents(), _subjet_def);
  vector<PseudoJet> subjets = cs->inclusive_jets();

  // signal that the cluster sequence should delete itself when
  // there are no longer any of its (sub)jets in scope anywhere
  cs->delete_self_when_unused();

  // get the selected subjets
  vector<PseudoJet> selected_subjets = _selector(subjets);
  // join them using the same recombiner as was used in the subjet_def
  PseudoJet result = join(selected_subjets, *_subjet_def.recombiner());
  return result;
}
```

This provides almost all the basic functionality that might be needed from a filter, including access to the `pieces()` of the filtered jet since it is formed with the `join(...)` function. The one part that is potentially missing is that the user does not have any way of accessing information about the subjets that were not kept by the filter. This requires adding to the structural information that underlies the returned jet. The `join(...)` function creates a structure of type `CompositeJetStructure`. There is also a templated version, `join<ClassDerivedFromCompositeJetStructure>(...)`, which allows the user to choose the structure created by the `join` function. In this case we therefore create

```
class SimpleFilterStructure: public CompositeJetStructure {
public:
  // the form of constructor expected by the join<...> function
  SimpleFilterStructure(const vector<PseudoJet> & pieces,
                        const Recombiner *recombiner = 0) :
                                   CompositeJetStructure(pieces, recombiner) {}
  // provide access to the rejected subjets from the filtering
  const vector<PseudoJet> & rejected() const {return _rejected;}
private:
  vector<PseudoJet> _rejected;
  friend class SimpleFilter;
};
```

and then replace the last few lines of the `SimpleFilter::result(...)` function with

```
// get the selected and rejected subjets
vector<PseudoJet> selected_subjets, rejected_subjets;
_selector.sift(subjets, selected_subjets, rejected_subjets);

// join the selected ones, now with a user-chosen structure
PseudoJet result = join<SimpleFilterStructure>(selected_subjets, *_subjet_def.recombiner());

// and then set the structure's additional elements
SimpleFilterStructure * structure =
                  static_cast<SimpleFilterStructure *>(result.structure_non_const_ptr());
structure->_rejected = rejected_subjets;
```

```
  return result;
```

Finally, with the replacement of the `typedef` in the `SimpleFilter` class with

```
  typedef SimpleFilterStructure StructureType;
```

then on a jet returned by the `SimpleFilter` one can simply call

```
  filtered_jet.structure_of<SimpleFilter>().rejected();
```

as with the fully fledged `Filter` of section 9.1.1.

A second way of extending the structural information of an existing jet is to "wrap" it. This can be done with the help of the `WrappedStructure` class.

```
/// a class to wrap and extend existing jet structures with  information about
///"rejected" pieces
class SimpleFilterWrappedStructure: public WrappedStructure {
public:
  SimpleFilterWrappedStructure(const SharedPtr<PseudoJetStructureBase> & to_be_wrapped,
                               const vector<PseudoJet> & rejected_pieces) :
             WrappedStructure(to_be_wrapped), _rejected(rejected_pieces) {}

  const vector<PseudoJet> & rejected() const {return _rejected;}
private:
  vector<PseudoJet> _rejected;
};
```

The `WrappedStructure`'s constructor takes a `SharedPtr` to an existing structure and simply redirects all standard structural queries to that existing structure. A class derived from it can then reimplement some of the standard queries, or implement non-standard ones, as done above with the `rejected()` call. To use the wrapped class one might proceed as in the following lines:

```
// create a jet with some existing structure
PseudoJet result = join(selected_subjets, *_subjet_def.recombiner());
// create a new structure that wraps the existing one and supplements it with new info
SharedPtr<PseudoJetStructureBase> structure(new
   SimpleFilterWrappedStructure(result.structure_shared_ptr(), rejected_subjets));
// assign the new structure to the original jet
result.set_structure_shared_ptr(structure);
```

The `SharedPtr`s ensure that memory allocated for the structural information is released when no jet remains that refers to it. For the above piece of code to be used in the `SimpleFilter` it would then suffice to include a

```
  typedef SimpleFilterWrappedStructure StructureType;
```

line in the `SimpleFilter` class definition.

In choosing between the templated `join<...>` and `WrappedStructure` approaches to providing advanced structural information, two elements are worth considering: on one hand, the `WrappedStructure` can be used to extend arbitrary structural information; on the other, while `join<...>` is more limited in its scope, it involves fewer pointer indirections when accessing structural information and so may be a little more efficient.

E Error handling

FastJet provides warning and error messages through the classes `fastjet::LimitedWarning` and `fastjet::Error` respectively. Both are handled internally, and a user does not normally need to interact with them. They do, however, provide some customization facilities, especially in terms of redirection of their output.

Each different kind of warning is written out a maximum number of times (the current default is 5) before its output is suppressed. The program is allowed to continue. At the end of the run (or at any other stage) it is possible to obtain a summary of all warnings encoutered, both explicit or supressed, through the following static member function of the LimitedWarning class:

```
#include "fastjet/LimitedWarning.hh"
// ...
cout << LimitedWarning::summary() << endl;
```

The throwing of an `Error` aborts the program. One can use

```
/// controls whether the error message (and the backtrace, if its printing is enabled)
/// is printed out or not
static void set_print_errors(bool print_errors);

/// controls whether the backtrace is printed out with the error message or not.
/// The default is "false".
static void set_print_backtrace(bool enabled);
```

to control whether an error message is printed (default = `true`) and whether a full backtrace is also given (default = `false`). Switching off the printing of error messages can be useful, for example, if the user expects to repeatedly catch FastJet errors.

The output of both `LimitedWarning` and `Error`, which by default goes to `std::cerr`, can be redirected to a file using the `set_default_stream(std::ostream * ostr)` function. For instance,

```
#include "fastjet/LimitedWarning.hh"
#include "fastjet/Error.hh"
#include <iostream>
#include <fstream>
// ...
ostream * myerr = new ofstream("warnings-and-errors.txt");
LimitedWarning::set_default_stream(myerr);
Error::set_default_stream(myerr);
Error::set_print_backtrace(true);
// ...
cout << LimitedWarning::summary() << endl;
```

will send the output of both classes to the file `warnings-and-errors.txt` (as well as provide the backtrace of errors). Note that the output of `LimitedWarning::summary()` still goes to standard output, and will only be present if the program did not abort earlier due to an error.

F FastJet history

Version 1 of `FastJet` provided the first fast implementation of the longitudinally invariant k_t clustering [9, 10], based on the factorisation of momentum and geometry in that algorithm's distance measure [1].

Version 2.0 brought the implementation of the inclusive Cambridge/Aachen algorithm [11, 12] and of jet areas and background estimation [27, 26]; other changes include a new interface,[26] and new algorithmic strategies that could provide a factor of two improvement in speed for events whose number N of particles was $\sim 10^4$. Choices of recombination schemes and plugins for external jet algorithms were new features of version 2.1. The initial set of plugins included SISCone [15], the CDF midpoint [7] and JetClu [19] cones and PxCone. The plugins helped provide a uniform interface to a range of different jet algorithms and made it possible to overlay `FastJet` features such as areas onto the external jet algorithms. Version 2.2 never made it beyond the beta-release stage, but introduced a number of the features that eventually were released in 2.3. The final 2.3 release included the anti-k_t algorithm [13], a broader set of area measures, improved access to background estimation, means to navigate the ClusterSequence and a new build system (GNU autotools). Version 2.4 included the new version 2.0 of SISCone, as well as plugins to the DØ Run II cone, the ATLAS cone, the CMS cone, TrackJet and a range of e^+e^- algorithms, and also further tools to help investigate jet substructure. It also added a wrapper to `FastJet` allowing one to run SISCone and some of the sequential recombination algorithms from Fortran programs.

A major practical change in version 3.0 was that PseudoJet acquired knowledge (where relevant) about its underlying ClusterSequence, allowing one to write *e.g.* `jet.constituents()` and it became possible to associate extra information with a PseudoJet beyond just a user index. It brought the first of a series of `FastJet` tools to help with advanced jet analyses, namely the Selector class, filters, pruners, taggers and new background estimation classes. Version 3 also added the D0-Run I cone [20] plugin and support for native jet algorithms to be run with $R > \pi/2$.

G Deprecated and removed features

While we generally aim to maintain backwards compatibility for software written with old versions of FastJet, there are occasions where old interfaces or functionality no longer meet the standards that are demanded of a program that is increasingly widely used. Table 4 lists the cases where such considerations have led us to deprecate and/or remove functionality.

[26]The old one was retained through v2

Feature, class or include file	Dep.	Rem.	Suggested replacement
FjClusterSequence.hh	2.0	3.0	fastjet/ClusterSequence.hh
FjPseudoJet.hh	2.0	3.0	fastjet/PseudoJet.hh
CS::set_jet_finder(...)	2.1	3.0	JetDefinition
CS::set_jet_algorithm(...)	2.1	3.0	JetDefinition
CS::CS(particles, R, ...)	2.1	3.0	CS::CS(particles, jet_def)
SISConePlugin.hh	2.3	3.0	fastjet/SISConePlugin.hh (idem. other plugins)
ActiveAreaSpec	2.3	–	AreaDefinition & GhostedAreaSpec
ClusterSequenceWithArea	2.3	–	ClusterSequenceArea
default $f = 0.5$ in some cone plugins	–	2.4	include f explicitly in c'tor
default $R = 1$ in JetDefinition	–	2.4	include R explicitly in c'tor
RangeDefinition	3.0	–	Selector(s)
CircularRange	3.0	–	SelectorCircle
CSAB::median_pt_per_unit_area(...)	3.0	–	BackgroundEstimator
CSAB::parabolic_pt_per_unit_area(...)	3.0	–	BackgroundEstimator (cf. section 8.2)
GAS::set_fj2_placement(...)	3.0	–	use new default ghost placement instead

Table 4: Summary of interfaces and features of earlier versions that have been deprecated and/or removed. For brevity we have used the following abbreviations: Dep. = version since which a feature has been deprecated, c'tor = constructor, Rem. = version where removed, CS = ClusterSequence, CSAB = ClusterSequenceAreaBase, GAS = GhostedAreaSpec.

References

[1] M. Cacciari and G. P. Salam, Phys. Lett. B **641** (2006) 57 [hep-ph/0512210].

[2] C. Buttar *et al.*, arXiv:0803.0678 [hep-ph].

[3] T. M. Chan, "Closest-point problems simplified on the RAM," in Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), p. 472 (2002).

[4] M. R. Anderberg, Cluster Analysis for Applications, (Number 19 in Probability and Mathematical Statistics, Academic Press, New York, 1973).

[5] L. Sonnenschein, Ph.D. Thesis, RWTH Aachen 2001;
`http://cmsdoc.cern.ch/documents/01/doc2001_025.ps.Z`

[6] A. Fabri *et al.*, Softw. Pract. Exper. **30** (2000) 1167; J.-D. Boissonnat *et al.*, Comp. Geom. **22** (2001) 5; `http://www.cgal.org/`

[7] G. C. Blazey *et al.*, hep-ex/0005012.

[8] `http://hepforge.cedar.ac.uk/ktjet/`; J. M. Butterworth, J. P. Couchman, B. E. Cox and B. M. Waugh, Comput. Phys. Commun. **153**, 85 (2003) [hep-ph/0210022].

[9] S. Catani, Y. L. Dokshitzer, M. H. Seymour and B. R. Webber, Nucl. Phys. B **406** (1993) 187.

[10] S. D. Ellis and D. E. Soper, Phys. Rev. D **48** (1993) 3160 [hep-ph/9305266].

[11] Y. L. Dokshitzer, G. D. Leder, S. Moretti and B. R. Webber, JHEP **9708**, 001 (1997) [hep-ph/9707323];

[12] M. Wobisch and T. Wengler, "Hadronization corrections to jet cross sections in deep-inelastic arXiv:hep-ph/9907280; M. Wobisch, "Measurement and QCD analysis of jet cross sections in deep-inelastic DESY-THESIS-2000-049.

[13] M. Cacciari, G. P. Salam and G. Soyez, JHEP **0804** (2008) 063 [arXiv:0802.1189 [hep-ph]].

[14] S. Catani, Y. L. Dokshitzer, M. Olsson, G. Turnock and B. R. Webber, Phys. Lett. B **269**, 432 (1991);

[15] G.P. Salam and G. Soyez, JHEP **0705** 086 (2007), [arXiv:0704.0292 [hep-ph]]; standalone code available from `http://projects.hepforge.org/siscone`.

[16] S. D. Ellis, J. Huston and M. Tonnesmann, in *Proc. of the APS/DPF/DPB Summer Study on the Future of Particle Physics (Snowmass 2001)* ed. N. Graf, p. P513 [hep-ph/0111434].

[17] TeV4LHC QCD Working Group *et al.*, hep-ph/0610012.

[18] The CDF code has been taken from `http://www.pa.msu.edu/~huston/Les_Houches_2005/JetClu+Midpoi`

[19] F. Abe *et al.* [CDF Collaboration], "The Topology of three jet events in $\bar{p}p$ collisions at $\sqrt{s} = 1.8$ TeV," Phys. Rev. D **45** (1992) 1448.

[20] B. Abbott *et al.* [D0 Collaboration], FERMILAB-PUB-97-242-E.

[21] P.A. Delsart, K. Geerlings, J. Huston, B. Martin and C. Vermilion, SpartyJet, `http://projects.hepforge.org/spartyjet`

[22] M. H. Seymour and C. Tevlin, JHEP **0611** (2006) 052 [arXiv:hep-ph/0609100].

[23] L. A. del Pozo and M. H. Seymour, unpublished.

[24] W. Bartel *et al.* [JADE Collaboration], Z. Phys. C **33** (1986) 23;

[25] S. Bethke *et al.* [JADE Collaboration], Phys. Lett. B **213** (1988) 235.

[26] M. Cacciari, G. P. Salam and G. Soyez, JHEP **0804** (2008) 005, [arXiv:0802.1188 [hep-ph]].

[27] M. Cacciari and G. P. Salam, Phys. Lett. B **659** (2008) 119 [arXiv:0707.1378 [hep-ph]].

[28] S. Fortune, Algorithmica **2** (1987) 1.

[29] M. Cacciari, G. P. Salam, S. Sapeta, JHEP **1004** (2010) 065. [arXiv:0912.4926 [hep-ph]].

[30] T. Sjostrand, S. Mrenna, P. Z. Skands, Comput. Phys. Commun. **178** (2008) 852-867. [arXiv:0710.3820 [hep-ph]].

[31] J. M. Butterworth, A. R. Davison, M. Rubin and G. P. Salam, Phys. Rev. Lett. **100** (2008) 242001 [arXiv:0802.2470 [hep-ph]].

[32] D. Krohn, J. Thaler and L. T. Wang, JHEP **1002** (2010) 084 [arXiv:0912.1342 [hep-ph]].

[33] S. D. Ellis, C. K. Vermilion, J. R. Walsh, Phys. Rev. **D80** (2009) 051501. [arXiv:0903.5081 [hep-ph]].

[34] A. Abdesselam, E. B. Kuutmann, U. Bitenc, G. Brooijmans, J. Butterworth, P. Bruckman de Renstrom, D. Buarque Franzosi, R. Buckingham *et al.*, Eur. Phys. J. **C71** (2011) 1661. [arXiv:1012.5412 [hep-ph]].

[35] D. E. Kaplan, K. Rehermann, M. D. Schwartz, B. Tweedie, Phys. Rev. Lett. **101** (2008) 142001 [arXiv:0806.0848 [hep-ph]].

[36] J. M. Butterworth, J. R. Ellis, A. R. Raklev, G. P. Salam, Phys. Rev. Lett. **103** (2009) 241803. [arXiv:0906.0728 [hep-ph]].

[37] J. H. Kim, Phys. Rev. D **83** (2011) 011502 [arXiv:1011.1493 [hep-ph]].

[38] D. Eppstein J. Experimental Algorithmics **5** (2000) 1-23 [cs.DS/9912014].