FastJet 3.0alpha3 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` provides fast $(N \ln N, N^2)$ implementations of the longitudinally invariant k_t, anti-k_t and Cambridge/Aachen jet algorithms for pp collisions, based in part on tools and methods from the Computational Geometry community, as well as a native implementation of the e^+e^- k_t algorithm. Further jet algorithms, including most of the other commonly used pp and e^+e^- algorithms, can be accessed from the FastJet interface using a plugin mechanism. `FastJet` also provides ways of determining jet areas.

Contents

1 Introduction

This note documents the `FastJet` package, which provides efficient geometrically-based implementations [1] for the longitudinally invariant k_t [2, 3], inclusive Cambridge/Aachen [4, 5] and anti-k_t[6] jet algorithms. It also provides access to tools that allow one to determine the areas of individual jets, which is of importance when correcting for underlying event and pileup contamination. Finally external jet algorithms can be accessed through the fastjet interface using a "plugin" facility.

The implementation of the inclusive Cambridge algorithm and of jet areas are new features of version 2 of `FastJet`; other changes include a new interface,[1] and new algorithmic strategies that can provide a factor of two improvement in speed for events whose number N of particles $\sim 10^4$. Choices of recombination schemes and plugins are new features of version 2.1. Version 2.2 introduces a broader set of area measures and the anti-k_t algorithm [6]. A plugin facility allows one to access external jet algorithms through the `FastJet` interface, and overlays `FastJet` features such as areas onto the external jet algorithms. Plugins are included for the fortran PxCone code and for the CDF C++ JetClu and MidPoint cone jet algorithms (all infrared unsafe), as well as the recent Seedless Infrared-Safe Cone (SISCone) jet algorithm [7]. Version 2.3 introduced a new build system (GNU autotools), a broader range of areas and tools to help navigate the ClusterSequence. 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 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. In version 3.0(alpha1) a PseudoJet knows (where relevant) about its underlying ClusterSequence, allowing one to write *e.g.* `jet.constituents()`, and can be associated with extra information beyond just a user index. This version also adds the D0-Run I cone as a new plugin and support for native jet algorithms to be run with $R > \pi/2$. It also brings the first of a series of FastJet tools to help with advanced jet analyses, namely the Selector class. The current implementation of the *pp* sequential-recombinaion algorithms is restricted to the so-called ΔR distance measure (recommended in [8]), with a choice of recombination schemes. The k_t algorithm has both exclusive [2] and inclusive modes [3], the latter currently being in more widespread use, though the exclusive mode may have physical advantages in certain cases.

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<br>
cd fastjet-X.Y.Z/
```

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

```
./configure --prefix=‘pwd‘/../fastjet-install
```

[1] The old one though retained through v2 is deprecated and will be removed in v3

```
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 two particles:    px py pz    E
  particles.push_back( PseudoJet( 100.0, 0, 0, 100.0) );
  particles.push_back( PseudoJet(-100.0, 0, 0, 100.0) );

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

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

  // 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;
  }
}
```

- Then compile and run it with

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

```
        pt y phi
jet 0: 100 0 3.14159
jet 1: 100 0 0
```

3 Jet-finding interface

The FastJet code is written in C++. From the point of view of simple usage its interface has a number of similarities to that of KtJet [10], fundamental differences being highlighted below.

All classes are contained in the fastjet namespace. For basic usage, the user is exposed to three main classes:

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

fastjet::PseudoJet provides a jet object with a four-momentum and some internal indices to situate it in the context of a jet-clustering sequence. fastjet::ClusterSequence is the class that carries out jet-clustering and provides access to the final jets.

The class fastjet::JetDefinition contains a specification of how jet clustering is to be performed. Such a class was not present in version 1 of FastJet (nor in KtJet), but was found to be useful to 'contain' the complexity of the interface as new features such as alternative jet algorithms and jet-area determination were introduced.

3.1 fastjet::PseudoJet

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

```
fastjet::PseudoJet (double px, double py, double pz, double  E);
template<class T> fastjet::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.[2] 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
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
```

[2] fastjet::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 m2()       const ; // returns squared invariant mass
double m()        const ; // returns invariant mass (-sqrt(-m^2) if m^2 < 0)
double mperp2()   const ; // returns the squared transverse mass = k_t^2 + m^2
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 information 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 involves a pointer to a derived class of `PseudoJet::UserInfoBase`. It is discussed in appendix B.

A `PseudoJet` can be reset with

```
/// reset the 4-momentum according to the supplied components,
/// put the user and history indices back to their default values (-1
inline void reset(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, the user and internal indices are inherited).

Finally, the `+`, `-`, `*` and `/` operators are defined, with `+`, `-` acting on pairs of `PseudoJets` and `*`, `/` acting on a `PseudoJet` and a `double` coefficient. Analogous `+=`, etc., operators, are also defined.

We also provide routines for taking an unsorted vector of `fastjet::PseudoJets` and returning a sorted vector,

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

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

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

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

3.2 fastjet::JetDefinition

The class `fastjet::JetDefinition` contains a full specification of how to carry out the clustering. According to the Les Houches convention detailed in [11], a 'jet definition' should include the jet

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 5.6).

algorithm name, its parameters (often the radius R) and the recombination scheme. Its constructor is[3]

```
fastjet::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 `fastjet::JetAlgorithm enum`[4]:

```
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. (1,2,3,4,5). The recombination scheme should be one of those listed in table 1. If it is omitted the E-scheme is chosen.

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

```
fastjet::JetDefinition(fastjet::JetAlgorithm jet_algorithm,
                       fastjet::RecombinationScheme recomb_scheme = E_scheme,
                       fastjet::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

```
fastjet::JetDefinition(fastjet::JetAlgorithm jet_algorithm,
                       double R,
                       double p,
                       fastjet::RecombinationScheme recomb_scheme = E_scheme,
                       fastjet::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.

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

[4]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.

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

The strategy selects the algorithmic strategy to use while clustering and is an `enum` of type `fastjet::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 [12] (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" [13] 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,[5] 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 [15] 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.

[5] 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 [14].

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 `fastjet::ClusterSequence` object,[6] through the following constructor

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

where `input_particles` is the vector of initial particles of any type (`fastjet::PseudoJet`, `HepLorentzVector`, etc.) that can be used to initialise a `fastjet::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<fastjet::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,[7] 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.
vector<fastjet::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.
vector<fastjet::PseudoJet> exclusive_jets (const int & njets) const;
```

Note that these two member functions have the same name, and the compiler selects the correct one based on the type of the arguments, as is standard in C++. The `fastjet::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:

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

[7] 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.

```
/// 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 9.4.2).

Unclustered particles. User-supplied plugin jet algorithms (see section 9) 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<fastjet::PseudoJet> unclustered = clust_seq.unclustered_particles();
```

3.4 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. [8]

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

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

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

[8] 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.

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.

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;

/// 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.

3.5 Composite jets

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, 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.

For a jet formed in this way, one can find out which pieces it has been composed from with the function

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

which, in the above example, would simply return a vector of size 2 containing `jet1` and `jet2`. For jets not formed in this (or a related) way `pieces()` will throw an error.

Inquiries 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 kinds of structural information it has. Accordingly a number of inquiry functions are available:

```
bool has_structure();         // returns true if it has some kind of structural info
bool has_constituents();      // returns 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
```

The dichotomy between cluster-sequence type structural information and `pieces` helps provide a distinction between different origins for structure — either that arising diretly from a jet algorithm, or that applied a posteriori. There is a balance between the usefulness of such a dichotomy and the added complication it brings. In this respect version 3.0alpha3 should be considered a snapshot of ongoing thought about the question.

Features not yet documented The following features still need documenting.

```
PseudoJetStructureBase;
structure<StructureType>();
has_structure_of<ToolName>();
structure_of<ToolName>();
```

3.6 Version information

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

```
std::string fastjet::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 Example program

For full details see the example program example/fastjet_example.cc that is distributed with the FastJet code. For the reader who is familiar with KtJet [10], the example program is also given as it would be written for KtJet, in the file example/ktjet_example.cc.

A simplified version of the FastJet example program is given below.

```
#include "fastjet/PseudoJet.hh"
#include "fastjet/ClusterSequence.hh"
using namespace std;

int main (int argc, char ** argv) {
  vector<fastjet::PseudoJet> input_particles;

  // read in input particles
  double px, py , pz, E;
  while (cin >> px >> py >> pz >> E) {
    // push fastjet::PseudoJet of (px,py,pz,E) on back of input_particles
    input_particles.push_back(fastjet::PseudoJet(px,py,pz,E));
  }

  // create an object that represents your choice of jet algorithm and
  // its associated parameters
  double Rparam = 1.0;
  fastjet::Strategy strategy = fastjet::Best;
  fastjet::JetDefinition jet_def(fastjet::kt_algorithm, Rparam, strategy);

  // run the jet clustering with the above jet definition
  fastjet::ClusterSequence clust_seq(input_particles, jet_def);

  // extract the inclusive jets with pt > 5 GeV
  double ptmin = 5.0;
  vector<fastjet::PseudoJet> inclusive_jets = clust_seq.inclusive_jets(ptmin);

  // extract the exclusive jets with dcut = 25 GeV^2 and sort them
```

[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.

```
  // in order of increasing pt
  double dcut = 25.0;
  vector<fastjet::PseudoJet> exclusive_jets = sorted_by_pt(
                                        clust_seq.exclusive_jets(dcut));

  // print out the details for each jet
  for (unsigned int i = 0; i < exclusive_jets.size(); i++) {
    // get constituents of the jet. NB this is through a member function
    // of clust_seq because that is where the information is held.
    vector<fastjet::PseudoJet> constituents = exclusive_jets[i].constituents();

    printf("%5u %15.8f %15.8f %15.8f %8u\n",
           i, exclusive_jets[i].rap(), exclusive_jets[i].phi(),
           exclusive_jets[i].perp(), constituents.size());
  }
}
```

5 FastJet native jet algorithms

5.1 k_t jet algorithm

The definition of the inclusive k_t jet algorithm that is coded is as follows (and corresponds to [3], modulo small changes of notation):

1. For each pair of particles i, j work out the k_t distance

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

   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 $d_{iB} = k_{ti}^2$.

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 longitudinally invariant k_t jet algorithm [2] 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.

5.2 Cambridge/Aachen jet algorithm

Currently the Cambridge/Aachen jet algorithm is provided only in an inclusive version [5], whose formulation is identical to that of the k_t jet algorithm, except as regards the distance measures, which

are:

$$d_{ij} = \Delta R_{ij}^2/R^2\,, \tag{2a}$$
$$d_{iB} = 1\,. \tag{2b}$$

Attempting to extract exclusive jets from the Cambridge/Aachen 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 [4] 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.

5.3 Anti-k_t jet algorithm

This new algorithm, introduced and studied in [6], is defined exactly like the standard k_t algorithm, except for the distance measures which are now given by

$$d_{ij} = \min(1/k_{ti}^2, 1/k_{tj}^2)\,\Delta R_{ij}^2/R^2\,, \tag{3a}$$
$$d_{iB} = 1/k_{ti}^2\,. \tag{3b}$$

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 [6].

5.4 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:

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

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 ($p = -1$).

5.5 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 [9], but extended in analogy with the generalised pp algorithm of [6] and section 5.4. We define the two following distances:

$$d_{ij} = \min(E_i^{2p}, E_j^{2p})\frac{(1-\cos\theta_{ij})}{(1-\cos R)}\,, \tag{5a}$$
$$d_{iB} = E_i^{2p}\,, \tag{5b}$$

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. (5), 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.

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 [9], often referred to also as the Durham algorithm, which has a single distance:

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

Note the difference in normalisation between the d_{ij} in eqs. (5) and (6), and the fact in neither case have we normalised to the total energy Q in the event, contrary to the convention adopted originally in [9] (where the distance measure was called y_{ij}).

5.6 Recombination schemes

When merging particles in step 2 of 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 [8], and is the default option in `FastJet`.

Other schemes for *pp* collisions. Other schemes provided by earlier k_t-clustering implementations [10] 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

$$p_{t,r} = p_{t,i} + p_{t,j}\,, \tag{7a}$$
$$\phi_r = (w_i\phi_i + w_j\phi_j)/(w_i + w_j)\,, \tag{7b}$$
$$y_r = (w_i y_i + w_j y_j)/(w_i + w_j)\,, \tag{7c}$$

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 A.

6 Selectors

6.1 Essential usage

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 `Selector` class, available through

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

As an example of how it is used, if have a vector of jets, `jets`, and wish to select those that have rapidities $|y| < 2.5$ and transverse momenta above 20 GeV, then 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.

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 may 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 in C++ higher precedence than boolean operators.

6.2 Available selectors

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, M, 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 GeV
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`($y_{\min}, y_{\max}, \phi_{\min}, \phi_{\max}$).

Mostly for completeness, there is a `SelectorIdentity` which selects everything. And, as a helper for the situations where ghost areas are computed using explicit ghosts, `SelectorIsPureGhost` selects only pure-ghost jets.

In addition there are also some selectors that take a *reference jet*. These 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.

One example of a selector that takes a reference jet is:

```
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. The other selectors taking a reference that are available by default are `SelectorStrip`, `SelectorDoughnut`, `SelectorRectangle` and `SelectorPtFractionMin`.

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

6.3 Other information about selectors

Selectors contain a certain amount of additional information that can provide useful hints in some contexts (for example, in anticipation of their future use for background estimation).

One such piece of information is a selector's rapidity extent, accessible through a `get_rapidity_extent(rapmin,rapmax)` call. 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.

Selectors may also have areas associated with them (in analogy with jet areas, section 7). The `has_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).

6.4 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 are relocatable), users may wish to examine the implementation of the existing workers and/or consult the authors.

[**to come**]:

- areas?

7 Jet areas, background evaluation and subtraction

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 [16] 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 [16]. 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 [17], 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::BackgroundEstimator
class fastjet::Subtractor
```

These classes are described in detail below and an example program is given in section 7.5. [**Check it before release...?**]

7.1 fastjet::AreaDefinition

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

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

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

```
fastjet::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 fastjet::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.[10]

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.[11] This is done via the class `fastjet::GhostedAreaSpec` whose constructor is

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

[10] 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 [16]. For other algorithms it defaults back to the `one_ghost_passive_area` approach.

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

```
/// 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

```
fastjet::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 `fastjet::ClusterSequenceArea` by passing the proper `fastjet::VoronoiAreaSpec` specification to `fastjet::AreaDefinition`. Its constuctors are

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

/// constructor that allows you to set effective_Rfact
fastjet::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 [18].

One use for the Voronoi area is in background determination with the k_t algorithm (below, section 7.3): 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[12] 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 `fastjet::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 fastjet::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 fastjet::PseudoJet & jet) const ;

/// return a PseudoJet whose 4-vector is defined by the following integral
///
///       ∫ dydφ PseudoJet(y,φ,p_t = 1) * Θ("y,φ inside jet boundary")
///
/// where PseudoJet(y,φ,p_t = 1) is a 4-vector with the given
/// rapidity (y), azimuth (φ) and p_t = 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 fastjet::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.

[12] `ClusterSequenceArea` makes use of one among `ClusterSequenceActiveArea`, `ClusterSequenceActiveAreaExplicitGhosts`, `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.

7.3 Background estimation

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 7.4.

One of the difficulties 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 [17] and studied in detail also in [33]. This is 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.

Note that we recommend that you use only the k_t or Cambridge/Aachen algorithms for calculating the UE/pileup density ρ in this way. In particular, it is important to avoid algorithms with extreme behaviours for the areas. Specifically, one should not use anti-k_t (which has many jets with near-zero area), or SISCone (which can have jets with near-zero area, as well as monster jets with huge areas if the overlap parameter f is too small). The choice of jet radius can also have an impact on the determination, especially for events that are relatively sparse. We have found that a choice in the range $R = 0.4 - 0.6$ is generally adequate [17, 33] 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.

Two elements are needed for FastJet's background estimation: a cluster sequence with area information and a `Selector` to specify the subset of jets that will be used for the background estimation. The class that provides background estimation facilities is `BackgroundEstimator`. The simplest way to construct it is:

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

// constructor for a BackgroundEstimator that will use the
// inclusive jets from csa that pass the cuts specified by
// the selector in order to estimate the background density
BackgroundEstimator(const ClusterSequenceAreaBase & csa,
                    const Selector & selector);
```

If, for example, one wants to use all jets within $|y| < 4.5$ to estimate the background, one should choose `selector = SelectorAbsRapMax(4.5)`. The background density ρ and its fluctuations σ can then be accessed through the functions:

```
// the median of (p_t/area) for jets that pass the selection cut,
// making use also of information on empty area in the event
double rho()   const;

// 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>)
double sigma() const;
```

Note that ρ and σ determinations count empty area within the selected region as consisting of jets of zero p_t. Thus (roughly speaking), if more that half of the area covered by the selector is empty the median estimate for ρ will be zero. **[empty areas should be discussed more in comments to empty area calls below — check it's OK later]**

7.3.1 Positional dependence of background

One drawback of a ρ estimate based on all jets 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 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 functions

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

a `selector.set_reference(jet)` call is made to centre the selector on the specified jet. Then the inclusive jets in `csa` that pass the cut specified by this newly positioned `selector` are used to estimate ρ or σ.[13] 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))
```

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.[14]

In unpublished studies for the case of pp pileup events (with Pythia 8.145, tune 4C for the pileup [34]), 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. One first creates an object that encodes the shape

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

(for other shapes, derive a class from `FunctionOfPseudoJet<double>` — see appendix D) and then one tells the `BackgroundEstimator` about the rescaling with the call

```
// tell the BackgroundEstimator about the rescaling
set_rescaling_class(*rescaling);
```

[13] 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.

[14] 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.

Subsequent calls will then take the median of the distribution $p_t/A/\texttt{rescaling}(y)$ (rather than p_t/A) and any calls to `rho(jet)` and `sigma(jet)` will including an additional factor of $\texttt{rescaling}(y_{\texttt{jet}})$. 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()`.

7.3.2 Other facilities

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)
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 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 ($0.55\pi R^2$ [16]). All four function above return a result corresponding to the last call to `rho` or `sigma` (as long as the cluster sequence or selector has not changed in the meantime).

To allow flexibility in the user's workflow, there are a number of alternative constructors to `BackgroundEstimator`, together with elements to (re)set the cluster sequence and/or jets, and also the selector:

```
// the estimator will use the supplied jets instead of inclusive jets from a ClusterSequence
BackgroundEstimator(const std::vector<PseudoJet> & jets, const Selector & rho_range);
// information about the ClusterSequence or jets will be set later
BackgroundEstimator(const Selector & rho_range);
// information about the ClusterSequence/jets and Selector will be set later
BackgroundEstimator();

// (re)set the cluster sequence to be used by future calls to rho() etc.
void set_cluster_sequence(const ClusterSequenceAreaBase & csa);
// (re)set the jets to be used by future calls to rho() etc.
void set_jets(const std::vector<PseudoJet> & jets);
// (re)set the selector to be used for future calls to rho() etc.
void set_selector(const Selector & rho_range_selector);
```

Jets passed to any of the above calls must originate from a common originate from a common `ClusterSequenceAreaBase` type class.

Finally, there are cases where the user may wish to determine medians of quantities other than p_t/A. This may arise in estimating the hadron-mass contribution to pileup contamination of jet masses, or in the determination of moments of fragmentation functions. For this purpose, the user may use the following call

```
void set_jet_density_class(const FunctionOfPseudoJet<double> * jet_density_class);
```

For each jet that passes the selection, `jet_density_class`'s `FunctionOfPseudoJet<double>::operator(PseudoJet & jet)` function is called[15] and it returns the quantity that substitutes p_t/A. [**Do we discuss the `BackgroundJetPtMDensity` class as an example here?**]

7.3.3 Backwards compatibility

The `BackgroundEstimator` class is new to FastJet 3.0. 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 `BackgroundEstimator`. 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. A call to `set_provide_fj2_sigma(true)` causes `BackgroundEstimator` to reproduce that behaviour.

7.4 Background subtraction

The `Subtractor` class is defined in `include/tools/Subtractor.hh`. Its constructor takes a pointer to a background estimator:

```
BackgroundEstimator bge(....);
Subtractor subtractor(&bge);
```

The subtractor can then be used as follows:

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

[15] As explained in Appendix D describing functions of `PseudoJet`, user-defined density classes derived from `FunctionOfPseudoJet<double>` should overload `FunctionOfPseudoJet<double>::result()`.

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. In both cases, the returned jet retains the user and structural information of the original jet. Note that `Subtractor` derives from `FunctionOfPseudoJet<PseudoJet>` (cf. appendix D).

7.5 Example program with areas and subtraction

See `example/07-subtraction-new.cc`.

8 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. As of version 3.0, `FastJet` 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 `Transformer`s. We then describe some that have already been implemented. Finally, we outline some of the techniques that can help when creating new transformers.

8.1 Generic considerations about `Transformers`

The common interface to all the transformers, declared in the `fastjet/tools/Transformer.hh` header, is the following:

```
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(){}
};
```

It is essentially the same as that of the `FunctionOfPseudoJet<PseudoJet>` from which it derives (cf. appendix D). While the action of a `Transformer` is to be implemented in the `result(...)` member function, typically it will be accessed 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;
```

i.e.

```
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

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

which would give direct access to the structural information produced by `MyTransformer`. Using `jet.has_structure_of<MyTransformer>()`, it is possible to check if `jet` is compatible with the structure provided by `MyTransformer`.

8.2 Jet Filtering and Trimming using `Filter`

Filtering was first introduced in [19] 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 $R_{\rm filt}$ and only keep the 3 hardest of these subjets as the pieces of the final filtered jet, rejecting the others. Similarly, trimming [20] also unclusters a jet into subjets but selects the subjets to be kept based on a p_t cut.

The `Filter` class, defined in `fastjet/tools/Filter.hh`, handles both filtering and trimming. Its constructor

```
class Filter : public Transformer {
  Filter(JetDefinition subjet_def, Selector selector, double rho = 0.0);
  ...
};
```

allows one to specify a filter/trimmer that reclusters the jet's constituents with the jet definition `subjet_def` 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.

Two alternative constructors are supported:

```
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. For 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 [19] where $R_{\rm filt} = \min(0.3, R_{b\bar{b}})$, with $R_{b\bar{b}}$ the distance between the parents of the jet). In both cases, the Cambridge/Aachen algorithm is used.

Constraints. The jet being filtered must have constituents. Furthermore, if `rho` is non-zero, the input jet must be the result of a `ClusterSequence` with area support and explicit ghosts. If any of these requirements fail, an exception will be thrown.

Structure of the result. The `pieces()` of the resulting filtered/trimmed jet correspond to the subjets that were kept. Additional structural information is available as follows:

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

Usage examples. Filtering involving obtaining the subjets with the Cambridge/Aachen algorithm with radius $R_{\rm filt}$ and keeping the $n_{\rm filt}$ hardest subjets found can be achieved using

```
Filter filter(Rfilt, SelectorNHardest(nfilt));
```

Trimming, which keeps the subjets with a p_t larger than a fixed fraction of the input jet, is done using

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

Implementation note. Usually, the subjets are obtained simply by clustering the input jet's constistuents with the subjet definition. However, 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 recluster its constituents.

8.3 2-pronged and 3-pronged taggers

A number of the taggers developed to distinguish 2- or 3-pronged decays of massive objects from plain QCD jets 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 two of the taggers proposed for boosted Higgs reconstruction: the `MassDropTagger` introduced in [19] and a rest-frame `NSubjettinessTagger` from Ref. [21]. We refer the reader to these articles for an extensive description of the taggers and shall only give a very brief overview in what follows.

8.3.1 The mass-drop tagger

Introduced in [19], this tagger 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, $y \equiv \min(p^2_{tj_1}, p^2_{tj_2})\Delta R^2_{j_1 j_2}/m^2_j > y_{\rm cut}$, 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.

Constraints. [rest of documentation still to be written]

Structure of the result.

8.3.2 The rest-frame N-subjettiness tagger

Constraints.

Structure of the result.

8.3.3 The Johns-Hopkins top tagger

[still to be implemented]

8.4 User-defined transformers

9 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[16] to be used within the same framework.

9.1 Generic plugin use and construction

Any plugin is to be derived from the abstract base class `fastjet::JetDefinition::Plugin`,

```
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;
```

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

```
    /// 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 p_t << 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;

  };
```

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
  fastjet::JetDefinition jet_def = fastjet::JetDefinition( & cdf_midpoint);

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

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 later for SISCone.

Example plugin jet algorithms that are provided with `FastJet` are the `CDFMidPointPlugin` illustrated above and also a `CDFJetCluPlugin`. These interface code available at [22]. The `PxConePlugin` interfaces the PxCone code [23]. All three of these cone jet algorithms, though widely used, are infrared or collinear unsafe (depending on the parameters). The `SISConePlugin`, described in detail below, interfaces the infrared safe seedless cone algorithm of [7]. Examples using these plugins and some further documentation are provided in the `plugins/` directory.

9.2 SISCone Plugin

SISCone [7] 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 [8] split–merge procedure.[17] 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 [24], SISCone can carry out multiple stable-cone search passes (as advocated in [25]): 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 [8], 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).[18]

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

[17] 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.

[18] 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`, $\gtrsim 0.6$.

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 };
```

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 [8]) 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<fastjet::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.

9.3 Other plugins for *pp*

Not all plugins are enabled by default in `FastJet`. At configuration time `./configure.sh --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.

9.3.1 CDF Midpoint.

One of the two algorithms used by CDF in Run II of the Tevatron, based on [8]. 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 [24].

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 (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 [7]; with `cone_area_fraction`$\neq 1$ it becomes IR_{2+1} unsafe [25]). It is to be deprecated for new experimental or theoretical analyses.

9.3.2 CDF JetClu

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

```
#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 fo 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.

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.

9.3.3 DØ Run I cone

The main algorithm used by DØ in Run I of the Tevatron, 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 [32].

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.

9.3.4 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.

9.3.5 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 [30].

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.

9.3.6 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 [6]. It is to be deprecated for new experimental or theoretical analyses.

9.3.7 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 [27].

The underlying code for this algorithm was taken from a google search for PxCone!

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

9.3.8 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.

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

9.4 Other plugins for e^+e^-

9.4.1 Cambridge algorithm

The original e^+e^- cambridge [4] 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.

9.4.2 Jade algorithm

The JADE algorithm [28, 29], 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 [28]. 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 5.6,A) to modify the recombination scheme.

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

9.5 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 [26] 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.

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.

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` [15].[19] At configure time the `--enable-cgal`

[19] 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.)

option may be used to specify that CGAL support should be included.

`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 [12].

Acknowledgements

We wish to thank Timothy Chan for helpful exchanges about the details of his dynamic closest pair algorithm. We thank Markus Wobisch and Andreas Oehler for comments on the documentation, and Sebastian Sapeta for bug reports and testing of beta-version of the code, as well as to Juan Rojo for discussions related to various features of the code.

We are grateful to CDF, Joey Huston, Lars Sonnenschein and Mike Seymour for providing permission to distribute their cone jet algorithm codes as plugins for fastjet and to Salvatore Rappoccio for assistance in validating the CMS cone. We thank Andy Buckley for numerous comments on the documentation and build system.

A 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.

B 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, if an input particle can come from one of several HepMC events).

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 extra info has not been set, an error will be thrown.

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.[20]

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

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

[20] 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 then imply that every other class in `FastJet` should then also be templated.

	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.

It behaves almost identically to the `C++0x shared_ptr`.[21] 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();
```

C Structural information for various inputs

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(...)`

[21] 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.

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.

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.

D Functions of a PseudoJet

A concept that is new to FastJet v3.0 is that of a FunctionOfPseudoJet. Functions of PseudoJets 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 7.3.1.

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 PseudoJets.[22].

This definition allows one for example to pass functions of PseudoJets as arguments. This is *e.g.* used for the background rescalings in section 7.3.1 which are just derived from FunctionOfPseudoJet<double>. It is also used for the Transformers of section 8, which all derive

[22]Having result(...) and operator(...) doing the same thing may seems 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.

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.

References

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

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

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

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

[5] 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.

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

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

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

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

[10] `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].

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

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

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

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

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

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

[17] M. Cacciari and G. P. Salam, arXiv:0707.1378 [hep-ph].

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

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

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

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

[22] The CDF code has been taken from `http://www.pa.msu.edu/~huston/Les_Houches_2005/JetClu+Midpoint-StandAlone.tgz`.

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

[24] 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].

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

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

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

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

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

[30] P.A. Delsart, K. Geerlings and J. Huston, SpartyJet, `http://www.pa.msu.edu/~huston/SpartyJet/SpartyJet.html`

[31] J. E. Huth *et al.*, FNAL-C-90-249-E, published in the proceedings of the 1990 Summer Study on High Energy Physics, Research Directions for the Decade, Snowmass, Colorado, June 25 – July 13, 1990.

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

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

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