CERN-PH-TH/2011-297

FastJet user manual
(for version 3.1.1)

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

[1]LPTHE, UPMC Univ. Paris 6 and CNRS UMR 7589, Paris, France
[2]Université Paris Diderot, Paris, France
[3]CERN, Physics Department, Theory Unit, Geneva, Switzerland
[4]Department of Physics, Princeton University, Princeton, NJ 08544, USA
[5]Institut de Physique Théorique, CEA Saclay, France

Abstract

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

Contents

1 Introduction

Jets are the collimated sprays of hadrons that result from the fragmentation of a high-energy quark or gluon. They tend to be visually obvious structures when one looks at an experimental event display, and by measuring their energy and direction one can approach the idea of the original "parton" that produced them. Consequently jets are both an intuitive and quantitatively essential part of collider experiments, used in a vast array of analyses, from new physics searches to studies of Quantum Chromodynamics (QCD). For any tool to be so widely used, its behaviour must be well defined and reproducible: it is not sufficient that one be able to visually identify jets, but rather one should have rules that project a set of particles onto a set of jets. Such a set of rules is referred to as a jet algorithm. Usually a jet algorithm involves one or more parameters that govern its detailed behaviour. The combination of a jet algorithm and its parameters is known as a jet definition. Suitable jet definitions can be applied to particles, calorimeter towers, or even to the partonic events of perturbative QCD calculations, with the feature that the jets resulting from these different kinds of input are not just physically close to the concept of partons, but can be meaningfully be compared to each other.

Jet finding dates back to seminal work by Sterman and Weinberg [1] and several reviews have been written describing the various kinds of jet finders, their different uses and properties, and even the history of the field, for example [2, 3, 4, 5, 6].

It is possible to classify most jet algorithms into one of two broad classes: sequential recombination algorithms and cone algorithms.

Sequential recombination algorithms usually identify the pair of particles that are closest in some distance measure, recombine them, and then repeat the procedure over and again, until some stopping criterion is reached. The distance measure is usually related to the structure of divergences in perturbative QCD. The various sequential recombination algorithms differ mainly in their particular choices of distance measure and stopping criterion.

Cone algorithms put together particles within specific conical angular regions, notably such that the momentum sum of the particles contained in a given cone coincides with the cone axis (a "stable cone"). Because QCD radiation and hadronisation leaves the direction of a parton's energy flow essentially unchanged, the stable cones are physically close in direction and energy to the original partons. Differences between various cone algorithms are essentially to do with the strategy taken to search for the stable cones (e.g. whether iterative or exhaustive) and the procedure used to deal with cases where the same particle is found in multiple stable cones (e.g. split–merge procedures).

One of the aims of the `FastJet` C++ library is to provide straightforward, efficient implementations for the majority of widely used sequential-recombination algorithms, both for hadron-hadron and e^+e^- colliders, and easy access also to cone-type jet algorithms. It is distributed under the terms of version 2 of the GNU General Public License (GPL) [7].

To help introduce the terminology used throughout `FastJet` and this manual, let us consider the longitudinally-invariant k_t algorithm for hadron colliders [8, 9]. This was the first jet algorithm to be implemented in `FastJet` [10] and its structure, together with that of other sequential recombination algorithms, has played a key role in the design of `FastJet`'s interface. The k_t algorithm involves a (symmetric) distance measure, d_{ij}, between all pairs of particles i and j,

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

where p_{ti} is the transverse momentum of particle i with respect to the beam (z) direction and $\Delta R_{ij}^2 =$

$(y_i - y_j)^2 + (\phi_i - \phi_j)^2$, with $y_i = \frac{1}{2}\ln\frac{E_i+p_{zi}}{E_i-p_{zi}}$ and ϕ_i respectively i's rapidity and azimuth. The k_t algorithm also involves a distance measure between every particle i and the beam

$$d_{iB} = p_{ti}^2 \,. \tag{2}$$

R in eq. (1), usually called the jet radius, is a parameter of the algorithm that determines its angular reach. In the original, so-called "exclusive" formulation of the k_t algorithm [8] (proposed with $R \equiv 1$), one identifies the smallest of the d_{ij} and d_{iB}. If it is a d_{ij}, one replaces i and j with a single new object whose momentum is $p_i + p_j$ — often this object is called a "pseudojet", since it is neither a particle, nor yet a full jet.[1] If instead the smallest distance is a d_{iB}, then one removes i from the list of particles/pseudojets and declares it to be part of the "beam" jet. One repeats this procedure until the smallest d_{ij} or d_{iB} is above some threshold $d_{\rm cut}$; all particles/pseudojets that are left are then that event's (non-beam) jets.[2]

In the "inclusive" formulation of the k_t algorithm [9], the d_{ij} and d_{iB} distances are the same as above. The only difference is that when a d_{iB} is smallest, then i is removed from the list of particles/pseudojets and added to the list of final "inclusive" jets (this is instead of being incorporated into a beam jet). There is no $d_{\rm cut}$ threshold and the clustering continues until no particles/pseudojets remain. Of the final jets, generally only those above some transverse momentum are actually used.[3] Because the distance measures are the same in the inclusive and exclusive algorithms, the clustering sequence is common to both formulations (at least up to $d_{\rm cut}$), a property that will be reflected in `FastJet`'s common interface to both formulations.

Having seen these descriptions, the reader may wonder why a special library is needed for sequential-recombination jet finding. Indeed, the k_t algorithm can be easily implemented in just a few dozen lines of code. The difficulty that arises, however, is that at hadron colliders, clustering is often performed with several hundreds or even thousands of particles. Given N particles, there are $N(N-1)/2$ d_{ij} distances to calculate, and since one must identify the smallest of these $\mathcal{O}(N^2)$ distances at each of $\mathcal{O}(N)$ iterations of the algorithm, original implementations of the k_t algorithm [11, 12] involved $\mathcal{O}(N^3)$ operations to perform the clustering. In practice this translates to about 1 s for $N = 1000$. Given that events with pileup can have multiplicities significantly in excess of 1000 and that it can be necessary to cluster hundreds of millions of events, N^3 timing quickly becomes prohibitive, all the more so in time-critical contexts such as online triggers. To alleviate this problem, `FastJet` makes use of the observation [10] that the smallest pairwise distance remains the same if one uses the following alternative (non-symmetric) d_{ij} distance measure:

$$d_{ij} = p_{ti}^2 \frac{\Delta R_{ij}^2}{R^2} \,, \qquad d_{ji} = p_{tj}^2 \frac{\Delta R_{ij}^2}{R^2} \tag{3}$$

For a given i, the smallest of the d_{ij} is simply found by choosing the j that minimises the ΔR_{ij}, i.e. by identifying i's geometrical nearest neighbour on the $y-\phi$ cylinder. Geometry adds many constraints to closest pair and nearest neighbour type problems, e.g. if i is geometrically close to k and j is geometrically close to k, then i and j are also geometrically close; such a property is not true for the d_{ij}. The factorisation of the problem into momentum and geometrical parts makes it possible to calculate and search for minima among a much smaller set of distances. This is sufficiently

[1] In `FastJet` we actually will use `PseudoJet` to denote any generic object with 4-momentum.

[2] One can also choose to have the exclusive algorithm stop when it reaches a configuration with a predetermined number of remaining particles/pseudojets, which then become the event's (non-beam) jets.

[3] This transverse momentum cut has some similarity to $d_{\rm cut}$ in the exclusive case, since in the exclusive case pseudojets with $p_t < \sqrt{d_{\rm cut}}$ become part of the beam jets, i.e. are discarded.

powerful that with the help of the external Computational Geometry Algorithms Library (CGAL) [13] (specifically, its Delaunay triangulation modules), `FastJet` achieves expected $N \ln N$ timing for many sequential recombination algorithms. This $N \ln N$ strategy is supplemented in `FastJet` with several other implementations, also partially based on geometry, which help optimise clustering speed up to moderately large multiplicities, $N \lesssim 30000$. The timing for $N = 1000$ is thus reduced to a few milliseconds. The same techniques apply to a range of sequential recombination algorithms, described in section 4.

At the time of writing, sequential recombination jet algorithms are the main kind of algorithm in use at CERN's Large Hadron Collider (LHC), notably the anti-k_t algorithm [14], which simply replaces p_t^2 with p_t^{-2} in eqs. (1,2). Sequential recombination algorithms were also widely used at HERA and LEP. However at Fermilab's Tevatron, and in much preparatory LHC work, cone algorithms were used for nearly all studies. For theoretical and phenomenological comparisons with these results, it is therefore useful to have straightforward access also to cone algorithm codes. The main challenge that would be faced by someone wishing to write their own implementation of a given cone algorithm comes from the large number of details that enter into a full specification of such algorithms, e.g. the precise manner in which stable cones are found, or in which the split–merge step is carried out. The complexity is such that in many cases the written descriptions that exist of specific cone algorithms are simply insufficient to allow a user to correctly implement them. Fortunately, in most cases, the authors of cone algorithms have provided public implementations and these serve as a reference for the algorithm. While each tends to involve a different interface, a different 4-momentum type, etc., `FastJet` has a "plugin" mechanism, which makes it possible to provide a uniform interface to these different third party jet algorithms. Many plugins (and the corresponding third party code) are distributed with `FastJet`. Together with the natively-implemented sequential-recombination algorithms, they ensure easy access to all jet algorithms used at colliders in the past decade (section 5). Our distribution of this codebase is complemented with some limited curatorial activity, e.g. solving bugs that become apparent when updating compiler versions, providing a common build infrastructure, etc.

In the past few years, research into jets has evolved significantly beyond the question of just "finding" jets. This has been spurred by two characteristics of CERN's LHC experimental programme. The first is that the LHC starts to probe momentum scales that are far above the the electroweak scale, M_{EW}, e.g. in the search for new particles or the study of high-energy WW scattering. However, even in events with transverse momenta $\gg M_{EW}$, there can simultaneously be hadronic physics occurring on the electroweak scale (e.g. hadronic W decays). Jet finding then becomes a multi-scale problem, one manifestation of which is that hadronic decays of W's, Z's and top quarks may be so collimated that they are entirely contained within a single jet. The study of this kind of problem has led to the development of a wide array of jet substructure tools for identifying "boosted" object decays, as reviewed in [15]. As was the situation with cone algorithms a few years ago, there is considerable fragmentation among these different tools, with some public code available from a range of different sources, but interfaces that differ from one tool to the next. Furthermore, the facilities provided with version 2 of `FastJet` did not always easily accommodate tools to manipulate and transform jets. Version 3 of `FastJet` aims to improve this situation, providing implementations of the most common jet substructure tools[4] and a framework to help guide third party authors who wish to write further such tools using a standard interface (section 9). In the near future we also envisage the creation of a `FastJet` "contrib" space, to provide a common location for access to these new tools as they are developed.

[4]To some extent there is overlap here with SpartyJet [16], however we believe there are benefits to being able to easily carry jet structure manipulations natively within the framework of `FastJet`.

The second characteristic of the LHC that motivates facilities beyond simple jet finding in FastJet is the need to use jets in high-noise environments. This is the case for proton-proton (pp) collisions, where in addition to the pp collision of interest there are many additional soft "pileup" pp collisions, which contaminate jets with a high density of low-momentum particles. A similar problem of "background contamination" arises also for heavy-ion collisions (also at RHIC) where the underlying event in the nuclear collision can generate over a TeV of transverse momentum per unit rapidity, part of which contaminates any hard jets that are present. One way of correcting for this involves the use of jet "areas", which provide a measure of a given jet's susceptibility to soft contamination. Jet areas can be determined for example by examining the clustering of a large number of additional, infinitesimally soft "ghost" particles [17]. Together with a determination of the level of pileup or underlying-event noise in a specific event, one can then perform event-by-event and jet-by-jet subtraction of the contamination [18, 19]. FastJet allows jet clustering to be performed in such a way that the jet areas are determined at the same time as the jets are identified, simply by providing an "area definition" in addition to the jet definition (section 7). Furthermore it provides the tools needed to estimate the density of noise contamination in an event and to subtract the appropriate amount of noise from each jet (section 8). The interface here shares a number of characteristics with the substructure tools, some of which also serve to remove noise contamination. Both the substructure and pileup removal make use also of a "selectors" framework for specifying and combining simple cuts (section 6).

While FastJet provides a broad range of facilities, usage for basic jet finding is straightforward. To illustrate this, a quick-start guide is provided in section 2, while the core classes (PseudoJet, JetDefinition and ClusterSequence) are described in section 3. For more advanced usage, one of the design considerations in FastJet has been to favour user extensibility, for example through plugins, selectors, tools, etc. This is one of the topics covered in the appendices. Further information is also available from the extensive "doxygen" documentation, available online at http://fastjet.fr.

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

  ```
  curl -O http://fastjet.fr/repo/fastjet-X.Y.Z.tar.gz
  tar zxvf fastjet-X.Y.Z.tar.gz
  cd fastjet-X.Y.Z/
  ```

  replacing X.Y.Z with the appropriate version number. On some systems you may need to replace "curl -O" with "wget".

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

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

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

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

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

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

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

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

  // print out some info
  cout << "Clustered with " << jet_def.description() << endl;

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

- Then compile and run it with

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

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

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

```
Clustering with Longitudinally invariant anti-kt algorithm with R = 0.7
and E scheme recombination
        pt y phi
jet 0: 103 0 0
    constituent 0's pt: 99.0001
    constituent 1's pt: 4.00125
jet 1: 99 0 3.14159
    constituent 0's pt: 99
```

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

3 Core classes

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

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

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

3.1 fastjet::PseudoJet

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

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

where the second form allows the initialisation to be obtained from any class T that allows subscripting to return the components of the momentum (running from $0 \ldots 3$ in the order p_x, p_y, p_z, E). 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...2pi
double phi_std()  const ; // returns the azimuthal angle in range -pi...pi
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 pt2()      const ; // returns the squared transverse momentum
double pt()       const ; // returns the transverse momentum
double perp2()    const ; // returns the squared transverse momentum
double perp()     const ; // returns the transverse momentum
double m2()       const ; // returns squared invariant mass
double m()        const ; // returns invariant mass (-sqrt(-m^2) if m^2 < 0)
double mt2()      const ; // returns the squared transverse mass = k_t^2 + m^2
double mt()       const ; // returns the transverse mass
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;
```

The reader may have observed that in some cases more than one name can be used to access the same quantity. This is intended to reflect the diversity of common usages within the community.[5]

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

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

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

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

Further details are to be found in appendix B and in `example/09-user_info.cc`.

A `PseudoJet` can be reset with

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

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

[5]The `pt()`, `pt2()`, `mt()`, `mt2()` names are available only from version 3.0.1 onwards.

Additionally, the +, -, * and / operators are defined, with +, - acting on pairs of PseudoJets and *, / acting on a PseudoJet and a double coefficient. The analogous +=, etc., operators, are also defined.[6]

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

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

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

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

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

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

A number of further PseudoJet member functions provide access to information on a jet's structure. They are documented below in sections 3.5 and 3.6.

3.2 fastjet::JetDefinition

The class JetDefinition contains a full specification of how to carry out the clustering. According to the Les Houches convention detailed in [20], a 'jet definition' should include the jet algorithm name, its parameters (often the radius R) and the recombination scheme. Its constructor is

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

The jet algorithm is one of the entries of the JetAlgorithm enum:[7]

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

Each algorithm is described in detail in section 4. The ... represent additional values that are present for internal or testing purposes. They include plugin_algorithm, automatically set when plugins are used (section 5) and undefined_jet_algorithm, which is the value set in JetDefinition's default constructor.

[6] The +, - operators return a PseudoJet with default user information; the * and / operators return a PseudoJet with the same user information as the original PseudoJet; the +=, -=, etc., operators all preserve the user information of the PseudoJet on the left-hand side of the operator.

[7] 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.

The parameter `R` specifies the value of R that appears in eq. (1) and in the various definitions of section 4. For one algorithm, `ee_kt_algorithm`, there is no R parameter, so the constructor is to be called without the `R` argument. For the generalised k_t algorithm and its e^+e^- version, one requires R and (immediately after R) an extra parameter p. Details are to be found in sections 4.4–4.6. If the user calls a constructor with the incorrect number of arguments for the requested jet algorithm, a `fastjet::Error()` exception will be thrown with an explanatory message.

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

The strategy selects the algorithmic strategy to use while clustering and is an `enum` of type `Strategy`. The default option of `Best` automatically determines and selects a strategy that should be close to optimal in speed for each event, based on its multiplicity. A discussion of the main available strategies together with their performance is given in appendix A. Different strategies give identical clustering results, except potentially in the presence of distance degeneracies, where different strategies may resolve those degeneracies differently.

A textual description of the jet definition can be obtained by a call to the member function `std::string description()`.

As of `FastJet` version 3.1, the `JetDefinition` class allows one to directly perform the clustering and access the inclusive jets through a `()` operator:

```
template<class L> std::vector<PseudoJet> JetDefinition::operator()(
                                        const std::vector<L> & particles
                                        ) const;
```

Doing

```
fastjet::JetDefinition AKT4(fastjet::antikt_algorithm,0.4);
vector<fastjet::PseudoJet> akt4_jets = AKT4(particles);
```

returns the list of all the inclusive jets given by clustering with the anti-k_t algorithm with radius parameter $R = 0.4$. Jets from the anti-k_t algorithm and other hadron-collider algorithms are returned sorted in decreasing transverse momentum, while those from e^+e^- algorithms are returned sorted in decreasing energy. This behaviour differs from that of `ClusterSequence::inclusive_jets()`, where the ordering of the jets is not a priori determined (i.e. the user must manually sort the jets themselves).

3.3 `fastjet::ClusterSequence`

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

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

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

3.3.1 Accessing inclusive jets

Inclusive jets correspond to all objects that have undergone a "beam" clustering (i.e. d_{iB} recombination) in the description following Eq. (2). For nearly all hadron-collider algorithms, the "inclusive" jets above some given transverse momentum cut are the ones usually just referred to as the "jets".

To access inclusive jets, the following member function should be used

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

where `ptmin` may be omitted, then implicitly taking the value zero. Note that the order in which the inclusive jets are provided depends on the jet algorithm. To obtain a specific ordering, such as decreasing p_t, the user should perform a sort themselves, e.g. with the `sorted_by_pt(...)` function, described in section 3.1.

With a zero transverse momentum cut, the number of jets found in the event is not an infrared safe quantity (adding a single soft particle can lead to one extra soft jet). However it can still be useful to talk of all the objects returned by `inclusive_jets()` as being "jets", e.g. in the context of the estimation underlying-event and pileup densities, cf. section 8.

3.3.2 Accessing exclusive jets

There are two ways of accessing exclusive jets, 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<PseudoJet> exclusive_jets (const double dcut) const;

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

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

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

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

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

```
/// return the dmin corresponding to the recombination that went from n+1 to n jets
/// (sometimes known as d_{n,n+1}).
double exclusive_dmerge (const int n) 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 n) 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 n jets. For this reason the second function `exclusive_dmerge_max` is provided — using a d_{cut} below its return value is guaranteed to provide a final state with more than n jets, while using a larger value will return a final state with n or fewer jets.

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

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

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

3.3.3 Other functionality

Unclustered particles. Some jet algorithms (e.g. a number of the plugins in section 5) have the property that not all particles necessarily participate in the clustering. In other cases, particles may take part in the clustering, but not end up in any final inclusive jet. Two member functions are provided to obtain the list of these particles. One is

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

which returns the list of particles that never took part in the clustering. The other additionally returns objects that are the result of clustering but that never made it into a inclusive jet (i.e. into a "beam" recombination):

```
vector<PseudoJet> childless = clust_seq.childless_pseudojets();
```

A practical example where this is relevant is with plugins that perform pruning [21], since they include a condition for vetoing certain recombinations.[8]

Copying and transforming a ClusterSequence. A standard copy constructor is available for `ClusterSequence`s. Additionally it is possible to copy the clustering history of a `ClusterSequence` while modifying the momenta of the `PseudoJet`s at all (initial, intermediate, final) stages of the clustering, with the `ClusterSequence` member function

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

`FunctionOfPseudoJet<PseudoJet>` is an abstract base class whose interface provides a `PseudoJet operator()(const PseudoJet & jet)` function, i.e. a function of a `PseudoJet` that returns a `PseudoJet`

[8] To obtain the list of all initial particles that never end up in any inclusive jet, one should simply concatenate the vectors of constituents of all the childless `PseudoJet`s.

E_scheme
pt_scheme
pt2_scheme
Et_scheme
Et2_scheme
BIpt_scheme
BIpt2_scheme
WTA_pt_scheme
WTA_modp_scheme

Table 1: Members of the `RecombinationScheme` enum; the schemes prefixed by "BI" boost-invariant version of the p_t and p_t^2 schemes (as defined in section 3.4).

(cf. appendix D). As the clustering history is copied to the target `ClusterSequence`, each `PseudoJet` in the target `ClusterSequence` is set to the result of `action_on_jet(original_pseudojet)`. One use case for this is if one wishes to obtain a Lorentz-boosted copy of a `ClusterSequence`, which can be achieved as follows

```
#include "fastjet/tools/Boost.hh"
// ...
ClusterSequence original_cs(...);
ClusterSequence boosted_cs;
Boost boost(boost_4momentum);
boosted_cs.transfer_from_sequence(cs, &boost);
```

3.4 Recombination schemes

When merging particles (i.e. `PseudoJet`s) during the clustering procedure, one must specify how to combine the momenta. The simplest procedure (E-scheme) simply adds the four-vectors. This has been advocated as a standard in [3], was the main scheme in use during Run II of the Tevatron, is currently used by the LHC experiments, and is the default option in `FastJet`. Other choices are listed in table 1, and are described below.

Other schemes for *pp* collisions. Other schemes provided by earlier k_t-clustering implementations [11, 12] 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

$$p_{t,r} = p_{t,i} + p_{t,j}\,, \tag{4a}$$
$$\phi_r = (w_i\phi_i + w_j\phi_j)/(w_i + w_j)\,, \tag{4b}$$
$$y_r = (w_i y_i + w_j y_j)/(w_i + w_j)\,, \tag{4c}$$

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. As a workaround for this issue, we propose 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. They are available as `BIpt_scheme` and `BIpt2_scheme`.

A generalisation of the p_t and p_t^2 schemes is to take a weighting by $w_i = p_t^n$. An interesting limit is then $n \to \infty$, referred to as the winner-takes-all scheme (WTA), because the new rapidity and azimuth in Eq. (4) coincide with those of the harder of the two particles [64]. This gives the `WTA_pt_scheme`. The `WTA_pt_scheme` does not perform any preprocessing, but differs from the `BIpt...` schemes in that the result of a recombination acquires the mass of the higher-p_t of the two particles (instead of setting the mass to zero). This treatment ensures, for example, that recombination of a massive particle with an infinitesimally soft particle leaves the mass unchanged.

Other schemes for e^+e^- collisions. The default E-scheme is a sensible choice also for e^+e^- collisions. The only non-standard e^+e^--specific scheme currently implemented is the `WTA_modp_scheme`, in which the recombination of p_i ad p_j points in the direction and has the mass of the particle with larger 3-vector modulus ($|\vec{p}|$), and has a 3-vector modulus $|\vec{p}_r| = |\vec{p}_i| + |\vec{p}_i|$. [9]

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

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

3.5 Constituents and substructure of jets

For any `PseudoJet` that results from a clustering, the user can obtain information about its constituents, internal substructure, etc., through member functions of the `PseudoJet` class.[10]

Jet constituents. The constituents of a given `PseudoJet jet` can be obtained through

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

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

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:

```
/// Returns 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;
```

[9] It is tempting to define also a `WTA_E_scheme` in which the recombination would point in the direction of the particle with the largest energy. This is however dangerous in situations where the most energetic particle is at rest. As a consequence, this scheme is not provided in `FastJet`. A similar consideration applies for a `WTA_mt_scheme`.

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

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

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

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

```
/// If the jet has parents in the clustering, returns true and sets parent1 and parent2
/// equal to them. If it has no parents returns false and sets parent1 and parent2 to 0
bool has_parents(PseudoJet & parent1, PseudoJet & parent2) const;

/// If the jet has a child then returns true and sets the child jet otherwise returns
/// false and sets the child to 0
bool has_child(PseudoJet & child) const;

/// If this jet has a child (and so a partner), returns true and sets the partner,
/// otherwise returns false and sets the partner to 0
bool has_partner(PseudoJet & partner) const;
```

Accessibility of structural information. 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_valid_cluster_sequence() const;

// returns a (const) pointer to the parent ClusterSequence (throws if inexistent
// or no longer valid)
const ClusterSequence* validated_cluster_sequence() const;
```

There are also `has_associated_cluster_sequence()` and `associated_cluster_sequence()` member functions. The first returns true even when the cluster sequence is not valid, and the second returns a null pointer in that case. Further information is to be found in appendix C.

There are contexts in which, within some function, one 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 the `ClusterSequence` was created via a `new` operation, then one 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 (e.g. a jet, subjet or jet constituent). Note that `ClusterSequence` tends to be a large object, so this should be used with care.

3.6 Composite jets, general considerations on jet structure

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

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

The 4-momenta are added,[11] and in addition the `resonance` remembers that it came from `jet1` and `jet2`. So, for example, a call to `resonance.constituents()` will return the constituents of both `jet1` and `jet2`. It is possible to `join` 1, 2, 3 or 4 jets or a `vector` of jets. If the jets being joined had areas (section 7) then the joined jet will also have an area.

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

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

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

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

```
bool has_structure();         // true if the jet has some kind of structural info
bool has_constituents();      // true if the jet 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 the jet has jet-area information
string description();         // returns a textual description of the type
                              // of structural info associated with the jet
```

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

3.7 Version information

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

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

(defined in `fastjet/JetDefinition.hh`). In line with recommendations for other programs in high-energy physics, the user should include this information in publications and plots so as to

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

facilitate reproducibility of the jet-finding.[12] 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 this manual and, optionally, the original `FastJet` paper [10].

As of version 3.1, it is also possible to conditionally compile code based on the the `FastJet` version number, which is encoded in the `FASTJET_VERSION_NUMBER` preprocessor symbol as a single integer Mmmpp: M is the major version number, mm the minor version number and pp the patch level. Thus version 3.1.12 would be represented as 30112. Code requiring at least version 3.1.0 could be included as follows:

```
#include "fastjet/config.h"

#if FASTJET_VERSION_NUMBER >= 30100
// code that needs version 3.1.0 or higher
# else
// alternative code that works also with version 3.0.x
#endif
```

In versions prior to 3.1.0 the `FASTJET_VERSION_NUMBER` symbol is undefined and is accordingly treated by the preprocessor as if it were zero.

4 `FastJet` native jet algorithms

4.1 Longitudinally invariant k_t jet algorithm

The longitudinally invariant k_t jet algorithm [8, 9] comes in inclusive and exclusive variants. The inclusive variant (corresponding to [9], modulo small changes of notation) is formulated as follows:

1. For each pair of particles i, j work out the k_t distance[13]

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

   with $\Delta R_{ij}^2 = (y_i - y_j)^2 + (\phi_i - \phi_j)^2$, where p_{ti}, y_i and ϕ_i are the transverse momentum (with respect to the beam direction), rapidity and azimuth of particle i. R is a jet-radius parameter usually taken of order 1. For each parton i also work out the beam distance $d_{iB} = p_{ti}^2$.

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.

[12] We devote significant effort to ensuring that all versions of the `FastJet` program give identical, correct clustering results, and that any other changes from one version to the next are clearly indicated. However, as per the terms of the GNU General Public License (v2), under which `FastJet` is released, we are not able to provide a warranty that `FastJet` is free of bugs that might affect your use of the program. Accordingly it is important for physics publications to include a mention of the `FastJet` version number used, in order to help trace the impact of any bugs that might be discovered in the future.

[13] In the soft, small angle limit for i, the k_t distance is the (squared) transverse momentum of i relative to j.

The exclusive variant of the longitudinally invariant k_t jet algorithm [8] is similar, except that (a) when a d_{iB} is the smallest value, that particle is considered to become part of the beam jet (i.e. is discarded) and (b) clustering is stopped when all d_{ij} and d_{iB} are above some d_{cut}. In the exclusive mode R is commonly set to 1.

The inclusive and exclusive variants are both obtained through

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

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

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

4.2 Cambridge/Aachen jet algorithm

The *pp* Cambridge/Aachen (C/A) jet algorithm [22, 23] is provided in the form proposed in Ref. [23]. Its formulation is identical to that of the (inclusive) *pp* k_t jet algorithm, except as regards the distance measures, which are:

$$d_{ij} = \Delta R_{ij}^2/R^2\,, \tag{6a}$$

$$d_{iB} = 1\,. \tag{6b}$$

To use this algorithm, define

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

and then extract inclusive jets from the cluster sequence.

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

We note that the original formulation of the Cambridge algorithm [22] (in e^+e^-) instead makes use of an auxiliary (k_t) distance measure and 'freezes' pseudojets whose recombination would involve too large a value of the auxiliary distance measure. Details are given in section 5.4.1.

4.3 Anti-k_t jet algorithm

This algorithm, introduced and studied in [14], is defined exactly like the standard k_t algorithm, except for the distance measures which are now given by

$$d_{ij} = \min(1/p_{ti}^2, 1/p_{tj}^2)\,\Delta R_{ij}^2/R^2\,, \tag{7a}$$

$$d_{iB} = 1/p_{ti}^2\,. \tag{7b}$$

While it is 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 [14]. To use this algorithm, define

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

and then extract inclusive jets from the cluster sequence. We advise against the use of exclusive jets in the context of the anti-k_t algorithm, because of the lack of physically meaningful hierarchy in the clustering sequence.

4.4 Generalised k_t jet algorithm

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

$$d_{ij} = \min(p_{ti}^{2p}, p_{tj}^{2p})\,\Delta R_{ij}^2/R^2\,, \tag{8a}$$

$$d_{iB} = p_{ti}^{2p}\,. \tag{8b}$$

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

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

and then extract inclusive jets from the cluster sequence (or, for $p \geq 0$, also the exclusive jets).

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

`FastJet` also provides native implementations of clustering algorithms in spherical coordinates (specifically for e^+e^- collisions) along the lines of the original k_t algorithms [24], but extended following the generalised pp algorithm of [14] and section 4.4. We define the two following distances:

$$d_{ij} = \min(E_i^{2p}, E_j^{2p})\frac{(1-\cos\theta_{ij})}{(1-\cos R)}\,, \tag{9a}$$

$$d_{iB} = E_i^{2p}\,, \tag{9b}$$

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

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

and then extract inclusive jets from the cluster sequence.

For values of $R > \pi$, `FastJet` replaces the factor $(1-\cos R)$ in the denominator of eq. (9a) with $(3+\cos R)$. With this choice (as long as $R < 3\pi$), the only time a d_{iB} will be relevant is when there is just a single particle in the event. The `inclusive_jets(...)` will then always return a single jet consisting of all the particles in the event. In such a context it is only the `exclusive_jets(...)` call that provides non-trivial information.

4.6 k_t algorithm for e^+e^- collisions

The e^+e^- k_t algorithm [24], often referred to also as the Durham algorithm, has a single distance:

$$d_{ij} = 2\min(E_i^2, E_j^2)(1 - \cos\theta_{ij})\,. \tag{10}$$

Note the difference in normalisation between the d_{ij} in eqs. (9) and (10), and the fact that in neither case have we normalised to the total energy Q in the event, contrary to the convention adopted originally in [24] (where the distance measure was called y_{ij}). To use the e^+e^- k_t algorithm, define

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

and then extract exclusive jets from the cluster sequence.

Note that the `ee_genkt_algorithm` with $\pi < R < 3\pi$ and $p = 1$ gives a clustering sequence that is identical to that of the `ee_kt_algorithm`. The normalisation of the d_{ij}'s will however be different.

5 Plugin jet algorithms

It can be useful to have a common interface for a range of jet algorithms beyond the native (k_t, anti-k_t and Cambridge/Aachen) algorithms, notably for the many cone algorithms that are in existence. It can also be useful to be able to use `FastJet` features such as 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[14] to be used within the same framework.

Generic plugin use is described in the next subsection. The plugins distributed with `FastJet` are described afterwards in sections 5.2–5.4. They are all accessible within the `fastjet` namespace and their code is to be found in `FastJet`'s `plugins/` directory. New user-defined plugins can also be implemented, as described in section E.2. Some third-party plugins are linked to from the tools page at `http://fastjet.fr/`.

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

5.1 Generic plugin use

Plugins are classes derived from the abstract base class `fastjet::JetDefinition::Plugin`. A `JetDefinition` can be constructed by providing a pointer to a `JetDefinition::Plugin`; the resulting `JetDefinition` object can then be used identically to the normal `JetDefinition` objects used in the previous sections. We illustrate this with an example based on the SISCone plugin:

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

// allocate a new plugin for SISCone (for other plugins, simply
// replace the appropriate parameters and plugin name)
double cone_radius = 0.7;
```

[14] Except those for which one particle may be assigned to more than one jet, e.g. algorithms such as ARCLUS [25], which performs $3 \to 2$ clustering.

```
double overlap_threshold = 0.75;
JetDefinition::Plugin * plugin = new SISConePlugin(cone_radius, overlap_threshold);

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

// run the jet algorithm and extract the jets
ClusterSequence clust_seq(particles, jet_def);
vector<PseudoJet> inclusive_jets = clust_seq.inclusive_jets();

// then analyse the jets as for native fastjet algorithms
...

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

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

5.2 SISCone Plugin

SISCone provides infrared and collinear-safe implementations of the two most common kinds of cone algorithm. Cone algorithm execution generally involves two phases. The first is the search for stable cones (SC). Then, because a given particle can appear in more than one stable cone, there are two options for the second phase: the standard approach is to apply a 'split–merge' (see e.g. Ref. [3]) procedure, which ensures that no particle ends up in more than one jet. Alternatively, one can identify the highest-p_t stable cone, call it a jet, remove all particles that it contained and then repeat the stable-cone search and removal over and over until no particles remain (we call this progressive removal).

The stable cones are identified using an $\mathcal{O}\left(N^2 \ln N\right)$ seedless approach [26]. This (and some care in the 'split–merge' procedure, if used) ensures that the jets it produces are insensitive to additional soft particles and collinear splittings, i.e. the algorithm is infrared and collinear safe.

By default SISCone runs with a split–merge step. The implementation with progressive removal was made public in version 3.0 of SISCone, first released with version 3.1 of `FastJet`.

5.2.1 Default mode: SISCone with a split–merge step (SISCone-SM)

The original implementation of SISCone [26] corresponded to a stable-cone jet algorithm with a split–merge step (SC-SM in the notation of [5]). This remains SISCone's default and, unless explicitly mentioned otherwise, it corresponds to the version of SISCone used in the literature.

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

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

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

A cone centred 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 [3] split–merge procedure. 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 [27], SISCone can carry out multiple stable-cone search passes (as advocated in [28]): 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 ∞.

Concern had at some point been expressed that an excessive number of stable cones might complicate cone jets in events with high noise [3], and in particular lead to large "monster" jets. 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), so as to limit this issue. A large value of the split–merge overlap threshold, e.g. 0.75, also helps to disfavour the production of these monster jets through repeated merge operations.

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

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

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

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 [3]) 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 was given in section 5.1. As can be seen there, SISCone jets are to be obtained by requesting inclusive jets from the cluster sequence. Note that it makes no sense to ask for exclusive jets from a SISCone based `ClusterSequence`.

Some cone algorithms provide information beyond that simply about the jets. Such additional information, where available, can be accessed with the help of the `ClusterSequence::extras()` function. In the case of SISCone, one can access that information as follows:

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

To determine the pass at which a given jet was found, one then uses[15]

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

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

```
const vector<PseudoJet> & stable_cones = extras->stable_cones();
```

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

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);
```

For this to work, `plugin` must explicitly be of type `SISConePlugin *` rather than `JetDefinition::Plugin *`.

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 $\mathcal{O}\left(N^2 \ln N\right)$ time to find jets, and the memory use is $\mathcal{O}\left(N^2\right)$; 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.[16]

Note that the underlying implementation of SISCone is optimised for large N. An alternative implementation that is faster for $N \lesssim 10$ was presented in [29].

5.2.2 SISCone with progressive removal (SISCone-PR)

After creating an instance of `SISConePlugin`, the progressive-removal option can be enabled by calling

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

The value of the split–merge overlap threshold is then ignored and caching is not available. The `SplitMergeScale` argument in the constructor, instead of being interpreted as the choice of scale to

[15] In versions of `FastJet` prior to 3.0.1, a jet's user index indicated the pass at which it had been found. The value was however incorrectly set for single-particle jets. The current choice is to leave the user index unchanged from its default.

[16] In area determinations, the ghost particles are not entirely random, but distributed close to a grid pattern, all with similar transverse momenta. Run times and memory usage are then, 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` (cf. section 7), 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 many situations, is to use a passive area, since this is relatively fast to calculate with SISCone.

use in the split-merge step, is instead interpreted as the scale used to order the stable cones (the default is `SISConePlugin::SM_pttilde`, i.e. the scalar sum of p_t's of the constituents) and it is the hardest stable cone according to that choice that is removed. The stable cone finding is then repeated on the remaining set of particles, the new hardest cone is removed, and so forth. As for other progressive-removal cone algorithms, SISCone-PR guarantees that the first jet that is found has area πR^2.[17]

There can be interest in trying out other choices of scale variable for the ordering of stable cones. This can be achieved by calling

```
plugin->set_user_scale(&user_scale);
```

where `user_scale` is an instance of any class that derives from the `SISConePlugin::UserScaleBase` class (actually defined in `SISConeBasePlugin`).

In discussing the speed of SISCone with progressive removal, it is useful to distinguish N, the total number of particles in the event from n, the number inside a typical circle of radius R. SISCone with split–merge actually takes a time $Nn \ln n$. SISCone with progressive removal, in its current implementation, repeats the full stable cone finding $\mathcal{O}(N/n)$ times. Thus it takes a time $\sim N^2 \ln n$. This could be reduced by repeating the stable-cone finding only for particles in the neighbourhood of the last stable-cone that was removed: that more limited stable-cone finding should take a time $n^2 \ln n$ at each iteration, so that the total time would then be $Nn \ln n$, albeit with a larger coefficient than for the split–merge variant. We leave such an improvement to future work, should there be demand for it.

5.3 Other plugins for hadron colliders

Most of the algorithms listed below are cone algorithms. They are usually either infrared (IR) or collinear unsafe. The details are indicated for each algorithm following the notation of Ref. [5]: 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 of IR and/or collinear unsafe jet algorithms. Interfaces to these algorithms have been provided mainly for legacy comparison purposes.

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

5.3.1 CDF Midpoint

This is one of the two cone algorithms used by CDF in Run II of the Tevatron, based on [3]. 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,
```

[17] The `inclusive_jets()` function returns the jets in the order in which they were found. Each jet's `extras->pass(jet)` value also corresponds to the pass of stable-cone finding at which it was obtained.

```
                    double cone_area_fraction = 1.0);
```

The overlap threshold (f) used by CDF is usually 0.5, the seed threshold is 1 GeV. With a cone area fraction $\alpha < 1$, the search for stable cones is performed with a radius that is $R \times \sqrt{\alpha}$, i.e. it becomes the searchcone algorithm of [27]. CDF has used both $\alpha = 0.25$ and $\alpha = 1.0$. It is our understanding that the particular choice of α is not always clearly documented in the corresponding publications.

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

The original underlying code for this algorithm was provided on a webpage belonging to Joey Huston [30] (with minor modifications to ensure reasonable run times with optimising compilers for 32-bit Intel processors — these modifications do not affect the final jets).

This algorithm is IR_{3+1} unsafe in the limit of zero seed threshold [26] (with $\alpha \neq 1$ it becomes IR_{2+1} unsafe [28]). With a non-zero seed threshold (and no preclustering into calorimeter towers) it is collinear unsafe. It is to be deprecated for new experimental or theoretical analyses.

5.3.2 CDF JetClu

JetClu is the other cone algorithm used by CDF during Run II, as well as their main algorithm during Run I [31]. It is an iterative cone with split-merge and optional "ratcheting" if `iratch == 1` (particles that appear in one iteration of a cone are retained in future iterations). It can be obtained as follows:

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

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 original underlying code for this algorithm was provided on a webpage belonging to Joey Huston [30].

This algorithm is IR_{2+1} unsafe (for zero seed threshold; 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 together into calorimeter towers, with CDF-type geometry, before running the jet algorithm.

5.3.3 DØ Run I cone

This is the main algorithm used by DØ in Run I of the Tevatron [32]. It is an iterative cone algorithm with a split-merge step. 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 recombination scheme used to determine the jet momenta once each jet's constituents have been identified. In the pre-1996 version, a hybrid between an E-like scheme and an E_t scheme recombination is used, while in the post-1996 version it is just the E_t scheme [32].

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, corresponding to a cut of 4 GeV on cones. The publication that describes this algorithm [32] mentions the use of a 1 GeV seed threshold applied to preclustered calorimeter towers in order to obtain the seeds for the stable cone finding. Such a threshold and the pre-clustering appear not to be included in the code distributed with `FastJet`.

The underlying code for this algorithm was provided by Lars Sonnenschein. Permission to redistribute this code with `FastJet` has been granted by the DØ collaboration under the terms of the GPL license.

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. It is to be deprecated for new experimental or theoretical analyse

5.3.4 DØ Run II cone

This is the main algorithm used by DØ in Run II of the Tevatron. It is a midpoint type iterative cone with split-merge step. The DØ collaboration usually refers to Ref. [3] when introducing the algorithm in its articles. That generic Tevatron document does not reflect all details of the actual DØ algorithm, and for a complementary description the reader is referred to Ref. [33].

```
#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. There is a cut on the minimum E_t of the cones during iteration, which by default is half of `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).

As concerns seed thresholds and preclustering, DØ describes them as being applied to calorimeter towers in data and in Monte Carlo studies that include detector simulation [33]. However, for NLO calculations and Monte Carlo studies based on stable particles, no seed threshold is applied. The code distributed with `FastJet` does not allow for seed thresholds.

The underlying code for this algorithm was provided by Lars Sonnenschein. Permission to redistribute this code with `FastJet` has been granted by the DØ collaboration under the terms of the GPL license.

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

5.3.5 ATLAS iterative cone

This iterative cone algorithm, with a split-merge step, was 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 an early version of SpartyJet [16] (which itself was distributed under the GPL license). Since version 3.0 of `FastJet` it is a slightly modified version that we distribute, where an internal `sort` function has been replaced with a `stable_sort`, to ensure reproducibility of results across compilers and architectures (results are unchanged when the results of the sort are unambiguous).

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

5.3.6 CMS iterative cone

This iterative cone algorithm with progressive removal was 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. Permission to redistribute the resulting code with `FastJet` has been granted by CMS under the terms of the GPL license. The 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 [14]. It is to be deprecated for new experimental or theoretical analyses.

5.3.7 PxCone

The PxCone algorithm 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);
```

It includes a threshold on the minimum transverse energy for a cone (jet) 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 [34].

The underlying code for this algorithm was written by Luis del Pozo and Michael Seymour with input also from David Ward [35] and they have granted permission for their code to be distributed with `FastJet` under the terms of the GPL license.

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

5.3.8 TrackJet

This algorithm has been used at the Tevatron for identifying jets from charged-particle tracks in underlying-event studies [36].

```
#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 (GPL) Rivet implementation, written by Andy Buckley with input from Manuel Bahr and Rick Field. Since version 3.0 of `FastJet` it is a slightly modified version that we distribute, where an internal `sort` function has been replaced with a `stable_sort`, to ensure reproducibility of results across compilers and architectures (results are unchanged when the results of the sort are unambiguous, which is the usual case).

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

5.3.9 GridJet

GridJet allows you to define a grid and then cluster particles such that all particles in a common grid cell combine to form a jet. Its main interest is in providing fast clustering for high multiplicities (the clustering time scales linearly with the number of particles). The jets that it forms are not always physically meaningful: for example, a genuine physical jet may lie at the corner of 4 grid cells and so

be split up somewhat arbitrarily into 4 pieces. It is therefore not intended to be used for standard jet finding. However for some purposes (such as background estimation) this drawback is offset by the greater uniformity of the area of the jets. Its interface is as follows

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

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

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

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

5.4 Plugins for e^+e^- collisions

5.4.1 Cambridge algorithm

The original e^+e^- Cambridge [22] algorithm is provided as a plugin:

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

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

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

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

5.4.2 Jade algorithm

The JADE algorithm [37, 38], 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 [37]. To modify the recombination scheme, the user may first construct the jet definition and then use either of

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

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

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

5.4.3 Spherical SISCone algorithm

The spherical SISCone algorithm is an extension [39] to spherical coordinates of the hadron-collider SISCone algorithm [26].

```
#include "fastjet/SISConeSphericalPlugin.hh"
// ...
SISConeSphericalPlugin(double R,
                       double overlap_threshold
                       double protojet_Emin = 0.0,
                       bool   caching = false,
                       SISConeSphericalPlugin::SplitMergeScale
                         split_merge_scale = SISConeSphericalPlugin::SM_Etilde,
                       double split_merge_stopping_scale = 0.0);
```

The functionality follows directly that of SISConePlugin, including options for using a split–merge step (the default) or progressive removal.

Note that the underlying implementation of spherical SISCone is optimised for large N. An alternative implementation that is faster for $N \lesssim 10$ was presented in [29]. That reference also contains a nice description of the algorithm.

6 Selectors

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

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

6.1 Essential usage

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

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

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

Here, Selector is a class, while SelectorAbsRapMax and SelectorPtMin are functions that return an instance of the Selector class containing the internal information needed to carry out the selection. Selector::operator(const vector<PseudoJet> & jets) takes the jets given as input and returns a

vector containing those that pass the selection cuts. The logical operations `&&`, `||` and `!` enable different selectors to be combined.

Nearly all selectors, like those above, apply jet by jet (the function `Selector::applies_jet_by_jet()` returns `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 jets with largest transverse momenta. If there are fewer than n jets, then all jets pass.

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

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

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

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

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

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

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

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

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

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

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

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

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

A user can obtain a string describing a given `Selector`'s action by calling its `description()` member function. This behaves sensibly also for compound selectors.

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

6.1.1 Other information about selectors

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

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

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

6.2 Available selectors

6.2.1 Absolute kinematical cuts

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

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

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

```
Min, Max, Range
```

A couple of examples are

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

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

6.2.2 Relative kinematical cuts

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

The available selectors of this kind are:

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

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

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

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

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

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

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

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

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

6.2.3 Other selectors

The following selectors are also available:

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

7 Jet areas

Jet areas provide a measure of the surface in the y-ϕ plane over which a jet extends, or, equivalently, a measure of a jet's susceptibility to soft contamination.

Since a jet is made up of only a finite number of particles, one needs a specific definition in order to make its area an unambiguous concept. Three definitions of area have been proposed in [17] and 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. Active areas give a measure of a jet's sensitivity to diffuse background noise.

- *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. Passive areas give a measure of a jet's sensitivity to point-like background noise.

- 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 in general for other algorithms) the Voronoi area of a jet coincides with its passive area.

In the limit of very densely populated events, all area definitions lead to the same jet-area results [17].[18]

The area of a jet 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; 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.

To access jet areas, the user is exposed to two main classes:

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

with input particles, a jet definition and an area definition being supplied to a `ClusterSequenceArea` in order to obtain jets with area information. Typical usage would be as follows:

```
#include "fastjet/ClusterSequenceArea.hh"
// ...
double ghost_maxrap = 5.0; // e.g. if particles go up to y=5
AreaDefinition area_def(active_area, GhostedAreaSpec(ghost_maxrap));
ClusterSequenceArea clust_seq(input_particles, jet_def, area_def);
vector<PseudoJet> jets = sorted_by_pt(clust_seq.inclusive_jets());
double area_hardest_jet = jets[0].area();
```

Details are to be found below and an example program is given as `example/06-area.cc`.

When jet areas are to be used to establish the level of a diffuse noise that might be present in the event (e.g. from underlying event particles or pileup), and maybe subtract it from jets, further classes such as `fastjet::JetMedianBackgroundEstimator` and `fastjet::Subtractor` are useful. This topic is discussed in Section 8 and an example program is given in `example/07-subtraction.cc`.

7.1 AreaDefinition

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

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

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

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

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

[18] This can be useful when one area is particularly expensive to calculate: for example active areas for SISCone tend to be memory and CPU intensive; however, for dense events, they can be adequately replaced with passive areas, which, for SISCone, are computationally more straightforward.

Information about the current `AreaDefinition` can be retrieved with the help of `description()`, `area_type()`, `ghost_spec()` and `voronoi_spec()` member functions.

7.1.1 Ghosted Areas (active and passive)

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

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

The two active variants give identical results for the areas of hard jets. The second one explicitly includes the ghosts when the user requests the constituents of a jet and also leads to the presence of "pure ghost" jets. 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.[19]

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.[20] This is done via the class `GhostedAreaSpec` whose constructor is

```
GhostedAreaSpec(double ghost_maxrap,
                int    repeat        = 1,   double ghost_area    = 0.01,
                double grid_scatter  = 1.0, double pt_scatter    = 0.1,
                double mean_ghost_pt = 1e-100);
```

The ghosts are distributed on a uniform grid in y and ϕ, with small random fluctuations to avoid clustering degeneracies.

The `ghost_maxrap` variable defines the maximum rapidity up to which ghosts are generated. If one places ghosts well beyond the particle acceptance (at least R beyond it), then jet areas also stretch beyond the acceptance, giving a measure of the jet's full extent in rapidity and azimuth. If ghosts are placed only up to the particle acceptance, then the jet areas are clipped at that acceptance and correspond more closely to a measure of the jet's susceptibility to contamination from accepted soft particles. This is relevant in particular for jets within a distance R of the particle acceptance boundary. The two choices are illustrated in fig. 1. To define more complicated ghost acceptances it is possible to replace `ghost_maxrap` with a `Selector`, which must be purely geometrical and have finite rapidity extent.

The `ghost_area` parameter in the `GhostedAreaSpec` constructor 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 adequate for most applications. Smaller ghost areas are beneficial mainly for high-precision determinations of areas of jets with small R.

[19] 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 [17] (though some approximations are made for 4-vector areas). For other algorithms it defaults back to the `one_ghost_passive_area` approach.

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

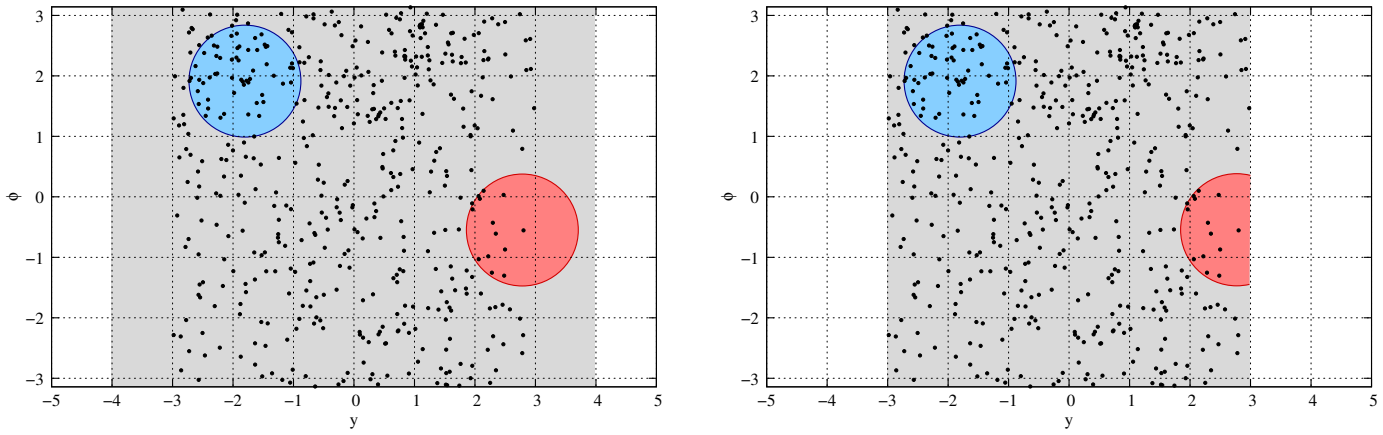

Figure 1: Two choices for ghost placement. The grey area in each plot indicates the region containing ghosts, while the dots are particles, which here are accepted up to $|y| < 3$. The circular regions indicate the areas that will be found for two particular jets. In the left-hand case, with ghosts that extend well beyond the acceptance for particles, jet areas are unaffected by the particle acceptance; in the right-hand case, with ghosts placed only up to the acceptance limit, jet areas are clipped at the edge of the acceptance.

By default, one set of ghosts is generated for each event that is clustered. The small random fluctuations in ghost positions and p_t's, introduced to break clustering degeneracies, mean that for repeated clustering of the same event a given hard jet's area will be different after each clustering. This is especially true for sparse events, where a jet's particle content fails to accurately delineate the boundaries of the jet. For the `active_area` choice (and certain passive areas), specifying `repeat` > 1 causes `FastJet` to directly cluster the same hard event with multiple ghost sets. This results in a pseudo-Monte Carlo type evaluation of the jet areas. A statistical uncertainty on the area is available for each jet, through the `jet.area_error()` call. It is calculated as the standard deviation of areas obtained for that jet, divided by $\sqrt{\texttt{repeat} - 1}$. While there are situations in which this facility is useful, for most applications of jet areas it is sufficient to use `repeat` $= 1$.[21]

After initialisation, the parameters can be modified and retrieved respectively with calls such as `set_ghost_area(...)` and `ghost_maxrap()` (similarly for the other parameters[22]). A textual description of the `GhostedAreaSpec` can be obtained, as usual, with the `description()` member function.

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.

[21] Several parameters are available to control the properties and randomness of the ghosts: each ghost's position differs from an exact grid vertex by a random amount distributed uniformly in the range $\pm\frac{1}{2}$`grid_scatter` times the grid spacing in both the rapidity and azimuth directions. Each ghost's p_t is distributed randomly in the range $(1 \pm \frac{1}{2}$`pt_scatter`$)$ times `mean_ghost_pt`. For nearly all applications, it makes sense to use the default values. Facilities are also available to set and retrieve the seeds for the random-number generator, notably through the `set_random_status(...)` and `get_random_status(...)` members of `GhostedAreaSpec`.

[22] In versions of `FastJet` prior to 3.0.1, the names `mean_ghost_kt` and `kt_scatter` should be used rather than `mean_ghost_pt` and `pt_scatter`. The former names will be maintained for the foreseeable future.

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

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

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

The second constructor allows one to modify (by a multiplicative factor `effective_Rfact`) the radius of the circle which is intersected with the Voronoi cells. With `effective_Rfact` $= 1$, for the k_t algorithm, the Voronoi area is equivalent to the passive area. Information about the specification in use is returned by `effective_Rfact()` and `description()` member functions.

The Voronoi areas are calculated with the help of Fortune's $(N \ln N)$ Voronoi diagram generator for planar static point sets [40].

One use for the Voronoi area is in background determination with the k_t algorithm (see below, section 8): with the choice `effective_Rfact` $\simeq 0.9$ it provides an acceptable approximation to the k_t algorithm's active area and is often significantly faster to compute than the active area. Note that it is not currently possible to clip Voronoi areas with a given particle acceptance. As a result, given particles up to $|y| < y_{\max}$, only jets with $|y| \lesssim y_{\max} - R$ will have areas that reflect the jets' sensitivity to accepted particle contamination. It is only these jets that should then be used for background determinations.

7.2 `ClusterSequenceArea`

This is the class[23] used for producing a cluster sequence that also calculates jet areas. Its constructor is

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

and the class includes the methods

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

/// Returns an estimate of the area contained within the selector that is free of jets.
/// The selector needs to have a finite area and be applicable jet by jet.
/// The function returns 0 if active_area_explicit_ghosts was used.
virtual double empty_area(const Selector & selector) const;
```

As long as an instance of this class is in scope, a user can access information about the area of its jets using the following methods of `PseudoJet`:

[23] `ClusterSequenceArea` is derived from `ClusterSequenceAreaBase` (itself derived from `ClusterSequence`) and makes use of one among `ClusterSequenceActiveAreaExplicitGhosts`, `ClusterSequenceActiveArea`, `ClusterSequencePassiveArea`, `ClusterSequence1GhostPassiveArea` or `ClusterSequenceVoronoiArea` (all of them in the `fastjet` namespace of course), according to the choice made with `AreaDefinition`. The user can also use these classes directly. `ClusterSequenceActiveAreaExplicitGhosts` is particularly useful in that it allows the user to specify their own set of ghost particles. This is exploited to provide area support in a number of the transformers of section 9.

```
/// Returns the scalar area associated with the given jet
double area = jet.area();

/// Returns the error (uncertainty) associated with the determination of the
/// scalar area of the jet; gives zero when the repeat=1 and for passive/Voronoi areas
double area_error = jet.area_error();

/// Returns 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 1
/// when y,φ define a direction inside the jet boundary and 0 otherwise.
PseudoJet area_4vector = jet.area_4vector();

/// When using active_area_explicit_ghosts, returns true for jets made
/// exclusively of ghosts and for ghost constituents.
bool is_pure_ghost = jet.is_pure_ghost();
```

In the limit of small-R jets, the transverse component of the 4-vector area is close to the scalar area; for moderate R values, the transverse component of the 4-vector area is smaller by a factor that reads $1 - R^2/8 + R^4/192 + \mathcal{O}(R^6)$ for the case of circular jets [41]. Note that 4-vector areas are not currently computed exactly for Voronoi areas of jets in sparse events, insofar as the 4-vector area of each particle's Voronoi cell is approximated as massless and pointing in the direction of the particle.

8 Background estimation and subtraction

Events with hard jets are often accompanied by a more diffuse "background" of relatively soft particles, for example from the underlying event (in pp or PbPb collisions) or from pileup (in pp collisions). For many physical applications, it is useful to be able to estimate characteristics of the background on an event-by-event basis, for example the p_t per unit area (ρ), or fluctuations from point to point (σ). One use of this information is to correct the hard jets for the soft contamination, as discussed below in section 8.1.2.

One of the issues in characterising the background is that it is difficult to introduce a robust criterion to distinguish "background" jets from hard jets. The main method that is available in `FastJet` involves the determination of the distribution of p_t/A for the jets in a given event (or region of the event) and then taking the median of the distribution as an estimate of ρ, as proposed in [18] and studied in detail also in [42, 43]. This is largely insensitive to the presence of a handful of hard jets, and avoids any need for introducing a p_t scale to distinguish hard and background jets.

The original form of this method used the k_t or Cambridge/Aachen jet algorithms to find the jets. These algorithms have the advantage that the resulting jets tend to have reasonably uniform areas[24] In the meantime a variant of the approach that has emerged is to cluster the particles into rectangular grid cells in y and ϕ and determine their median p_t/A. This has the advantage of simplicity and much

[24]Whereas anti-k_t and SISCone suffer from jets with near zero areas or, for SISCone, sometimes huge, "monster" jets, biasing the ρ determination. They are therefore not recommended.

greater speed. One may worry that a hard jet will sometimes lie at a corner of multiple grid cells, inducing larger biases in the median than with a normal jet finder jets, however we have found this not to be an issue in practice [43].

8.1 General Usage

8.1.1 Background estimation

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

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

where the selector is used to indicate which jets are used for background estimation (for simple use cases, one just specifies a rapidity range, e.g. `SelectorAbsRapMax(4.5)` to use all jets with $|y| < 4.5$), together with a jet definition and an area definition. We have found that the k_t or Cambridge/Aachen jet algorithms with $R = 0.4 - 0.6$ generally provide adequate background estimates, with the lower range of R values to be preferred if the events are likely to be busy [42, 43]. An active area with explicit ghosts is generally recommended.[25]

For the grid based method one creates a `fastjet::GridMedianBackgroundEstimator`,

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

We have found grid spacings in the range $0.5 - 0.7$ to be adequate [43], with lower values preferred for events that are likely to have high multiplicities.

Both of the above background estimators derive from a `fastjet::BackgroundEstimatorBase` class and the remaining functionality is common to both. In particular, for each event, one tells the background estimator about the event particles,

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

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

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

// an estimate of the fluctuations in the p_t density per unit sqrt(A),
```

[25] With the k_t algorithm one may also use a Voronoi area (`effective_Rfact = 0.9` is recommended), which has the advantage of being deterministic and faster than ghosted areas. In this case however one must use a selector that is geometrical and selects only jets well within the range of event particles. E.g. if particles are present up to $|y| = y_{\max}$ one should only use jets with $|y| \lesssim y_{\max} - R$. When using ghosts, the selector can instead go right up to the edge of the acceptance, if the ghosts also only go right up to the edge, as in the right-hand plot of fig. 1.

```
// 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 < ρ − σ/√⟨A⟩
sigma = bge.sigma();
```

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

8.1.2 Background subtraction

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

$$p_{t,jet}^{sub} = p_{t,jet}^{raw} - \rho A_{jet} \tag{11}$$

or its 4-vector version

$$p_{\mu,jet}^{sub} = p_{\mu,jet}^{raw} - \rho A_{\mu,jet}\,, \tag{12}$$

as first proposed in [18].

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

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

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

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

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

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

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

8.2 Positional dependence of background

The background density in pp and heavy-ion collisions usually has some non-negligible dependence on rapidity (and sometimes azimuth). This dependence is not accounted for in the standard estimate of ρ based on all jets or grid cells from (say) $|y| < 4.5$. Two techniques are described below to help alleviate this problem. In each case the properties of the background are to be obtained through the methods (available for both `JetMedianBackgroundEstimator` and `GridMedianBackgroundEstimator`)

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

8.2.1 Local estimation (jet based)

The first technique, "local estimation", available for now only in the case of the jet-based estimator, involves the use of a more local range for the determination of ρ, with the help of a `Selector` that is able to take a reference jet, e.g. `SelectorStrip(`Δy`)`, a strip of half-width Δy (which might be of order 1) centred on whichever jet is set as its reference. With this kind of selector, when the user calls either `rho(jet)` or `sigma(jet)` a `selector.set_reference(jet)` call is made to centre the selector on the specified jet. Then only the jets in the event that pass the cut specified by this newly positioned `selector` are used to estimate ρ or σ.[26] 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.[27] This is however not always very effective, because one may not know how many hard jets to remove.

8.2.2 Estimation in regions (grid based)

The grid-based estimator does not currently provide for local estimates, in the sense that the set of tiles used for a given instance of the grid-based estimator is always fixed. However, as of `FastJet` 3.1, it is possible to obtain relatively fine control over which fixed set of tiles a grid-based estimator uses. This is done with the help of the `RectangularGrid` class

```
RectangularGrid grid(rap_min, rap_max, rap_spacing, phi_spacing, selector);
GridMedianBackgroundEstimator bge(grid);
```

A given grid tile will be used only if a massless `PseudoJet` placed at the centre of the tile passes the `selector`. So, for example, to obtain an estimate for ρ based on the activity in the two forward regions of a detector, one might set `rap_min` and `rap_min` to cover the whole detector and then supply a `SelectorAbsRapRange(rap_central_max, rap_max)` to select just the two forward regions.

8.2.3 Rescaling method

A second technique to account for positional dependence of the background is "rescaling". First one parametrises the average shape of the rapidity dependence from some number of pileup events. Then for subsequent event-by-event background determinations, one carries out a global ρ determination

[26] 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 also in effect, cf. section 8.2.3).

[27] If you use non-geometric selectors such as this 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.

and then applies the previously determined average rescaling function to that global determination to obtain an estimate for ρ in the neighbourhood of a specific jet.

The rescaling approach approach is available for both grid and jet-based methods. To encode the background shape, one defines an object such as

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

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

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

Subsequent calls to `rho()` will return the median of the distribution $p_t/A/\texttt{rescaling}(y)$ (rather than p_t/A). Any calls to `rho(jet)` and `sigma(jet)` will include 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()` (which are in any case less meaningful when a rapidity dependence is being assumed for the background).

In ongoing studies [43], we have found that despite its use of an average background shape, the rescaling method generally performs comparably to local estimation in terms of its residual p_t dispersion after subtraction. Additionally, it has the advantage of reduced sensitivity to biases in events with high multiplicities of hard jets.

8.3 Handling masses

There are several subtleties in handling masses in subtraction. The first is related to the fact that hadrons have masses. In some contexts those masses are ignored and set to zero, e.g. because they are experimentally unconstrained: detectors do not systematically give information on the mass for every particle. However, if particle masses are kept non-zero, then Eq. (12) must be extended [44] to read

$$p^\mu_{\text{jet,sub}} = p^\mu_{\text{jet}} - [\rho A^x_{\text{jet}},\, \rho A^y_{\text{jet}},\, (\rho+\rho_m)A^z_{\text{jet}},\, (\rho+\rho_m)A^E_{\text{jet}}]\,, \tag{13}$$

where ρ_m accounts the massive component of the energy flow. It can be approximately determined as the median across patches of a quantity $m_{\delta,\text{patch}}/A_{\text{patch}}$, where

$$m_{\delta,\text{patch}} = \sum_{i\in\text{patch}} \left(\sqrt{m_i^2+p_{t,i}^2} - p_{ti}\right)\,. \tag{14}$$

By default, as of `FastJet` 3.1, the grid and jet-median background estimators automatically determine ρ_m. It is returned from a call to `rho_m()`, with fluctuations accessible through `sigma_m()`. The determination of ρ_m involves a small speed penalty and can be disabled with a call to `set_compute_rho_m(false)` for either of the background estimators.

To avoid changes in results relative to version 3.0, by default `FastJet` 3.1 does not use ρ_m in the `Subtractor`, i.e. it uses Eq. (12). However, for a given subtractor, a call to `set_use_rho_m(true)`, will cause it to instead use Eq. (13) for all subsequent subtractions. We *strongly recommend* switching this

on if your input particles have masses, and in future versions of FastJet we may change the default so that it is automatically switched on.[28] An alternative is to make the input particles massless.

A second issue, relevant both for Eqs. (12) and (13), is that sometimes the resulting squared jet mass is negative.[29] This is obviously unphysical. By default the 4-vector returned by the subtractor is left in that unphysical state, so that the user can decide what to do with it. For most applications a sensible behaviour is to adjust the 4-vector so as to maintain its p_t and azimuth, while setting the mass to zero. This behaviour can be enabled for a given subtractor by a call to its `set_safe_mass(true)` function (available since v.3.1). In this case the rapidity is then taken to be that of the original unsubtracted jet. In future versions of FastJet, the default behaviour may be changed so that "safe" subtraction is automatically enabled.

A general note with regards to jet masses is that a number of techniques have been proposed as alternatives to 4-vector subtraction for jet masses. They include Jet Cleansing [45] (which requires charged-track information; see also the discussion in Ref. [46]), Constituent Subtraction [47], the SoftKiller [48] method and PUPPI [49]. Some of these can give significant improvements in the resolution of the subtraction for the jet masses, though potentially at the expense of some bias. SoftKiller and PUPPI appear to give improved resolution also for the jet p_t (again at the potential expense of modest biases). Implementations of most of these techniques are, or will soon become available in `fjcontrib`. There one will also find the `GenericSubtractor` contrib, with code for area-based subtraction of jet shapes and other generic observables as discussed in Refs. [44, 50].

8.4 Other facilities

The `JetMedianBackgroundEstimator` has a number of enquiry functions to access information used internally within the median ρ and σ determination.

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

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

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

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

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

[28] It is also possible to construct a `Subtractor` with explicit ρ and ρ_m values, `Subtractor subtractor(rho, rho_m)`; if this is done, then ρ_m use *is enabled* by default.

[29] Recall that if $m^2 < 0$, `m()` returns $-\sqrt{-m^2}$, to avoid having to introduce complex numbers just for this special case.

range, with the number of empty jets then being calculated based on the average jet area for ghost jets ($0.55\pi R^2$ [17]). All four functions above return a result corresponding to the last call to `rho` or `sigma` (as long as the particles, cluster sequence or selector have not changed in the meantime).

The `Subtractor` has an option

```
void set_known_selectors(const Selector & sel_known_vertex,
                         const Selector & sel_leading_vertex);
```

The idea here is that there are contexts where it is possible, for some of a jet's constituents, to identify which vertex they come from. In that case it is possible to provide a user-defined a selector (section 6) that indicates whether a particle comes from an identified vertex or not and a second user-defined selector that indicates whether that vertex was the leading vertex. The 4-momentum from the non-leading vertex is then discarded, that from the leading vertex is kept, and subtraction is applied to component that is not from identified vertices. It follows that ρ must correspond only to the momentum flow from non-identified vertices. With this setup, the jet p_t is bounded to be at least equal to that from the leading vertex, as is the mass if the "safe" subtraction option is enabled.

8.5 Alternative workflows

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

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

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

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

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

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

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

8.6 Recommendations

While the basic idea of pileup subtraction is rather simple, in practice a few details are relevant to obtaining the best possible performance. Here we summarise some useful recommendations:

1. The `GridMedianBackgroundEstimator` is significantly faster than the `JetMedianBackgroundEstimator` and performs equally well in nearly all cases.

2. Pileup usually has non-negligible rapidity dependence (and in the case of heavy-ion collisions, the underlying event also has azimuthal dependence). It is often important to account for this dependence, which can be done either with the rescaling method (section 8.2.3) or via a local background estimator (section 8.2.1) or the use of regions (section 8.2.2). For their own work in pileup subtraction the `FastJet` authors tend to prefer the rescaling method, with local estimation also a useful option e.g. in the case of heavy-ion collisions.

3. If you are interested in jet masses, and wish to use non-zero input particle masses, make sure you use the ρ_m component in Eq. (13) by calling your subtractor's `set_use_rho_m()` method (section 8.3). You should also pay attention to what happens with negative squared masses, and consider calling the subtractor's `set_safe_mass()` option. For reasons of backwards compatibility, both of these options are disabled by default; in future versions of `FastJet`, this may change.

4. For jet masses and shapes, there are various other techniques that may be of use, see e.g. Refs. [50, 44, 45, 46, 47, 48, 49].

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

Performing post-clustering actions on jets has in recent years become quite widespread: for example, numerous techniques have been introduced to tag boosted hadronically decaying objects, and various methods also exist for suppressing the underlying event and pileup in jets, beyond the subtraction approach discussed in section 8. `FastJet` 3 provides a common interface for such tools, intended to help simplify their usage and to guide authors of new ones. Below, we first discuss generic considerations about these tools, which we call `fastjet::Transformer`s. We then describe some that have already been implemented. New user-defined transformers can be implemented as described in section E.4.

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

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

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

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

which returns a reference to an object of type `MyTransformer::StructureType`. This is illustrated below on a case-by-case basis for each of the transformers that we discuss. Using the Boolean function

`transformed_jet.has_structure_of<MyTransformer>()` it is possible to check if `transformed_jet` is compatible with the structure provided by `MyTransformer`.

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

9.1 Noise-removal transformers

In section 8.1.2 we already saw one transformer for noise removal, i.e. `Subtractor`. Others have emerged in the context of jet substructure studies and are described here.

9.1.1 Jet Filtering and Trimming using `Filter`

Filtering was first introduced in [51] to reduce the sensitivity of a boosted Higgs-candidate jet's mass to the underlying event. Generally speaking, filtering clusters a jet's constituents with a smaller-than-original jet radius $R_{\rm filt}$. It then keeps just the $n_{\rm filt}$ hardest of the resulting subjets, rejecting the others. Trimming [52] is similar, but selects the subjets to be kept based on a p_t cut. The use of filtering and trimming has been advocated in number of contexts, beyond just the realm of boosted object reconstruction.

The `fastjet::Filter` class derives from `Transformer`, and can be constructed using a `JetDefinition`, a `Selector` and (optionally) a value for the background density,

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

This reclusters the jet's constituents with the jet definition `subjet_def`[30] 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 is subtracted (using the specified value for the background density) prior to the selection of the kept subjets. Alternatively, the user can set a `Subtractor` (see section 8.1.2), e.g.

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

When this is done, the subtraction operation is performed using the `Subtractor`, independently of whether a value had been set for `rho`.

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

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

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

In the first one, only the radius parameter is specified instead of the full subjet definition. In the second, one has to provide a (pointer to) a class derived from `FunctionOfPseudoJet<double>` which dynamically computes the filtering radius as a function of the jet being filtered (as was originally used in [51] where $R_{\rm filt} = \min(0.3, R_{b\bar{b}/2})$, with $R_{b\bar{b}}$ the distance between the parents of the jet).

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

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

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

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

Additional structural information is available as follows:

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

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

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

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

Note that the jet being filtered must have constituents. Furthermore, if `rho` is non-zero or if a `Subtractor` is set, the input jet must come from a cluster sequence with area support and explicit ghosts. If any of these requirements fail, an exception is thrown. In cases where the filter/trimmer has been defined with just a jet radius, the reclustering of the jet is performed with the same recombination scheme as was used in producing the original jet (assuming it can be uniquely determined).

9.1.2 Jet pruning

Pruning was introduced in [21]. It works by reclustering a jet's constituents with some given sequential recombination algorithm, but vetoing soft and large-angle recombinations between pseudojets i and j, specifically when the two following conditions are met

1. the geometric distance between i and j is larger than a parameter `Rcut`, with `Rcut` = `Rcut_factor`$\times 2m/p_t$, where m and p_t are the mass and transverse momentum of the original jet being pruned;
2. one of p_t^i, p_t^j is $<$ `zcut`$\times p_t^{i+j}$.

When the veto condition occurs, the softer of i and j is discarded, while the harder one continues to participate in the clustering.

Pruning bears similarity to filtering in that it reduces the contamination of soft noise in a jet while aiming to retain hard perturbative radiation within the jet. However, because by default the parameters for the noise removal depend on the original mass of the jet, the type of radiation that is discarded depends significantly on the initial jet structure. As a result pruning, in its default form, is better thought of as a noise-removing boosted-object tagger (to be used in conjunction with a pruned-jet mass cut) rather than a generic noise-removal procedure.

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

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

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

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

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

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

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

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

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

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

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

The `pruned_jet.structure_of<Pruner>()` object also provides `Rcut()` and `zcut()` members, in order to retrieve the actual $R_{\rm cut}$ and $z_{\rm cut}$ values used for that jet.

9.2 Boosted-object taggers

A number of the taggers developed to distinguish 2- or 3-pronged decays of massive objects from plain QCD jets (see the review [15]) 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` with zero momentum, i.e. one that has the property `jet == 0`.

At the time of writing, we provide only a small set of taggers. These include one main two-body tagger, the `fastjet::MassDropTagger` introduced in [51] and one main boosted top tagger, `fastjet::JHTopTagger` from [53] (`JHTopTagger` derives from the `fastjet::TopTaggerBase` class, expressly included to provide a common framework for all top taggers capable of also returning a W).

In addition, to help provide a more complete set of examples of coding methods to which users may refer when writing their own taggers, we have also included the `fastjet::CASubJetTagger` introduced in [54], which illustrates the use of a `WrappedStructure` (cf. appendix E.4) and the rest-frame `fastjet::RestFrameNSubjettinessTagger` from Ref. [55], which makes use of facilities to boost a cluster sequence.

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

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

9.2.1 The mass-drop tagger

Introduced in [51] for the purpose of identifying a boosted Higgs decaying into a $b\bar{b}$ pair, this is a general 2-pronged tagger. It starts with a fat jet obtained with a Cambridge/Aachen algorithm (originally, $R = 1.2$ was suggested for boosted Higgs tagging). Tagging then proceeds as follows:

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

The tagger can be constructed with

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

and applied using

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

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

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

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

Note that in [51] the mass-drop element of the tagging was followed by a filtering stage using $\min(0.3, R_{jj}/2)$ as the reclustering radius and selecting the three hardest subjects. That can be achieved with

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

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

A significantly updated version of the mass-drop tagger, modified as in Ref. [56], is available as part of the RecursiveTools `fjcontrib` module (see section 12).

9.2.2 The Johns-Hopkins top tagger

The Johns Hopkins top tagger [53] is a 3-pronged tagger specifically designed to identify top quarks. It recursively breaks a jet into pieces, finding up to 3 or 4 subjets and then looking for a W candidate among them. The parameters used to identify the relevant subjets include a momentum fraction cut and a minimal separation in Manhattan distance ($|\Delta y| + |\Delta\phi|$) between subjets obtained from a declustering.

The tagger will run with any jet that comes from a `ClusterSequence`, however to conform with the original formulation of [53], the `ClusterSequence` should be based on the C/A algorithm. A warning will be issued if this is not the case. If the `JetDefinition` used in the `ClusterSequence` involves a non-default recombiner, that same recombiner will be used when joining the final two prongs to form the boosted particle candidate. The tagger can be used as follows:

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

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

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

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

TopTaggerBase provides (via the associated structure)

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

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

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

9.2.3 The Cambridge/Aachen subjet tagger

The Cambridge/Aachen subjet tagger [54], originally implemented in a 3-pronged context, is really a generic 2-body tagger, which can also be used in a nested fashion to obtained multi-pronged tagging. It can be obtained through the include

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

As it is less widely used than the taggers mentioned above, we refer the user to the online doxygen documentation for further details.

9.2.4 The rest-frame N-subjettiness tagger

The rest-frame N-subjettiness tagger [55], meant to identify a highly boosted colour singlet particle decaying to 2 partons, can be obtained through the include

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

As it is less widely used than the taggers mentioned above, we refer the user to the online doxygen documentation for further details.

10 Compilation notes

Compilation and installation make use of the standard

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

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

In order to access the NlnN strategy for the k_t algorithm, the FastJet library needs to be compiled with support for the Computational Geometry Algorithms Library CGAL [13]. This same strategy gives $N \ln N$ performance for Cambridge/Aachen and $N^{3/2}$ performance for anti-k_t (whose sequence for jet clustering triggers a worst-case scenario for the underlying computational geometry methods.) CGAL can be enabled with the --enable-cgal at the configure stage. CGAL may be obtained in source form from http://www.cgal.org/ and is also available in binary form for many common Linux

distributions. For CGAL versions 3.4 and higher, the user can specify `--with-cgaldir=...` if the CGAL files are not installed in a standard location.[31]

The `NlnNCam` strategy does not require CGAL, since it is based on a considerably simpler computational-geometry structure [57].

11 FJcore

The `fjcore` package provides a simple, compact way of accessing the main `FastJet` functionality. It consists of just two files `fjcore.hh` and `fjcore.cc`. A program can access the `fjcore` functionality by linking with `fjcore.cc` instead of the full `FastJet` library,

```
g++ main.cc fjcore.cc
```

where `main.cc` includes `fjcore.hh` rather than the usual `FastJet` headers. Clustering results are identical to those obtained by linking to the full `FastJet` distribution.

One of the main intended uses of `fjcore` is to provide jet-finding code that can be easily distributed with 3rd party code (e.g. it is currently distributed with Pythia 8 [58] and MadGraph5_aMC@NLO). We emphasize that the `FastJet` licensing conditions *must* be respected when incorporating `fjcore`. This includes a mention of `fjcore`'s GPL license in the third-party package's own LICENSE or COPYING file. Furthermore for interactive usage, the `fjcore` banner may not be removed. This is because it contains crucial information, such as the `fjcore` version number, that users are expected to quote in their scientific publications. In order to make it possible for `fjcore` and the full `FastJet` (e.g. used in a separate user program) to coexist, the `fjcore` package uses the `fjcore` namespace instead of `fastjet`.

In particular, `fjcore` provides:

- access to all native hadron-collider and e^+e^- algorithms, k_t, anti-k_t, C/A. For C/A, the $N \ln N$ method is available, while anti-k_t and k_t are limited to the N^2 one (still the fastest for $N < 100$k particles)
- access to selectors, for implementing cuts and selections
- access to all functionalities related to PseudoJets (e.g. a jet's structure or user-defined information)

Instead, it does *not* provide:

- jet-area functionality
- background estimation
- access to other algorithms via plugins
- interface to CGAL
- fastjet tools, e.g. filters, taggers

[31] For events with near degeneracies in their Delaunay triangulation, issues have been found with versions 3.7 and 3.8 of CGAL. We recommend the use of earlier or later versions.

If these functionalities are needed, the full FastJet installation must be used. The code will be fully compatible, with the sole replacement of the header files and of the fjcore namespace with the fastjet one.

The fjcore package has been available since release 3.0.4 of FastJet, and tracks FastJet versions. It is available for download from the http://fastjet.fr/ web site.

12 FastJet Contrib

The FastJet update schedule is geared towards providing stability. Typically, a minor (e.g. 3.0 → 3.1) release is made every two or three years, and a new minor or major release may only become widely adopted (notably by the experiments) a further year or two down the line. In comparison, the field of jet physics, especially jet substructure, and the development of associated tools proceeds at a much faster pace.

To accommodate this, part way through the 3.1 development cycle, in 2013, FastJet Contrib (fjcontrib) was introduced. This provides a home for a variety of contributed tools. New tools from 3rd-party authors can usually be added in the space of days or weeks, after some basic review of the interface, so far usually carried out by the FastJet authors. Contents, instructions and information for contributing are provided on the fjcontrib web pages [59], http://fastjet.hepforge.org/contrib/.

Acknowledgements

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

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

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

A Clustering strategies and performance

The constructor for a JetDefinition can take a strategy argument (cf. section 3.2), which selects the algorithmic "strategy" to use while clustering. It is an enum of type Strategy with relevant values listed in table 2. Nearly all strategies are based on the factorisation of energy and geometrical distance components of the d_{ij} measure [10]. 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

`N2Plain`	Plain N^2 algorithm
`N2Tiled`	Tiled N^2 algorithm
`N2MinHeapTiled`	Tiled N^2 algorithm with a heap for tracking the min. of d_{ij}
`N2MHTLazy9`	Variant of `N2MinHeapTiled`, with "lazy" (see text) evaluation of distances to particles in the set of $9 = 3 \times 3$ nearby tiles
`N2MHTLazy25`	Like `N2MHTLazy9`, but uses tiles of size $R/2$ and a set of $25 = 5 \times 5$ neighbours
`N2MHTLazy9AntiKtSeparateGhosts`	Similar to `N2MHTLazy9`, but neglects ghost-ghost clusterings (to be considered preliminary in FJ3.1)
`NlnN`	Voronoi-based $N \ln N$ algorithm
`NlnNCam`	Based on Chan's $N \ln N$ closest pairs algorithm, suitable only for the Cambridge jet algorithm
`Best`	Automatic selection of the best of these based on N and R
`BestFJ30`	Automatic strategy selection as was done in FastJet 3.0

Table 2: The more interesting of the various algorithmic strategies for clustering. Other strategies are given in `JetDefinition.hh` — however, 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 N range in which a given strategy is optimal depends on R and on the rapidity extent of the particles. See figure 2 for details.

implemented hadron-collider jet algorithms. The two exceptions are: `NlnNCam`, which is based on a computational geometry algorithm for dynamic maintenance of closest pairs [57] (rather than the more involved nearest neighbour graph), and is suitable only for the Cambridge algorithm, whose distance measure is purely geometrical; and `N2MHTLazy9AntiKtSeparateGhosts`, intended specifically for area calculations with the anti-k_t algorithm (to be considered preliminary in v3.1; it can also be used for passive area calculations for other algorithms).

The `N2Plain` strategy uses a "nearest-neighbour heuristic" [60] 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,[32] 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. Both use tiles of size at least $R \times R$, and search for a particle's nearest neighbours in the 3×3 set of tiles centred on that particle's own tile. The `N2MHTLazy9` strategy is similar, however before considering a neighbouring tile it establishes whether the edge of the tile is closer than the particle's nearest same-tile neighbour. If not, it skips the tile, hence the name "lazy". The extra checks and bookkeeping introduce a speed penalty for moderate N, but at large N that is more than compensated for by the fact that one searches for neighbours in a smaller set of tiles. The `N2MHTLazy25` strategy is almost identical except that it uses a 5×5 set of tiles of size at least $R/2$. All the lazy algorithms are new in `FastJet` 3.1. The `NlnN` strategy uses CGAL's Delaunay Triangulation [13] for the maintenance of the nearest-neighbour graph. Note that $N \ln N$ performance 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 (or 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

[32] 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 [61].

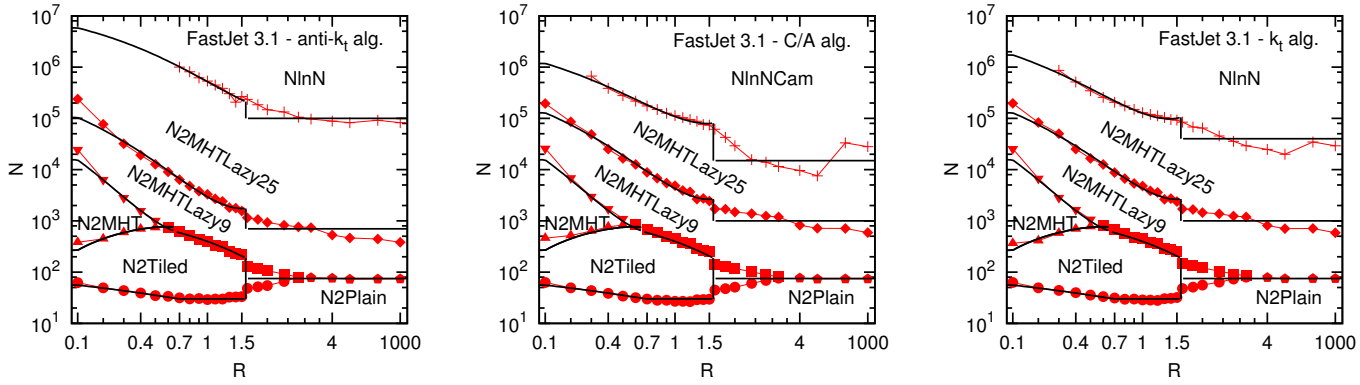

Figure 2: Transitions between fastest strategies in the plane of particle multiplicity N and radius R. The events consist of one hard scatter and a variable number pileup collisions, with particles accepted up to rapidity $|y| < 5$. To probe the small N region, only the N hardest particles in the event are kept. The red lines with symbols indicate the measured transitions, while the black solid lines are the approximate transitions used for the `Best` strategy. The three plots are, from left to right, for the anti-k_t, C/A and k_t algorithms.

impact only for $N \gtrsim 10^5$; we believe that it leads to $N^{3/2}$ asymptotic time for clustering).[33]

Given the many strategies, it is not entirely trivial to select the fastest one for any given event. The optimal choice depends on the jet algorithm, its radius parameter, the event multiplicity, and the way in which the particles are distributed across the event (e.g. all concentrated at small rapidities, or covering a large rapidity range). Figure 2 shows our determination of the optimal strategy for different R and N for events covering rapidities up to $|y| < 5$.[34] The solid black lines indicate the transitions encoded in the "`Best`" meta-strategy, which automatically selects one of the actual strategies for execution. Note that the choice of optimal strategy can to some extent depend also on the processor, compiler version and optimisation flags. Accordingly, the choices made in the `Best` strategy may not be perfectly optimal on all systems.

Illustrative timings for the `Best` strategy are shown as a function of N in figure 3 for the anti-k_t, k_t and the Cambridge/Aachen algorithms, with $R = 0.4$ on the left and $R = 1.2$ on the right. For comparison the figure also shows the timings for the anti-k_t algorithm in `FastJet` 3.0.6 and additionally for SISCone ($R = 0.4$ only). Kinks in the timings of the native algorithms are visible at the N values where there is a switch from one strategy to another. The speed improvements from the strategies introduced in version 3.1 set in starting with a couple of thousand (hundred) particles for $R = 0.4$ (1.2) and reach about a factor of ten improvement for $N \simeq 10^5$ (20 000); they also result in the transition to the $N \ln N$ strategies occurring at somewhat larger N than in version 3.0, typically at or beyond $N = 10^5$. Taking $N = 10^4$ particles as a reference (e.g. corresponding to moderate pp pileup plus ghosts for area subtraction[35]), the improvement in speed is about a factor of 2 for $R = 0.4$ and 7 for $R = 1.2$.

We note that there are a few places where there remains scope for timing improvements. In

[33] In versions of `FastJet` prior to 3.1.0, the $N \ln N$ strategy had the limitation that it could not be used in events with perfectly collinear particles. This was related to the fact that the underlying computational geometry structures cannot cleanly accommodate multiple particles in the same location. As of version 3.1.0, this limitation has been eliminated.

[34] The optimal transition to the `NlnNCam` strategy may depend strongly on the size of the cache.

[35] For the specific case of the anti-k_t algorithm with ghosts, the `N2MHTLazy9AntiKtSeparateGhosts` can bring further benefits in this region.

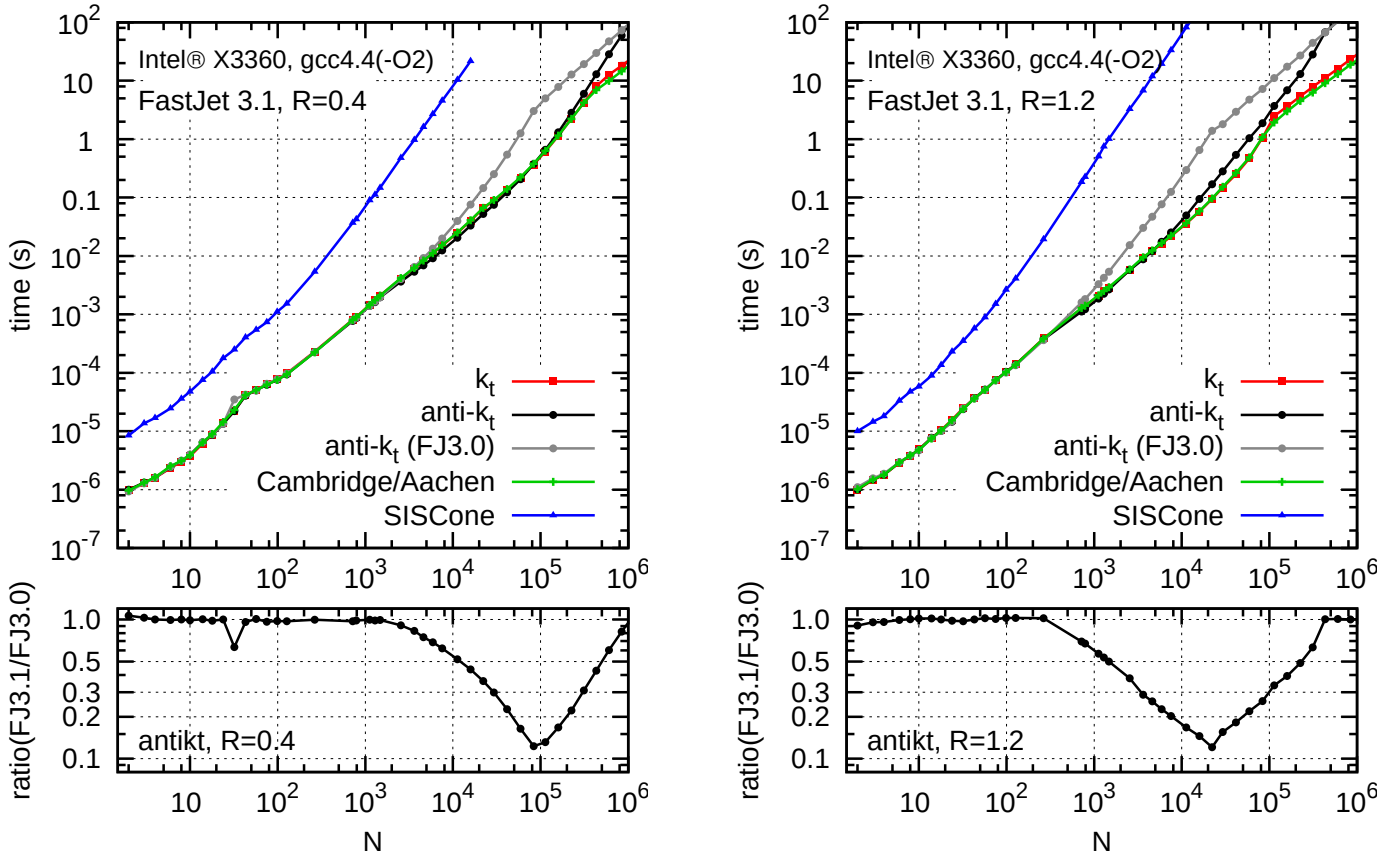

Figure 3: Time required to perform the clustering of N particles in `FastJet` 3.1.0 with the `Best` strategy. The anti-k_t, k_t, and Cambridge/Aachen (C/A) native algorithms are shown for $R = 0.4$ (left) and $R = 1.2$ (right). For comparison, the figure also includes the timings for the anti-k_t algorithm in version 3.0.6 of `FastJet`. The timings were obtained on an Intel Xeon X3360 processor with 6 MB of level-3 cache, running at 2.83GHz. The code was compiled with `g++` version 4.4, using the `-O2` optimisation flag. For small N, N was varied by taking a single hard dijet event generated with Pythia 6 [62] and extracting the N hardest particles. Large N values were obtained by taking a single hard dijet event and adding simulated minimum-bias events to it. Particles are limited to rapidities $|y| < 5$. The results include the time to extract the inclusive jets with $p_t > 5\,\text{GeV}$ and sort them into decreasing p_t. Note that timings observed in practice can differ from those given here for events with substantially different distributions of particles.

particular at low N the overheads related to copying and sorting a vector of PseudoJet objects are a relevant fraction of the total time, and could be reduced. Additionally, for the Cambridge/Aachen algorithm at moderate to large N, the use of multiple grid sizes could bring some benefit. The improvements contained in the N2MHTLazy9AntiKtSeparateGhosts strategy could be further extended to the Lazy25 case and made available automatically. Finally the Best strategy selection could try to better take into account the rapidity extent of the event, and special cases like single-jet reclustering. Should users have applications where such improvements are critical, they are encouraged to contact the FastJet authors.

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. such as an input particle's barcode in a HepMC event. However, it can quickly show its limitations; for example, when simulating pileup one might have several HepMC events and it is then useful for each particle to additionally store information about which HepMC event it comes from.

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

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

The function set_user_info(...) transfers ownership of the pointer to the PseudoJet. This is achieved internally with the help of a shared pointer. Copies of the PseudoJet then point to the same user_info. When the PseudoJet and all its copies go out of scope, the user_info is automatically deleted. Since nearly all practical uses of user_info 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 explicitly performs the cast of the extra info to type L. If the cast fails, or the user info has not been set, an error will be thrown.[36]

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.[37]

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

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

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

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

It behaves almost identically to the C++0x shared_ptr.[38] 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();
```

More advanced examples can be provided on request, including code that helps handle particle classes from third party tools such as Pythia 8 [58].

C Structural information for various kinds of PseudoJet

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

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(), is_pure_ghost()) will return the information from the ClusterSequenceAreaBase, or from the pieces in case of a composite jet. An error will be thrown if the jet does not have area information.

Internal storage of structural information. The means by which information about a jet's structure is stored is generally transparent to the user. The main exception that arises is when the

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

	particle	jet	jet (no CS)	constituent	join(j_1, j_2)	join(p_1, p_2)
has_associated_cs()	false	true	true	true	false	false
associated_cs()	NULL	CS	NULL	CS	NULL	NULL
has_valid_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.

user wishes to create jets with a new kind of structure, for example when writing boosted-object taggers. Here, we simply outline the approach adopted. For concrete usage examples one can consult section 9 and appendix E.4, where we discuss transformers and taggers.

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

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

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

D Functions of a PseudoJet

A concept that is new to FastJet 3 is that of a fastjet::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. Other func-

tions return a floating-point number associated with the jet: for example jet shapes, but also the rescaling functions used to provide local background estimates in section 8.2.

To help provide a uniform interface for functions of a `PseudoJet`, `FastJet` has 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`.[39].

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

E User-defined extensions of `FastJet`

E.1 External Recombination Schemes

A user who wishes to introduce a new recombination scheme may do so by 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).
```

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

```
    virtual void preprocess(PseudoJet & p) const {};

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

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

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

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

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

E.2 Implementation of a plugin jet algorithm

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

```
  class JetDefinition::Plugin{
  public:
    /// returns a textual description of the jet-definition implemented in this plugin
    virtual std::string description() const = 0;

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

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

    //------- ignore what follows for simple usage! ---------
    /// returns true if 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;}

    /// sets the ghost separation scale for passive area determinations
```

```
    /// in future runs (NB: const, so should set internal mutable var)
    virtual void set_ghost_separation_scale(double scale) const;
    virtual double ghost_separation_scale() const;

  };
```

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

```
using namespace fastjet;

void CDFMidPointPlugin::run_clustering(ClusterSequence & clust_seq) {

  // when run_clustering is called, the clust_seq.jets() has already been
  // filled with the initial particles
  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_j], with the specified dij, and return the index (newjet_k)
  /// allocated to the new jet. The recombined PseudoJet is determined by
  /// applying the JetDefinition's recombiner to the two input jets.
  /// (By default E-scheme recombination, i.e. a 4-vector sum)
  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 information 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; this jet will then be returned to
  /// the user when they request inclusive_jets() from the cluster sequence.
  void plugin_record_iB_recombination(int jet_i, double diB);
```

The `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 fictitious 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 an abstract base class, which the plugin should derive from 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
ClusterSequence::Extras * extras() const;
```

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

Note that when calculating areas, the `extras()` member works only for active areas with explicit ghosts and for Voronoi areas. The reason for this is that other types of area first carry out a clustering with explicit ghosts and then edit the `ClusterSequence` to remove the information related to pure ghost jets. This modifies the internal indices of the jets, making it very challenging, for example, for the `Extras` class to handle queries about individual jets.

E.2.1 Building new sequential recombination algorithms

To enable users to more easily build plugins for new sequential recombination algorithms, `FastJet` also provides a class `NNH`, which provides users with access to an implementation of the nearest-neighbour heuristic for establishing and maintaining information about the closest pair of objects in a dynamic set of objects (see [63] for an introduction to this and other generic algorithms). In good cases (C/A-like) this allows one to construct clustering that runs in N^2 time, though its worst case can be as bad as N^3 (e.g. anti-k_t). 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.

E.3 Implementing new selectors

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

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

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

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

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

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

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

E.4 User-defined transformers

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

```
class Transformer : public FunctionOfPseudoJet<PseudoJet> {
public:
  // the result of the Transformer acting on the PseudoJet.
  // this has to be overloaded in derived classes
  virtual PseudoJet result(const PseudoJet & original) const = 0;

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

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

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

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

To help understand how to create user-defined transformers, it is perhaps easiest to consider the example of a filtering/trimming class. The simplest form of such a class is the following:[40]

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

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

  virtual PseudoJet result(const PseudoJet & jet) const;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
typedef SimpleFilterStructure StructureType;
```

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

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

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

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

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

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

```
};
```

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

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

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

```
typedef SimpleFilterWrappedStructure StructureType;
```

line in the `SimpleFilter` class definition.

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

F Error handling

`FastJet` provides warning and error messages through the classes `fastjet::LimitedWarning` and `fastjet::Error` respectively. A user does not normally need to interact with them, however, they do provide some customisation facilities, especially to redirect and summarise their output.

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

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

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

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

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

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

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

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

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

With a suitable design of the output stream, the output redirection facility can also be used by the user to record additional information when an error or warning occurs, for example the event number. One only `stream << string` type operation is performed for each warning or error, so as to help with formatting in such cases.

As well as performing output of warnings and errors, `FastJet` also outputs a banner the first time that clustering is performed. If the user wishes to have the banner appear before the first clustering (e.g. during the initialisation phase of their program), they may call the static `ClusterSequence::print_banner()` function.

G Evolution of FastJet across versions

G.1 History

Version 1 of `FastJet` provided the first fast implementation of the longitudinally invariant k_t clustering [8, 9], based on the factorisation of momentum and geometry in that algorithm's distance measure [10].

Version 2.0 brought the implementation of the inclusive Cambridge/Aachen algorithm [22, 23] and of jet areas and background estimation [18, 17]; other changes include a new interface,[41] and new algorithmic strategies that could provide a factor of two improvement in speed for events whose number N of particles was $\sim 10^4$. Choices of recombination schemes and plugins for external jet algorithms were new features of version 2.1. The initial set of plugins included SISCone [26], the CDF midpoint [3] and JetClu [31] cones and PxCone [35, 34]. The plugins helped provide a uniform interface to a range of different jet algorithms and made it possible to overlay `FastJet` features such

[41] The old one was retained through v2

as areas onto the external jet algorithms. Version 2.2 never made it beyond the beta-release stage, but introduced a number of the features that eventually were released in 2.3. The final 2.3 release included the anti-k_t algorithm [14], a broader set of area measures, improved access to background estimation, means to navigate the ClusterSequence and a new build system (GNU autotools). Version 2.4 included the new version 2.0 of SISCone (including the spherical variant), as well as plugins to the DØ Run II cone, the ATLAS cone, the CMS cone, TrackJet and a range of e^+e^- algorithms, and also further tools to help investigate jet substructure. It also added a wrapper to `FastJet` allowing one to run SISCone and some of the sequential recombination algorithms from Fortran programs.

A major practical change in version 3.0 was that `PseudoJet` acquired knowledge (where relevant) about its underlying ClusterSequence, allowing one to write *e.g.* `jet.constituents()` It also became possible to associate extra information with a `PseudoJet` beyond just a user index. It brought the first of a series of `FastJet` tools to help with advanced jet analyses, namely the `Selector` class, filters, pruners, taggers and new background estimation classes. Version 3.0 also added the D0-Run I cone [32] plugin and support for native jet algorithms to be run with $R > \pi/2$. Version 3.1 gave significant speed improvements for multiplicities from a few thousand up to a few hundred thousand and various other small additions. The period in between versions 3.0.0 and 3.1.0 also saw the first releases of the `fjcore` compact subset of `FastJet` (section 11) and of the `fjcontrib` project (section 12).

G.2 Deprecated and removed features

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

As of version 3.1.0, the `FASTJET_VERSION_NUMBER` symbol can be used to provide version-dependent code compilation, as discussed in section 3.7.

G.3 Backwards compatibility of background estimation facilities

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

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

```
set_use_area_4vector(false);
```

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

Feature, class or include file	Dep.	Rem.	Suggested replacement
FjClusterSequence.hh	2.0	3.0	fastjet/ClusterSequence.hh
FjPseudoJet.hh	2.0	3.0	fastjet/PseudoJet.hh
CS::set_jet_finder(...)	2.1	3.0	pass a JetDefinition to constructor
CS::set_jet_algorithm(...)	2.1	3.0	pass a JetDefinition to constructor
CS::CS(particles, R, ...)	2.1	3.0	CS::CS(particles, jet_def)
JD(jet_alg, R, strategy)	2.1	-	JD(jet_alg, R, recomb_scheme, strategy)
JetFinder	2.3	-	JetAlgorithm
SISConePlugin.hh	2.3	3.0	fastjet/SISConePlugin.hh (idem. other plugins)
ActiveAreaSpec	2.3	–	AreaDefinition & GhostedAreaSpec
ClusterSequenceWithArea	2.3	–	ClusterSequenceArea
default $f = 0.5$ in some cone plugins	–	2.4	include f explicitly in constructor
default $R = 1$ in JetDefinition	–	2.4	include R explicitly in constructor
RangeDefinition	3.0	–	Selector(s)
CircularRange	3.0	–	SelectorCircle
CSAB::median_pt_per_unit_area(...)	3.0	–	BackgroundEstimator
CSAB::parabolic_pt_per_unit_area(...)	3.0	–	BackgroundEstimator (cf. section 8.2)
GAS::set_fj2_placement(...)	3.0	–	use new default ghost placement instead
CS::plugin_associate_extras(auto_ptr)	3.1	–	CS::plugin_associate_extras(Extras *)

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

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

References

[1] G. Sterman and S. Weinberg, "Jets From Quantum Chromodynamics," Phys. Rev. Lett. **39** (1977) 1436.

[2] S. Moretti, L. Lonnblad and T. Sjostrand, "New and old jet clustering algorithms for electron positron events," JHEP **9808** (1998) 001 [arXiv:hep-ph/9804296].

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

[4] S. D. Ellis, J. Huston, K. Hatakeyama, P. Loch and M. Tonnesmann, "Jets in Hadron-Hadron Collisions," Prog. Part. Nucl. Phys. **60** (2008) 484 [arXiv:0712.2447 [hep-ph]].

[5] G. P. Salam, Eur. Phys. J. **C67** (2010) 637-686 [arXiv:0906.1833 [hep-ph]].

[6] A. Ali, G. Kramer, Eur. Phys. J. **H36** (2011) 245-326. [arXiv:1012.2288 [hep-ph]].

[7] `http://www.gnu.org/licenses/gpl-2.0.html`

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

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

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

[11] M. Seymour, `http://hepwww.rl.ac.uk/theory/seymour/ktclus/`.

[12] `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].

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

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

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

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

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

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

[19] M. Cacciari, J. Rojo, G. P. Salam, G. Soyez, Eur. Phys. J. **C71** (2011) 1539. [arXiv:1010.1759 [hep-ph]].

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

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

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

[23] 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.

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

[25] L. Lonnblad, Z. Phys. **C58** (1993) 471-478.

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

[27] 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].

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

[29] S. Weinzierl, arXiv:1108.1934 [hep-ph].

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

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

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

[33] V. M. Abazov *et al.* [D0 Collaboration], arXiv:1110.3771 [hep-ex].

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

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

[36] T. Affolder *et al.* [CDF Collaboration], Phys. Rev. **D65** (2002) 092002.

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

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

[39] G. P. Salam and G. Soyez, April 2009, unpublished.

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

[41] M. Dasgupta, L. Magnea and G. P. Salam, JHEP **0802** (2008) 055 [arXiv:0712.3014 [hep-ph]].

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

[43] M. Cacciari, G. P. Salam and G. Soyez, contribution in preparation to proceedings of "Workshop on TeV Colliders", Les Houches, June 2011.

[44] G. Soyez, G. P. Salam, J. Kim, S. Dutta and M. Cacciari, Phys. Rev. Lett. **110** (2013) 16, 162001 [arXiv:1211.2811 [hep-ph]].

[45] D. Krohn, M. D. Schwartz, M. Low and L. T. Wang, arXiv:1309.4777 [hep-ph].

[46] M. Cacciari, G. P. Salam and G. Soyez, arXiv:1404.7353 [hep-ph].

[47] P. Berta, M. Spousta, D. W. Miller and R. Leitner, JHEP **1406** (2014) 092 [arXiv:1403.3108 [hep-ex]].

[48] M. Cacciari, G. P. Salam and G. Soyez, arXiv:1407.0408 [hep-ph].

[49] D. Bertolini, P. Harris, M. Low and N. Tran, arXiv:1407.6013 [hep-ph].

[50] M. Cacciari, P. Quiroga-Arias, G. P. Salam and G. Soyez, Eur. Phys. J. C **73** (2013) 2319 [arXiv:1209.6086 [hep-ph]].

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

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

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

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

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

[56] M. Dasgupta, A. Fregoso, S. Marzani and G. P. Salam, JHEP **1309** (2013) 029 [arXiv:1307.0007 [hep-ph]].

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

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

[59] `FastJet Contrib`, `http://fastjet.hepforge.org/contrib/`.

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

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

[62] T. Sjostrand, S. Mrenna and P. Skands, "`Pythia` 6.4 physics and manual," JHEP **0605** (2006) 026, [arXiv:hep-ph/0603175].

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

[64] D. Bertolini, T. Chan and J. Thaler, JHEP **1404** (2014) 013 [arXiv:1310.7584 [hep-ph]];
A. J. Larkoski, D. Neill and J. Thaler, JHEP **1404** (2014) 017 [arXiv:1401.2158 [hep-ph]];
G. P. Salam, unpublished.