FastJet 2.1 user manual

Matteo Cacciari and Gavin P. Salam
LPTHE, Université Pierre et Marie Curie – Paris 6,
Université Denis Diderot – Paris 7, CNRS UMR 7589

Abstract

FastJet provides fast $(N \ln N, N^2)$ implementations of the longitudinally invariant k_t and Cambridge/Aachen jet finders, based in part on tools and methods from the Computational Geometry community. It also provides ways of determining jet areas. Other jet finders can be accessed from the FastJet interface using a plugin mechanism.

Contents

1 Introduction

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

The implementation of the inclusive Cambridge algorithm and of jet areas are new features of version 2 of `FastJet`; other changes include a new interface (the old one will be retained), and new algorithmic strategies that can provide a factor of two improvement in speed for events whose number N of particles $\sim 10^4$. Choices of recombination schemes and plugins are new features of version 2.1. We anticipate that the `FastJet` facilities may also be made available at a later date with a more developed interface as part of the planned multi-purpose `HepJet` jet-finding package [6]. In the meantime the plugin feature provides a similar functionality. Plugins are included for the fortran

PxCone code and for the CDF C++ JetClu and MidPoint cone jet finders (all infrared unsafe), as well as the recent Seedless Infrared-Safe Cone (SISCone) jet finder [7].

There are a range of variants possible for the both the k_t and Cambridge jet-finders. For the k_t jet finder, `FastJet` implements the so-called ΔR distance measure (as recommended in [8]) with various recombination schemes. It has both exclusive [2] and inclusive modes [3], the latter currently being in more widespread use, though the exclusive mode has physical advantages in certain cases. We provide access to both modes.

1.1 k_t jet finder

The definition of the inclusive k_t jet finder that is coded is as follows:

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

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

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

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

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

The exclusive jet algorithm 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.

1.2 Cambridge/Aachen jet finder

Currently the Cambridge/Aachen jet-finder is provided only in an inclusive version, whose formulation is identical to that of the k_t jet finder, except as regards the distance measures, which are:

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

$$d_{iB} = 1\,. \tag{2b}$$

Attempting to extract exclusive jets from the Cambridge/Aachen with a d_{cut} parameter simply provides the set of jets obtained up to the point where all $d_{ij}, d_{iB} > d_{cut}$. The true exclusive formulation of the Cambridge algorithm [4] instead makes use an auxiliary (k_t) distance measure and 'freezes' pseudojets whose recombination would involve too large a value of the auxiliary distance measure.

1.3 Recombination schemes

When merging particles in step 2 of the clustering procedure, one must specify how to combine the momenta. The simplest procedure (E-scheme) simply adds the four-vectors. This has been advocated as a standard in [8], and is the default option in FastJet.

Other schemes provided by earlier k_t-clustering implementations [9] 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

$$p_{t,r} = p_{t,i} + p_{t,j}\,, \tag{3a}$$
$$\phi_r = (w_i\phi_i + w_j\phi_j)/(w_i + w_j)\,, \tag{3b}$$
$$y_r = (w_i y_i + w_j y_j)/(w_i + w_j)\,, \tag{3c}$$

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

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

2 Jet-finding interface

The `FastJet` code is written in C++. From the point of view of simple usage its interface has a number of similarities to that of `KtJet` [9], fundamental differences being highlighted below.

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

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

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

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

2.1 fastjet::PseudoJet

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

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

where the second form allows the initialisation to be obtained from any class `T` that allows subscripting to return the components of the momentum (running from $0 \ldots 3$ in the order p_x, p_y, p_z, E), for example

the CLHEP HepLorentzVector class.[1] It 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 to pi
double rap()      const ; // returns the rapidity
double rapidity() const ; // returns the rapidity
double kt2()      const ; // returns the squared transverse momentum
double perp2()    const ; // returns the squared transverse momentum
double perp()     const ; // returns the transverse momentum
double m2()       const ; // returns squared invariant mass
double m()        const ; // returns invariant mass (-sqrt(-m2()) if m2() < 0)
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;
```

It also allows the user to set and access an index, in case the user wishes to keep track of the identity of a fastjet::PseudoJet object

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

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

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

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

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

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

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

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

E_scheme
pt_scheme
pt2_scheme
Et_scheme
Et2_scheme
BIpt_scheme
BIpt2_scheme

Table 1: Members of the `RecombinationScheme` enum; the last two refer to boost-invariant version of the p_t and p_t^2 schemes (as defined in section 1.3).

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

Table 2: The more interesting of the various algorithmic strategies for clustering. (For remaining strategies see `JetDefinition.hh`)

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

2.2 fastjet::JetDefinition

The class `fastjet::JetDefinition` contains a full specification of how to carry out the clustering. Its constructor is[2]

```
fastjet::JetDefinition(fastjet::JetFinder jet_finder = kt_algorithm,
                       double R = 1.0,
                       fastjet::RecombinationScheme recomb_scheme = E_scheme,
                       fastjet::Strategy strategy = Best);
```

The jet finder is one of the entries of the `fastjet::JetFinder enum`:

```
enum JetFinder {kt_algorithm, cambridge_algorithm};
```

`R` specifies the value of R that appears in eqs. (1,2). The recombination scheme should be one of those listed in table 1. If it is omitted the E-scheme is chosen.

The strategy selects the algorithmic strategy to use while clustering and is an `enum` of type `fastjet::Strategy` with potentially interesting values listed in table 2: All strategies apply equally well to the k_t and Cambridge jet finders, except `NlnNCam`, which being based on a computational geometry algorithm for dynamic maintenance of closest pairs [10] (rather than the more involved

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

nearest neighbour graph), is suitable only for the Cambridge algorithm whose distance measure is purely geometrical.

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

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

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

2.3 fastjet::ClusterSequence

To run the jet clustering, create a `fastjet::ClusterSequence` object,[3] through the following constructor

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

where `input_particles` is the vector of initial particles of any type (`fastjet::PseudoJet`, `HepLorentzVector`, etc.) that can be used to initialise a `fastjet::PseudoJet` and `jet_def` contains the full specification of the clustering (see Section 2.2).

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

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

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

There are two ways of accessing exclusive jets,[4] one where one specifies d_{cut}, the other where one specifies that the clustering is taken to be stopped once it reaches the specified number of jets.

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

/// return a vector of all jets when the event is clustered (in the
/// exclusive sense) to exactly njets.
vector<fastjet::PseudoJet> exclusive_jets (const int & njets) const;
```

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

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

Note that these two member functions have the same name, and the compiler selects the correct one based on the type of the arguments, as is standard in C++. The `fastjet::PseudoJet` vectors returned by the above routines can all be sorted with the routines described at the end of section 2.1.

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

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

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

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

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

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

Finally the user may obtain the constituent particles of a given jet, via

```
/// return a vector of the particles that make up jet
vector<fastjet::PseudoJet> constituents (const fastjet::PseudoJet & jet);
```

Note that this is a member function of the `fastjet::ClusterSequence` and not of `fastjet::PseudoJet`, because jets are only meaningful when referred to a given `fastjet::ClusterSequence`.[5] If the user wishes to identify the constituents with the original particles provided to `fastjet::ClusterSequence` she or he should have set a unique index for each of the original particles with the `fastjet::PseudoJet::set_user_index` function.

Subjet analysis. Currently no dedicated subjet analysis is provided. It is however trivial (though non-optimal) to obtain a subjet analysis as follows:

```
// run clustering on vector<fastjet::PseudoJet> all_particles
```

[5]This is a further respect in which the interface differs from that of `KtJet`.

```
  fastjet::ClusterSequence clust_seq(all_particles);

  // get the exclusive jets with the event viewed as two hard jets
  vector<fastjet::PseudoJet> jets = clust_seq.exclusive_jets(2);

  // get the constituents of the first of these jets
  vector<fastjet::PseudoJet> jet0_constituents = clust_seq.constituents(jets[0]);

  // now rerun the jet clustering on jet0_constituents
  fastjet::ClusterSequence jet0_seq(jet0_constituents);

  // run whatever subjet analysis is of interest here on jet0_seq.
```

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

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

3 Example program

For full details see the example program `example/fastjet_example.cc` that is distributed with the `FastJet` code. For the reader who is familiar with `KtJet` [9], the example program is also given as it would be written for `KtJet`, in the file `example/ktjet_example.cc`.

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

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

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

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

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

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

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

  // extract the exclusive jets with dcut = 25 GeV^2 and sort them
  // in order of increasing pt
  double dcut = 25.0;
  vector<fastjet::PseudoJet> exclusive_jets = sorted_by_pt(
                                       clust_seq.exclusive_jets(dcut));

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

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

4 Jet areas

Since a jet is made up of only a finite number of particles, one needs a specific definition in order to make its *area* (i.e. the surface in the y-ϕ plane over which it extends) an unambiguous concept. Currently only a single kind of area definition exists, which we refer to as an *active* area [11], however since future releases are expected to implement additional definitions of areas, it was deemed useful to provide a base class defining a common interface to access information about jet areas.

4.1 fastjet::ClusterSequenceWithArea

This is a base class for representing cluster sequences that include information about jet areas. It is derived from `fastjet::ClusterSequence`. It includes the methods

```
  /// return the area associated with the given jet
  double area (const fastjet::PseudoJet & jet) const ;

  /// return the error (uncertainty) associated with the determination
  /// of the area of the jet;
  double area_error (const fastjet::PseudoJet & jet) const ;
```

```
/// return a PseudoJet whose 4-vector is defined by the following integral
///
///       \int drap d\phi PseudoJet("rap,phi,pt=one") *
///                           * Theta("rap,phi inside jet boundary")
///
/// where PseudoJet("rap,phi,pt=one") is a 4-vector with the given
/// rapidity (rap), azimuth (phi) and pt=1, while Theta("rap,phi
/// inside jet boundary") is a function that is 1 when rap,phi
/// define a direction inside the jet boundary and 0 otherwise.
///
fastjet::PseudoJet area_4vector(const PseudoJet & jet) const ;
```

4.2 Active areas

Currently, `FastJet` implements what is called an *active* area [11]. It is determined by adding a large number of extremely soft "ghost" particles to the event, and letting them cluster with the user's event (as well as between themselves). The ghosts are distributed on a uniform grid (with small random fluctuations to avoid degeneracies) in y and ϕ. The number of ghosts that end up in a given jet provides a measure of its area. Because the ghosts are extremely soft (and because the k_t and Cambridge/Aachen jet finders are infrared safe), the presence of the ghosts does not affect the set of user particles that end up in a given jet.

4.2.1 fastjet::ActiveAreaSpec

In order to carry out a clustering with an active-area determination, the user must first create an object that specifies how to distribute the ghosts. This is done via the class `fastjet::ActiveAreaSpec` whose constructor is

```
fastjet::ActiveAreaSpec(double ghost_maxrap,
                        int    repeat = 1,
                        double ghost_area = 0.01,
                        double grid_scatter = 1e-4,
                        double kt_scatter = 0.1,
                        double mean_ghost_kt = 1e-100);
```

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

For sparse events, the set of ghost particles that end up in a given jet is not unique, and depends on the degeneracy-breaking random shifts added to the ghost positions, as compared to a perfect grid distribution. To obtain a reliable area one may then repeat the area determination several times, the number of times being specified by the `repeat` variable. For hadron-level events a value of 5 is sufficient to give jet areas that are determined to within a few percent. In practice it is usually

satisfactory even to set `repeat`= 1 and this is the default since it runs faster. For events with a dense distribution of true particles, there is no degeneracy in the ghost clustering and there is no need at all to use `repeat`> 1. If `repeat` > 1, a statistical uncertainty on the area, given by $\sigma/\sqrt{\texttt{repeat}-1}$, is provided for each jet.

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

4.2.2 fastjet::ClusterSequenceActiveArea

This is the main interface for the active area determination. The user may create an object of this type in order to carry out a jet clustering that also provides information on the areas of the jets. Its constructor is

```
template<class L> fastjet::ClusterSequenceActiveArea
       (const std::vector<L> & pseudojets,
        const fastjet::JetDefinition & jet_def,
        const fastjet::ActiveAreaSpec & area_spec,
        const bool & writeout_combinations = false) ;
```

This class is derived from `fastjet::ClusterSequenceWithArea`. The set of jets that it returns, as well as their particle content, is identical to that of `ClusterSequence`. In particular the ghosts particles are not included in the jets returned to the user. Furthermore, jets composed exclusively of ghosts are eliminated from the list of jets (since the number and momenta of any such jets fluctuates according to the random shifts of the ghosts on the grid).[6]

In addition to providing clustering and jet-area determination, this class includes some methods for extracting information about the estimated background in the clustered event. These methods are to be considered *experimental*: it is likely that their interfaces will change in a future release, and further refinements may be added. It is also possible that their functionality will be transferred to the base class.

```
/// enum providing a variety of tentative strategies for estimating
/// the background (e.g. non-jet) activity in a highly populated event; the
/// one that has been most extensively tested is median.
enum fastjet::mean_pt_strategies{median=0, non_ghost_median, pttot_over_areatot,
                                 pttot_over_areatot_cut, mean_ratio_cut, play,
                                 median_4vector} ;

/// return the transverse momentum per unit area according to one
/// of the above strategies; for some strategies (those with "cut"
/// in their name) the parameter "range" allows one to exclude a
/// subset of the jets for the background estimation, those that
/// have pt/area > median(pt/area)*range.
double pt_per_unit_area(fastjet::mean_pt_strategies strat=median,
```

[6] In order to extract jet including their ghosts (and any pure ghost jets), use the class `ClusterSequenceActiveAreaExplicitGhosts` whose interface is documented in the corresponding include file.

```
                              double range=2.0 ) const;

  /// fits a form pt_per_unit_area(y) = a + b*y^2 in the range
  /// abs(y) < raprange (for negative raprange, it defaults to
  /// abs(y) < ghost_maxrap - Rparam).
  void parabolic_pt_per_unit_area(double & a, double & b,
                                  double raprange=-1.0,
                                  double exclude_above=-1.0,
                                  bool use_area_4_vector=false) const;
```

`pt_per_unit_area` can usually be used for high luminosity events with pileup, while `parabolic_pt_per_unit_area` has been found to provide a better description of the background (its transverse momentum per unit area being parabolic in y) in simulated Pb-Pb collisions at the LHC.

The `mean_pt_strategies` are options of `pt_per_unit_area`. They dictate the strategy with which the transverse momentum per unit area of the background of each event is determined. Common choices are:

> `non_ghost_median` - For each jet i one defines the quantity $\rho_i = p_{t,i}/A_i$ where A_i is the jet area. This option returns the median value of the ρ_i for the set of all true jets (those that contain at least one user particle — as opposed to ghost jets, which contain only ghost particles).
>
> `median` - This is the default choice. As above, but the median is taken over *all* jets, including purely ghost jets. This definition coincides with the previous one for densely populated events, and provides a more faithful estimation of the underlying activity for more sparsely populated events.
>
> `median_4vector` - Like `median`, but the areas are calculated using the `area_4vector()` function. Gives a more faithful estimation of the background for highly populated events. Should eventually become the default.

The jets that are considered with the above options are those whose area is considered reliable, i.e. those with $|y| <$ `ghost_maxrap` $- R$.

The `parabolic_pt_per_unit_area` member function returns the coefficients a and b of the least square fit of all the $\rho_i <$ `exclude_above` to the parabolic function $a + by^2$. If `exclude_above` < 0 (default behaviour) no upper limit is used. Its only strategy option is the boolean `use_area_4vector`. When set to true, the `area_4vector()` function is used internally to evaluate the areas.

5 Plugin jet finders

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

5.1 Generic plugin use and construction

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

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

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

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

A `JetDefinition` can be constructed by providing a pointer to a `JetDefinition::Plugin`; the resulting `JetDefinition` object can then be used identically to the normal `JetDefinition` objects used in the previous sections.

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

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

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

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

```
using namespace fastjet;

void CDFMidPointPlugin::run_clustering(ClusterSequence & clust_seq) {

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

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

// ...
```

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

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

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

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

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

Certain (cone) jet algorithms do not perform pairwise clustering — in these cases the plugin must invent a ficititious series of pairwise recombinations that leads to the same final jets.

Example plugin jet finders that are provided with `FastJet` are the `CDFMidPointPlugin` illustrated above and also a `CDFJetCluPlugin`. These interface code available at [12]. The `PxConePlugin` interfaces the PxCone code [13]. All three of these cone jet finders, though widely used, are infrared unsafe. The `SISConePlugin`, described in detail below, interfaces the infrared safe seedless cone algorithm of [7]. Examples using these plugins and some further documentation are provided in the `plugins/` directory.

5.2 SISCone Plugin

SISCone [7] is an implementation of a cone type jet finder. As with most modern cone algorithms, it is divided into two parts: first it searches for stable cones; then, because a particle can appear in

more than one stable cone, a 'split–merge' procedure is applied, which ensures that no particle ends up in more than one jet. The stable cones are identified using an $(N^2 \ln N)$ seedless approach. This (and some care in the the 'split–merge' procedure) ensures that the jets it produces are insensitive to additional soft particles and collinear splittings (i.e. the jets are infrared and collinear safe).

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

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

A cone centered at y_c, ϕ_c is stable if the sum of momenta of all particles i satisfying $\Delta y_{ic}^2 + \Delta\phi_{ic}^2 <$ `cone_radius`^{2} has rapidity y_c, ϕ_c. The `overlap_threshold` is the fraction of overlapping momentum above which two protojets are merged in a Tevatron Run II type [8] split–merge procedure. 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 [14], SISCone can carry out multiple stable-cone search passes (as advocated in [15]): in each pass one searches for stable cones using just the subset of particles not present in any stable cone in earlier passes. Up to `n_pass_max` passes are carried out, and the algorithm automatically stops at the highest pass that gives no new stable cones. The default of `n_pass_max` $= 0$ is equivalent to setting it to ∞. Since concern has been expressed that an excessive number of stable cones may complicate cone jets in events with high noise [8], the `protojet_ptmin` parameter allows one to use only protojets with $p_t \geq$ `protojet_ptmin` in the split–merge phase (all others are thrown away).[7]

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 [8]) is not

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

boost-invariant, and $m_t = \sqrt{m^2 + p_t^2}$ is IR unsafe for events whose hadronic component conserves momentum and stems from the decay of two identical particles.

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

```
// define a generic plugin pointer
fastjet::JetDefinition::Plugin * plugin;

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

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

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

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

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

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

Some extra output information is appropriate for a cone algorithm that is not of relevance in clustering algorithms. Accordingly SISCone 'misappropriates' the `user_index()` of the jets, using it to return the pass in which a jet was found.[8] The user may also obtain a list of the positions of original stable cones as follows:

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

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

Since SISCone is infrared safe, it may meaningfully be used also with the `ClusterSequenceActiveArea` class. Note however that in that case ones loses the cone-specific

[8] Owing to the current internal representation of jets in `FastJet`, this does not work for jets consisting of a single particle — the user index of the jet is then just the user index originally attributed to that particle (by default, -1).

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 should 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 $(N^2 \ln N)$ time to find jets, and the memory use is (N^2); taking $N = 10^3$ as a reference point, it runs in a few tenths of a second, making it about 100 times slower than `FastJet`. These are 'expected' results, i.e. valid for a suitably random set of particles. In area determinations, the ghost particles are anything but random, and run times and memory usage are, in practice, somewhat larger than for a normal QCD event with the same number of particles. We therefore recommend running with not too small a `ghost_area` (e.g. ~ 0.05) and using `grid_scatter` $= 1$, which helps to reduce the number of stable cones (and correspondingly, the time and memory usage of the subsequent split–merge step).

6 Compilation notes

In order to access the `NlnN` strategy the library needs to be compiled with the Computational Geometry Algorithms Library `CGAL` [16].

The user should first download `CGAL` from `http://www.cgal.org/` and then compile and install it. Under linux, she or he will have to ensure that an environment variable `CGAL_MAKEFILE` points to the Makefile generated by `CGAL` at install time, containing various definitions of locations of include files. `CGAL` will encourage the user to set this variable during the install process.

The choice of whether or not CGAL will be used is made in the Makefile in the top directory of the distribution, through the value of the variable `USE_CGAL` which should be either `yes` or `no`. For more details see the `INSTALL` file in the top directory.

The `NlnNCam` strategy does not require CGAL, since it is based on a considerably simpler computational-geometry structure [10].

Acknowledgements

We wish to thank Timothy Chan for helpful exchanges about the details of his dynamic closest pair algorithm. We thank Markus Wobisch and Andreas Oehler for comments on the documentation.

We are grateful to Joey Huston and Mike Seymour for providing permission to distribute their cone jet-finder codes as plugins for fastjet.

A External Recombination Schemes

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

```
class JetDefinition::Recombiner {
public:
  /// return a textual description of the recombination scheme
  /// implemented here
```

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

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

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

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

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

```
  JetDefinition(JetFinder jet_finder,
                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.

References

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

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

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

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

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

[6] `HepJet` is being planned in collaboration with authors of `KtJet` [9].

[7] G. P. Salam and G. Soyez, in preparation; standalone code available from `http://projects.hepforge.org/siscone`.

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

[9] 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].

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

[11] M. Cacciari and G. P. Salam, in preparation.

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

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

[14] 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].

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

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