MAN/HEP/2002/02
UCL/HEP 2002-02
October 2002

KtJet : A C++ implementation of the $K_\perp$ clustering algorithm

J. M. Butterworth[1], J. P. Couchman[1],
B. E. Cox[2] and B. M. Waugh[1]

[1]Department of Physics and Astronomy
University College London
Gower St. London WC1E 6BT
England

[2]Department of Physics and Astronomy
University of Manchester
Manchester M13 9PL
England

Abstract

A C++ implementation of the $K_\perp$ jet algorithm for high energy particle collisions is presented. The time performance of this implementation is comparable to the widely used Fortran implementation. Identical algorithmic functionality is provided, with a clean and intuitive user interface and additional recombination schemes. A short description of the algorithm and examples of its use are given.

PACS:
Keywords: Jet Algorithms; QCD

1 Introduction

This paper is intended to be an introduction to the use of KtJet. For a detailed explanation of the $K_\perp$ clustering algorithm, its properties, and the physics motivation behind it, the reader is referred to [1] and references therein. The KtJet package implements all the features of the Fortran implementation of the $K_\perp$ algorithm [2]. The philosophy throughout has been to design an interface that users of the Fortran code will recognise, but which exploits the advantages of object-oriented design. Therefore the names and functions of the input parameters are retained wherever possible. The KtJet library, examples of its use and detailed documentation are all available on the KtJet website, `http://www.ktjet.org/`, which also provides a link to a CVS repository from which the latest version of the source code may be obtained.

2 The $K_\perp$ algorithm

The $K_\perp$ algorithm can function in several distinct ways depending on the nature of the colliding beams and the physics to be studied. We deal first with hadron-hadron and lepton-hadron collisions. There are two modes of operation, the 'inclusive' and 'exclusive' modes. The difference between the two cases, described in sections 2.1 and 2.2, is in the definition of the hard final state jets, and the separation of these jets from the beam remnants.

For the analysis of e^+e^- data, in contrast, there is of course no concept of beam remnants. The algorithm proceeds in a very similar way to that employed in a jet substructure analysis. We deal with these two cases together in section 2.3.

2.1 The inclusive mode

The algorithm proceeds as follows:

1. For every final state object[1] h_k and for every pair h_k and h_l, compute the resolution variables d_{kB} and d_{kl}. The precise definition of these variables can be chosen by the user (using the parameter `angle`), and will be described in detail in section 2.4. They always have the property that in the small angle limit they reduce respectively to the squared relative transverse momentum of the object with respect to the beam direction, and the squared relative transverse momentum of one object with respect to the other.

[1]Final state objects could be, for example, partons, hadrons, calorimeter cells, tracks etc.

$$
\begin{aligned}
d_{kB} &\simeq E_k^2\theta_{kB}^2 && (1)\\
&\simeq k_{\perp kB}^2, \text{for } \theta \to 0\\
d_{kl} &\simeq \min(E_k^2, E_l^2)\theta_{kl}^2 && (2)\\
&\simeq k_{\perp kl}^2, \text{for } \theta_{kl} \to 0
\end{aligned}
$$

At this stage, a dimensionless parameter R is introduced [3], which plays a radius-like role in defining the extent of the jets. This is usually set to 1.0.

2. Scale the d_{kB} by R^2

$$d_k = d_{kB}R^2 \tag{3}$$

3. Find the smallest value among the d_k and d_{kl}. If a d_{kl} is the smallest, h_k and h_l are combined into a single object with momentum $p_{(kl)}$ according to a user specified recombination scheme (parameter `recom`) which will be described in section 2.5. As an example, `recom = 1` would correspond to 4-vector addition. If a d_k is the smallest, object k is defined to be a jet and is removed from the list of objects to be merged.

4. Repeat until all objects have been included in jets.

2.2 The exclusive mode

In this mode, the algorithm separates the 'hard final state' from the soft 'beam remnants' explicitly. Jets are defined in the hard final state by a stopping parameter d_{cut} with dimensions of energy squared.

1. d_{kB} and d_{kl} are defined as in section 2.1.

2. Find d_{min}, the smallest value among the d_{kB} and d_{kl}. If $d_{min} > d_{cut}$, all remaining objects in the event will be classified as jets, and the algorithm is complete.

3. If a d_{kl} is the smallest, h_k and h_l are combined into a single object with momentum $p_{(kl)}$ according to a user specified recombination scheme (parameter `recom`). If a d_{kB} is the smallest, object k is included in a 'beam jet' and removed from the list. Go back to (2).

Note that the stopping parameter d_{cut} defines the hard scale of the process ($\Lambda_{QCD}^2 \ll d_{cut} \leq s$, where $\sqrt{s}$ is the centre-of-mass energy). One can therefore think of the hard subprocess being factorised from the low-$p_\perp$ scattering fragments, which are thrown away into the beam jets.

Instead of specifying the stopping scale d_{cut}, one can choose to stop merging when a given number of jets is reached.

2.3 Subjet analysis and e^+e^- mode

Once the hadronic final state has been decomposed into jets, the structure of the jets themselves can be investigated. This is commonly known as a subjet analysis. The procedure is physically (and practically) identical to that employed in the analysis of an e^+e^- event. In the e^+e^- case, all final state objects are used as input to the algorithm. In a subjet analysis on a particular jet, only the particles within that jet are used as input. The steps are as follows;

1. Define a resolution parameter

$$y_{cut} = Q_0^2/E_{cut}^2. \tag{4}$$

   The parameter E_{cut}^2 may be chosen by the user, but is conventionally taken to be the square of the total energy of the e^+e^- event (or the p_t of the jet) in the frame in which the algorithm is run. These are the default settings in KtJet.

2. For each pair of objects h_k and h_l, construct the rescaled resolution variable y_{kl}

$$y_{kl} = d_{kl}/E_{cut}^2 \tag{5}$$

   where d_{kl} is defined as in section 2.1.

3. Find y_{min}, the smallest of the y_{kl}. If $y_{min} < y_{cut}$, h_k and h_l are combined into a single object with momentum $p_{(kl)}$ according to a user specified recombination scheme (parameter `recom`).

4. Repeat the procedure until all pairs of objects have $y_{kl} > y_{cut}$. The remaining objects are called subjets.

As described in section 2.2, one can choose to stop merging when a given number of jets is reached, instead of specifying the stopping scale y_{cut}.

2.4 Jet resolution variables

The small-angle behaviour of the resolution variables d_{kB} and d_{kl} given in equations 1 and 2 controls the behaviour of the $K_\perp$ algorithm in the soft and collinear limits.

Away from the collinear limit, the user has a choice of angular definitions, controlled by the parameter `angle`. Note that the resolution variables are defined such that they remain monotonic functions of angle at large angles. Here, and throughout this paper, we use η to denote true rapidity, i.e. $\eta = \frac{1}{2} \ln \frac{E+p_z}{E-p_z}$.

2.4.1 The angular scheme, `angle = 1`

$$
\begin{aligned}
d_{kB} &= 2E_k^2(1 - cos\theta_{kB}), \\
d_{kl} &= 2\min(E_k^2, E_l^2)(1 - cos\theta_{kl}).
\end{aligned} \tag{6}
$$

This definition was originally proposed for jets in DIS [4] (where the $K_\perp$ algorithm should be implemented in the Breit frame).

2.4.2 The ΔR scheme, `angle = 2`

$$
\begin{aligned}
d_{kB} &= p_{tk}^2, \\
d_{kl} &= \min(p_{tk}^2, p_{tl}^2)R_{kl}^2,
\end{aligned} \tag{7}
$$

where

$$
R_{kl}^2 = (\eta_k - \eta_l)^2 + (\phi_k - \phi_l)^2. \tag{8}
$$

This definition corresponds to that used in cone algorithms, and is the most common choice for hadron-hadron collisions.

2.4.3 The QCD emission scheme, `angle = 3`

An alternative definition of R_{kl}^2, motivated by the form of the QCD matrix elements for multi-parton emissions:

$$
R_{kl}^2 = 2[cosh(\eta_k - \eta_l) - cos(\phi_k - \phi_l)]. \tag{9}
$$

2.5 Recombination schemes

The recombination scheme defines how two objects h_k and h_l are merged into a single object with 4-momentum $p_{(kl)}$. Five schemes are available, selected by the parameter `recom`.

2.5.1 The E scheme, `recom=1`

Simple 4-vector addition:

$$p_{kl} = p_k + p_l. \tag{10}$$

This procedure results in massive final state jets.

2.5.2 The p_t scheme, `recom=2`

$$\begin{aligned} p_{t(kl)} &= p_{tk} + p_{tl}, \\ \eta_{kl} &= \frac{p_{tk}\eta_k + p_{tl}\eta_l}{p_{t(kl)}}, \\ \phi_{kl} &= \frac{p_{tk}\phi_k + p_{tl}\phi_l}{p_{t(kl)}}. \end{aligned} \tag{11}$$

This definition constrains only the 3 spatial components of the object's 4-vector. The combined object is made massless by setting its energy equal to the magnitude of its 3-momentum. If massive objects are input (for instance particle 4-vectors from a simulated event) they are made massless in the same way before the d_{kB} and d_{kl} are calculated. Compare the E_t scheme (section 2.5.4).

2.5.3 The p_t^2 scheme, `recom=3`

$$\begin{aligned} p_{t(kl)} &= p_{tk} + p_{tl}, \\ \eta_{kl} &= \frac{p_{tk}^2\eta_k + p_{tl}^2\eta_l}{p_{tk}^2 + p_{tl}^2}, \\ \phi_{kl} &= \frac{p_{tk}^2\phi_k + p_{tl}^2\phi_l}{p_{tk}^2 + p_{tl}^2}. \end{aligned} \tag{12}$$

This definition constrains only the 3 spatial components of the object's 4-vector. The energy is made equal to to the magnitude of its 3-momentum, thus making the combined object massless. Note also that this definition of the p_t^2 scheme is that used in the Fortran implementation of the algorithm [2]. It is not equivalent to the monotonic p_t^2 scheme defined in equations 16 and 17 of reference [1]. We discuss the issue of monotonicity further in section 3.1.1.

2.5.4 The E_t scheme, `recom=4`

$$E_{t(kl)} = E_{tk} + E_{tl},$$

$$
\begin{aligned}
\eta_{kl} &= \frac{E_{tk}\eta_k + E_{tl}\eta_l}{E_{t(kl)}}, \\
\phi_{kl} &= \frac{E_{tk}\phi_k + E_{tl}\phi_l}{E_{t(kl)}}.
\end{aligned} \tag{13}
$$

For massless input objects this definition is identical to the p_t scheme (section 2.5.2). It differs solely in the way it deals with massive input objects. If massive objects are input, the p_t scheme uses their transverse momentum, whereas the E_t scheme uses the transverse energy $E \sin\theta$. This can have a significant effect for low E_t jets, particularly for steeply falling distributions close to a cut-off value. All combined objects are massless in both cases.

2.5.5 The E_t^2 scheme, `recom=5`

$$
\begin{aligned}
E_{t(kl)} &= E_{tk} + E_{tl}, \\
\eta_{kl} &= \frac{E_{tk}^2\eta_k + E_{tl}^2\eta_l}{E_{tk}^2 + E_{tl}^2}, \\
\phi_{kl} &= \frac{E_{tk}^2\phi_k + E_{tl}^2\phi_l}{E_{tk}^2 + E_{tl}^2}.
\end{aligned} \tag{14}
$$

This bears the same relationship to the p_t^2 scheme that the E_t scheme bears to the p_t. All combined objects are massless.

3 Practical implementation of the algorithm

KtJet uses the `HepLorentzVector` class of the CLHEP package [7]. The input 4-vectors (the objects on which the algorithm will be run) may be `HepLorentzVector`s or `KtLorentzVector`s. The `KtLorentzVector` class, which inherits from `HepLorentzVector`, carries an internal index which allows the user to determine, for example, to which final state jet a particular input particle belongs. Examples of the use of `KtLorentzVector` are given in section 3.2. The output 4-vectors (the jets) are instantiated as `KtLorentzVector`s. The `KtLorentzVector` class has the constructors

```
KtLorentzVector(const HepLorentzVector &);
KtLorentzVector(const KtLorentzVector &);
KtLorentzVector(float Px, float Py, float Pz, float E);
```

type	Beam	Comments
1	ee	
2	ep	p in $-z$ direction
3	pe	p in $+z$ direction
4	pp	

Table 1: Possible input values for `type`.

The first step in running the algorithm is the creation of a `KtEvent` object. There are separate constructors to run the algorithm in inclusive and exclusive modes respectively:

```
KtEvent(const std::vector<KtLorentzVector> particles &, int type,
int angle, int recom, float rparameter);
```

```
KtEvent(const std::vector<KtLorentzVector> particles &, int type,
int angle, int recom);
```

There are also versions of these constructors taking vectors of `HepLorentzVectors` instead of `KtLorentzVectors`. The parameter `type` should be set to the colliding beam type, as defined in table 1. The parameters `angle` and `recom` are described in sections 2.4 and 2.5. `rparameter` is the parameter R defined in section 2.1 and should almost always be set to 1.0, although values smaller than 1.0 have been suggested for certain applications [5, 6].

3.1 KtEvent methods

For the inclusive case, the final state jets are now fully defined, and the jets can be recovered using the methods described below. In the exclusive case, the final state jets are defined by setting either the stopping parameter d_{cut}:

```
void findJetsD(float dcut);
```

or by forcing the final state to decompose into N jets:

```
void findJetsN(int N);
```

The d_{min} value at which the final state changes from $N+1$ to N jets can be recovered using the method

```
float getDMerge(int N) const;
```

In the case of an e^+e^- analysis, the variable y_{cut} may be used. The final state jets are defined by the method

```
void findJetsY(float yCut);
```

and the y_{min} value at which the final state changes from $N+1$ to N jets can be recovered using the method

```
float getYMerge(int N) const;
```

By default, y_{cut} is defined as in equation 4, where E_{cut} is taken to be the total energy in the event. The E_{cut} value can be set by the user if required:

```
void setECut(float ECut);
```

The `KtEvent` object has the following methods for recovering its final state jets:

```
std::vector<KtLorentzVector> getJets() const;
```
Returns jets without sorting.

```
std::vector<KtLorentzVector> getJetsE() const;
```
Returns jets in order of decreasing energy.

```
std::vector<KtLorentzVector> getJetsEt() const;
```
Returns jets in order of decreasing transverse energy.

```
std::vector<KtLorentzVector> getJetsPt() const;
```
Returns jets in order of decreasing transverse momentum.

```
std::vector<KtLorentzVector> getJetsRapidity() const;
```
Returns jets in order of decreasing rapidity.

```
std::vector<KtLorentzVector> getJetsEta() const;
```
Returns jets in order of decreasing pseudorapidity.

```
KtLorentzVector getJet(const KtLorentzVector &) const;
```
Returns the final state jet which contained the input particle `KtLorentzVector`. Note that this method is only available if `KtLorentzVector`s were used as input in the `KtEvent` constructor.

```
std::vector<const KtLorentzVector*> & getConstituents() const;
```
Returns a vector of pointers to all the input objects in the `KtEvent`.

```
std::vector<KtLorentzVector> copyConstituents() const;
```
Returns a vector of copies of the input objects in the `KtEvent`.

```
int getNConstituents() const;
```
Returns the number of objects in the `KtEvent`.

The parameters of a particular `KtEvent` can be retrieved using the methods

```
float getETot() const;
int getType() const;
int getAngle() const;
int getRecom() const;
float getECut() const;
bool isInclusive() const;
```

3.1.1 Monotonicity

Although in practice most users will not have to face the issue, it is worth bearing in mind that recombination schemes 1 to 5 are not guaranteed to lead to monotonic resolution variables, i.e. it is NOT necessarily true that $\min\{d_{kB}, d_{kl}\} \leq \min\{d'_{kB}, d'_{kl}\}$, where d_k and d'_k are the resolution variables before and after recombination respectively. This means that, physically, the question "How many jets are there at a particular scale d?" may not have a unique answer. In KtJet, as should be clear from section 2.2, for a particular value of d_{cut} set by `findJetsD`, the largest value of N (i.e. the first place at which $d_{min} > d_{cut}$) will be returned by `getNJets`. Similarly, there may be no value of d_{cut} for which a particular event has N jets. There will however always be a d_{min} value at which the event changed from $N+1$ to N jets, which is the value returned by `getDMerge`. The above discussion also applies to the y variables and associated methods.

3.2 KtLorentzVector Methods

These methods would normally be used to investigate the structure and constituents of final state jets defined in a particular `KtEvent`.

```
const std::vector<const KtLorentzVector*> & getConstituents() const;
```

Returns a reference to the vector of pointers to all the objects in the KtLorentzVector.

`std::vector<KtLorentzVector> copyConstituents() const;`
Returns a vector of copies of the objects in the KtLorentzVector.

`int getNConstituents() const;`
Returns the number of objects in the KtLorentzVector.

`bool contains (const KtLorentzVector &) const;`
Check if a jet contains a particular object. For example,
`if (JET.contains(PARTICLE)) { /* do something */ }`
where JET is the KtLorentzVector of a final state jet and PARTICLE is the KtLorentzVector of an input object in the KtEvent.

`KtLorentzVector & operator+= (const KtLorentzVector &);`
Adds a KtLorentzVector constituent to a jet using the E scheme (4-vector addition) and maintains an internal record of constituents (so for example the getConstituents method will work on the resulting KtLorentzVector).

`void add (const KtLorentzVector &, int recom);`
Adds a KtLorentzVector constituent to a jet using any of the available recombination schemes and maintains an internal record of constituents.

Note that the momentum of a jet (KtLorentzVector) will be given according to the recombination scheme used in its construction. If the user wishes to reconstruct the momenta according to a different recombination scheme (for example, to recover the mass for jets which were found using a massless recombination scheme) the getConstituents() or copyConstituents() methods may be used. The constituents can then be recombined in a new scheme using the add method.

4 Subjet analysis

A subjet analysis is performed on a particular final state jet by constructing a new KtEvent object:

```
KtEvent(const KtLorentzVector jet &, int type, int angle, int
recom);
```

The subsequent analysis and methods are identical to those described above for the e^+e^- case: for example, subjets can be defined using the findJetsY or findJetsN methods. E_{cut} is taken to be the transverse momentum of the jet, although this can be changed by the user using the setECut method.

5 Advanced features of the KtLorentzVector class

The KtLorentzVector class has several features which are used internally by KtJet, but which may be useful for the user.

A KtLorentzVector can be constructed in two different ways. When instantiated using the constructor KtLorentzVector(const HepLorentzVector &) (or KtLorentzVector(float Px,float Py,float Pz,float E)), the KtLorentzVector is simply a copy of a single HepLorentzVector with an internal index added, i.e. it has no constituents. This index is then used, for example, in the getJet method, to ascertain to which final state jet a particular input KtLorentzVector belongs. There are two methods which compare the *index* of two KtLorentzVectors:

```
bool operator== (const KtLorentzVector &) const;
bool operator!= (const KtLorentzVector &) const;
```

These would be useful if, for example, one wanted to ascertain whether the highest p_t particle in a particular jet was a given input KtLorentzVector. A KtLorentzVector may also be instantiated using the constructor

```
KtLorentzVector();
```

The KtLorentzVector can then be built up by adding other KtLorentzVectors to it using either the assignment operator += or the add method described in section 3.2. The KtLorentzVector will then also carry a list of pointers to the constituent KtLorentzVectors. To find out whether a KtLorentzVector has constituents, use the method

```
bool isJet() const;
```

which will be true for a KtLorentzVector with a constituent list.

6 Using KtJet

In this section we demonstrate a simple example analysis using the KtJet library. We run the inclusive $K_\perp$ algorithm on a proton-proton collision and do a subjet analysis on the highest E_t jet in the event.

```
// This header file must be included.
#include "KtJet.h"

std::vector<KtLorentzVector> jetvec;

// Loop over input particles.
for (int i=0; i<npart; i++) {
  KtLorentzVector  r = KtLorentzVector(p[i][0],p[i][1],p[i][2],p[i][3]);
  jetvec.push_back(r);
}
// We now have a vector of KtLorentzVector, jetvec,
// containing all the particle 4-vectors, p, in the event.

// Run the inclusive KT algorithm in PP mode using the covariant E-scheme
// (type=4, angle=2, recom=1, rparameter=1).
KtEvent ev(jetvec,4,2,1,1.0);

// Get jets, sorted in Et.
std::vector<KtLorentzVector> jetsEt = ev.getJetsEt();

// Perform subject analysis on highest Et jet (angle=2, recom=1).
KtEvent jet1(jetsEt[0],2,1);

// Decompose jet1 into 2 subjets.
jet1.findJetsN(2);

// Get the KtLorentzVectors of the 2 subjets
std::vector<KtLorentzVector> subjetsEt = jet1.getJetsEt();

// Write out the pseudorapidity of the highest Et subjet.
cout << subjetsEt[0].eta() << endl;
```

7 Optimisation issues

The time taken to process events grows with the multiplicity n of the input objects as n^3. As the multiplicity grows with energy, and current and future colliders have ever higher event rates, timing issues are therefore of increasing practical importance.

A few compilation options are worth considering if speed is critical. A compile option to switch from single to double precision arithmetic is available, but single precision is used by default as this increases the speed by around 10%. In gcc, the option `-O2` is essential; higher or lower optimisation levels seriously degrade performance. Choice of architecture is also important. For example using the gcc compiler flag `-march=athlon` on an Athlon processor gives a performance gain of around 5%. More details on performance issues are given at `http://www.ktjet.org/`.

8 Adding new functionality

KtJet interfaces with its recombination and jet resolution schemes via purely abstract base classes called `KtRecom` and `KtDistance` respectively. The packaged code comes supplied with 5 Recombination and 3 jet resolution schemes, shown in table 2 along with their `angle` and `recom` flags.

Users can define their own recombination and jet resolution scheme classes which inherit from the base classes `KtRecom` and `KtDistance` and use these instead of the supplied schemes.

To use their own scheme the user needs to pass pointers to the base classes, instantiated as their own scheme objects, in the `KtEvent` constructor instead of the integer flags.

As an example, if the user defines a new jet resolution scheme in the class `KtDistanceNew`, and wishes to use the E recombination scheme in an ep collision, the inclusive `KtEvent` constructor should be called as follows:

```
KtDistance* distance_scheme = new KtDistanceNew(2);
KtEvent ev(particles, 2, distance_scheme, 1, rparameter);
```

If the user defines a new recombination scheme in the class `KtRecomNew`, and wishes to use the `angle=2` scheme in a pp collision, the inclusive `KtEvent` constructor should be called as follows:

```
KtRecom* recom_scheme = new KtRecomNew();
```

Flag	Class Name
`angle = 1`	`KtDistanceAngle(int type)`
`angle = 2`	`KtDistanceDeltaR(int type)`
`angle = 3`	`KtDistanceQCD(int type)`
`recom = 1`	`KtRecomE`
`recom = 2`	`KtRecomPt`
`recom = 3`	`KtRecomPt2`
`recom = 4`	`KtRecomEt`
`recom = 5`	`KtRecomEt2`

Table 2: The names of the jet resolution and recombination classes included in KtJet.

```
KtEvent ev(particles, 4, 2, recom_scheme, rparameter);
```

Users are invited to submit any such additions to the authors for inclusion in future releases.

Acknowledgements

We would like to thank Mike Seymour for useful discussions, suggestions and encouragement throughout the project. This work was funded in the UK by PPARC and HEFCE.

References

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

[2] The Fortran code is also available from `http://www.ktjet.org/`.

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

[4] S. Catani, Y. L. Dokshitzer and B. R. Webber, Phys. Lett. B **285** (1992) 291.

[5] M. H. Seymour, Z. Phys. C **62** (1994) 127.

[6] M. H. Seymour, arXiv:hep-ph/0007051.

[7] L. Lönnblad, Comp. Phys. Comm. 84 (1994) 307. `http://wwwinfo.cern.ch/asd/lhc++/clhep/`