Alberta Thy 03-05

Optimal Jet Finder (v1.0 C++)

S. Chumakov

Department of Physics, M. V. Lomonosov Moscow State University, Moscow 119992, Russia

E. Jankowski

Department of Physics, University of Alberta, Edmonton, AB, T6G 2J1, Canada

F. V. Tkachov [1]

Institute for Nuclear Research of RAS, Moscow 117312, Russia

Abstract

We describe a C++ implementation of the Optimal Jet Definition for identification of jets in hadronic final states of particle collisions. We explain interface subroutines and provide a usage example. The source code is available from
`http://www.inr.ac.ru/~ftkachov/projects/jets/`

Keywords: hadronic jets, jet finding algorithms
PACS: 13.87.-a, 29.85.+c

[1] Corresponding author: ftkachov@ms2.inr.ac.ru

PROGRAM SUMMARY

Title of program: Optimal Jet Finder (v1.0 C++)

Catalogue identifier: (supplied by the Publisher)

Distribution format: (supplied by the Program Library)

Computer: any computer with a standard C++ compiler

Tested with:

(1) GNU gcc 3.4.2, Linux Fedora Core 3, Intel i686;

(2) Forte Developer 7 C++ 5.4, SunOS 5.9, UltraSPARC III+;

(3) Microsoft Visual C++ Toolkit 2003 (compiler 13.10.3077, linker 7.10.30777, option /EHsc), Windows XP, Intel i686.

Programming language used: C++

Memory required: ∼1 MB (or more, depending on the settings)

Number of bytes in distributed program, including examples and test data: ∼100 KB

Keywords: hadronic jets; jet finding algorithms

Nature of physical problem
Analysis of hadronic final states in high energy particle collision experiments often involves identification of hadronic jets. A large number of hadrons detected in the calorimeter is reduced to a few jets by means of a *jet finding algorithm*. The jets are used in further analysis which would be difficult or impossible when applied directly to the hadrons. Reference [1] provides brief introduction to the subject of jet finding algorithms and a general review of the physics of jets can be found in [2].

Method of solution
The software we provide is an implementation of the so-called *Optimal Jet Definition (OJD)*. The theory of OJD was developed in [3], [4], [5]. The desired jet configuration is obtained as the one that minimizes Ω, a certain function of the input particles and jet configuration. A FORTRAN 77 implementation of OJD is described in [6].

Restrictions on the complexity of the program
Memory required by the program is proportional to the number of particles in the

input $\times$ the number of jets in the output. For example, for 650 particles and 20 jets $\sim$300KB memory is required.

Typical running time
The running time (in the running mode with a fixed number of jets) is proportional to the number of particles in the input $\times$ the number of jets in the output $\times$ times the number of different random initial configurations tried (`ntries`). For example, for 65 particles in the input and 4 jets in the output, the running time is $\sim 4 \cdot 10^{-3}$s per try (Pentium 4 2.8GHz).

References
[1] D. Yu. Grigoriev, E. Jankowski, F. V. Tkachov, Phys. Rev. Lett. **91**, 061801 (2003).
[2] R. Barlow, Rep. Prog. Phys. **36**, 1067 (1993).
[3] F. V. Tkachov, Phys. Rev. Lett. **73**, 2405 (1994); Erratum, **74**, 2618 (1995).
[4] F. V. Tkachov, Int. J. Mod. Phys. **A12**, 5411 (1997).
[5] F. V. Tkachov, Int. J. Mod. Phys. **A17**, 2783 (2002).
[6] D. Yu. Grigoriev, E. Jankowski, F. V. Tkachov, Comput. Phys. Commun. **155**, 42 (2003).

Contents

1 Introduction

This paper introduces a C++ implementation of Optimal Jet Finder, a jet finding algorithm for use in high energy physics data analysis. The current version is based on the same algorithm and physics motivations as the previous FORTRAN 77 implementation published in [1], and the reader is referred there for more details.

The input of the algorithm is *an event* [2]: a collection of n *particles* from the detector (or n hit detector cells), indexed with $a = 1, 2, 3, ..., n$. Each particle is characterized by its energy, E_a, and its direction described by the standard angles θ_a, φ_a or equivalently by transverse energy, $E_a^\perp$, pseudorapidity, η_a, and the angle φ_a. The a-th particle in the input is assigned the 4-momentum p_a:

$$p_a = E_a \cdot (1,\, \sin\theta_a \cos\varphi_a,\, \sin\theta_a \sin\varphi_a,\, \cos\theta_a) \tag{1}$$

or

$$p_a = E_a^\perp \cdot (\cosh\eta_a,\, \cos\varphi_a,\, \sin\varphi_a,\, \sinh\eta_a)\,. \tag{2}$$

depending on which parameters are used to describe the particles. The output of the program is a set of N jets, indexed with $j = 1, 2, 3, ..., N$. The jet configuration is described by *recombination matrix* $\{z_{aj}\}$ components of which satisfy:

$$0 \le z_{aj} \le 1 \quad \text{for} \quad \text{all} \quad a, j, \tag{3}$$

$$\sum_{j=1}^{N} z_{aj} \le 1 \quad \text{for} \quad \text{all} \quad a. \tag{4}$$

The number z_{aj} gives the fraction of the a-th particle which goes into formation of the j-th jet. Each z_{aj} can take any value between 0 and 1. The final value of the recombination matrix $\{z_{aj}\}$ is the result of the algorithm. The 4-momentum q_j of the j-th jet is defined as

$$q_j = \sum_{a=1}^{n} z_{aj} p_a. \tag{5}$$

The final (optimal) jet configuration is the one that minimizes the value of some function $\Omega(\{z_{aj}\})$ depending on the recombination matrix $\{z_{aj}\}$ and all p_a as parameters.

[2] The following summary of the algorithm is excerpted from [1].

The definition of Ω follows some intermediate sub-definitions. The part of the a-th particle that does not go into formation of any jet:

$$\overline{z}_a \equiv 1 - \sum_{j=1}^{N} z_{aj}. \tag{6}$$

The rest of the definitions are given separately for spherical kinematics (lepton collisions) and for cylindrical kinematics (hadron collisions).

Spherical kinematics. Overall energy left outside jets E_{soft}, called *soft energy*:

$$E_{\text{soft}} \equiv \sum_{a=1}^{n} \overline{z}_a E_a. \tag{7}$$

The function Y, called *fuzziness*:

$$Y \equiv 2 \sum_{j=1}^{N} q_j \widetilde{q_j}, \tag{8}$$

where $\widetilde{q_j}$ is light-like ($\widetilde{q_j}^{\,2} = 0$) 4-direction defined:

$$\widetilde{q_j} \equiv (1,\, \sin\theta_j \cos\varphi_j,\, \sin\theta_j \sin\varphi_j,\, \cos\theta_j)\,, \tag{9}$$

with

$$\cos\theta_j \equiv \frac{(q_j)_z}{\sqrt{(q_j)_x^2 + (q_j)_y^2 + (q_j)_z^2}}, \tag{10}$$

$$\cos\varphi_j \equiv \frac{(q_j)_x}{\sqrt{(q_j)_x^2 + (q_j)_y^2}}, \tag{11}$$

$$\sin\varphi_j \equiv \frac{(q_j)_y}{\sqrt{(q_j)_x^2 + (q_j)_y^2}}. \tag{12}$$

Cylindrical kinematics. The soft energy is the overall *transverse* energy left outside the jets

$$E_{soft} \equiv \sum_{a=1}^{n} \overline{z}_a E_a^{\perp}. \tag{13}$$

The fuzziness Y is defined again by (8) with $\widetilde{q_j}$, light-like ($\widetilde{q_j}^2 = 0$) 4-direction given by:

$$\widetilde{q_j} \equiv (\cosh\eta_j, \cos\varphi_j, \sin\varphi_j, \sinh\eta_j)\,, \tag{14}$$

where

$$\eta_j \equiv \frac{\sum_{a=1}^{n} z_{aj} E_a^{\perp} \eta_a}{\sum_{a=1}^{n} z_{aj} E_a^{\perp}}, \tag{15}$$

and $\cos\varphi_j$, $\sin\varphi_j$ given by (11), (12).

Finally, **in both cases**, Ω is a linear combination of Y and E_{soft} with the parameter R weighting their relative contribution:

$$\Omega\left(\{z_{aj}\}\right) \equiv \frac{1}{R^2} Y + E_{soft}. \tag{16}$$

For a fixed number of jets, the program starts with some initial value of the recombination matrix z_{aj}, for example, chosen randomly, and finds a local minimum of the $\Omega\left(\{z_{aj}\}\right)$ function with respect to $\{z_{aj}\}$. Several (random) initial values of $\{z_{aj}\}$ are used (the parameter n_{tries}), and the corresponding local minima may differ; the value of the recombination matrix $\{z_{aj}\}$ that gives the smallest local minimum is the final jet configuration. (If the initial value of $\{z_{aj}\}$ is not chosen randomly, it is useless to do the minimization procedure more than once as the minimization algorithm is deterministic.)

If the number of jets is to be determined in the process of jet reconstruction, the procedure described above can be repeated for different number of jets, N, each time. The final jet configuration is the one that satisfies

$$\Omega\left(\{z_{aj}\}\right) < \omega_{\text{cut}} \tag{17}$$

with the minimal number of jets, N. (The above condition will be satisfied for a sufficiently large number of jets.) The parameter ω_{cut} is a (small) positive number, analogous to the jet resolution parameter of conventional recombination algorithms.

The current implementation is based on the verification version of Optimal Jet Finder [2], whereas the published FORTRAN 77 version (ojf_014) was based on an earlier Component Pascal code. In particular, the current version offers a somewhat more fine-grained control of the rounding errors. A correspondingly updated FORTRAN 77 version will be published in the due course.

The program is self-contained: it requires only standard C++ libraries and should compile with any standard C++ compiler.

The current implementation has been verified against the FORTRAN 77 version: ojf_015, available from [3]. The details can be found in section 5.

2 User interface classes and methods

All classes are contained within the `OptimalJetFinder namespace`. In this section, we describe several classes and methods most likely to be needed by the user. The reader may find it more practical to study `example.cpp` in the next section before browsing through this section.

2.1 class ***Event***

This class represents a high energy physics event: a collection of input particles (calorimeter cells, preclusters, etc.)

- `Event( Kinematics k )`
  - constructor. `Kinematics = enum { sphere, cylinder }`, where `sphere` applies to the center of mass kinematics (lepton collisions), and `cylinder` applies to the cylindrical kinematics of hadron collisions.
- `void AddParticleRaw( double px, double py, double pz )`
  adds a particle to the event. `px`, `py`, `pz` are the components of the momentum of the particle in arbitrary units.
- `void AddParticle( double E, double theta, double phi )`
  adds a particle to the event. `E` is the energy of the particle in arbitrary units and the standard angles `theta` and `phi` describe the direction of the particle. The angles are measured in degrees.
- `void Normalize()`
  has to be called before jets are searched. It normalizes the 4-momenta of the particles so that the sum of all energies or transverse energies of all particles is equal to one.
- `void Clear()`
  removes all particles from the event and releases memory accordingly.
- `Kinematics GetKinematics() const`
  returns the type of kinematics; see the constructor above.
- `Particle* GetFirst() const`
  returns the pointer to the first particle in the event or 0 if there are no particles.
- `bool IsNormalized() const`

returns `true/false` depending whether the event is already normalized; see `Normalize()` above.

- `double GetXEnergy() const`
  returns the sum of energy (for the spherical kinematics) or sum of transverse energy (for the cylindrical kinematics) of all particles in the event.
- `int GetNumber() const`
  returns the number of particles in the event.

2.2 class JetSearch

This is a simple jet search class.

- `JetSearch(const Event* P, double R, int ntries = 10)`
  - constructor. Initializes jet search. `P` is a pointer to the object of the `Event` class. `R` is the radius parameter R of eq.(20) in [1]. `ntries` is the number of different random initial jet configurations tried.
- `bool FindJetsForFixedNJets(int njets)`
  finds the final jet configuration with the number of jets equal to `njets` and returns `true` if successful and `false` otherwise. For each "try", it starts with a random initial jet configuration and finds a local minimum of Ω function, eq. (20) in ref. [1]. After a number of tries (set with
  `void SetNTries( int ntries )`; default = 10) the best jet configuration is chosen, i.e. the one that gives the smallest value of Ω (the deepest local minimum).
- `int FindJetsForOmegaCut(double omegaCut)`
  finds the final jet configuration for `omegaCut`=$\omega_{\rm cut}$ of relation (21) in [1] and returns the number of jets in the final jet configuration or 0 if the search is not successful. It runs
  `bool JetSearch::FindJetsForFixedNJets(int njets)` increasing the number of jets between the values set by
  `void JetSearch::SetNJetsBegin( int nBegin )`
  and `void JetSearch::SetNJetsEnd( int nEnd )`.
  The final jet configuration is the one with the smallest number of jets for which the value of Ω function (eq. (20) in ref. [1]) is smaller than $\omega_{\rm cut}$ parameter.
- `Jets* GetJets() const`
  can be used to access the final jet configuration.
- `void SetNTries( int ntries )`
  sets the number of different random initial jet configurations tried.
- `int GetNTries() const`
  returns the number of different random initial jet configurations tried.
- `void SetMaxIter( int MaxIter )`
  sets the maximal number of iterations in the minimization algorithm. The

default value is 2000. If the local minimum is not found within the maximal number of iterations the current jet search is terminated and `bool FindJetsForFixedNJets(int njets)` returns `false`, or `int FindJetsForOmegaCut(double omegaCut)` returns 0.

- `int GetMaxIter() const`
  returns the maximal number of iterations in the minimization algorithm.
- `void SetNJetsBegin( int nBegin )`
  sets the initial number of jets in `int FindJetsForOmegaCut(double omegaCut)`.
- `int GetNJetsBegin() const`
  returns the initial number of jets in `int FindJetsForOmegaCut(double omegaCut)`.
- `void SetNJetsEnd( int nEnd )`
  sets the maximal allowed number of jets in `int FindJetsForOmegaCut(double omegaCut)`.
- `int GetNJetsEnd() const`
  returns the maximal allowed number of jets in `int FindJetsForOmegaCut(double omegaCut)`.

2.3 class `Particle`

Objects of this class correspond to particles (or calorimeter cells, preclusters, etc.) in the event. In most cases, the user will not need to create instances of this class directly, but only use pointers to this class to access information about particles.

- `Particle(int Label, Kinematics k, const Event* P)`
  - constructor. In most cases, the user does not need to call the constructor directly but only through `Event::AddParticleRaw(double px, double py, double pz)` or `Event::AddParticle(double px, double py, double pz)`. If particles are entered using either of the two just mentioned methods, the first particle has label 1, the next 2, etc. Otherwise the `label` has an arbitrary value specified by the user.
- `double GetE() const`
  returns the energy of the particle in the same units as used in the input.
- `double GetPx() const`
- `double GetPy() const`
- `double GetPz() const`
  return the x(y,z)-component of the momentum of the particle in the same units as used in the input.
- `double GetXEnergy() const`
  returns the energy of the particle (for the spherical kinematics) or transverse

energy of the particle (for the cylindrical kinematics) in the same units as used in the input.

- `double GetXEta() const`
  returns the standard angle θ in degrees for the spherical kinematics or pseudorapidity η for the cylindrical kinematics.
- `double GetPhi() const`
  returns the standard angle ϕ in degrees.
- `double GetESoft() const`
  for the spherical kinematics, it returns the fraction of the energy of the particle that does not belong to any jet; for the cylindrical kinematics, it returns the fraction of the transverse energy of the particle that does not belong to any jet; in normalized units (see `Event::Normalize()`).
- `double GetFractionInJet(int j) const`
  returns the fraction of the particle that belongs to the `j`-th jet.
- `int GetLabel() const`
  returns the label of the particle. If particles are entered using `Event::AddParticleRaw(double px, double py, double pz)` or `Event::AddParticle(double px, double py, double pz)`, the first particle has label 1, the next 2, etc. Otherwise, the label has the value that was used in the constructor call.
- `Particle* GetNext() const`
  returns the pointer to the next particle in the event. This method allows to loop over all particles in the event.

2.4 class `Jets`

This class represents a configuration of jets. In most cases, the user will not need to create instances of this class directly, but only use pointers to this class to access information about the jet configuration.

- `Jets(int njets, const Event* P, double R)`
  - constructor. `njets` is the number of jets, `P` is a pointer to the object of the class `Event`, `R` is the radius parameter R of eq.(20) in [1].
- `const Event* GetEvent() const`
  returns the pointer the event with which the jets are associated.
- `double GetR() const`
  returns the radius parameter R of eq.(20) in [1].
- `int GetNumber() const`
  returns the number of jets.
- `Jet* GetFirst() const`
  returns the pointer to the first jet.
- `double GetESoft() const`
  For the spherical kinematics, it returns the soft energy in normalized units,

which is the part of the energy of the event that does not belong to any jet. For the cylindrical kinematics, it returns the fraction of the transverse energy of the event that does not belong to any jet.

- `Jet* GetJet( int n ) const`
  returns the pointer to the `n`-th jet.
- `double GetY() const`
  returns the value of Y of eq. (12) in ref. [1].
- `double GetOmega() const`
  returns the value of Ω of eq. (20) in ref. [1].

2.5 class `Jet`

This class represents a single jet. In most cases, the user will not need to create instances of this class directly, but only use pointers to objects of this class to access the information about the jets.

- `Jet(int label, Jets* Q, Kinematics k)`
  - constructor. `label` is the index of the jet, `Q` is the pointer to the jet configuration (to an object of the class `Jets`).
  `Kinematics = enum { sphere, cylinder }`, where `sphere` applies to the center of mass kinematics (lepton collisions), and `cylinder` applies to the cylindrical kinematics of hadron collisions.
- `double GetE() const`
  returns the energy of the jets in the same units as used in the input.
- `double GetPx() const`
- `double GetPy() const`
- `double GetPz() const`
  return the x(y,z)-component of the momentum of the jet in the same units as used in the input.
- `double GetXEnergy() const`
  returns the energy of the jet for the spherical kinematics or transverse energy of the jet for the cylindrical kinematics.
- `double GetXEta() const`
  returns the standard angle θ of the jet direction (in degrees) for the spherical kinematics or the pseudorapidity η of the jet for the cylindrical kinematics.
- `double GetPhi() const`
  returns the standard angle ϕ of the jet direction (in degrees).
- `int GetLabel() const`
  returns the label of the jet (the index of the jet).
- `Jets* GetJets() const`
  returns the pointer to the jet configuration to which the jet belongs.
- `Jet* GetNext() const`
  returns the pointer to the next jet. It allows to loop over jets.

3 Compilation

The program is self-contained and requires only a standard C++ compiler and the standard C++ libraries. It consists of the implementation files: `OJFZD.cpp`, `OJFKinematics.cpp`, `OJFJets.cpp`, `OJFSearch.cpp`, header files: `OJFZD.h`, `OJFKinematics.h`, `OJFJets.h`, `OJFSearch.h`, example program: `example.cpp`, input data for the example program `inputWW.dat`, and the `Makefile`. To compile and run the example program (with g++ under Linux)

```
>make example
>example
```

can be used or alternatively

```
>g++ OJFZD.cpp OJFKinematics.cpp OJFJets.cpp OJFSearch.cpp
 example.cpp -o example
>example
```

In the last three lines, the example program `example.cpp` can be replaced by the user's own program.

4 Usage example

The usage of Optimal Jet Finder is best explained with the following example.

4.1 Source code of `example.cpp`

```
#include "OJFKinematics.h"
#include "OJFJets.h"
#include "OJFSearch.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;
using namespace OptimalJetFinder;


int main() {


  //input data

  ifstream in( "inputWW.dat" );
```

```
//create a new event
Event *P = new Event( sphere );
//use "cylinder" instead of "sphere" for cylindrical kinematics

double px, py, pz;
while( in>>px>>py>>pz ) {
  P->AddParticleRaw( px, py, pz ); //input a particle
}

in.close();

//input data ends


//normalize input momenta so that the sum of input energies = 1
//(or the sum transverse energies for cylindrical kinematics = 1)
P->Normalize();


cout << P->GetNumber() << " particles in the event." << endl;

//set the seed for the random number generator
OJFRandom::SetSeed( 13 );

double radius = 1.0; // R parameter of eq. (20) in reference [1]
unsigned ntries = 3; // number of tries

//new jet search created
JetSearch* js = new JetSearch( P, radius, ntries );

//find jets for a given value of Omega_cut
unsigned njets = js->FindJetsForOmegaCut(0.05);
if( njets == 0 ) { cout << "Jets lost." << endl; exit(1); }

//alternatively, find jet configuration for a fixed number of jets
//bool success = js->FindJetsForFixedNJets(4);
//if (  ! success  ) { cout << "Jets lost." << endl; exit(1); }

//get the jet configuration
Jets* Q = js->GetJets();

//display the number of jets
// and parameters /Omega, Y, Esoft/ of the jet configuration
cout << Q->GetNumber() << " jets found." << endl;
```

```
  cout << "Omega: " << Q->GetOmega() << ",   "
       << "Y: " << Q->GetY() << ",   "
       << "Esoft (normalized): " << Q->GetESoft() << "." << endl;

  //display the details of the jets
  cout << "The details of the jets (E px py pz):" << endl;

  Jet* jet = Q->GetFirst();
  while( jet ) {
    cout << setw( 10 ) << jet->GetE() << "  "
         << setw( 10 ) << jet->GetPx() << "  "
         << setw( 10 ) << jet->GetPy() << "  "
         << setw( 10 ) << jet->GetPz() << endl;
    jet = jet->GetNext();
  }

  //the user is responsible for deleting
  //what they created themselves with new
  delete P;
  delete js;

}
```

4.2 Output of ***example.cpp***

```
65 particles in the event.
4 jets found.
Omega: 0.0464792,   Y: 0.0382961,   Esoft (normalized): 0.00818312.
The details of the jets (E px py pz):
   38.1886    -18.5112     25.5879     19.5718
   59.2424    -37.2752    -43.3007    -12.9766
   49.1723      45.698    -9.17652    -11.8903
   29.8772     10.4448     26.5263     6.43505
```

5 Comparison between FORTRAN 77 and current version

We have run several test programs to compare the output of the FORTRAN ojf_015 version [1] and the current C++ version (both compiled with GNU gcc 3.4.2 on Linux Fedora Core 3, Intel i686).

In each test, we compute

$$\Delta = \begin{cases} \left| \dfrac{x_{\mathrm{C++}} - x_{\mathrm{FORTRAN}}}{x_{\mathrm{FORTRAN}}} \right| & (x_{\mathrm{FORTRAN}} \neq 0) \\ \\ |x_{\mathrm{C++}} - x_{\mathrm{FORTRAN}}| & (x_{\mathrm{FORTRAN}} = 0) \end{cases}, \tag{18}$$

where x is any of the following quantities: Ω, Y, E_{soft}, E_j, $p_j^{(x)}$, $p_j^{(y)}$, $p_j^{(z)}$, and E_j, θ_j, ϕ_j (spherical kinematics) or $E_j^{\perp}$, η_j, ϕ_j (cylindrical kinematics); j runs over all reconstructed jets. We characterize each event by $\Delta_{\max}$, the maximal value of all Δ's calculated for this event. Tables 1 and 2 present the distribution of $\Delta_{\max}$'s for two multi event tests. Tables 3 and 4 show the parameters and the results of single-event tests.

All events were generated with Pythia 6.222 [4].

Note that different stochastic minimum search algorithms must find the same set of local minima – but not necessarily in the same order (if only because of different floating point machine codes generated by different compilers). However, it proved possible to adjust the current implementation (the control parameters, etc.) so as to ensure that even the order of the local minima found is the same as with the FORTRAN 77 version for the same seed of the random number generator – without spoiling the high precision of the computations. Whatever minor numerical differences remain (see the comparison tables) must be attributed to the observed differences in computation of hyperbolic sines, etc. by the different routines provided by the C++ and FORTRAN 77 compilers.

References

[1] D. Yu. Grigoriev, E. Jankowski, F. V. Tkachov, Comput. Phys. Commun. **155**, 42 (2003).

[2] F. V. Tkachov, e-print: hep-ph/0111035.

[3] `http://www.inr.ac.ru/~ftkachov/projects/jets/`

[4] T. Sjostrand, P. Eden, C. Friberg, L. Lonnblad, G. Miu, S. Mrenna, E. Norrbin, Comput. Phys. Commun. **135**, 238 (2001).

Table 1
Distribution of $\Delta_{\rm max}$ for a sample of 10^6 $e^+e^- \to WW \to$ hadrons events at 180 GeV. Spherical kinematics. Three-momenta used in the input. $R = 1.0$, $n_{\rm tries} = 1$, $n_{\rm jets} = 4$, seed = 13.

$\Delta_{\rm max}$ RANGE	FRACTION OF EVENTS IN THE RANGE
$10^{-18} - 10^{-17}$	0.000002
$10^{-17} - 10^{-16}$	0.068718
$10^{-16} - 10^{-15}$	0.508784
$10^{-15} - 10^{-14}$	0.409418
$10^{-14} - 10^{-13}$	0.010992
$10^{-13} - 10^{-12}$	0.001616
$10^{-12} - 10^{-11}$	0.000369
$10^{-11} - 10^{-10}$	0.000074
$10^{-10} - 10^{-9}$	0.000021
$10^{-9} - 10^{-8}$	0.000005
$10^{-8} - 10^{-7}$	0.000001

Table 2
Distribution of $\Delta_{\rm max}$ for a sample of 10^5 $pp \to tt + X \to$ hadrons events at 14 TeV. Cylindrical kinematics. Three-momenta used in the input. $R = 1.0$, $n_{\rm tries} = 1$, $n_{\rm jets} = 6$, seed = 13. Two events yielded different jet configurations in the FORTRAN and C++ versions, corresponding to different local minima. The value of Ω was smaller for the C++ version by approximately 10^{-4} and 0.25.

$\Delta_{\rm max}$ RANGE	FRACTION OF EVENTS IN THE RANGE
$< 10^{-18}$	0.00004
$10^{-18} - 10^{-17}$	0.00051
$10^{-17} - 10^{-16}$	0.00336
$10^{-16} - 10^{-15}$	0.11745
$10^{-15} - 10^{-14}$	0.22692
$10^{-14} - 10^{-13}$	0.24575
$10^{-13} - 10^{-12}$	0.23479
$10^{-12} - 10^{-11}$	0.13154
$10^{-11} - 10^{-10}$	0.03411
$10^{-10} - 10^{-9}$	0.00536
$10^{-9} - 10^{-8}$	0.00015

Table 3
A single $e^+e^- \to WW \to$ hadrons event at 180 GeV. Spherical kinematics. In the input, three-momenta are used for tests B01-B17, and angles are used for tests C01-C04.

TEST ID	R	n_{tries}	n_{jets}	ω_{cut}	seed	$\Delta_{\max}$
B01	1.0	1	2	-	13	$8.5 \cdot 10^{-15}$
B02	1.0	1	4	-	13	$1.6 \cdot 10^{-16}$
B03	1.0	1	12	-	13	$3.2 \cdot 10^{-15}$
B04	1.0	1	20	-	13	$5.4 \cdot 10^{-15}$
B05	0.1	1	4	-	13	$8.3 \cdot 10^{-14}$
B06	0.2	1	4	-	13	$1.4 \cdot 10^{-16}$
B07	0.7	1	4	-	13	$1.4 \cdot 10^{-16}$
B08	10.0	1	4	-	13	$2.3 \cdot 10^{-15}$
B09	1.0	2	4	-	13	$2.9 \cdot 10^{-15}$
B10	1.0	3	4	-	13	$2.9 \cdot 10^{-15}$
B11	1.0	100	4	-	13	$2.9 \cdot 10^{-15}$
B12	1.0	3	4	-	6969	$2.9 \cdot 10^{-15}$
B13	1.0	50	-	0.005	13	$5.6 \cdot 10^{-15}$
B14	1.0	50	-	0.01	13	$2.4 \cdot 10^{-16}$
B15	1.0	50	-	0.02	13	$8.2 \cdot 10^{-15}$
B16	1.0	50	-	0.04	13	$3.3 \cdot 10^{-15}$
B17	1.0	50	-	0.06	13	$2.9 \cdot 10^{-15}$
C01	0.7	3	-	0.021	1313	$1.9 \cdot 10^{-14}$
C02	1.5	50	-	0.04	1313	$2.4 \cdot 10^{-16}$
C03	0.7	1	4	-	1313	$1.9 \cdot 10^{-16}$
C04	1.5	2	5	-	1313	$1.9 \cdot 10^{-15}$

Table 4
A single $pp \to tt + X \to$ hadrons event at 14 TeV. Cylindrical kinematics. In the input, three-momenta are used for tests D01-D13, and angles are used for tests E01-E04.

TEST ID	R	n_{tries}	n_{jets}	ω_{cut}	seed	Δ_{max}
D01	1.0	1	2	-	13	$4.2 \cdot 10^{-16}$
D02	1.0	1	6	-	13	$6.2 \cdot 10^{-14}$
D03	1.0	1	12	-	13	$4.1 \cdot 10^{-13}$
D04	1.0	1	20	-	13	$1.5 \cdot 10^{-11}$
D05	0.1	1	6	-	13	$4.2 \cdot 10^{-13}$
D06	10.0	1	6	-	13	$3.8 \cdot 10^{-13}$
D07	1.0	2	6	-	13	$4.2 \cdot 10^{-13}$
D08	1.0	3	6	-	13	$4.2 \cdot 10^{-13}$
D09	1.0	100	6	-	13	$3.4 \cdot 10^{-14}$
D10	1.0	3	6	-	6969	$3.3 \cdot 10^{-14}$
D11	1.0	50	-	0.05	13	$2.4 \cdot 10^{-13}$
D12	1.0	50	-	0.1	13	$3.4 \cdot 10^{-14}$
D13	1.0	50	-	0.2	13	$9.6 \cdot 10^{-16}$
E01	0.7	3	-	0.1	1313	$1.9 \cdot 10^{-12}$
E02	1.5	50	-	0.2	1313	$4.5 \cdot 10^{-16}$
E03	0.7	1	6	-	1313	$7.3 \cdot 10^{-16}$
E04	1.5	2	7	-	1313	$1.4 \cdot 10^{-12}$