Towards FastJet 3

Gavin Salam

CERN, Princeton & LPTHE/CNRS (Paris)

Work in progress with Matteo Cacciari and Gregory Soyez
alpha releases at `http://fastjet.fr/`

Boost 2011
PCTS, Princeton, May 2011

Jan 2006 (FJ 1.0):

- Fast implementation of *pp* k_t algorithm — Cacciari & GPS
  N^2 and $N \ln N$ timings for clustering N particles v. N^3 with earlier codes
  $N \ln N$ strategy relies on external package CGAL

Oct 2006 (FJ 2.0):

- Implementation of Cambridge/Aachen algorithm
  including coding of Chan's Closest Pair algorithm
- Introduction of jet areas and background estimation/subtraction
- New interface — for long-term stability

Apr 2007 (FJ 2.1):

- Plugin mechanism giving common interface to external jet finders
- Inclusion of plugins that wrap CDF (JetClu, Midpoint) code and PxCone
- Inclusion of SISCone as a plugin

FastJet

Jan 2008 (FJ 2.3):

Soyez joined development team

- Added the anti-k_t algorithm (fast, native implementation)
- Added "passive" and "Voronoi" areas
- Switched to autotools for compilation/installation
- Better access to information for subjet studies
- Basic Fortran wrapper

April 2009 (FJ 2.4):

- Added plugins for DØRunIICone, ATLAS cone, CMS cone, TrackJet

  DØ and Trackjet code contributed by Sonnenschein

  ATLAS code taken from SpartyJet
- Added gen-k_t + e^+e^- algorithms (k_t, Cambridge, Jade, e^+e^- anti-k_t)
- Framework for handling user-supplied clustering distances (NNH)

Why no new major release since 2009?

Why no new major release since 2009?

Why no new major release since 2009?

Why no new major release since 2009?

[we also wanted to take the time for future code developments to be informed by the range of uses that are actually out there]

Jets ecosystem: users

Individuals

- Anyone needing basic jet finding — wants stable, simple interface
- People playing with new jet ideas — needs flexible interface
- Theorists who still like Fortran — only basic features available

Analysis/detector-simulation frameworks

- Rivet — One of the drivers for inclusion of “legacy” jet algorithms
- SpartyJet — See talk by Chris Vermilion
- Delphes detector simulation

Experiments

- The four main LHC experiments all use FastJet for jet analyses
- So do STAR, H1, ZEUS and occasionally CDF
- ATLAS and CMS use FastJet in the high-level triggers

Stability is paramount!

Jets ecosystem: extensions etc.

External plugins for FastJet: (not included in 2.x releases)

- Variable R plugin — Krohn, Thaler & Wang '09
- Pruning plugin — Ellis, Vermillion & Walsh '09
- Trimming plugin — Krohn, Thaler & Wang '10

SpartyJet — Delsart, Geerlings, Huston, Martin & Vermilion

- Provides `root` interface to FastJet, including PyRoot access
- Visualisation tools, file-reading utilities, taggers, event storage, etc.

FastJet Tools page

- A range of boosted-object finders (Higgs, top, etc.), filtering, etc.

Our own, links to other people's, and our implementations of other people's

All oriented to FastJet 2.x

FastJet sees about 4000 downloads a year

Why FastJet v3?

To make it easier and safer for users
to do advanced things with jets

Incorporating lessons we've learned while writing taggers,
mimicking real analyses (particle ID's, acceptances, etc.)
& performing background subtraction
[as well as some frequent requests]

The core development?
`PseudoJet` is now a much more powerful object:

It knows about its internal structure
There are new ways of building a `PseudoJet` with structure
It can be associated with arbitrary user-specified information

Accessing internal jet structure

Accessing a jet's constituents in FastJet 2.x

```
ClusterSequence cs(particles, jet_def);
vector<PseudoJet> jets = cs.inclusive_jets();

// info about jet's structure comes through the cluster sequence
vector<PseudoJet> constituents = cs.constituents(jets[0]);
```

What changes in FastJet 3.0

```
// info about jet's structure directly from the jet
vector<PseudoJet> constituents = jets[0].constituents();
```

And similarly for all other structural info: e.g. `has_parents(...)`

NB: the cluster sequence must still exist for this to work

Accessing internal jet structure

Accessing a jet's constituents in FastJet 2.x

```
ClusterSequence cs(particles, jet_def);
vector<PseudoJet> jets = cs.inclusive_jets();

// info about jet's structure comes through the cluster sequence
vector<PseudoJet> constituents = cs.constituents(jets[0]);
```

What changes in FastJet 3.0

```
// info about jet's structure directly from the jet
vector<PseudoJet> constituents = jets[0].constituents();
```

And similarly for all other structural info: e.g. `has_parents(...)`

NB: the cluster sequence must still exist for this to work

Building jets with new structure

Suppose you have a top tagger

```
// some procedure gives you 3 subjets
PseudoJet W = subjet[0] + subjet[1];
PseudoJet b = subjet[2];
PseudoJet top = b + W; // addition just combines the 4-momenta
return top;            // you cannot ask for top.constituents()
```

FJ3: use `join(...)` to add momenta *and* structure

```
PseudoJet W = join(subjet[0], subjet[1]);
PseudoJet b = subjet[2];
PseudoJet top = join(b, W); // top.constituents() is now sensible
return top;                 // top.pieces() returns the b and W
```

Calls like `jet.has_constituents()` and `jet.has_pieces()` return true if it is legitimate to ask for constituents and pieces

Building jets with new structure

Suppose you have a top tagger

```
// some procedure gives you 3 subjets
PseudoJet W = subjet[0] + subjet[1];
PseudoJet b = subjet[2];
PseudoJet top = b + W; // addition just combines the 4-momenta
return top;            // you cannot ask for top.constituents()
```

FJ3: use `join(...)` to add momenta *and* structure

```
PseudoJet W = join(subjet[0], subjet[1]);
PseudoJet b = subjet[2];
PseudoJet top = join(b, W); // top.constituents() is now sensible
return top;                 // top.pieces() returns the b and W
```

Calls like `jet.has_constituents()` and `jet.has_pieces()` return true if it is legitimate to ask for constituents and pieces

User information inside PseudoJets

FastJet 2.x had only a `user_index`

```
PseudoJet particle;          // could easily associate one index;
particle.set_user_index(n); // no space for PDG ID, vertex number
```

FJ3 can store arbitrary information through `user_info`

```
class MyInfo : public PseudoJet::UserInfoBase {
public:
  MyInfo(int id, int vertex): pdg_id(id), vertex_number(vertex) {}
  int pdg_id, vertex_number; };

// allocate new instance of MyInfo for each particle
particle.set_user_info(new MyInfo(13,0)); // muon from vertex 0
// access the info
int id = particle.user_info<MyInfo>().pdg_id;
```

FastJet deletes the MyInfo pointer when it's no longer needed

User information inside PseudoJets

FastJet 2.x had only a `user_index`

```
PseudoJet particle;          // could easily associate one index;
particle.set_user_index(n); // no space for PDG ID, vertex number
```

FJ3 can store arbitrary information through `user_info`

```
class MyInfo : public PseudoJet::UserInfoBase {
public:
  MyInfo(int id, int vertex): pdg_id(id), vertex_number(vertex) {}
  int pdg_id, vertex_number; };

// allocate new instance of MyInfo for each particle
particle.set_user_info(new MyInfo(13,0)); // muon from vertex 0
// access the info
int id = particle.user_info<MyInfo>().pdg_id;
```

FastJet deletes the MyInfo pointer when it's no longer needed

The ancillary development: a framework of helper classes

`Selector` for defining particle / jet cuts

`BackgroundEstimator` for more flexible background estimation, Subtractors, Filters, Taggers, etc. (still in progress)

It's trivial to write cuts on particles and jets. But suppose you want to pass cuts as an argument to a function? One solution: C++0x lambda functions

But too new to be widely supported

FJ3 solution: create a Selector object

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

Selector pt_selector = SelectorPtMin(20.0);      // pt > 20
Selector eta_selector = SelectorAbsEtaMax(2.5);  // |η| < 2.5
Selector selector = pt_selector && eta_selector; // logical and

vector<PseudoJet> electrons = ...; // e.g. get electrons from MC
// select the ones that have pt > 20 and |η| < 2.5
vector<PseudoJet> selected_electrons = selector(electrons);
```

the last line makes use of `Selector::operator()(...)`

`selector.description()` tells you what a given selector does

Writing your own selectors is straightforward

It's trivial to write cuts on particles and jets. But suppose you want to pass cuts as an argument to a function?

One solution: C++0x lambda functions
But too new to be widely supported

FJ3 solution: create a Selector object

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

Selector pt_selector = SelectorPtMin(20.0);       // pt > 20
Selector eta_selector = SelectorAbsEtaMax(2.5);  // |η| < 2.5
Selector selector = pt_selector && eta_selector; // logical and

vector<PseudoJet> electrons = ...; // e.g. get electrons from MC
// select the ones that have pt > 20 and |η| < 2.5
vector<PseudoJet> selected_electrons = selector(electrons);
```

the last line makes use of Selector::operator()(...)
selector.description() tells you what a given selector does
Writing your own selectors is straightforward

Illustrating selectors for filtering / trimming

Filtering and Trimming through a single interface

```
PseudoJet jet = ...;
double Rfilt = 0.3;

// define a filter that reclusters jet constituents on scale
// Rfilt and then select the 2 hardest subjets
Filter filter(Rfilt, SelectorNHardest(2));
PseudoJet filtered_jet = filter(jet);

// recluster jet constituents on scale Rfilt, and select subjets
// that carry at least 5% of original jet's momentum
Filter trimmer(Rfilt, SelectorPtFractionMin(0.05));
PseudoJet trimmed_jet = trimmer(jet);

// obvious query functions just work
vector<PseudoJet> kept_subjets = trimmed_jet.pieces();
vector<PseudoJet> constituents = trimmed_jet.constituents();
```

Illustrating selectors for filtering / trimming

Filtering and Trimming through a single interface

```
PseudoJet jet = ...;
double Rfilt = 0.3;

// define a filter that reclusters jet constituents on scale
// Rfilt and then select the 2 hardest subjets
Filter filter(Rfilt, SelectorNHardest(2));
PseudoJet filtered_jet = filter(jet);

// recluster jet constituents on scale Rfilt, and select subjets
// that carry at least 5% of original jet's momentum
Filter trimmer(Rfilt, SelectorPtFractionMin(0.05));
PseudoJet trimmed_jet = trimmer(jet);

// obvious query functions just work
vector<PseudoJet> kept_subjets = trimmed_jet.pieces();
vector<PseudoJet> constituents = trimmed_jet.constituents();
```

Illustrating selectors for filtering / trimming

Filtering and Trimming through a single interface

```
PseudoJet jet = ...;
double Rfilt = 0.3;

// define a filter that reclusters jet constituents on scale
// Rfilt and then select the 2 hardest subjets
Filter filter(Rfilt, SelectorNHardest(2));
PseudoJet filtered_jet = filter(jet);

// recluster jet constituents on scale Rfilt, and select subjets
// that carry at least 5% of original jet's momentum
Filter trimmer(Rfilt, SelectorPtFractionMin(0.05));
PseudoJet trimmed_jet = trimmer(jet);

// obvious query functions just work
vector<PseudoJet> kept_subjets = trimmed_jet.pieces();
vector<PseudoJet> constituents = trimmed_jet.constituents();
```

The rest

What else is already in FJ 3.0alpha2?

- We've lifted the restriction of $R < \frac{\pi}{2}$ for native algorithms
- Significantly improved online (doxygen) documentation
- A broader set of example programs

What is forthcoming?

- More flexible pileup subtraction framework

  Designed to integrate easily with other tools
- A framework of boosted taggers

  Main restriction will be that they should not have external (e.g. ROOT) dependencies
- Other small changes

  e.g. `PseudoJet` defaults to zero momentum. Features to help with memory management

Conclusions

Try out the current α release

[the features described here are already mostly stable]

Let us know if anything doesn't work
or if you think important features are missing

Stay tuned for forthcoming $\alpha/\beta/3.0$ releases