SpartyJet 4.0 User's Manual

Pierre-Antoine Delsart, Kurtis L. Geerlings, Joey Huston,
Brian T. Martin, and Christopher K. Vermilion

January 17, 2012

Contents

1 Overview

SpartyJet is a set of software tools for jet finding and analysis, built around the FastJet library of jet algorithms [1, 2]. SpartyJet provides four key extensions to FastJet: a simple Python interface to most FastJet features, a powerful framework for building up modular analyses, extensive input file handling capabilities, and a graphical browser for viewing analysis output and creating new on-the-fly analyses. Many of these capabilities rely on a ROOT-based backend [3]. Beyond finding jets, many jet tools in SpartyJet perform measurement of jet or event variables, referred to here as jet or event "moments". These moments are available to subsequent tools and stored in the final output. SpartyJet can be downloaded from HepForge at `http://projects.hepforge.org/spartyjet`.

Sec. 2 walks through a simple example. Sec. 3 gives instruction on installation. Sec. 4 describes the overall structure of a SpartyJet run. Sec. 5 describes and enumerates the various jet tools available in SpartyJet. Secs. 6 and 7 describe SpartyJet's input and output facilities. Finally, Sec. 8 describes the capabilities of SpartyJet's graphical interface.

2 A simple example

The simplest way to get a feel for how SpartyJet works is to consider an example; in this section we walk through the script `examples_py/simpleExample.py`. This script runs the anti-k_T algorithm on the first 10 events listed in `data/J1_Clusters.dat`, makes a simple measurement on the found jets, and stores the results.

- Load the libraries that are needed for the algorithms that you are running (you need to `source setup.sh` in the main directory first to set up environment variables):

```
from  spartyjet import *
```

- Create a `JetBuilder` object to manage SpartyJet analyses. The argument sets the output message level — options are {`DEBUG, INFO, WARNING, ERROR`}. Log messages throughout the code are tagged with a message level; messages with level lower than the current output level are suppressed. Note that the `SpartyJet` namespace is aliased as `SJ`.

```
builder = SJ.JetBuilder(SJ.INFO)
```

- Create an input object with the name of the file containing the events, using the `getInputMaker` helper function. This function guesses the type of input file based on the file extension. Many types of input are available; see Section 6.

```
input = getInputMaker('../data/J1_Clusters.dat')
builder.configure_input(input)
```

- Define the jet analyses that you want to run. Most analyses begin with a jet algorithm, implemented in FastJet via the `FastJetFinder` tool. See Section for 5.1 for more options. Note that the `fastjet` namespace is aliased as `fj`.

```
name = 'AntiKt4'
alg = fj.antikt_algorithm
R = 0.4
antikt4Finder = SJ.FastJet.FastJetFinder(name, alg, R)
analysis = SJ.JetAnalysis(antikt4Finder)
```

- Pass analysis to your `JetBuilder`. A sequence of jet tools defines a jet analysis. A jet finder is one example of a `JetTool`; see Section 5 for many more.

```
builder.add_analysis(analysis)
```

- Insert another tool in the chain, in this case making a measurement of jets' angular moments in η and ϕ. These will also be stored in the output ROOT file. `add_jetTool()` can take a second argument, the name of an algorithm to add the tool to; otherwise the tool is added to all algorithms. Tools can also be added directly to analyses via `JetAnalysis::add_tool()`.

```
builder.add_jetTool(SJ.EtaPhiMomentTool())
```

- Configure (optional) simple text output for quick visual check of results.

```
builder.add_text_output('../data/output/text_simple.dat')
```

- Configure the Ntuple output by specifying the name of the tree and the `ROOT` file you want the data to be stored in. This output can be manipulated via your own `ROOT` scripts or be viewed with the SpartyJet GUI.

```
builder.configure_output('SpartyJet_Tree','../data/output/simple.root')
```

- Give the command to run the algorithms on the first 10 events

```
builder.process_events(10)
```

Running the script will process the first 10 events on the file specified in the input object and produce the `.root` file specified in `configure_output`. To view the output with the GUI, run

```
sparty data/output/simple.root
```

(`spartyjet/bin` must be in your `$PATH`.) This is meant as a basic introduction, and there are many more functions than listed here. See the other examples for more.

3 Installation

3.1 Requirements

SpartyJet should build and run on any Unix-like operating system, including Mac OS X. A Cygwin build is presumably possible but not tested. The standard build system is via Makefiles; there is also a CMake build system which requires CMake. SpartyJet uses ROOT extensively for input, output, and Python wrapping via PyROOT. SpartyJet 4.0 has been tested with versions 5.30 and 5.32, but older versions may be OK. To use the Python interface to SpartyJet you must have built ROOT with PyROOT enabled. We have tested with Python versions 2.6 and 2.7. SpartyJet now makes heavy use of recent FastJet features, so FastJet version 3.0+ is required. If you do not already have FastJet installed, SpartyJet can build an internal version. If you want to use the StdHEP input facility, you will need a Fortran compiler; we've tested with gfortran.

3.2 Compilation

To compile:

```
# Set up ROOT such that root-config is in your path
# for example source root/bin/thisroot.sh
cd spartyjet
source setup.sh
make
```

If you do not already have FastJet installed, do `make fastjet` and `source setup.sh` (again) before `make`. This builds FastJet inside SpartyJet and sets the relevant environment variables.

SpartyJet has several building options to note:

- **FastJet:** SpartyJet depends on FastJet for jet finding and some internal features, and the latest version is included in the SpartyJet distribution. If you prefer to use your own installation, simply add `your-fastjet/bin` to the environmental variable `$PATH` such that `fastjet-config` can be found.
  NOTE: If you have linking problems between your version of FastJet and SpartyJet, either recompile your FastJet with the `--with-pic` option enabled before compilation, or have SpartyJet compile its own

version. Note also that to enable all FastJet plugins, you must pass the `--enable-allcxxplugins` flag to `configure`. This is done by default for the built-in version.

- **StdHEP libraries:** These require a Fortran compiler and are automatically compiled if you have `gfortran`, `f77`, or `g77` in your `$PATH`. If you would like to try a different compiler, set the environmental variable `$F77` to the compiler binary.

- **Pythia 6/8 interface:** If you have ROOT compiled with the Pythia 6 and/or Pythia 8 interfaces enabled, you can use this from within SpartyJet to generate events in Pythia and feed the output directly to SpartyJet. To enable this, set the variable `$PYTHIA6DIR` and/or `$PYTHIA8DIR`. If you do not have `PYTHIA` support in ROOT, you can add it by doing:
```
cd $ROOTSYS
./configure --enable-pythia6 --enable-pythia8
--with-pythia6-libdir=/my/pythia6/
--with-pythia8-incdir=/my/pythia8145/include/
--with-pythia8-libdir=/my/pythia8145/lib/
make
```

Building SpartyJet creates a set of libraries in `spartyjet/lib` that you can load from a ROOT session or Python script, or you can link to to build an executable.

libs/libExternal.so	-	FastJet and other code SpartyJet depends on
libs/libJetCore.so	-	Core infrastructure
libs/libIO.so	-	Facilities for reading and writing a variety of file formats
libs/libFastJet.so	-	Tools that rely on FastJet, including jet finding
libs/libJetTools.so	-	Other JetTools
libs/libEventShape.so	-	Thrust and other event shapes
libs/libSpartyDisplay.so	-	SpartyJet GUI
libs/libExternalTools.so	-	A set of third-party jet tools, with wrappers

3.3 Running

Working examples of how to use SpartyJet can be found in the following directories:

`spartyjet/examples_py`	:	Python scripts (recommended)
`spartyjet/examples_C`	:	Compiled programs in C++

The Python interface to SpartyJet is strongly preferred, and C++ access may be deprecated in a future release. To use the Python scripts, some environment variables needs to be set, which can be accomplished via:

```
source setup.sh
```

in the `spartyjet/` directory. This exports the relevant paths to your `LD_LIBRARY_PATH` and sets the environment variable `SPARTYJETDIR`, which allows the SpartyJet

Python modules and libraries to be accessible from any directory. The relevant lines of `setup.sh` could also be copied into your shell's `rc` file, e.g. `~/.bashrc`. You will also need `${ROOTSYS}/lib` in your `$PYTHONPATH`. (This is necessary to use PyROOT.)

3.4 CMake build

SpartyJet now includes a CMake build system. If you have CMake installed, you can build SpartyJet by creating a build directory (e.g., `spartyjet/build`) and running:

```
cmake ..
make
```

If you are using the built-in FastJet distribution, you need to run:

```
cmake ..
make fastjet
cmake ..
make
```

To set environment variables correctly, you may need to run `source setup.sh` in the main directory before running SpartyJet programs. In a future release, the CMake build system should allow natively building on Windows (not Cygwin), but this will likely require Windows builds for SpartyJet's dependencies.

CMake can generate project files for your IDE of choice, e.g., to build an Xcode project just do

```
cmake .. -G Xcode
```

4 JetBuilder

`JetBuilder` is the job manager for SpartyJet. `JetBuilder` takes the input from an `InputMaker` (see Section 6) and passes it through a set of `JetAnalyses`. A `JetAnalysis` is made up of a sequence of `JetTools`. The final list of jets, and associated moments, is passed to an `NtupleMaker`, which prepares the output. This is shown schematically in Fig. 1.

4.1 Input Functions

Example: `examples_py/inputExample.py`

Input is passed to `JetBuilder` via:

```
builder.configure_input(input, saveInput=True)
```

where input can be any class deriving from `InputMaker`. See Section 6 for examples. The `saveInput` flag, true by default, determines whether to save the input particles in the output file. Saving input particles is needed for most event displays but not for plotting run variables (jet mass, moments, etc.).

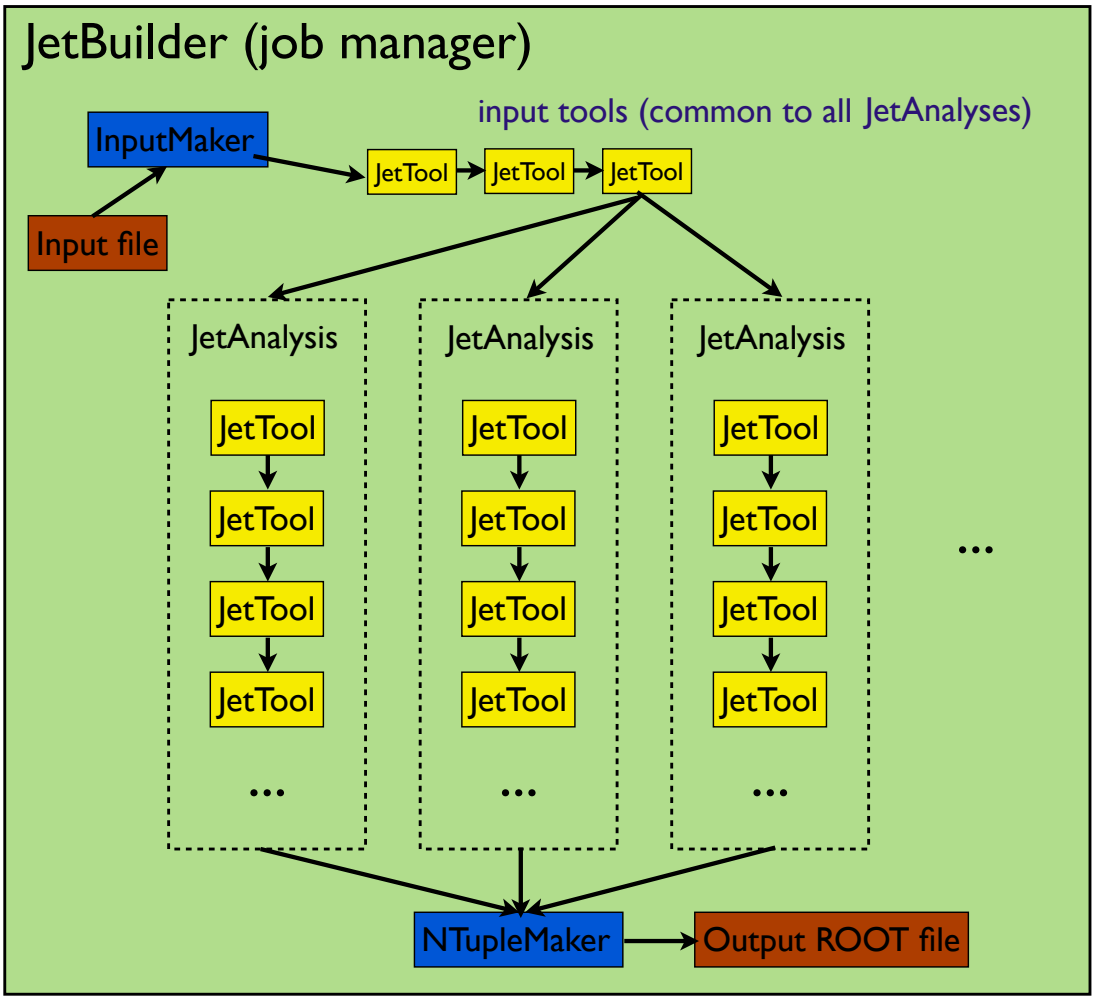

Figure 1: The structure of a SpartyJet analysis. A `JetBuilder` gets input from a file using an `InputMaker`, passes the input through a sequence of input `JetTools`, and passes the output to several `JetAnalyses` (labeled JetAlgorithms in this diagram). `JetAnalyses` consist of a sequence of `JetTools` that are run sequentially. The output of each `JetAnalysis` is passed to an `NTupleMaker` which stores the final jets and accompanying moments in a file.

4.2 `JetAnalysis` Options

`JetAnalyses` are blocks of code that act on a set of `Jet`s — implemented as a `JetCollection`. A set of `JetTool`s forms a `JetAnalysis`, and a `JetBuilder` holds a set of `JetAnalyses` that are each run on its inputs. Examples of specific tools can be found in Sec. 5.

`JetAnalyses` can be added to a run via:

```
builder.add_analysis(analyis)
```

Alternatively, analyses can be set up implicitly by just passing the `JetBuilder` a `JetTool`:

```
builder.add_default_analysis(tool)
```

where `tool` is typically a jet finder, which forms the basis of the tool chain. Default tools are added before and after — currently just negative energy correctors, if this option is enabled (it is not by default). To enable negative energy correction, call

```
builder.do_correct_neg_energy(True)
```

This flips the sign of the energy of any negative-energy input particles, then after jet finding adjusts jets' energies accordingly.

`JetTool`s can be added directly to analyses, or through the `JetBuilder`:

```
# add a tool a specific analysis
analysis.add_tool(tool)
# add a tool to all analyses registered with builder
builder.add_jetTool(tool)
# add a tool to a specific analysis via builder
builder.add_jetTool(tool, 'AnalysisName')
```

Tools can be added to the front of the sequence with:

```
analysis.add_tool_front(tool)
# or
builder.add_jetTool_front(tool)
```

Analyses are passed to the `JetBuilder` as pointers, so tools can be added to them at any time before calling `JetBuilder::process_events()`.

A single sequence of `JetTool`s is run on the input particles before they are passed to each `JetAnalysis`. This sequence starts empty; to add to it use:

```
builder.add_jetTool_input(tool)
```

This is useful for initial tools like η cuts or detector simulation that are common to all algorithms.

4.3 Output options

SpartyJet by default retains information about each jet's constituents. `JetBuilder::add_analysis()` and `JetBuilder::add_default_analysis()` each take a second bool argument that determines whether to save constituent information for each jet:

```
builder.add_analysis(analysis, withIndex=True)
builder.add_default_analysis(tool, withIndex=True)
```

The default is `True`. For this to be useful in the output, the input should be set up to store input particles in the output file as well. The full recombination history is stored in memory, but storage to disk has not yet been implemented. We hope to have this available in a future release.

A third option can be passed to `JetBuilder::add_analysis()`, which if `True` tells the builder to only save jet and event moments, not jet variables like η, p_T, etc. This can be useful to save space, or if only event-shape tools are used. In fact, `JetBuilder` has a method `add_eventshape_analysis(analysis)` that is simply an alias to `add_analysis(analysis, False, True)`.

A text file output option tells `JetBuilder` to produce an easily-readable text file that contains a list of all jets found from all algorithms for all events. To turn on the text file output, you must call `builder.add_text_output()`, and pass it the filename you want to create. For example:

```
builder.add_text_output('../data/output/text_output.dat')
```

The main output of SpartyJet is a ROOT file holding input particles, jet momenta, and jet and event moments. The format of this file as well as some additional `JetBuilder` options controlling it, are described in Sec. 7.

4.4 Minimum Bias Overlay

Example: `examples_py/overlayExample.py`

`JetBuilder` allows the user to add minimum bias (MB) events to the signal events to study the effects of pileup. To enable MB events, one must:

1. Create another input object:

   ```
   MBinput = SJ.StdTextInput('../data/MB_Clusters.dat')
   ```

2. Tell `JetBuilder` to add `n` MB events to each signal event:

   ```
   builder.add_minbias_events(n, MBinput, poisson=False)
   ```

`JetBuilder` will start at the beginning of the MB data file and read the first `n` events for the first data event, then the second `n` events for the second data event, so on. When the end of the MB file is found, it will simply continue from the beginning. If the third, optional, argument is `True`, the `JetBuilder` will draw the number of MB events from a Poisson distribution (default is `False`). In this case, `n` is the Poisson mean or expected number of MB events.

4.5 Options controlling screen output

SpartyJet can be configured to produce screen output at varying levels of verbosity. The most important point of control is the global message level, which can be one of {`DEBUG, INFO, WARNING, ERROR`}. Each message produced in the course has an associated level, and will be printed to the screen if its level is at least as high as the global level: e.g., if the global level is `WARNING`, `WARNING` and `ERROR` messages will be printed. The global message level can be set in the

`JetBuilder` constructor, or later via the `JetBuilder::set_message_level()` method:

```
builder = SJ.JetBuilder(SJ.INFO)
# or
builder.set_message_level(SJ.WARNING)
# equivalent to set_message_level(SJ.ERROR):
builder.silent_mode()
```

SpartyJet will report on the progress of the run by printing "Processed Event: N" lines, by default every event. You can change the frequency via:

```
# prints every 100 events
builder.print_event_every(100)
```

4.6 Running

Once all analyses have been set up and all options loaded, it's time to run. To run over all events in the input, simply do

```
builder.process_events()
```

You can also specify how many events to process, and what event number to begin at:

```
# process 100 events, starting at the 250th
builder.process_events(100, 250)
# alias to process_events(1, N):
builder.process_one_event(N)
```

5 JetTools

`JetTool`s are blocks of code that act on a `JetCollection` for every event. They generally perform one or more of the following functions:

- Find jets
- Add or remove jets from the list.
- Modify the jets themselves.
- Add information about each jet as a `JetMoment`.
- Add information about each event as an `EventMoment`.

Jet tools can be added either before or after the primary jet finder (e.g., the k_T algorithm) in the `JetAnalysis`. This section describes the set of `JetTool`s shipped with SpartyJet. The relevant definitions can be found in `JetTools/` (core SpartyJet tools), `FastJetTools` (wrappers around FastJet tools), and `ExternalTools` (FastJet-based tools provided by third parties).

SpartyJet and FastJet have evolved alongside each other, and many features formerly implemented natively in SpartyJet are now outsourced to FastJet algorithms or tools. We have endeavored to keep pace, but realize there is still some duplication, especially with the suite of tools available in FastJet 3.0.

Where available, FastJet-based tools are the preferred means of performing jet finding (`fastjet::ClusterSequence`), selection (`fastjet::Selector`), manipulation (`fastjet::Transformer`), and measurement (`fastjet::FunctionOfPseudoJet`). The improved wrapping of FastJet code in SpartyJet 4.0 should enable essentially any FastJet tool to be used within SpartyJet. We expect that future development of jet tools will be done within the context of FastJet, with development in SpartyJet focused on the Python interface, I/O facilities, and graphical interaction.

As noted in [4], the planned future development of FastJet includes a "contrib" project of externally written code. We envision that many external tools currently provided with SpartyJet will migrate to this space when it becomes available. In the meantime, see `ExternalTools/README` for details on the origins and authorship of the tools in that directory. To add your own tools to SpartyJet, you will need to either create a shared library that a) links against compiled C++ code or b) is loaded by ROOT. See the comments in `UserPlugins/Makefile` for more information on creating such a library, or including your own code in SpartyJet's build. `UserPlugins/ExamplePlugin` gives an example.

Note: In this section we sometimes switch from Python to C++ syntax for clarity of argument types.

5.1 Jet finders

All jet finding is done via FastJet jet algorithms. The basic wrapper is defined in `FastJetTools/FastJetFinder.hh`. There are two basic operations: jet finding, and reclustering.

`FastJetFinder`: Finds jets using FastJet algorithms. It has two constructors:

```
FastJetFinder(string name, fastjet::JetAlgorithm alg, double R=0.4,
    bool area=false)
FastJetFinder(fastjet::JetDefinition, string name, bool area=false)
```

The second form allows you to pass your own `fastjet::JetDefinition`, including a plugin algorithm; see `FJExample.py` for examples. By default, a `FastJetFinder` finds inclusive jets with $p_T > 5$ GeV. This can be modified via `set_ptmin(double)`, `set_dcut(double)`, and `set_njets(int)`. The last two correspond to *exclusive* jet finding; note that `dcut` has dimension GeV^2. Setting any of these overrides the others.

`FastJetRecluster`: Similar to `FastJetFinder`, but uses a given jet algorithm to recluster the constituents of each jet in an event. Has two constructors, with the same arguments as `FastJetFinder`.

The full set of default algorithms from FastJet is available. To use the included but optional plugin algorithms, uncomment the relevant lines in `External/ExternalLinkDef.hpp` and recompile SpartyJet.

Two additional plugin algorithms are available in the `ExternalTools` directory:

`NjettinessPlugin`: Finds jets by minimizing the N-jettiness measure [5, 6] over the event with a fixed number (N) of axes.

`QjetsPlugin`: Implements the "Qjets" algorithm of [7].

5.2 Selectors

These tools allow the user to remove some Jets from an input/output JetCollection. Most selectors can now be formed with the standard FastJet `Selector`s via the `SelectorTool` class:

```
selector = fj.SelectorPtMin(200.0) * fj.SelectorNHardest(2)
```

See the FastJet manual or `fastjet-install/include/fastjet/Selector.hh` for the full set of available selectors. The compound assignment operators `&=` and `|=` are supported, as well as (via some duck punching in the SpartyJet Python wrapper) the binary operations `*`, `&&`, and `||`. Note that `(s1 * s2)` applies `s2`, then `s1`; `(s1 && s2)` applies both separately. The result is the same iff the selectors commute.

Many native SpartyJet selectors are defined in `JetTools/JetSelectorTool.hh`; the ones that do not duplicate a FastJet selector (and are thus not deprecated) are:

`JetInputPdgIdSelectorTool(std::vector<int> pdgIds)`: Removes input "jets" with given PDG IDs. (Useful when the input jets are just single particles from a Monte Carlo where leptons, neutrinos, etc. are included.) The input file must have included PDG IDs.

`JetMomentSelectorTool<T>(std::string momentName, T min, T max)`: Finds the given jet moment (calculated by another tool) and requires it be within (min, max). T can be any type supporting less-than comparison.

5.3 Transformers

FastJet 3.0 provides a common base class for jet manipulation: `fastjet::Transformer`. Transformers can remove particles (e.g., filtering), re-arrange substructure, or tag/reject jets. To add a given transformer to an analysis, use the `TransformerTool`:

```
Trimmed = SJ.JetAnalysis([... some jet finder ...])
trimmer = fj.Filter(0.35, fj.SelectorPtFractionMin(0.03))
Trimmed.add_tool(SJ.FastJet.TransformerTool("TrimmerTool", trimmer))
```

In C++ code any class deriving from `fastjet::Transformer` is acceptable; in Python a ROOT dictionary must be generated. See the comments in `UserPlugins/Makefile` for more information on including your own classes in SpartyJet. The currently available set of `Transformer`s is:

FastJet native:

`Boost`: Boost a jet to the rest frame of a reference jet.

`CASubJetTagger`: Versatile and generic substructure identification for Cambridge/Aachen jets.

`Filter`: Recluster a jet with a smaller R and only keep subjets passing a given criterion [8].

`GridMedianBackgroundEstimator`: Two different ways of estimating background radiation.

`JetMedianBackgroundEstimator`

`MassDropTagger`: Tagger that peels off soft radiation until a splitting is found where the two parents have significantly less mass than the child [8].

`RestFrameNSubjettinessTagger`: Implements the rest-frame version of the N-subjettiness jet shape [9].

`Subtractor`: Subtracts estimated background energy.

`JHTopTagger`: The Johns Hopkins top tagger [10].

`Unboost`: Unboost a jet from the rest frame of a reference jet.

`Pruner`: Rebuilds a jet with a new cluster sequence, vetoing soft, large-angle mergings [11, 12].

External tools included with SpartyJet:

`HEPTopTagger`: An implementation of the `HEPTopTagger` of [13, 14], provided by the authors.

`CMSTopTagger`: An implementation of the "CMS" top tagger [15, 16], written by one of the SpartyJet authors, based on `JHTopTagger`.

`TopTaggerDipolarityTool`: Adds dipolarity [17] to any top tagger, running the tagger, then storing the dipolarity measured on its substructure.

The FastJet-native transformers are described further in the FastJet documentation. Examples of using many of these transformers can be found in `examples_py/FJToolExample.py` and `examples_py/Boost2011.py`.

5.4 Jet and event moment tools

These tools calculate moments for each jet or event, which can be any type that ROOT knows how to store, most commonly `double`s or `int`s. The generic `JetMomentTool` and `EventMomentTool` calculate and store any user-implemented `JetMoment<T>` or `EventMoment<T>` object. See `JetTools/JetMomentTool.hh` for some specific examples, and `FJToolExample.py` for moment tools in action.

The FastJet 3 base class `FunctionOfPseudoJet<T>` provides a common interface for jet measurements. This is wrapped in SpartyJet by the `PseudoJetMomentTool<T>` class. The template argument `T` gives the type of the function output.

```
# An example of using a FunctionOfPseudoJet measurement
nsub3 = fj.Nsubjettiness(3, Njettiness.onepass_kt_axes, 1.0, 1.0)
AntiKt10.add_tool(SJ.FastJet.PseudoJetMomentTool(double)(nsub3, "tau3"))
```

Most jet and event moments in SpartyJet are of type `Double32_t`. This is a ROOT-defined type that behaves like a `double` in memory but is stored as `float`; we feel this is a reasonable compromise between precision mid-calculation and saving storage space. In the Python package `ROOT.Double32_t` is imported as simply `double`, as in the above example.

The following `FunctionOfPseudoJet`s are available, all defined in `spartyjet/ExternalTools`. Examples of their use can be found in `examples_py/Boost2011.py`.

`MinMassFunction`: Finds three exclusive subjets, then measures the smallest invariant mass between any two, as in the CMS top tagger [15, 16].

`zCellFunction`: Measures the z_{cell} variable of the "Thaler and Wang" top tagger [18].

`zCutFunction`: Measures the z^i_{cut} of the ATLAS top tagger [19].

`Nsubjettiness`: Measures the version of N-subjettiness described in [20].

The following are other moment-storing tools available. They may be phased out in favor of more explicitly FastJet-based versions.

`HullMomentTool`: This tool finds the convex hull enclosing each jet and saves the hull length and area as `Hull` and HullA respectively.

`EtaPhiMomentTool`: This tool calculates angular second moments in η and ϕ of each jet stores them as `M2eta` and `M2phi`.

`PtDensityTool`: This tool calculates each event's p_T density using FastJet. It does this by finding all the jets in the event with no minimum p_T requirement. It then extracts the p_T density from these jets by selecting the mean p_T density for each bin in η. These p_T densities are stored as `ptDensity` and the η bin limits are stored as `ptDensityBins`.

`JetAreaCorrectionTool`: This tool uses the area of each jet and the pTDensity found by the `PtDensityTool` to calculate a correction (stored as jet moment `JetAreaCorr`) to the jet's p_T.

`YSplitterTool`: This tool uses FastJet to calculate the y values associated with a set of recombinations. In this implementation, SpartyJet will run a FastJet algorithm of the user's choice on the constituents of a given jet. It can be called with either of the following constructors:

```
YSplitterTool(float R, fastjet::JetAlgorithm alg, int ny, int njet)
YSplitterTool(fastjet::JetDefinition *jet_def, int ny, int njet)
```

where `njet` is the number of jets for which the y values will be calculated and `ny` is the number of y values to calculate for each jet.

5.5 Miscellaneous tools

`ForkToolParent` and `ForkToolChild`: This pair of tools allows the forking of `JetTool` chains. `ForkToolParent` merely saves a copy of its input. A `ForkToolChild` is associated with a specific parent, and it reads in the `JetCollection` saved by its parent. This allows, for example, one jet algorithm to be run combined with several different jet-modifying tool chains for comparison. See `FJToolExample.py` for a usage example.

`CalorimeterSimTool`: This tool applies a very simple calorimeter simulation to its input jets. Inputs are sorted into calorimeter cells on a specified η–ϕ grid. For each non-empty cell, a massless output particle is created with the direction of the cell and the total energy of all particles in the cell.

`RadialSmearingTool`: This tool wraps Peter Loch's `DetectorModel` code for calorimeter simulation. For more information, see Peter's website. A usage example is given in `Boost2011.py`.

`JetNegEnergyTool`: This tool is meant to be run twice: once before jet finding and once afterward. On first run, the tool finds and stores all input particles with negative energy. For each such particle it inverts the energy to be positive. On second run, the `JetNegEnergyTool` loops over jets, and for each constituent that initially had a negative energy, it corrects the jet energy by subtracting twice the constituent's (positive) energy. The `JetBuilder` method `do_correct_neg_energy(true)` inserts this pair of tools before and after all jet finders added with `add_default_analysis(tool)`; by default this is not done.

`EConversionTool`: This tool simply converts the units of all the jets between MeV and GeV. The user can convert to arbitrary units as well.

`HardProcessMatchTool`: Assuming the input file includes information about the hard scattering (this is true of HepMC files, e.g.), finds the closest hard parton for each jet and stores the ΔR distance as a jet moment.

`AngularCorrelationTool`: Measures the angular correlation functions described in [21], storing R_n and m_n for each peak found up to three as jet moments.

`QjetsTool`: Runs the "Qjets" plugin algorithm repeatedly on each jet, storing the average pruned jet mass and its "volatility", as described in [7].

`WTaggerTool`: This tool wraps the W-tagging method of [22], which includes a large number of substructure and mass cuts. The cuts are taken from data files in `external/wtag-1.00/data`. So far there is no way to re-train the cuts from within SpartyJet— the idea is to take a pre-trained W-tagging method and plug it into a SpartyJet analysis. Since the tagger is taken "out-of-the-box", there are no input parameters:

```
WTaggerTool()
```

6 Input

All SpartyJet jobs need an `InputMaker` object to read input from some data file and prepare a list of four-vectors for `JetAnalyses` to process. There are several types of `InputMaker`s available. An example of the implementation for each type of input can be seen in `examples_py/inputExample.py`. For Python scripts, the top-level module `python/spartyjet/__init__.py` defines the helper function `getInputMaker(fileName)` which will create the appropriate `InputMaker` by looking at the filename extension.

6.1 NtupleInputMaker

Sample: `data/J2_clusters.root`

This form of input reads ROOT files. This `InputMaker` requires the components of the input 4-vectors to be stored in separate branches of a ROOT `TTree`. The following definitions are supported:

- `px, py, pz, E`
- `(psuedo)rapidity, phi, pt, E`
- `(psuedo)rapidity, phi, pt, m`

You need only specify how the information is stored (array or vector / float or double) and the names of the branches.

As an example, to set up an `NtupleInputMaker` to read the file `data/J2_clusters.root`, if you don't know how variables are internally stored in your ntuple, do the following:

```
root -l data/J2_clusters.root
root [0] clusterTree->MakeClass("test");
```

Open the file test.h and check to see how the variables are stored. In this example, we see lines like:

```
vector<float> *Cluster_eta;
```

indicating that our 4-vectors are stored in vectors of floats. Now to configure SpartyJet to accept this, we need to:

- Create an `NtupleInputMaker` of the correct type: (in our case: `vector<float>` for (`eta, phi, pt, E`)).

  ```
  input = SJ.NtupleInputMaker(SJ.NtupleInputMaker.EtaPhiPtE_vector_float)
  ```

  For full list of Input codes see: `JetCore/InputMaker_Ntuple.hh`

- Configure the names of the `TBranches`

  ```
  input.set_prefix('Cluster_')
  input.set_n_name('N')
  input.set_variables('eta','phi','p_T','e')
  input.setFileTree('../data/J2_clusters.root', 'clusterTree')
  ```

- Specify if input is massless, only useable in `eta,phi,pt,E` mode (if true, `pt` is ignored):

```
input.set_masslessMode(True)
```

- Set the input file and tree names:

```
input.setFileTree('../data/J2_clusters.root', 'clusterTree')
```

Python Shortcut:
To allow SpartyJet to configure your Ntuple using some assumptions use the following helper function:

```
input = createNtupleInputMaker('../data/J2_clusters.root', inputprefix='
    Cluster')
```

DelphesInput

`DelphesInputMaker` is a minor extension of `NTupleInputMaker` that reads the ROOT files produced by the detector simulator Delphes. Only calorimeter cells are read in.

6.2 StdTextInput

Sample: `data/J1_Clusters.dat`
This form of input reads ASCII files. To separate events, put one of the following lines between the events:

```
.Event
.event
N
n
```

(only the .E or .e is important in the first two).
The form of the four vectors should be:

```
E px py pz
```

This input is configured simply with:

```
input = SJ.StdTextInput('../data/J1_Clusters.dat')
```

If the form is the opposite (px py pz E), then call the function

```
input.invert_input_order(True)
```

and it will be read in properly.
An example of this input can be seen in `data/J1_Clusters.dat`

6.3 StdHepInput

Sample: `data/ttbar_smallrun_pythia_events.hep`

This form of input reads StdHEP format XDR files. It will look for particles with the status code of 1 (final state). It is also able extract the PDG ID code for each particle from the input data to allow further filtering and matching. Access to intermediate particles such as the participants in the hard scattering, or B hadrons from b quark decays, should be possible in the near future.

This input is configured simply with:

```
SJ.StdHepInput('../data/ttbar_smallrun_pythia_events.hep')
```

NOTE: To read StdHep files, you must enable StdHep compilation as explained in Section 3.

6.4 CalchepPartonTextInput

Sample: `data/gg_ggg_events.dat`

This form of input reads output from CalcHEP. It reads in the number of initial and final state particles, and then for each event saves only the information for the final state particles.

This input is configured simply with:

```
SJ.CalchepPartonTextInput('../data/gg_ggg_events.dat')
```

6.5 HepMCInput

Sample: `data/Zprime_ttbar.hepmc`

This form of input reads HepMC (version 2) format ASCII files. This class reads in the four-vectors and PDG IDs of the particles denoted with a status code of 1 (not decayed, final state).

6.6 PythiaInput

This form of input generates and reads events directly from Pythia, without ever having to write them to a file. This requires ROOT's Pythia interface; versions 6 and 8 will both work. See `examples_py/pythiaExample.py` for an example of using Pythia in this way.

6.7 FourVecInput

A `FourVecInput` takes four-vectors from some other code; the input class is templated on a "Reader" class that must provide a simple interface for retrieving four-vectors. See `IO/FourVecInput.hh` for a complete specification, and `examples_C/FourVecExample.cc` for an example. This does not currently work in Python, since you would have to generate a dictionary for `FourVecInput<YourReaderClass>`.

6.8 Input Options

Multiple input files

The `MultiInput` class can be used to string a set of input files together. See `examples_py/mergedInputExample.py` for an example. For ROOT files, the current implementation opens all input files before beginning, which may be inefficient. Other files are opened sequentially.

Rejecting bad input

The `InputMaker` can be set to remove four-vectors with negative energy and non-physical momenta by using the following function.

```
input.reject_bad_input(False)
```

The current default is `False`; no checks will be done. (An alternative to this method of dealing with bad input is to use the `JetNegEnergyTool`, described in Sec. 5.)

Reading of PDG ID codes

The `InputMaker` can be set to read PDG ID codes from the input data with the following function. This is done by default.

```
input.readPdgId(True)
```

This makes the PDG IDs available for input selection and saves the IDs of the input particles for offline analysis.

7 Output

After a SpartyJet run, the result is a ROOT `TTree` containing jet variables for each algorithm added, plus variables for input particles to the jet algorithms. Jet and event moments are also stored.

The output ROOT file contains a `TTree` (the file and `TTree` name are set via `builder.configure_output(treename, filename)`. The `TTree` contains a branch for each jet variable, named by default {`AnalysisName_eta`, `AnalysisName_phi`, `AnalysisName_e`, `AnalysisName_mass`, `AnalysisName_pt` }. (For this reason it is important that each analysis have a unique name!) If input particles are being stored, there are similar branches for the input particles. Every jet moment has an associated name and is stored in a branch named `AnalysisName_momentName`. For each analysis, an integer variable `AnalysisName_N` is stored, giving the number of jets for each event.

7.1 Output variable type

It is possible to choose the type of the variable saved in the `TTree`. The choices are C array vs. STL vector, and floats vs. doubles.

```
# can also do SJ.kArray and SJ.kDouble
builder.set_output_type(SJ.kVector, SJ.kFloat) # (default)
```

This affects the jet momentum variables, but not jet or event moments. Jet moments are always stored as `vector<T>` where `T` was the template parameter to the jet moment (the type that the moment returned. Event moments are simply stored as bare `T`s.

7.2 Constituents

For all analyses, constituent information is saved as follows, assuming that the analysis was added to the `JetBuilder` with option `withIndex=True` (this is the default). Assuming an analysis named `MyJet` has been added, two additional variables are stored in the ROOT `TTree`:

`MyJet_numC`

`MyJet_ind`

`MyJet_numC` is an array of size `MyJet_N`. `MyJet_numC[i]` is the number of constituents of `i` th jet. `MyJet_ind` is an array of size `InputJet_N`. `MyJet_ind[i]` is the index of the jet to which the `i` th input constituent has been assigned.

For example:

```
myTree.Draw('InputJet_e', 'AntiKt10_ind==0')
```

will give the energy distribution of constituents in jet number 0 (i.e. highest p_T jets) for the `AntiKt10` collection.

8 Graphical interface

SpartyJet output ROOT files can be explored with a graphical interface, launchable by running

```
$> sparty foo.root [bar.root ...]
```

(`spartyjet/bin` must be in your `$PATH`.) A screenshot of the GUI in action is shown in Fig. 2. The user can select which algorithms to view by ticking the boxes under "JetCollections". On the left are several event-by-event views, which will be drawn separately for each algorithm. The number of rows and columns are set in the upper left.

Additional analyses can be run (event-by-event) on the fly using the menu in the lower left. These are based on a new "live algorithm" facility in the SpartyJet Python package, which extends certain common analyses with knowledge of their input parameters so the GUI can build them live with user-provided parameters. These analyses can be run on the inputs from any file loaded on launch. In particular, you can launch the GUI with one or more ROOT files with no analyses run at all, only inputs. On-the-fly analyses provide a simple and powerful way to explore jet analysis dynamics, and we hope to extend these capabilities in future releases. High on our wish list is event generator integration, which would make the GUI self-contained.

On the right side of the control panel, full-run plots can be selected, which are plotted for all algorithms together (an example of this output is shown in Fig. 3). Check boxes are provided for the standard four-momentum variables, but any stored jet moment can also be plotted by entering the name in the box below, e.g. `$$_tau3/$$_tau2` to plot the ratio of 3- to 2-subjettiness if the `Nsubjettiness` moment has been measured as in `Boost2011.py`. Note that `$$` is a placeholder for the analysis name, and the syntax is that of `TTree::Draw`. Moments found in the input files are also given in the box below the jet variables, but note that not all moments are stored for all analyses!

Cuts can be added using `TCut` syntax, e.g. `AntiKt10_mass > 150 && AntiK10_mass < 200`. Finally, legend labels for each algorithm can be given; the syntax is ROOT `TLatex` text, e.g. `#phi_{0}` produces ϕ_0. Several drawing options are provided below the `Draw` button. Finer control can be gained by opening the ROOT histogram editor in the output canvas.

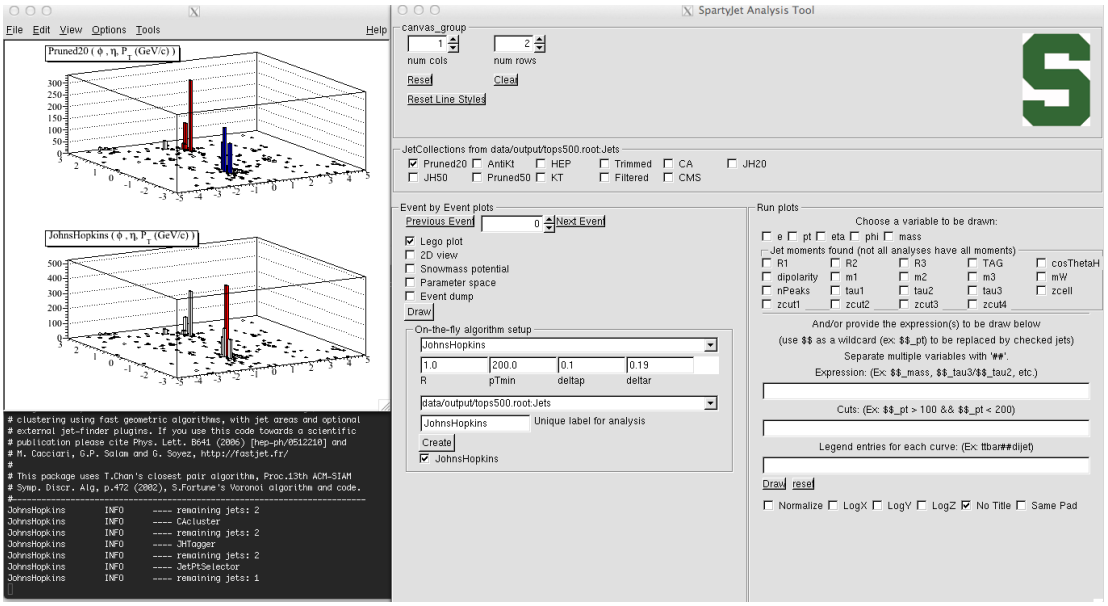

Figure 2: A screenshot of the SpartyJet graphical interface.

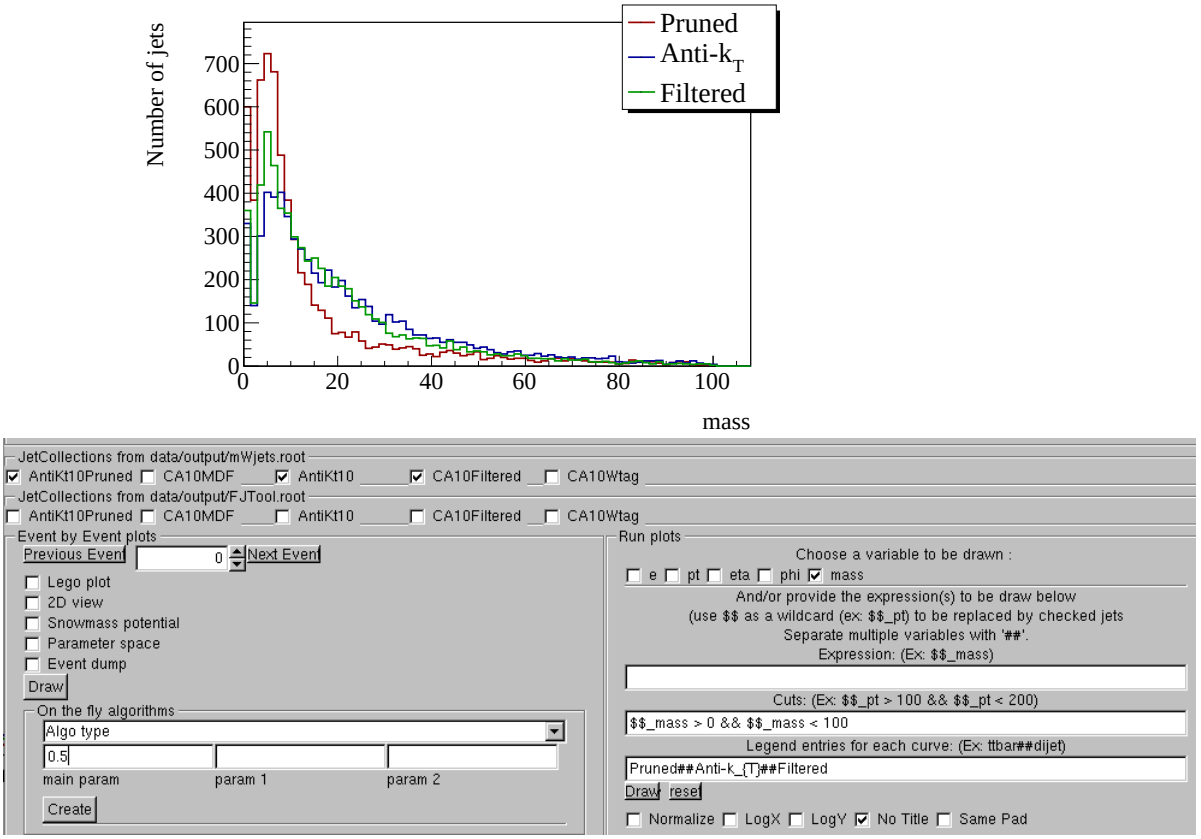

Figure 3: An example canvas produced by the run plot option (above), with the options that produced it (below).

Contact information

SpartyJet website: `http://projects.hepforge.org/spartyjet`

Any questions/comments/suggestions, email:

Pierre-Antoine Delsart:	delsart@ in2p3.fr
Joey Huston:	huston@ pa.msu.edu
Brian Martin:	marti347@ msu.edu
Chris Vermilion:	verm@ uw.edu

References

[1] M. Cacciari and G. P. Salam, "Dispelling the N^3 myth for the k_t jet-finder," *Phys. Lett.* **B641** (2006) 57–61, `arXiv:hep-ph/0512210`.

[2] M. Cacciari, G. P. Salam, and G. Soyez. `http://fastjet.fr`.

[3] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework," *Nucl.Instrum.Meth.* **A389** (1997) 81–86.

[4] M. Cacciari, G. P. Salam, and G. Soyez, "FastJet user manual," `arXiv:1111.6097 [hep-ph]`.

[5] I. W. Stewart, F. J. Tackmann, and W. J. Waalewijn, "N-Jettiness: An inclusive event shape to veto jets," *Phys. Rev. Lett.* **105** (2010) 092002, `arXiv:1004.2489 [hep-ph]`.

[6] J. Thaler and K. Van Tilburg, "Maximizing boosted top identification by minimizing N-subjettiness," `arXiv:1108.2701 [hep-ph]`.

[7] S. D. Ellis, A. Hornig, D. Krohn, T. S. Roy, and M. D. Schwartz, "Qjets: A Non-Deterministic Approach to Tree-Based Jet Substructure," `arXiv:1201.1914 [hep-ph]`.

[8] J. M. Butterworth, A. R. Davison, M. Rubin, and G. P. Salam, "Jet substructure as a new Higgs search channel at the LHC," *Phys. Rev. Lett.* **100** (2008) 242001, `arXiv:0802.2470 [hep-ph]`.

[9] J.-H. Kim, "Rest frame subjet algorithm with SISCone jet for fully hadronic decaying Higgs search," *Phys.Rev.* **D83** (2011) 011502, `arXiv:1011.1493 [hep-ph]`.

[10] D. E. Kaplan, K. Rehermann, M. D. Schwartz, and B. Tweedie, "Top tagging: A method for identifying boosted hadronically decaying top quarks," *Phys. Rev. Lett.* **101** (2008) 142001, `arXiv:0806.0848 [hep-ph]`.

[11] S. D. Ellis, C. K. Vermilion, and J. R. Walsh, "Techniques for improved heavy particle searches with jet substructure," *Phys. Rev.* **D80** (2009) 051501, arXiv:0903.5081 [hep-ph].

[12] S. D. Ellis, C. K. Vermilion, and J. R. Walsh, "Recombination algorithms and jet substructure: Pruning as a tool for heavy particle searches," *Phys. Rev.* **D81** (2010) 094023, arXiv:0912.0033 [hep-ph].

[13] T. Plehn, G. P. Salam, and M. Spannowsky, "Fat jets for a light Higgs," *Phys. Rev. Lett.* **104** (2010) 111801, arXiv:0910.5472 [hep-ph].

[14] T. Plehn, M. Spannowsky, M. Takeuchi, and D. Zerwas, "Stop reconstruction with tagged tops," *JHEP* **1010** (2010) 078, arXiv:1006.2833 [hep-ph].

[15] **CMS** Collaboration, R. Adolphi *et al.*, "A Cambridge-Aachen (C-A) based jet algorithm for boosted top jet tagging," tech. rep., 2009.

[16] **CMS** Collaboration, "Search for BSM $t\bar{t}$ production in the boosted all-hadronic final state," Tech. Rep. CMS-PAS-EXO-11-006, CERN, 2011.

[17] A. Hook, M. Jankowiak, and J. G. Wacker, "Jet dipolarity: Top tagging with color flow," arXiv:1102.1012 [hep-ph].

[18] J. Thaler and L.-T. Wang, "Strategies to identify boosted tops," *JHEP* **07** (2008) 092, arXiv:0806.0023 [hep-ph].

[19] **ATLAS** Collaboration, G. Brooijmans, "High pt hadronic top quark identification part 1 : Jet mass and ysplitter," Tech. Rep. ATL-PHYS-CONF-2008-008, CERN, Geneva, 2008.

[20] J. Thaler and K. Van Tilburg, "Identifying boosted objects with N-subjettiness," *JHEP* **03** (2011) 015, arXiv:1011.2268 [hep-ph].

[21] M. Jankowiak and A. J. Larkoski, "Jet substructure without trees," *JHEP* **1106** (2011) 057, arXiv:1104.1646 [hep-ph].

[22] Y. Cui, Z. Han, and M. D. Schwartz, "W-jet tagging: Optimizing the identification of boosted hadronically-decaying W bosons," *Phys.Rev.* **D83** (2011) 074023, arXiv:1012.2077 [hep-ph].