Unveiling the Jet Substructure using Julia

Sattwamo Ghosh

Department of Physical Sciences,
Indian Institute of Science Education and Research, Kolkata, India

Supervised by

Dr. Sanmay Ganguly

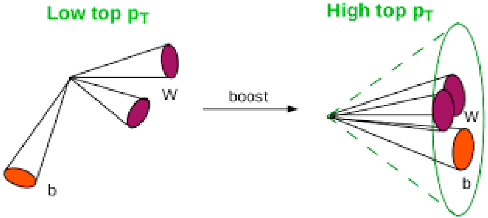
Assistant Professor, Department of Physics,
Indian Institute of Technology, Kanpur, India

October 1, 2024

1. Introduction
2. Implementation of Substructure Modules in Julia
3. Comparison of Performance with Pre Existing Bindings
4. Conclusion

What is Jet Substructure?

- Jet substructure refers to the internal structure of jets. It provides detailed information about the nature of the particles that initiated the jet (quarks, gluons, boosted heavy objects like W/Z/H bosons).
- Jet substructure analysis is essential for identifying boosted heavy particles in the search for new physics (e.g., Higgs boson, dark matter candidates).



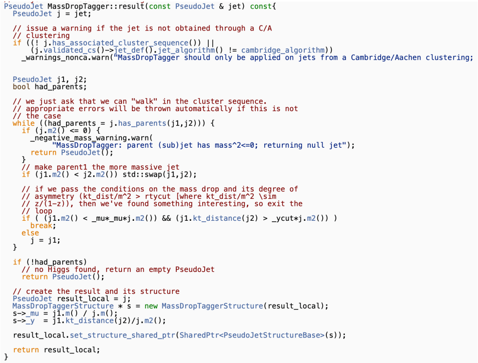
1. Introduction
2. Implementation of Substructure Modules in Julia
3. Comparison of Performance with Pre Existing Bindings
4. Conclusion

Code Overview

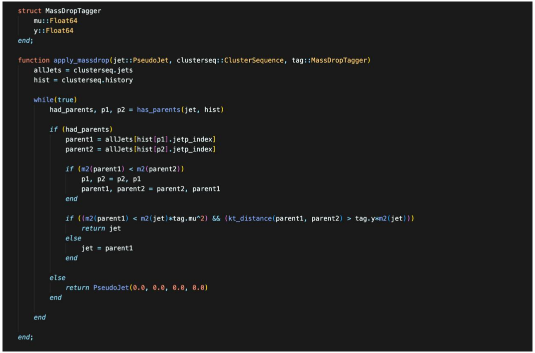
- The FastJet C++ package (and the corresponding Python bindings) provides numerous methods for substructure analysis.
- Out of these, we have successfully implemented the **MassDrop Tagger**, **SoftDrop Tagger**, **John-Hopkin's Top Tagger**, **Jet Filtering** and **Jet Trimming algorithms** into Julia.

Mass Drop Tagger

The MassDrop Tagging algorithm was implemented in Julia, following a similar structure to the corresponding FastJet code (a part of it is given below).



```cpp
PseudoJet MassDropTagger::result(const PseudoJet & jet) const{
  PseudoJet j = jet;

  // issue a warning if the jet is not obtained through a C/A
  // clustering
  if ((! j.has_associated_cluster_sequence()) ||
      (j.validated_cs()->jet_def().jet_algorithm() != cambridge_algorithm))
    _warnings_nonca.warn("MassDropTagger should only be applied on jets from a Cambridge/Aachen clustering;


  PseudoJet j1, j2;
  bool had_parents;

  // we just ask that we can "walk" in the cluster sequence.
  // appropriate errors will be thrown automatically if this is not
  // the case
  while ((had_parents = j.has_parents(j1,j2))) {
    if (j.m2() <= 0) {
      _negative_mass_warning.warn(
           "MassDropTagger: parent (sub)jet has mass^2<=0; returning null jet");
      return PseudoJet();
    }
    // make parent1 the more massive jet
    if (j1.m2() < j2.m2()) std::swap(j1,j2);

    // if we pass the conditions on the mass drop and its degree of
    // asymmetry (kt_dist/m^2 > rtycut [where kt_dist/m^2 \sim
    // z/(1-z)), then we've found something interesting, so exit the
    // loop
    if ( (j1.m2() < _mu*_mu*j.m2()) && (j1.kt_distance(j2) > _ycut*j.m2()) )
      break;
    else
      j = j1;
  }

  if (!had_parents)
    // no Higgs found, return an empty PseudoJet
    return PseudoJet();

  // create the result and its structure
  PseudoJet result_local = j;
  MassDropTaggerStructure * s = new MassDropTaggerStructure(result_local);
  s->_mu = j1.m() / j.m();
  s->_y  = j1.kt_distance(j2)/j.m2();

  result_local.set_structure_shared_ptr(SharedPtr<PseudoJetStructureBase>(s));

  return result_local;
}
```



```julia
struct MassDropTagger
    mu::Float64
    y::Float64
end;

function apply_massdrop(jet::PseudoJet, clusterseq::ClusterSequence, tag::MassDropTagger)
    allJets = clusterseq.jets
    hist = clusterseq.history

    while(true)
        had_parents, p1, p2 = has_parents(jet, hist)

        if (had_parents)
            parent1 = allJets[hist[p1].jetp_index]
            parent2 = allJets[hist[p2].jetp_index]

            if (m2(parent1) < m2(parent2))
                p1, p2 = p2, p1
                parent1, parent2 = parent2, parent1
            end

            if ((m2(parent1) < m2(jet)*tag.mu^2) && (kt_distance(parent1, parent2) > tag.y*m2(jet)))
                return jet
            else
                jet = parent1
            end

        else
            return PseudoJet(0.0, 0.0, 0.0, 0.0)
        end

    end

end;
```

Mass Drop Tagger

The same dataset was used for clustering and tagging in Python and Julia. The obtained results were concurrent with each other (as can be seen below).

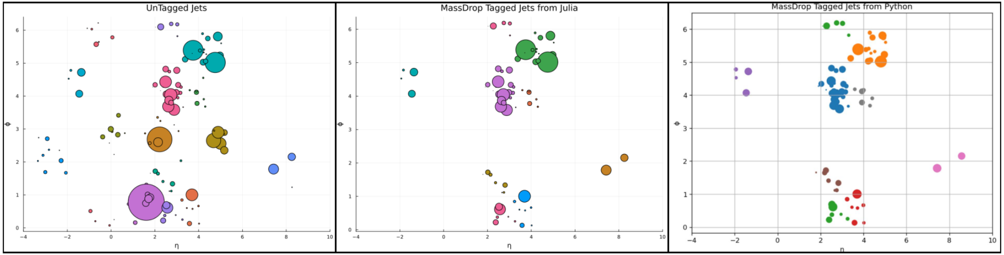


Figure 1: Plotting the $\eta-\phi$ space for a particular event and using MDT

John-Hopkins’ Top Tagger

The JH Top Tagger was implemented similarly and a similar analysis was carried out. The obtained results were concurrent with each other (as can be seen below).

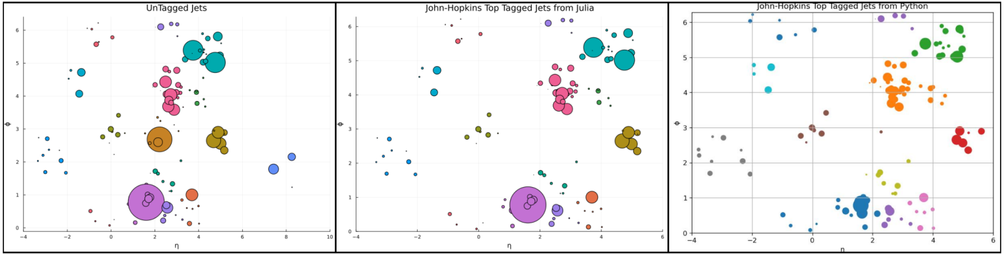
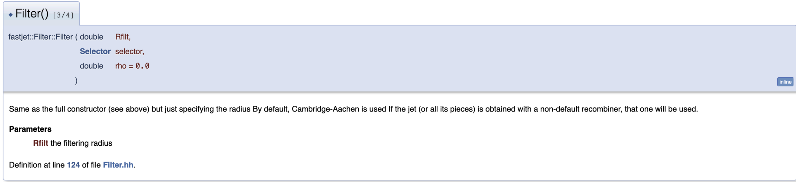


Figure 2: Plotting the $\eta - \phi$ space for a particular event and using JH Top Tagging

Jet Filtering & Jet Trimming

Coming to jet filtering and trimming,
we follow a similar approach for the Julia implementation.



◆ Filter() [3/4]

```
fastjet::Filter::Filter ( double    Rfilt,
                          Selector  selector,
                          double    rho = 0.0
                        )
```

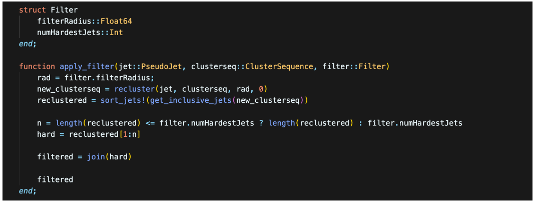
inline

Same as the full constructor (see above) but just specifying the radius By default, Cambridge-Aachen is used If the jet (or all its pieces) is obtained with a non-default recombiner, that one will be used.

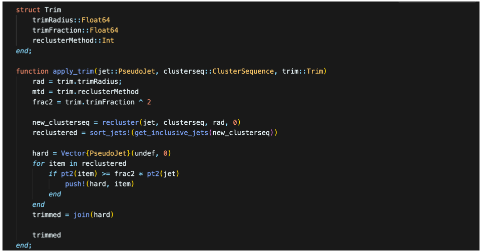
Parameters

Rfilt the filtering radius

Definition at line 124 of file Filter.hh.



```julia
struct Filter
    filterRadius::Float64
    numHardestJets::Int
end;

function apply_filter(jet::PseudoJet, clusterseq::ClusterSequence, filter::Filter)
    rad = filter.filterRadius;
    new_clusterseq = recluster(jet, clusterseq, rad, 0)
    reclustered = sort_jets!(get_inclusive_jets(new_clusterseq))

    n = length(reclustered) <= filter.numHardestJets ? length(reclustered) : filter.numHardestJets
    hard = reclustered[1:n]

    filtered = join(hard)

    filtered
end;
```



```julia
struct Trim
    trimRadius::Float64
    trimFraction::Float64
    reclusterMethod::Int
end;

function apply_trim(jet::PseudoJet, clusterseq::ClusterSequence, trim::Trim)
    rad = trim.trimRadius;
    mtd = trim.reclusterMethod
    frac2 = trim.trimFraction ^ 2

    new_clusterseq = recluster(jet, clusterseq, rad, 0)
    reclustered = sort_jets!(get_inclusive_jets(new_clusterseq))

    hard = Vector{PseudoJet}(undef, 0)
    for item in reclustered
        if pt2(item) >= frac2 * pt2(jet)
            push!(hard, item)
        end
    end
    trimmed = join(hard)

    trimmed
end;
```

Jet Filtering

The results of filtering in Julia was compared with Python, but this time, we visualise it a bit differently. In the adjoining plots, the y-axis represents the groomed parameters as obtained from Julia, and the x-axis represents the groomed parameters obtained from Python.

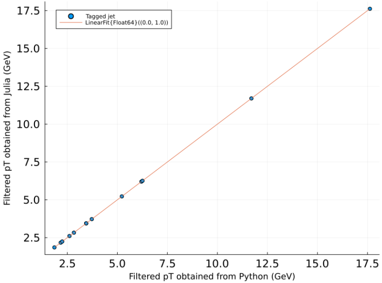


Figure 3: Comparing filtered pT

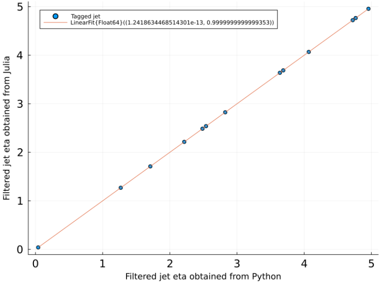


Figure 4: Comparing filtered eta

Jet Trimming

A similar analysis for the jet trimming algorithm was done, and the following plots were obtained.

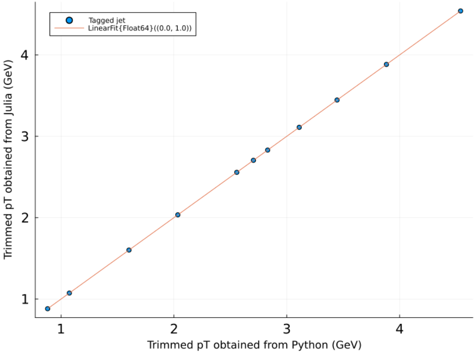


Figure 5: Comparing trimmed pT

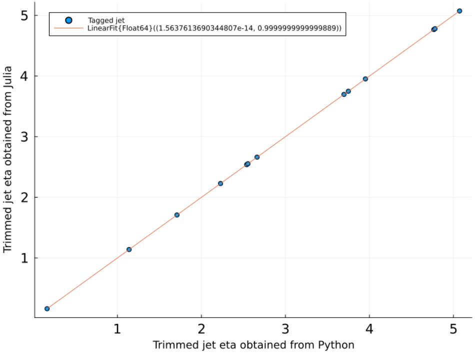


Figure 6: Comparing trimmed eta

1 Introduction

2 Implementation of Substructure Modules in Julia

3 Comparison of Performance with Pre Existing Bindings

4 Conclusion

Jet Filtering & Trimming

Clustering Algorithm: Cambridge-Aachen (with R = 0.6)

Jet Filtering parameters:

$R_{filt} = 0.3,\ n_{filt} = 3$

Jet Trim parameters:

$R_{trim} = 0.3,\ f_{trim} = 0.3$

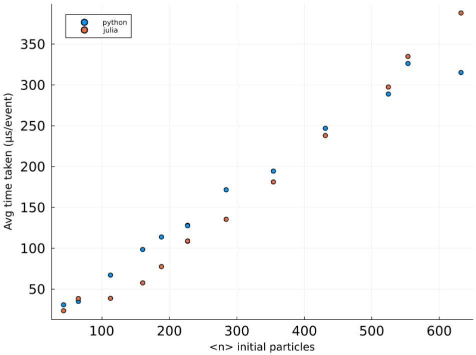


Figure 7: Time Comparison:Jet Filtering

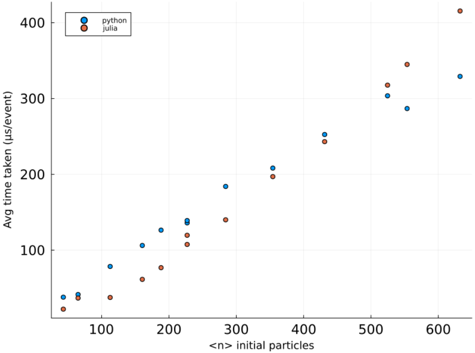


Figure 8: Time Comparison:Jet Trim

Mass Drop Tagger & JH Top Tagger

Mass Drop parameters:

$$\mu_{cut} = 0.67, \; y_{cut} = 0.09$$

JH Top Tagging parameters:

$$\Delta p_T = 0.1, \; \Delta r = 0.19, \; \Delta \cos\theta = 0.7$$

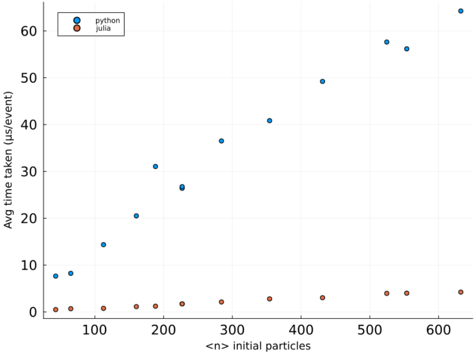


Figure 9: Time Comparison: Mass Drop Tagging

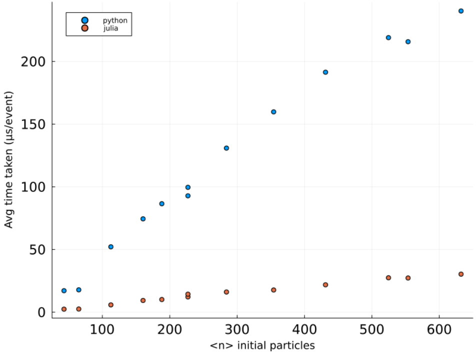


Figure 10: Time Comparison: JH Top Tagging

Overall Comparison

The following table contains the ratio of execution times, for each method, in Python to the corresponding Julia implementation, for various average no. of particles **(<n>)**

<n>	**Filter**	**Trim**	**MassDrop**	**JH Top Tag**
43	1.31	1.71	15.32	7.13
113	1.74	2.09	18.85	8.99
188	1.47	1.64	25.42	8.58
227	1.17	1.29	15.59	6.94
355	1.07	1.06	14.64	9.03
525	0.97	0.96	14.56	7.98
633	0.81	0.79	15.19	7.93

Table 1: Ratio of execution times **(Python:Julia)** for each method

1 Introduction

2 Implementation of Substructure Modules in Julia

3 Comparison of Performance with Pre Existing Bindings

4 Conclusion

Plans Moving Forward

- Optimise and improve the built modules
- Add the remaining modules
- Contribute to the already existing code base
- Add support for flavored jet tagging

I have added all the codes I have developed till now in this GitHub repository:
julia-JetSubstructure(https://github.com/sattwamo/julia-JetSubstructure)

THANK YOU.