Jet Substructure in Julia

Sattwamo Ghosh

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

Supervised by **Dr. Sanmay Ganguly**
Asst. Professor, Indian Institute of Technology, Kanpur

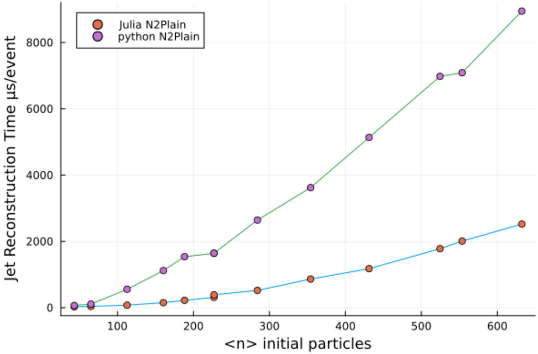
June 20, 2024

Initial Benchmarks

Comparing $N2$Plain Strategy: Julia and Python

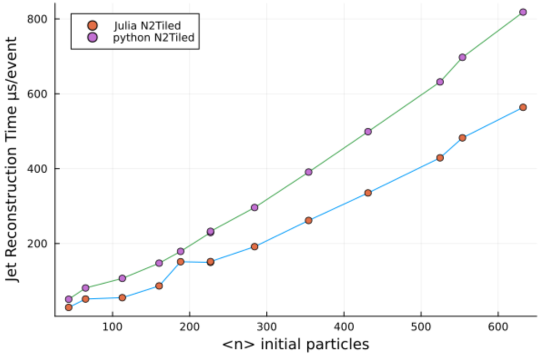
In this strategy, every particle is compared with every other particle to determine the nearest neighbors according to the chosen distance metric (e.g., k_t, anti-k_t, or Cambridge/Aachen)

The adjoining plot shows that, similar to C++, the python bindings show poor scaling compared to Julia at higher particle density



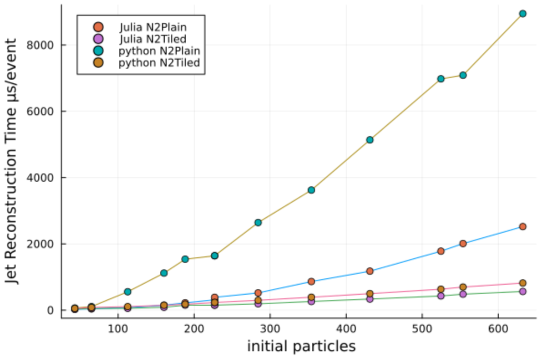
Comparing *N*2Tiled Strategy: Julia and Python

In this strategy, the space in which the particles are distributed (usually the rapidity-azimuth plane) is divided into a grid of tiles. Each particle is assigned to a tile based on its coordinates. Instead of comparing every particle with every other particle, the algorithm only compares particles within the same tile and neighboring tiles. This reduces the number of distance calculations significantly.



It can be seen that like C++, there is small advantage of Julia over Python also.

Overall comparison: Julia and Python



From this graph, we see that, in terms of efficiency we have:

$$N2TiledJulia > N2TiledPython > N2PlainJulia > N2PlainPython$$

With N2Tiled Julia being the most efficient, for higher particle density.

Jet Substructure Modules

Jet Filtering

The jet filtering algorithm was implemented in Julia, and the corresponding result was compared with Python. 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.

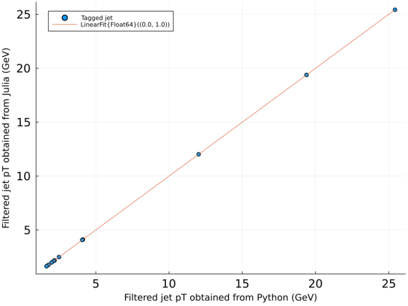


Figure: Comparing filtered pT

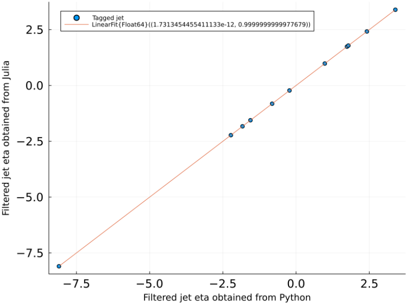
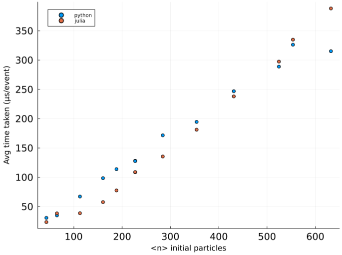
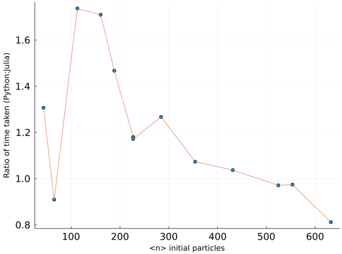


Figure: Comparing filtered eta

Jet Filtering: Efficiency

The adjoining plot compares the efficiency of the jet filtering algorithm implemented in Python and Julia.





Jet Trimming

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

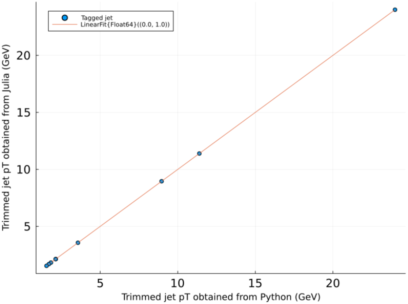


Figure: Comparing trimmed pT

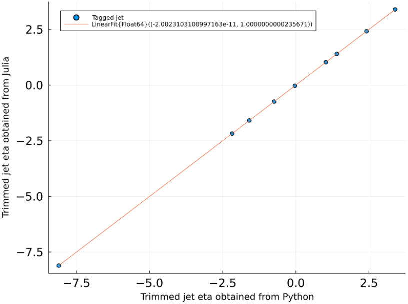
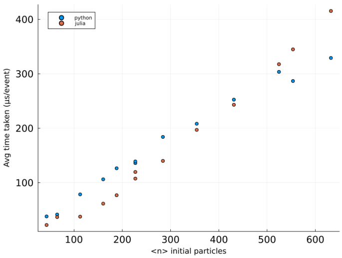
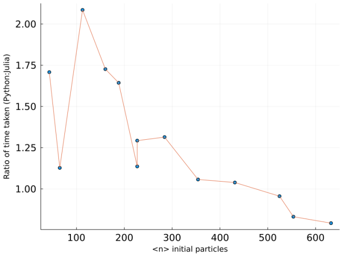


Figure: Comparing trimmed eta

Jet Trimming: Efficiency





The adjoining plot compares the efficiency of the jet trimming algorithm implemented in Python and Julia. This follows a similar trend to that of jet filtering.

Filtering & Trimming: Codes

◆ 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.

```
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;
```

```
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;
```

Mass Drop Tagger

The MassDrop Tagging algorithm was similarly implemented in Julia and compared with the corresponding Python bindings, for the same set of data. The obtained results were concurrent with each other (shown in the plots below).

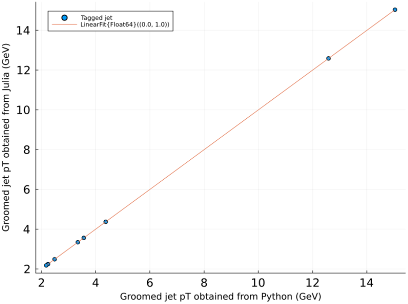


Figure: Comparing groomed pT

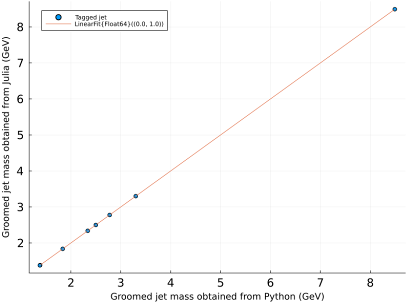
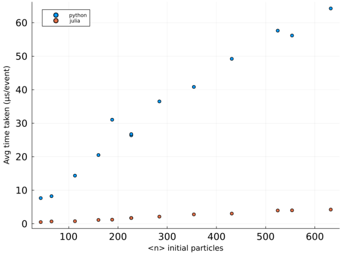
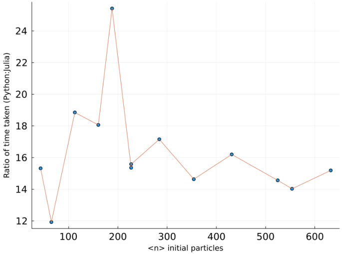


Figure: Comparing groomed mass

Mass Drop Tagger: Efficiency

As can be seen from the adjoining plot, the implemented Julia MassDrop Tagger is much more efficient compared to the corresponding Python bindings.





Mass Drop Tagger: Codes

```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;
```

Soft Drop Tagger

The SoftDrop Tagging algorithm was also implemented in Julia, but this time we compared it with the C++ FastJet Bindings, for the same set of data. The obtained results were concurrent with each other (shown in the plots below).

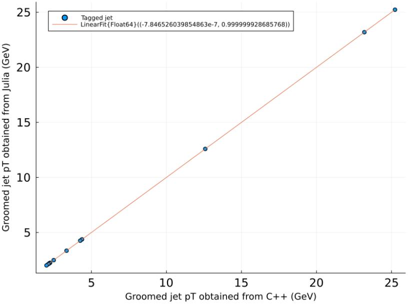


Figure: Comparing groomed pT

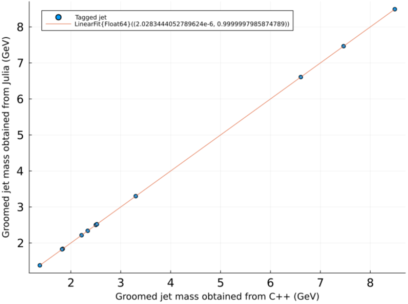


Figure: Comparing groomed mass

Plans Moving Forward

What More To Do

- Optimise and improve the built modules
- Add the remaining modules
- Build an analysis module
- Contribute to the already existing code base

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