Polyglot Jet Finding

Graeme Andrew Stewart[1,*], *Philippe* Gras[2], *Benedikt* Hegner[1], and *Atell* Krasnopolski[3]

[1]CERN, Esplanade des Particules 1, Geneva, Switzerland
[2]IRFU, CEA, Université Paris-Saclay, Gif-sur-Yvette, France
[3]Taras Shevchenko National University of Kyiv, Ukraine

Abstract. The evaluation of new computing languages for a large community, like HEP, involves comparison of many aspects of the languages' behaviour, ecosystem and interactions with other languages. In this paper we compare a number of languages using a common, yet non-trivial, HEP algorithm: the anti-k_{T} clustering algorithm used for jet finding. We compare specifically the algorithm implemented in Python (pure Python and accelerated with numpy and numba), and Julia, with respect to the reference implementation in C++, from Fastjet. As well as the speed of the implementation we describe the ergonomics of the language for the coder, as well as the efforts required to achieve the best performance, which can directly impact on code readability and sustainability.

1 Introduction

High energy physics (HEP), as a discipline, has undergone at least two major shifts in language after the widespread adoption of Fortran in the 1960s [1]. A first was a significant shift from Fortran to C++, starting with the BaBar experiment, then gathering pace at the end of the the Large Electron-Positron Collider (LEP) era, c. 2000, when the Large Hadron Collider (LHC) experiments adopted C++ more or less wholesale. The second shift happened with the gradual incorporation of Python into the language ecosystem of HEP, from about 2010.

In the first transition, Fortran was almost completely displaced by C++ in the HEP experiments; in the theory domain the evolution was more gradual and mixed, with Fortran and C++ still both used today. In the second, a different type of transition took place, where Python became more and more popular, but co-exists with C++. The C++ is largely used in performance critical areas, with Python finding traction when flexibility and rapid turn-around is needed, e.g., in configuration and steering. Python code is typically used to interface to higher performance C and C++ libraries, both generic (e.g., numpy) and specific HEP codes.

Although the field is, for reasons of stability and legacy, slow to move to new languages, there are some significant issues with the current language choices that make an exploration of alternatives worthwhile. For example, the interfaces between Python and C++ are a source of friction, both for passing data and error messages back and forth, as well as being obliged to switch languages and reimplement code on occasion, when moving from a prototype to production (assuming the the developer actually has skills in both languages, which is not a given). This *two language problem* has potentially been addressed in the *Julia* programming language [2, 3], with promising prospects for HEP, in particular, [4, 5] as well as other

*e-mail: graeme.andrew.stewart@cern.ch

Language	Repository
C++ (FastJet)	FastJet Website (release 3.4.1)
Python (Pure)	GitHub antikt-python
Python (Accelerated)	GitHub antikt-python
Julia	GitHub JetReconstruction.jl

Table 1. Code repositories used in this paper. See [9] for exact commits and instructions.

STEM[1] areas [6]. Julia offers just in time compilation giving an ergonomic experience much like Python, but with runtime speeds comparable to C and C++. C++ is also a notoriously tricky language to use, particularly related to memory handling [7] and HEP C++ codes are frequently riddled with code defects [8].

Evaluation of the prospects for a language in any particular domain area should be done with a real problem from that domain, rather than any synthetic benchmark. In this paper we look at the problem of jet finding, or clustering, which is a use case from high energy physics used in calorimeter reconstruction. This is a good example as it is not trivial, but it is also not so complex that different implementations take too long to write.

The languages we examine here, along with links to the code used, are given in Table 1.

The evaluation itself can cover many aspects of a programming language and the experience of using it. Metrics such as runtime are easy to evaluate, but the ergonomics of using particular languages and the support offered by the language ecosystem for developers are also critical and we comment on these.

2 Anti-k_{T} Jet Clustering Algorithm

2.1 Algorithm

The anti-k_{T} clustering algorithm [10, 11] is an infrared and colinear safe jet clustering algorithm, which is robust against soft fragmentation components. We use the Fastjet implementation [12], that proceeds in the following way:

1. A radius parameter, R, is defined (0.4 is typical at the LHC).
2. For each active pseudojet A (that is, an initial particle or a merged cluster):
   (a) Considering all other PseudoJets, B, which are closer in geometric distance than R, measure the minimum geometric distance:
   $$d = \min\left(\sqrt{\Delta\eta_{AB}^2 + \Delta\phi_{AB}^2}\right),$$
   where $\Delta\eta_{AB}$ and $\Delta\phi_{AB}$ are the rapidity and azimuthal angle differences between A and B.
   If there are no other pseudojets within R, then $d = R$ for pseudojet A.
   (b) Define the anti-k_{T} distance, d_{ij}, as $d_{ij} = d\min(k_{\mathrm{T},A}^{-2}, k_{\mathrm{T},B}^{-2})$ where $k_{\mathrm{T},\{A,B\}}$ is the transverse momentum of the pseudojet $\{A, B\}$. If there is no neighbouring pseudojet, $d_{ij} = d\,k_{\mathrm{T},A}^{-2}$.
3. Choose the pseudojet with the lowest d_{ij}.

[1] Science, technology, engineering, and mathematics

(a) If this pseudojet has an active partner, B, merge these two pseudojets to a new pseudojet.

(b) If not, this jet is finalised and removed from the active list.

4. Repeat until no pseudojets remain active.

Note that the definition of d_{ij}, so-called anti-k_T, with a negative power favours merging jets with a high transverse momentum first, which provides stability against soft radiation, hence its popularity. (Considering a general metric distance of k_T^{2p}, $p = -1$ is anti-k_T merging, $p = 0$ is Cambridge/Achen merging and $p = 1$ is inclusive k_T, [11].)

The algorithm itself has a nice mixture of parallelisation opportunities (pairwise matching of pseudojet candidates) and serial steps (finding the minimum values of d or d_{ij}), which is a good test of a non-naive algorithm's performance.

2.2 Algorithm Implementations

We consider two different implementations of the algorithm described above, taken from FastJet [12, 13].

The first is a *plain implementation* in which, at each step, all jets are considered as possible neighbours of each other. This algorithm has scaling that runs roughly as N^2, where N is the number of initial particle hits (this is an improvement over the most naive scaling which would be N^3 [10]). This implementation is fastest for $\lesssim 30$ particles.

The second is a *tiled implementation*, in which the geometric space (η, ϕ) is split into tiles of size R. In this way the number of possible neighbours of any particular jet is limited to the jet's tile and to its immediate neighbours, as illustrated in Figure 1. This strategy reduces the amount of work that needs to be done, at the expense of extra bookkeeping of which jets are in each tile. This implementation is fastest for p-p collisions at the LHC.

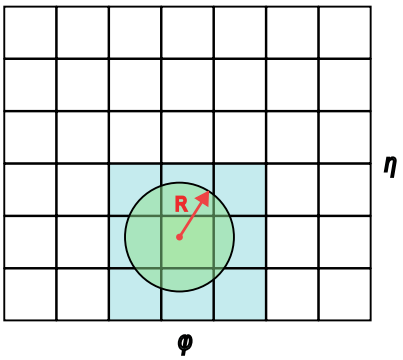

Figure 1. In the tiled algorithm implementation (η, ϕ) space is split into tiles of size R. When a pseudojet needs to rescan for neighbours (red dot) only pseudojets in tiles within the distance R need to be considered, here shaded in light blue.

3 Code Implementation Ergonomics

The code versions used are linked to in Table 1. We highlight some specific observations in this section.

3.1 C++, FastJet

The FastJet package [12, 13] is a well maintained code which is widely used in the HEP community. It is code which is of high quality and well scrutinised and tested. The general style of the code is more akin to C than C++, for reasons of minimising abstraction and increasing speed, although templates are used extensively (where any errors are not usually nicely handled by the compiler).

For the tiled implementation, a linked list structure is used, which requires pointers to pointers that are challenging to reason about for the programmer, as illustrated below.

```
// set up the initial nearest neighbour information
vector<Tile>::const_iterator tile;
for (tile = _tiles.begin(); tile != _tiles.end(); tile++) {
  // first do it on this tile
  for (jetA = tile->head; jetA != NULL; jetA = jetA->next) {
    for (jetB = tile->head; jetB != jetA; jetB = jetB->next) {
      double dist = _bj_dist(jetA,jetB);
      if (dist < jetA->NN_dist) {jetA->NN_dist = dist;
        jetA->NN = jetB;}
      if (dist < jetB->NN_dist) {jetB->NN_dist = dist;
        jetB->NN = jetA;}
    }
  }
  // then do it for RH tiles
  for (Tile ** RTile = tile->RH_tiles; RTile != tile->end_tiles; RTile++) {
    for (jetA = tile->head; jetA != NULL; jetA = jetA->next) {
      for (jetB = (*RTile)->head; jetB != NULL; jetB = jetB->next) {
        double dist = _bj_dist(jetA,jetB);
        if (dist < jetA->NN_dist) {jetA->NN_dist = dist;
          jetA->NN = jetB;}
        if (dist < jetB->NN_dist) {jetB->NN_dist = dist;
          jetB->NN = jetA;}
      }
    }
  }
}
```

3.2 Python

3.2.1 Pure Python

Python is renowned for being a high productivity language and implementation of the jet finding algorithms is rather straightforward, with a clear logic. Mutability of classes allows code to be shared between the different implementations. e.g., an update scan for the basic case looks like this:

```
  def scan_for_my_nearest_neighbours(jetA: PseudoJet, jets: list[PseudoJet],
    R2: float):
    "Retest all other jets against the target jet"
    jetA.info.nn = None
    jetA.info.nn_dist = R2
```

```
    for ijetB, jetB in enumerate(jets):
        if not jetB.info.active:
            continue
        if ijetB == jetA.info.id:
            continue
        dist = geometric_distance(jetA, jetB)
        if dist < jetA.info.nn_dist:
            jetA.info.nn_dist = dist
            jetA.info.nn = ijetB
    jetA.info.akt_dist = antikt_distance(jetA, jets[jetA.info.nn]
      if jetA.info.nn else None)
```

Where specifically `info` is a mix-in class for bookkeeping pseudojets.

While the code for the tiled implementation involves more bookkeeping, it also remains clear.

3.2.2 Acelerated Python

Accelerated Python code, where both numba and numpy are employed brings some added difficulty. Not all operations are easily expressed as numpy array calculations, particularly for dynamic arrays holding active and inactive jets. This necessitated the use of masks, which need to be tracked. In addition, numba jitted functions are very picky on types that can be passed (at least without being explicitly *taught* how to deal with them), so instead of a structure, functions are called with many individual array elements, leading to complicated call signatures. e.g., the same function as above becomes

```
@njit
def scan_for_my_nearest_neighbours(ijet:int, phi: npt.ArrayLike,
                                   rap:npt.ArrayLike, inv_pt2:npt.ArrayLike,
                                   dist:npt.ArrayLike, akt_dist:npt.ArrayLike,
                                   nn:npt.ArrayLike,
                                   mask:npt.ArrayLike, R2: float):
    "Retest all other jets against the target jet"
    nn[ijet] = -1
    dist[ijet] = R2
    _dphi = np.pi - np.abs(np.pi - np.abs(phi - phi[ijet]))
    _drap = rap - rap[ijet]
    _dist = _dphi*_dphi + _drap*_drap
    _dist[ijet] = R2 # Avoid measuring the distance 0 to myself!
    _dist[mask] = 1e20 # Don't consider any masked jets
    iclosejet = _dist.argmin()
    dist[ijet] = _dist[iclosejet]
    if iclosejet == ijet:
        nn[ijet] = -1
        akt_dist[ijet] = dist[ijet] * inv_pt2[ijet]
    else:
        nn[ijet] = iclosejet
        akt_dist[ijet] = dist[ijet] * (inv_pt2[ijet] if inv_pt2[ijet]
          < inv_pt2[iclosejet] else inv_pt2[iclosejet])
        # As this function is called on new PseudoJets it's possible
```

```
        # that we are now the NN of our NN
        if dist[iclosejet] > dist[ijet]:
            dist[iclosejet] = dist[ijet]
            nn[iclosejet] = ijet
            akt_dist[iclosejet] = dist[iclosejet] * (inv_pt2[ijet]
              if inv_pt2[ijet] < inv_pt2[iclosejet] else inv_pt2[iclosejet])
```

numba also has some surprising omissions from the numpy functions which it can JIT, e.g., array index ravelling, that required explicit reimplementation.

3.3 Julia

Julia is gaining in popularity because it is a language that is easy to use. We found numerous nice features that allow code to be clear, e.g., using the broadcast syntax for calculations on arrays is very compact:

```
kt2 = (JetReconstruction.pt.(objects) .^ 2) .^ p
```

Here `.^` (raise to power) operates on each member of the `pt` value of the `objects` array.

Like the FastJet code, loops can be used without sacrificing speed, so the code checking for new nearest neighbours is

```
# Finds new nearest neighbour for pseudojet i
# and cross checks distance for other pseudojets back to i
# Note that nndist, near_neighbour, eta and phi are *Vectors*
function update_nearest_neighbour_crosscheck!(nndist, near_neighbour,
  i::Int, from::Int, to::Int, eta, phi, R2)
    new_nndist = R2
    new_nn = i
    @inbounds @simd for j in from:to
        delta2 = dist(i, j, eta, phi)
        if delta2 < new_nndist
            new_nn = j
            new_nndist = delta2
        end
        if delta2 < nndist[j]
            nndist[j] = delta2
            near_neighbour[j] = i
        end
    end
    nndist[i] = new_nndist
    near_neighbour[i] = nn;
end
```

Note there are some optimisations applied here as Julia *macros*, e.g., `@simd`, which we discuss below. In particular, here we present updated results from those at the conference for the tiled algorithm in Julia from applying the `LoopVectorisation` package in a key area adding the `@turbo` macro:

```
find_best(diJ, n) = begin
    best = 1
```

```
    @inbounds diJ_min = diJ[1]
    @turbo for here in 2:n # Loop vectorisation marco
        newmin = diJ[here] < diJ_min
        best = newmin ? here : best
        diJ_min = newmin ? diJ[here] : diJ_min
    end
    diJ_min, best
end
```

4 Code Performance

The different implementations of the anti-k_T algorithm were tested on the same benchmark machine, a 64 core AMD EPYC 7302 3.00GHz with 24GB RAM, running CentOS7. The software versions used were gcc 11.3.0, Python 3.11.4 (with numba 0.57.1 and numpy 1.24.4) and Julia 1.9.2. More details on how to reproduce the measurements are given in [9].

Reconstruction of 100 LHC-like pp events[2] was run multiple times and the average reconstruction time per event is given in Table 2. These numbers are normalised to the FastJet tiled algorithm performance (which is 324 μs per event on the benchmark machine). Multiple repeats of the benchmark were done and jitter was observed to be extremely low, $< 1\%$, so is not given. In these measurements the time to read the events (in HepMC3 format) and the JIT time for Julia and numba is excluded.

Implementation	Basic Algorithm	Tiled Algorithm
C++ (FastJet)	16.4	1.00
Python (Pure)	504	110
Python (Accelerated)	28.5	113
Julia	2.83	0.94

Table 2. Relative run times for the reconstruction of 100 13TeV pp events, normalised to the time for FastJet's tiled algorithm. Results are stable and reproducible on the benchmark machine at $< 1\%$.

We observe that the benchmark C++ FastJet code, with the tiled algorithm, is one of fastest implementations. The increase in performance for the tiled code, over the plain one, is significant with the events we used, confirming this is both an excellent algorithm and implementation for LHC p-p data.

As expected, the pure Python codes run very slowly in comparison. More surprisingly, the accelerated Python codes have quite poor performance as well. This is due to the fact that not all parts of the algorithm can be accelerated - bookkeeping operations still run in normal Python and become dominant in the overall runtime. This is particularly true of the tiled algorithm, which deliberately reduces the work to be done (which can be parallelised and accelerated) at the cost of more bookkeeping. This significantly hurts the accelerated implementation, which ends up slower than the basic accelerated implementation; it is not even faster than the pure Python tiled implementation code.

Our Julia code exceeds the performance of FastJet code. In the case of the tiled algorithm, as noted in Section 3.3, a *loop vectorisation* optimisation was applied to the search across all d_{ij} to find the minimum value, which results in a 15% improved runtime on x86 architectures

[2] Hard QCD $2 \to 2$ processes generated with Pythia8 at 13TeV, with a minimum transverse momentum of 20 GeV.

cf. without this macro[3]. In the case of the basic algorithm the Julia code uses a structure of arrays layout, which the compiler can highly optimise; additional benefit is gained from macros like `@simd`, which allow the compiler to apply further optimisations, gaining an additional 5%.

There are some comments regarding these optimisations that should be made: the Julia compiler attempts to use SIMD instructions in any case; when using the `@simd` macro the developer is guaranteeing iterations are safe to reorder and to overlap, and that floating point operations can be reordered; `@turbo` also replaces some special functions with implementations that can be vectorised better, but may be of lower accuracy. Use of these marcos may lead to different numerical results so must be carefully validated (in our case we have checked that they are safe). One advantage in Julia is that these macros, as well as `@fastmath`, can be used and validated on a case-by-case basis (cf. the C++ compiler options such as `-O3` or `-ftree-vectorize`, which are applied per compilation unit, but more than likely are actually used globally). In addition, the JIT strategy of Julia and Python's numba will automatically target the binary architecture of the machine being used, avoiding portability issues that can hamper C++ compiled binaries on different microarchitectures.

5 Conclusions

We have implemented the anti-k_{T} algorithm in a number of different languages and examined code ergonomics as well as run time performance. The benchmark C++ code from FastJet is well written, but the hardest to reason on correctness, due to the nature of the language. Python is excellent for code logic and flexibility, but has a very poor run time performance; accelerating with numpy and numba unfortunately takes much of this advantage away, yet still fails to achieve a competitive run time. Julia performs extremely well, with an excellent 'out of the box' run time. The Julia compiler is able to find significant speed-ups and features like broadcast operators help to keep code clean and quick. Further, applying optimisations in Julia through the use of macros is extremely easy for the programmer to exploit and result in Julia having the best performance of all the codes that we tested.

It should be noted that the optimisations found by the Julia compiler could also be applied to the FastJet code to close the gap. However, the authors' experience is that doing this in C++ is considerably more difficult.

Ergonomically, C++ is also the most difficult language to use, with no package manager, no built in profiler, and where templates and memory management remain tricky. The breadth of libraries in C++ is impressive, although managing dependencies is not easy. In Python the situation is far better, albeit that the package managers are not quite standardised (pip vs. conda/mamba). Profiling and debugging when accelerated code is used in Python (which is how Python is used in data intensive science) is not easy, but package support in Python is really excellent. In Julia the ecosystem is very well integrated, with a built in package manager and excellent reproducibility. Julia libraries are not as extensive as for C++ and Python, although the speed of development of new scientific libraries (which is Julia's target community) is picking up quickly and most areas are covered (see the discussion in Eschel et al. [5]). Debugging and profiling in Julia are very well integrated.

We conclude that expanding the use of Julia in high energy physics would be very worthwhile, given its excellent performance and ergonomics.

[3] On Apple's M2Pro chip, the advantage for Julia is more significant, with the final Julia code running ×1.45 faster than FastJet for the tiled implementation cases

References

[1] J. Pisarski, *History and adoption of programming languages in NHEP* (2022), https://indico.jlab.org/event/505/contributions/9207/

[2] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, SIAM Review **59**, 65 (2017)

[3] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V.B. Shah, J. Vitek, L. Zoubritzky, Proc. ACM Program. Lang. **2** (2018)

[4] M. Stanitzki, J. Strube, Comput. Softw. Big Sci. **5**, 10 (2021), 2003.11952

[5] J. Eschle, T. Gal, M. Giordano, P. Gras, B. Hegner, L. Heinrich, U.H. Acosta, S. Kluth, J. Ling, P. Mato et al., *Potential of the Julia programming language for high energy physics computing* (2023), 2306.03675, https://doi.org/10.48550/arXiv.2306.03675

[6] J.M. Perkel, Nature **572**, 141 (2019)

[7] M. Miller, *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape* (2019), https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[8] A. Naumann, Journal of Physics: Conference Series **513**, 052023 (2014)

[9] G.A. Stewart, P. Gras, B. Hegner, A. Krasnopolski, *Polyglot jet finding - LaTeX sources and benchmark instructions* (2023), https://doi.org/10.5281/zenodo.8307668

[10] M. Cacciari, G.P. Salam, Phys. Lett. B **641**, 57 (2006), hep-ph/0512210

[11] M. Cacciari, G.P. Salam, G. Soyez, Journal of High Energy Physics **2008**, 063 (2008)

[12] M. Cacciari, G.P. Salam, G. Soyez, Eur. Phys. J. C **72**, 1896 (2012), 1111.6097

[13] *Fastjet*, https://fastjet.fr