Jet Reconstruction with Julia

Atell-Yehor Krasnopolski

IRIS-HEP Fellow
Taras Shevchenko National University of Kyiv

Benedikt Hegner
CERN

Graeme A Stewart
CERN

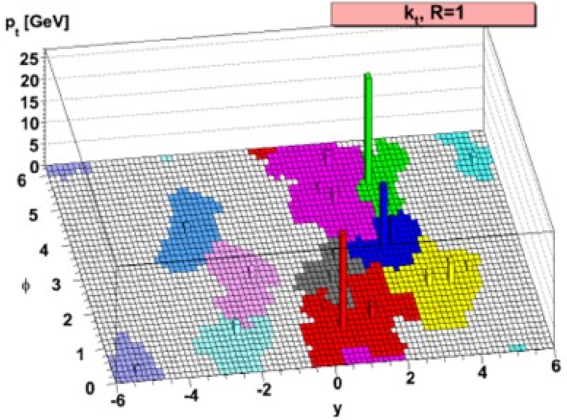
Jet Reconstruction

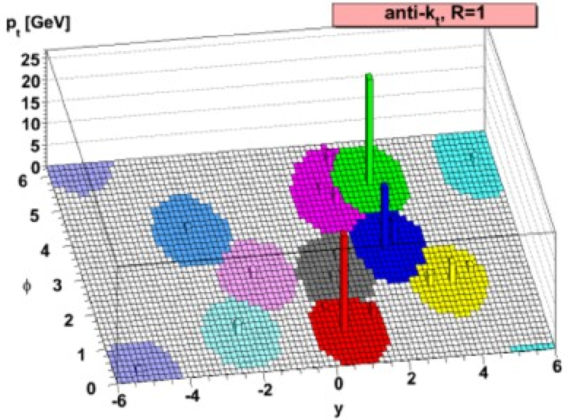
- Cluster together measurement points of the same origin
- Algorithm defines the jets
- Iterative process involving computing distances

$$d_{ij} = \min(k_{ti}^{2p}, k_{tj}^{2p})\frac{\Delta_{ij}^2}{R^2},$$

$$d_{iB} = k_{ti}^{2p},$$

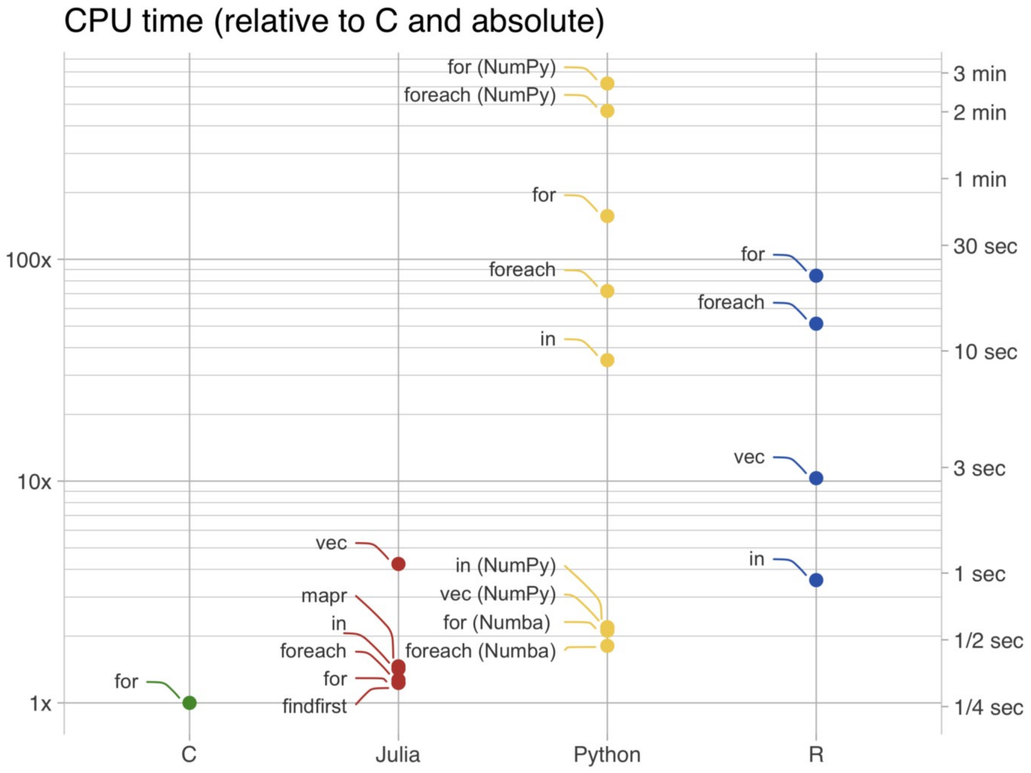
$$\Delta_{ij}^2 = (y_i - y_j)^2 + (\phi_i - \phi_j)^2$$





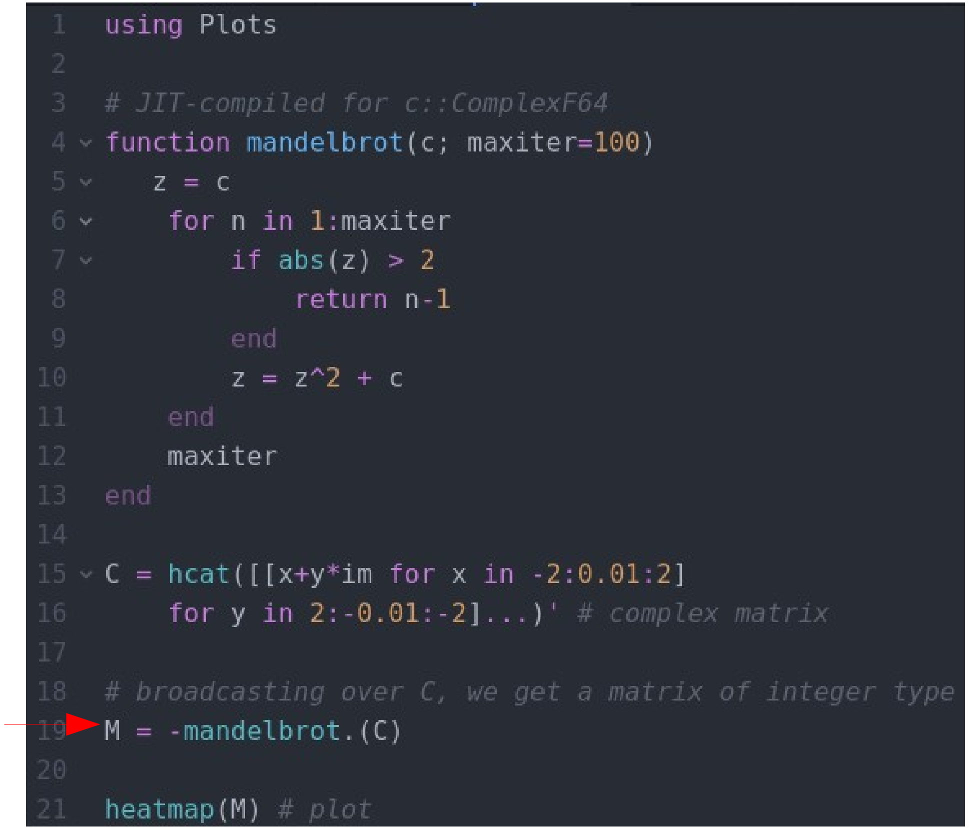
Why Julia (Speed)

- Python & C++ → Julia?
- Simple and efficient, way faster than Python
- Fast by design, not because of packages
- JIT-compiled
- Can interact with C, FORTRAN & Python

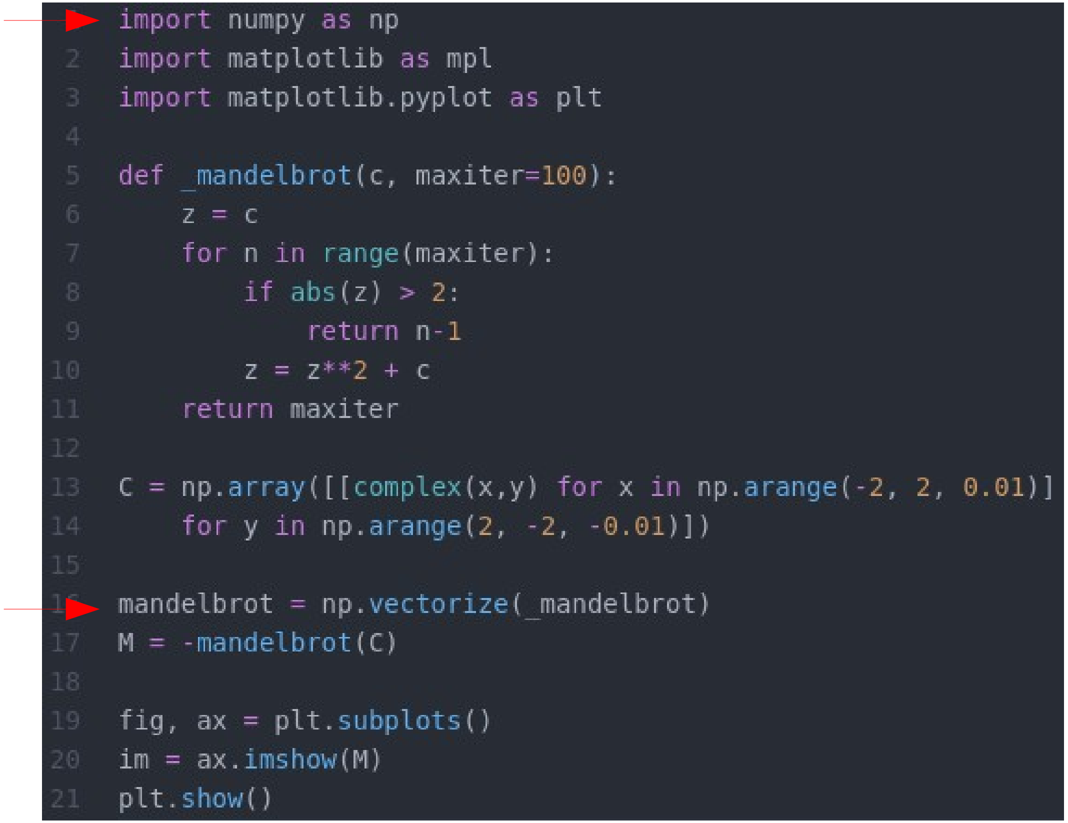


(towardsdatascience.com/r-vs-python-vs-julia-90456a2bcbab)

Why Julia (Convenience)



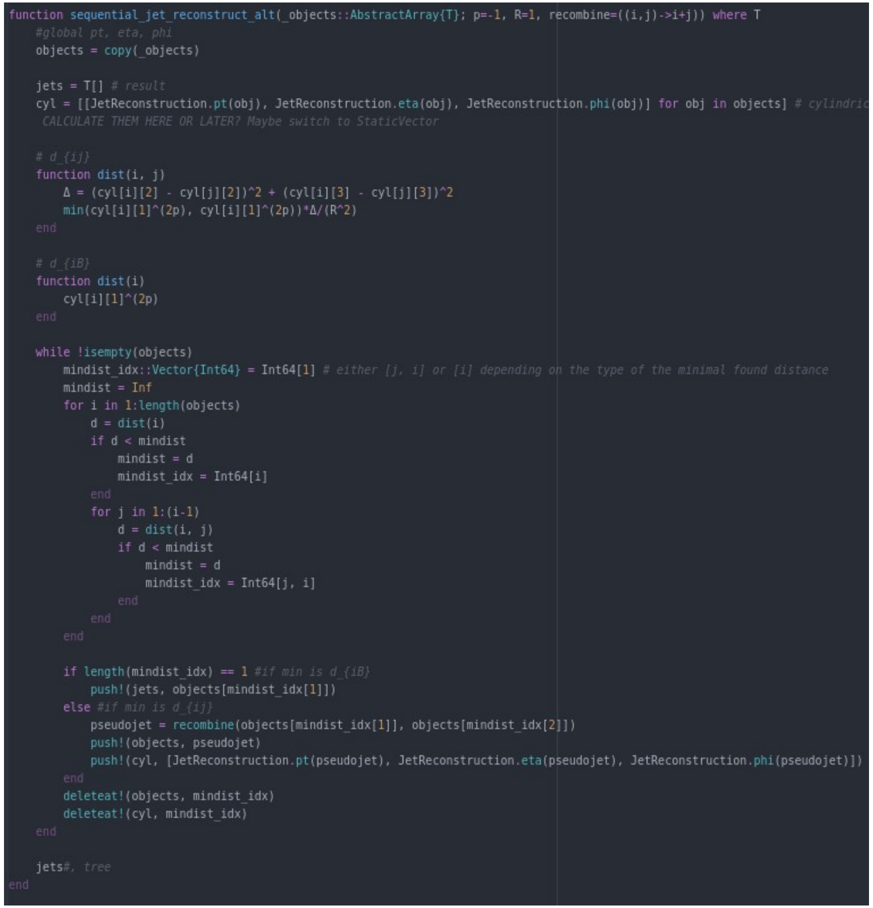
```julia
using Plots

# JIT-compiled for c::ComplexF64
function mandelbrot(c; maxiter=100)
    z = c
    for n in 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    maxiter
end

C = hcat([[x+y*im for x in -2:0.01:2]
    for y in 2:-0.01:-2]...)' # complex matrix

# broadcasting over C, we get a matrix of integer type
M = -mandelbrot.(C)

heatmap(M) # plot
```



```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

def _mandelbrot(c, maxiter=100):
    z = c
    for n in range(maxiter):
        if abs(z) > 2:
            return n-1
        z = z**2 + c
    return maxiter

C = np.array([[complex(x,y) for x in np.arange(-2, 2, 0.01)]
    for y in np.arange(2, -2, -0.01)])

mandelbrot = np.vectorize(_mandelbrot)
M = -mandelbrot(C)

fig, ax = plt.subplots()
im = ax.imshow(M)
plt.show()
```

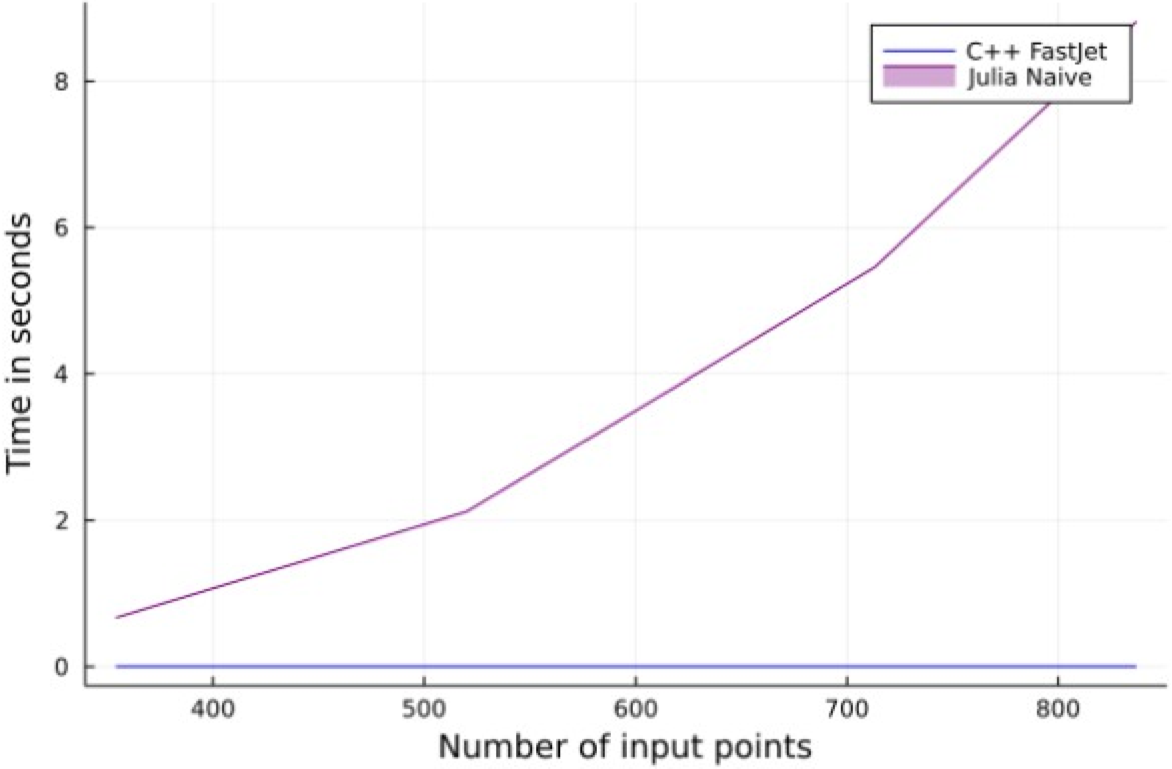
Naive Method

- Naive implementation takes ~42 lines of code
- Not a very sophisticated approach



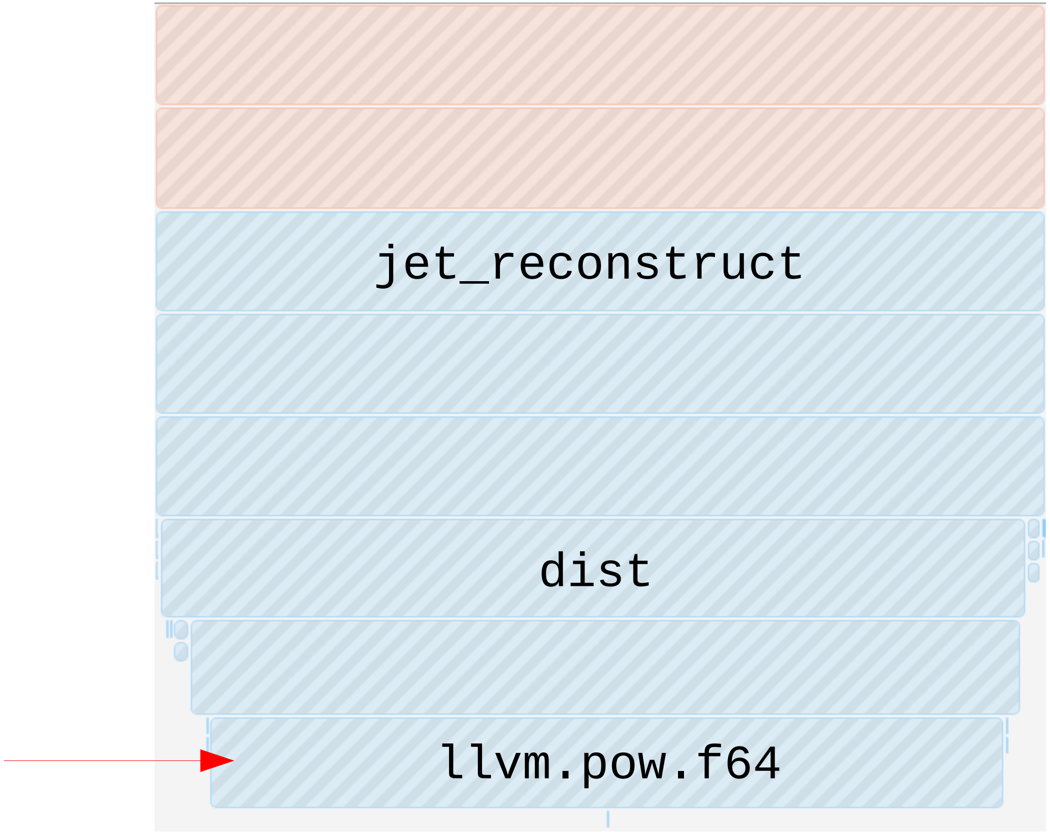
```
function sequential_jet_reconstruct_alt(_objects::AbstractArray{T}; p=-1, R=1, recombine=((i,j)->i+j)) where T
    #global pt, eta, phi
    objects = copy(_objects)

    jets = T[] # result
    cyl = [[JetReconstruction.pt(obj), JetReconstruction.eta(obj), JetReconstruction.phi(obj)] for obj in objects] # cylindric
     CALCULATE THEM HERE OR LATER? Maybe switch to StaticVector

    # d_{ij}
    function dist(i, j)
        Δ = (cyl[i][2] - cyl[j][2])^2 + (cyl[i][3] - cyl[j][3])^2
        min(cyl[i][1]^(2p), cyl[i][1]^(2p))*Δ/(R^2)
    end

    # d_{iB}
    function dist(i)
        cyl[i][1]^(2p)
    end

    while !isempty(objects)
        mindist_idx::Vector{Int64} = Int64[1] # either [j, i] or [i] depending on the type of the minimal found distance
        mindist = Inf
        for i in 1:length(objects)
            d = dist(i)
            if d < mindist
                mindist = d
                mindist_idx = Int64[i]
            end
            for j in 1:(i-1)
                d = dist(i, j)
                if d < mindist
                    mindist = d
                    mindist_idx = Int64[j, i]
                end
            end
        end

        if length(mindist_idx) == 1 #if min is d_{iB}
            push!(jets, objects[mindist_idx[1]])
        else #if min is d_{ij}
            pseudojet = recombine(objects[mindist_idx[1]], objects[mindist_idx[2]])
            push!(objects, pseudojet)
            push!(cyl, [JetReconstruction.pt(pseudojet), JetReconstruction.eta(pseudojet), JetReconstruction.phi(pseudojet)])
        end
        deleteat!(objects, mindist_idx)
        deleteat!(cyl, mindist_idx)
    end

    jets#, tree
end
```

Naive Method



FastJet is a package written in C++ that is considered a standard

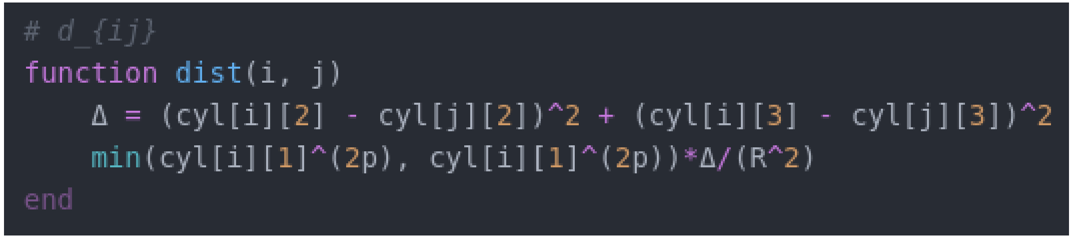
Profiling



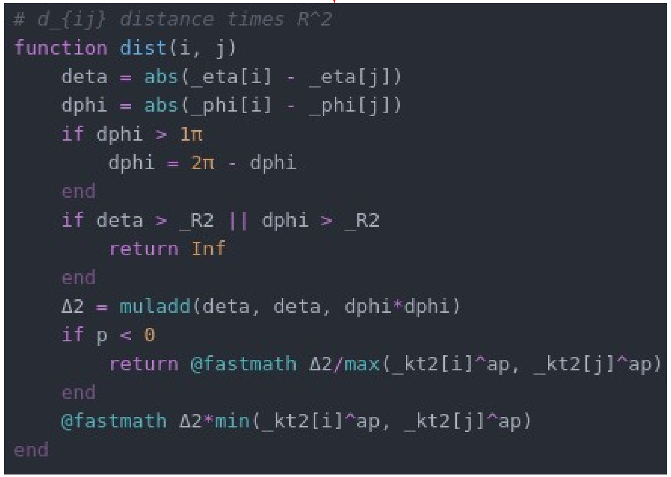
Julia is extremely easy to profile and optimise because of such native tools

Changes

- Cache the distances
- Optimise the distance function to avoid floating point powers
- Clean the code up

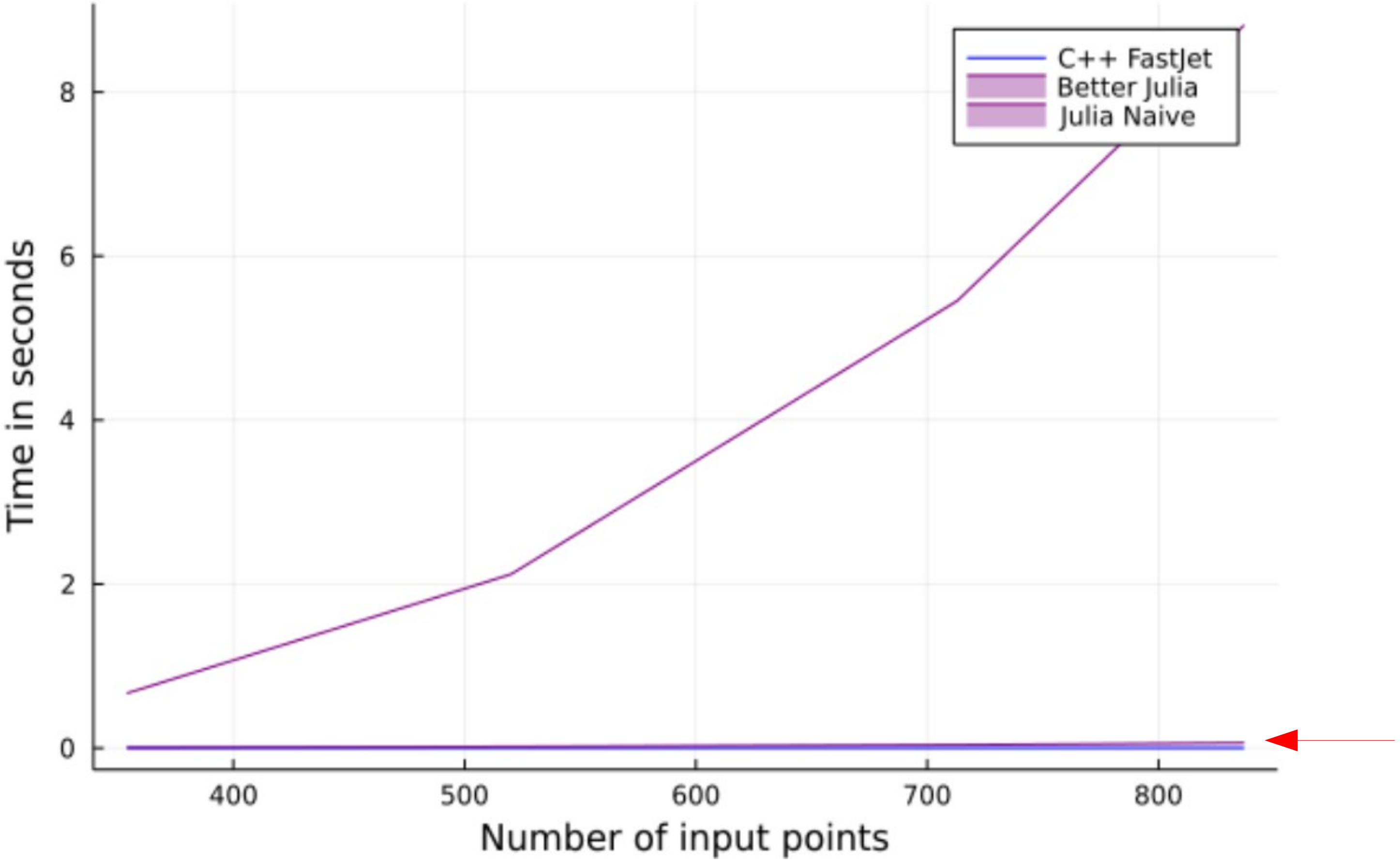


```julia
# d_{ij}
function dist(i, j)
    Δ = (cyl[i][2] - cyl[j][2])^2 + (cyl[i][3] - cyl[j][3])^2
    min(cyl[i][1]^(2p), cyl[i][1]^(2p))*Δ/(R^2)
end
```

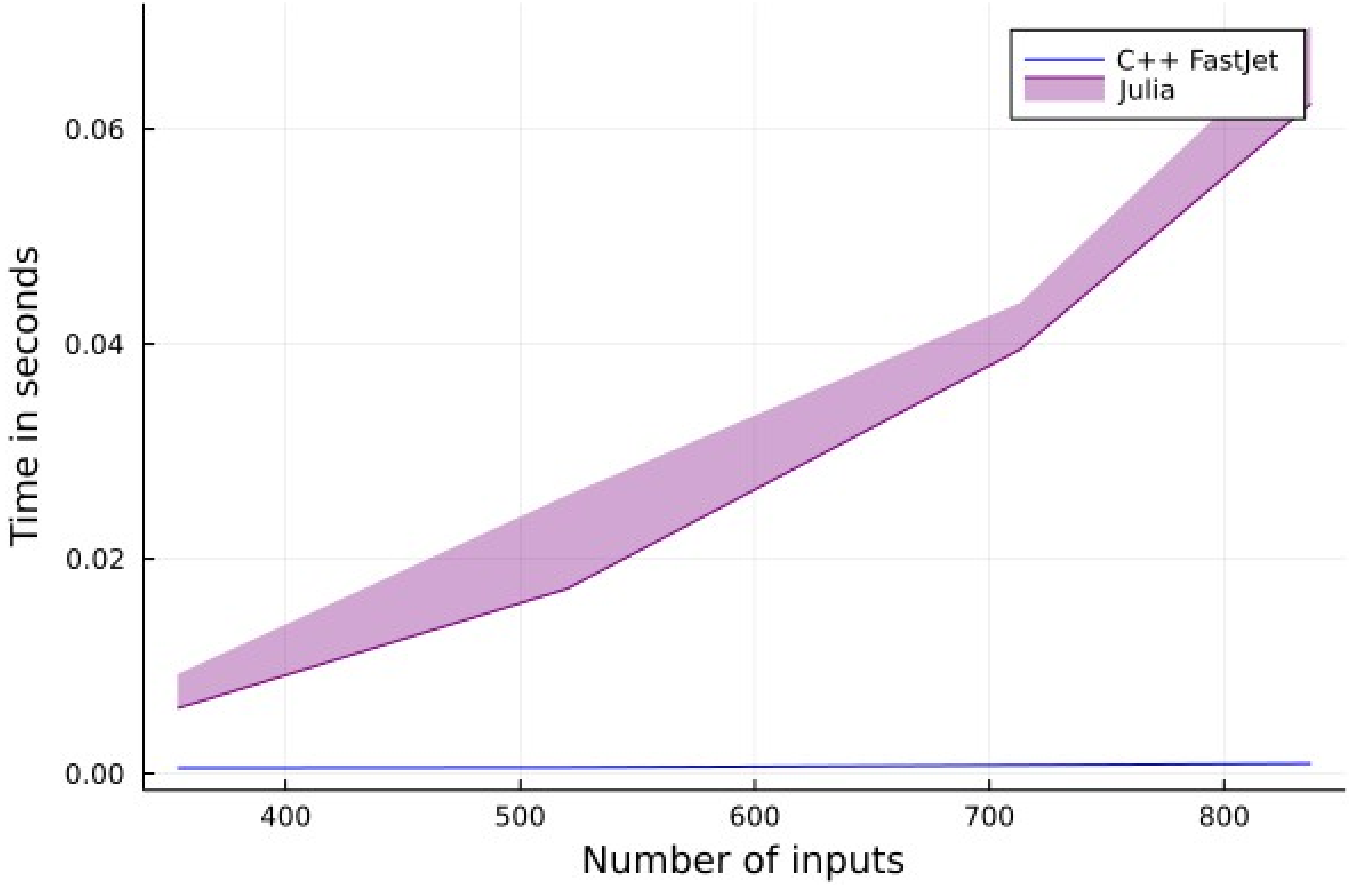


```julia
# d_{ij} distance times R^2
function dist(i, j)
    deta = abs(_eta[i] - _eta[j])
    dphi = abs(_phi[i] - _phi[j])
    if dphi > 1π
        dphi = 2π - dphi
    end
    if deta > _R2 || dphi > _R2
        return Inf
    end
    Δ2 = muladd(deta, deta, dphi*dphi)
    if p < 0
        return @fastmath Δ2/max(_kt2[i]^ap, _kt2[j]^ap)
    end
    @fastmath Δ2*min(_kt2[i]^ap, _kt2[j]^ap)
end
```

Better Method

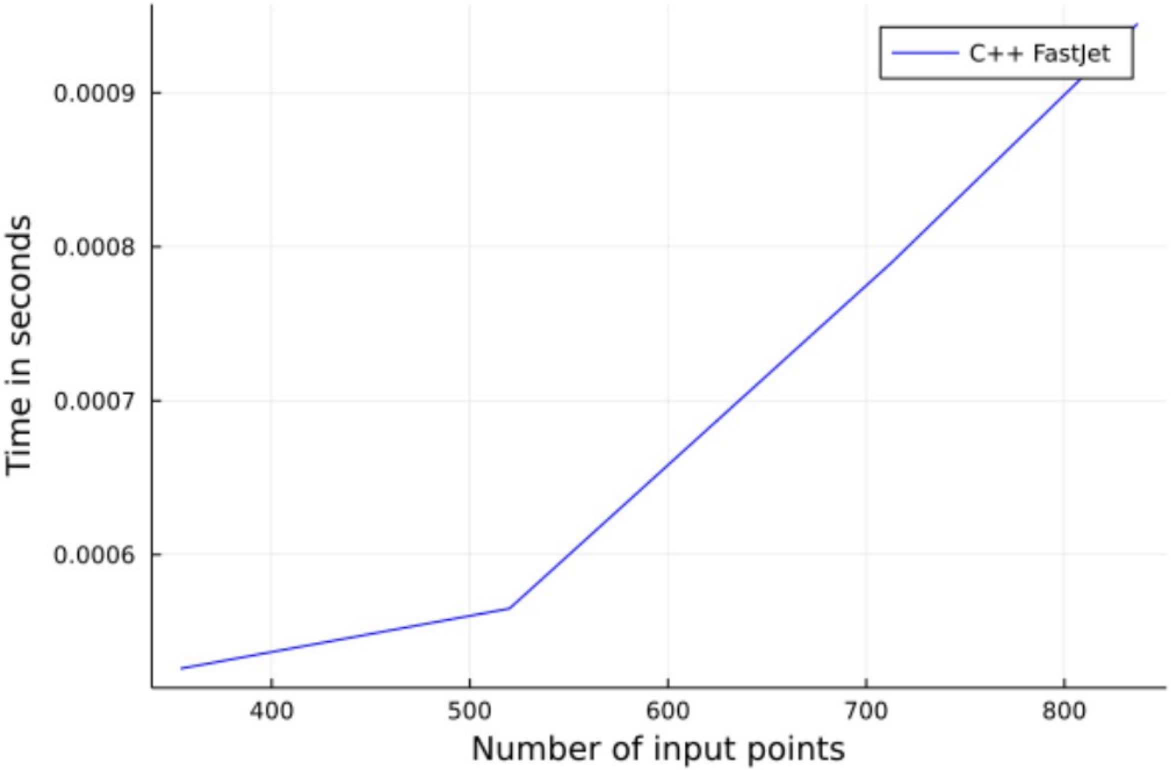


Better Method



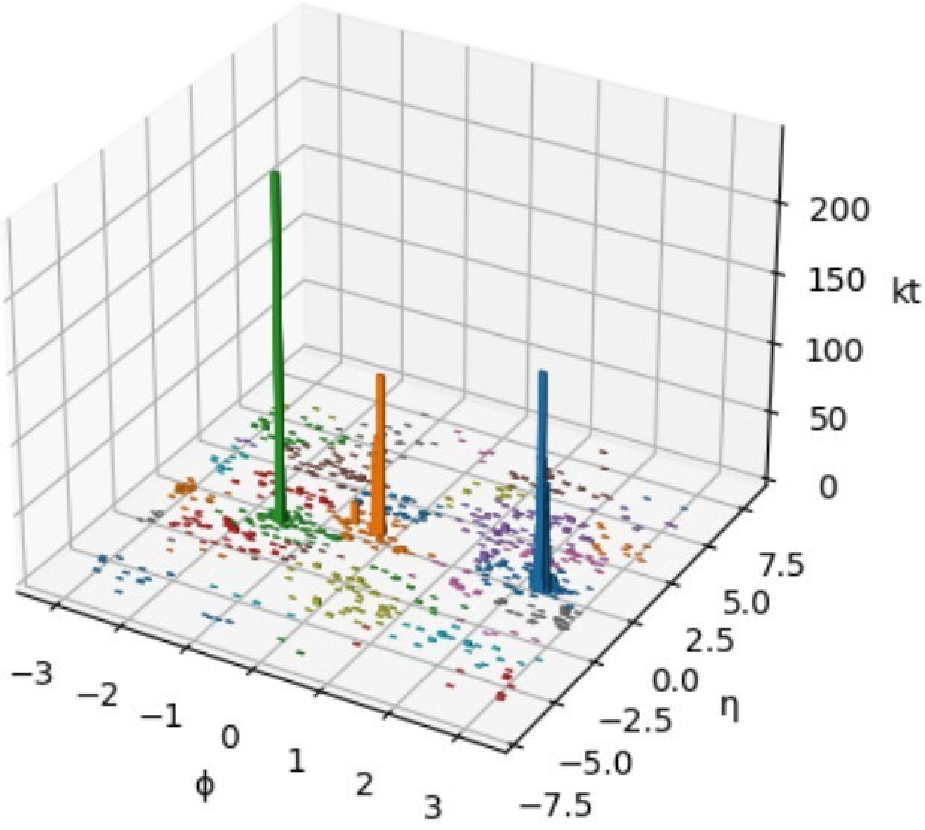
Next Steps

- Implement an exact copy of FastJet’s optimised version of the algorithm and test if this sophisticated approach is better
- Provide interfaces to other Julia HEP packages



Additional Features

- Possibility for other sequential recombination algorithms in Julia
- Plotting the results
- Comparison with FastJet (the C++ golden standard)
- Support for user-defined data types and extensions & GPU (because of multiple dispatch)



What Have I Learned?

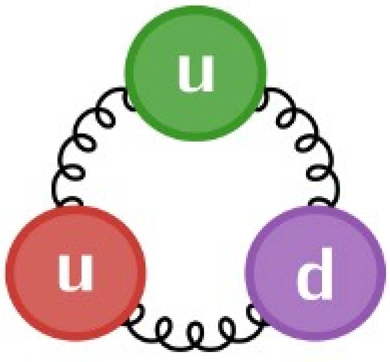
- A lot about high energy physics (tasks, software, algorithms)
- How to work with HEP data
- Automate tests for Julia repositories
- Use Julia’s profiling tools

Summary

- Julia can be fast if you know what to do
- Julia can be a good alternative to Python
- Still a lot to explore in the usage of Julia in HEP yet
- Conclusion is yet to be made

JetReconstruction.jl

github.com/gojakuch/JetReconstruction.jl



github.com/JuliaHEP