A new SymPy backend for vector: uniting experimental and theoretical physicists

Saransh Chopra[1] and *Jim* Pivarski[1,*]

[1]University College London
[2]Princeton University

Abstract. Vector is a Python library for 2D, 3D, and Lorentz vectors, especially arrays of vectors, to solve common physics problems in a NumPy-like [1] way. Vector can currently perform numerical computations, and through this paper, we introduce a new symbolic backend that extends vector's utility to theoretical physicists. The numerical backends of vector enable users to create pure Python Object, NumPy arrays, and Awkward arrays [2] of vectors. The Object and Awkward backends are also implemented in Numba [3] to leverage JIT-compiled vector calculations, and support for JAX [4] and Dask [5] is available within the Awkward backend. The new symbolic backend, built on top of SymPy [6] expressions, showcases vector's ability to support far-flung cases and allows SymPy methods and functions to work on vector classes. Moreover, apart from a few software, high energy physics has maintained a strict separation between tools used by theorists and experimentalists, and vector's SymPy backend aims to bridge this gap, providing a unified computational framework for both communities.

1 Introduction

The Scikit-HEP [7] ecosystem was primarily designed to be used by experimental physicists to manipulate and perform physics analysis on numerical data. Theoretical physicists have largely been aloof from Scikit-HEP and other experimental physics frameworks, as there exists a divide between the software used by them and experimental physicists. Moreover, only MC generators like Pythia [8] are routinely used by both groups, and there have been little to no efforts to unify the software used by theorists and experimentalists.

Vector was designed from the ground up to have multiple computational backends. The core compute functions of the library are written to be generic and library-agnostic, such that they operate only on the data containers and not the data itself. This design pattern has already led to the development of vector's three prominent backends -

- the Object backed for vectors that pure Python objects,
- the NumPy backend for NumPy arrays of vectors, and
- the Awkward backend for Awkward arrays of vectors.

[*]e-mail: pivarski@princeton.edu

All three backends share the same duck-typed compute functions, allowing vector to increase the number of backends without adding explicit compute functions for each of them. Therefore, this paper extends vector to operate on symbolic expressions to showcase the diversity of backends it can scale to and to enable both experimental and theoretical physicists to utilize the library in their work. Since the SymPy vector classes and their momentum equivalents operate on SymPy expressions, all of the standard SymPy methods and functions work on the vectors, vector coordinates, and the results of operations carried out on vectors. Moreover, vector's SymPy backend aims to create a stronger connection between software used by experimentalists and software used by theorists.

2 Implementation

Vector's design and duck typing of compute functions (allowing them to be shared amongst the backends) paved the way for the development of the SymPy backend. The SymPy backend's implementation consists of coordinate and vector classes that are capable of constructing vectors using SymPy's data containers and wrapping compute function results as SymPy expressions.

Figure 1 shows how the SymPy backend and other vector backends interact with the compute functions. Each backend has its own coordinate and vector classes that can accept numerical (for the case of Object, NumPy, and Awkward backend) or symbolic (for the case of SymPy backend) arguments. The classes, as well as their methods, are compatible with the respective backend libraries. The Object and Awkward backends use NumPy to perform all the arithmetic, and Awkward Array internally maps the NumPy calls to custom functions for ragged arrays.

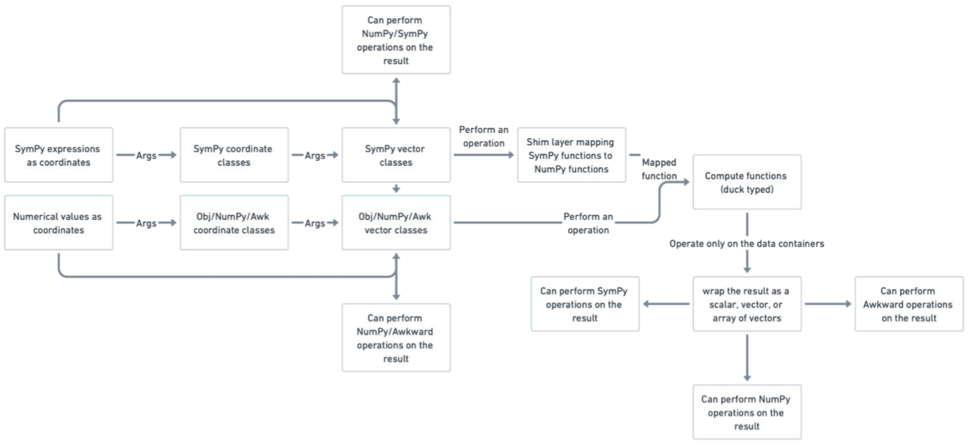
Can perform NumPy/SymPy operations on the result
SymPy expressions as coordinates
Args
SymPy coordinate classes
Args
SymPy vector classes
Perform an operation
Shim layer mapping SymPy functions to NumPy functions
Mapped function
Compute functions (duck typed)
Numerical values as coordinates
Args
Obj/NumPy/Awk coordinate classes
Args
Obj/NumPy/Awk vector classes
Perform an operation
Operate only on the data containers
Can perform SymPy operations on the result
wrap the result as a scalar, vector, or array of vectors
Can perform Awkward operations on the result
Can perform NumPy/Awkward operations on the result
Can perform NumPy operations on the result

Figure 1. Implementation of Vector's SymPy backend.

Finally, all the compute functions valid on the dimension of the constructed vector work with all the backends. At the moment, the compute functions switch between backend libraries using a shim layer. This shim layer is not required for the Object, NumPy, and Awkward backends because NumPy works with all of them. On the other hand, due to different naming conventions between SymPy and NumPy, the NumPy functions are mapped to the respective SymPy functions in the shim layer and are flown down to the compute functions.

The results from these compute functions are wrapped with values or appropriate data structures in the vector classes. This wrapped result can then be used in any of the functions provided by the backend libraries, making a strong cohesion between vector backends and the backend libraries.

3 Demonstration

Consider the example 1 performing `deltaR` operation on two Object type 4D Momentum vectors.

```
import vector

muon_1_obj = vector.MomentumObject4D(px=1, py=2, pz=3, E=10)
muon_2_obj = vector.MomentumObject4D(px=2, py=3, pz=4, E=11)

muon_1_obj.deltaR(muon_2_obj)
# 0.19249147660266414
```

Listing 1. Performing deltaR on Object vectors.

The exact same operation can be carried out using the SymPy backend with an almost identical syntax. Example 2 shows how SymPy `symbols` can be passed into `MomentumSymPy` as arguments just like numerical values are passed into `MomentumObject`. The `deltaR` operation on the SymPy vector returns a SymPy expression instead of a numerical value. The obtained SymPy expression is compatible with every SymPy method and function; hence, one can substitute (`subs`) and evaluate (`evalf`) the resultant expression to validate the theoretical expression.

```
import vector; import sympy

px_1, py_1, pz_1, E_1 = sympy.symbols(
    "px_1 py_1 pz_1 E_1", real=True
)
px_2, py_2, pz_2, E_2 = sympy.symbols(
    "px_2 py_2 pz_2 E_2", real=True
)

muon_1_sympy = vector.MomentumSympy4D(
    px=px_1, py=py_1, pz=pz_1, E=E_1
)
muon_2_sympy = vector.MomentumSympy4D(
    px=px_2, py=py_2, pz=pz_2, E=E_2
)

muon_1_sympy.deltaR(muon_2_sympy)
# sqrt((Mod(atan2(py_1, px_1) - atan2(py_2, px_2) + pi, 2*pi)
# - pi)**2 + (asinh(pz_1/sqrt(px_1**2 + py_1**2))
# - asinh(pz_2/sqrt(px_2**2 + py_2**2)))**2)

# take the values from object type vectors
muon_1_sympy.deltaR(muon_2_sympy).subs(
    {
        px_1: muon_1_obj.px,
        py_1: muon_1_obj.py,
        pz_1: muon_1_obj.pz,
        E_1: muon_1_obj.E,
        px_2: muon_2_obj.px,
        py_2: muon_2_obj.py,
        pz_2: muon_2_obj.pz,
        E_2: muon_2_obj.E,
    }
).evalf()
# 0.192491476602664
```

Listing 2. Performing deltaR on SymPy vectors.

Furthermore, 3 depicts how some of the other SymPy functionalities - using `simplify` to simplify expressions and converting expressions into code for a particular programming language - can be called on the symbolic results.

```
import vector; import sympy

x, y, z, t = sympy.symbols(
    "x y z t", real=True
)

v = vector.VectorSympy4D(x=x, y=y, z=z, t=t)

v.boost(v.to_beta3()).t
# t/sqrt(1 - x**2/t**2 - y**2/t**2 - z**2/t**2) + x**2/(t*
# sqrt(1 - x**2/t**2 - y**2/t**2 - z**2/t**2)) + y**2/(t*
# sqrt(1 - x**2/t**2 - y**2/t**2 - z**2/t**2)) + z**2/(t*
# sqrt(1 - x**2/t**2 - y**2/t**2 - z**2/t**2))

v.boost(v.to_beta3()).t.simplify()
# t*sqrt((t**2 - x**2 - y**2 - z**2)/t**2)*(t**2 + x**2 +
# y**2 + z**2)/(t**2 - x**2 - y**2 - z**2)

import sympy.printing.c

print(sympy.printing.c.ccode(boosted.t.simplify()))
# t*sqrt((pow(t, 2) - pow(x, 2) - pow(y, 2) - pow(z, 2))/
# pow(t, 2))*(pow(t, 2) + pow(x, 2) + pow(y, 2) + pow(z, 2))
#/(pow(t, 2) - pow(x, 2) - pow(y, 2) - pow(z, 2))
```

Listing 3. Simplifying expressions and converting theem into code for another programming language.

4 Challenges

The symbolic backend has two minor but intentional caveats. SymPy internally uses mpmath [9] to perform complex floating-point arithmetic, which has led to minor disagreements between the results obtained through the numerical (Python, NumPy, and Awkward) and the symbolic backend. This disagreement can be minimized by specifying more decimal points in the precision. Further, operations on SymPy vectors are only 100% compatible with numeric vectors if the vectors are positive time-like, that is, if -

$$t^2 > x^2 + y^2 + z^2 \tag{1}$$

The space-like and negative time-like cases have different sign conventions, which would have led to `if-else` statements in the code, making it impossible for SymPy to simplify symbolic expressions. Hence, to make SymPy's simplification work, space-like and negative time-like sign conventions are ignored in the shim layer. Given that most high energy physics analyses deal with positive time-like vectors, this caveat does not hinder the ability to use the vector's SymPy backend in theoretical calculations.

5 Future work

The SymPy backend offers users several new features, some of which are relatively under-explored. Symbolic differentiation and code generation for other languages are two such important but unexplored features.

On one hand, there has been a recent push to make high energy physics analysis pipelines differentiable for parameter tuning, such as the ongoing work on integrating Automatic Differentiation in the Analysis Grand Challenge. Vector already extends the JAX backend of Awkward Array, and with the development of SymPy backend, vector can be expected to perform both automatic and symbolic differentiation. On the other hand, code generation for another language through the SymPy backend can potentially help physicists flexibly switch languages and frameworks for their analyses. Both of these functionalities can be assessed and experimented with in the future to fill gaps in the software used for high energy physics.

6 Acknowledgements

The work on vector's symbolic backend was supported by NSF cooperative agreements OAC-1836650 and PHY-2323298 (IRIS-HEP).

References

[1] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Array programming with NumPy, Nature **585**, 357 (2020). 10.1038/s41586-020-2649-2

[2] J. Pivarski, I. Osborne, I. Ifrim, H. Schreiner, A. Hollands, A. Biswas, P. Das, S. Roy Choudhury, N. Smith, M. Goyal, Awkward Array (2018)

[3] S.K. Lam, A. Pitrou, S. Seibert, Numba: A llvm-based python jit compiler, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (2015), pp. 1–6

[4] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne et al., JAX: composable transformations of Python+NumPy programs (2018), `http://github.com/jax-ml/jax`

[5] M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, in *Proceedings of the 14th python in science conference* (Citeseer, 2015), 130-136

[6] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S.B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh et al., Sympy: symbolic computing in python, PeerJ Computer Science **3**, e103 (2017). 10.7717/peerj-cs.103

[7] E. Rodrigues, B. Krikler, C. Burr, D. Smirnov, H. Dembinski, H. Schreiner, J. Nandi, J. Pivarski, M. Feickert, M. Marinangeli et al., The scikit hep project overview and prospects, EPJ Web Conf. **245**, 06028 (2020). 10.1051/epjconf/202024506028

[8] T. Sjöstrand, S. Ask, J.R. Christiansen, R. Corke, N. Desai, P. Ilten, S. Mrenna, S. Prestel, C.O. Rasmussen, P.Z. Skands, An introduction to pythia 8.2, Computer Physics Communications **191**, 159–177 (2015). 10.1016/j.cpc.2015.01.024

[9] The mpmath development team, mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0) (2023), `http://mpmath.org/`