Awkward Array's JAX backend and complex-step autodiff as an alternative

Jim Pivarski

Princeton University – IRIS-HEP

December 17, 2024

Awkward Array has 4 backends

- `"cpu"`: buffers (user data, indexes, offsets, etc.) are NumPy arrays, computations are performed by NumPy functions or cpu-kernels.so

Awkward Array has 4 backends

- `"cpu"`: buffers (user data, indexes, offsets, etc.) are NumPy arrays, computations are performed by NumPy functions or cpu-kernels.so
- `"cuda"`: buffers are CuPy arrays, CUDA kernels are JIT-compiled by CuPy

Awkward Array has 4 backends

- `"cpu"`: buffers (user data, indexes, offsets, etc.) are NumPy arrays, computations are performed by NumPy functions or cpu-kernels.so
- `"cuda"`: buffers are CuPy arrays, CUDA kernels are JIT-compiled by CuPy
- `"typetracer"`: buffers are dataless Python objects that only infer types (intended for Dask; currently only used by Dask)

Awkward Array has 4 backends

- `"cpu"`: buffers (user data, indexes, offsets, etc.) are NumPy arrays, computations are performed by NumPy functions or cpu-kernels.so

- `"cuda"`: buffers are CuPy arrays, CUDA kernels are JIT-compiled by CuPy

- `"typetracer"`: buffers are dataless Python objects that only infer types (intended for Dask; currently only used by Dask)

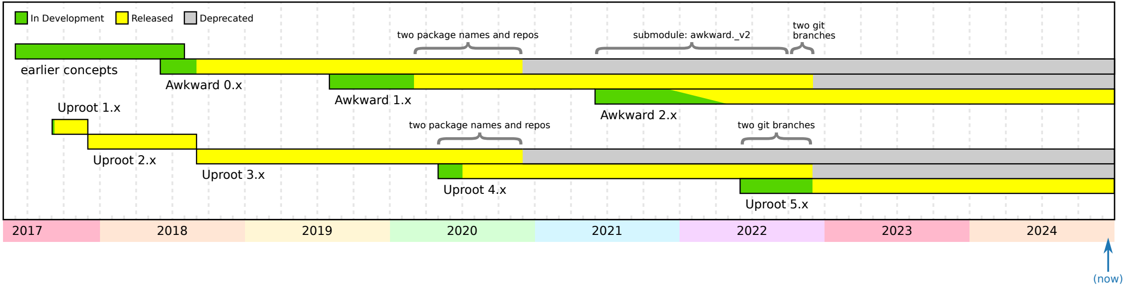
- `"jax"`: data buffers are JAX arrays, indexes are NumPy, `ak.Array` is flattened/unflattened as PyTrees

Awkward Array has 4 backends

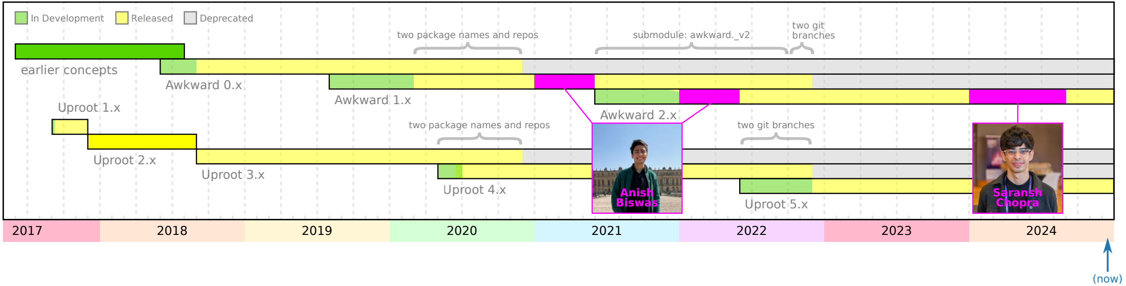
- `"cpu"`: buffers (user data, indexes, offsets, etc.) are NumPy arrays, computations are performed by NumPy functions or cpu-kernels.so
- `"cuda"`: buffers are CuPy arrays, CUDA kernels are JIT-compiled by CuPy
- `"typetracer"`: buffers are dataless Python objects that only infer types (intended for Dask; currently only used by Dask)
- `"jax"`: data buffers are JAX arrays, indexes are NumPy, `ak.Array` is flattened/unflattened as PyTrees

When a backend is untested, it gets out of date. `"jax"` has 44 tests. (`"cuda"` has 459 tests and `"cpu"` + `"typetracer"` has 2214 tests and are used daily.)

JAX backend development

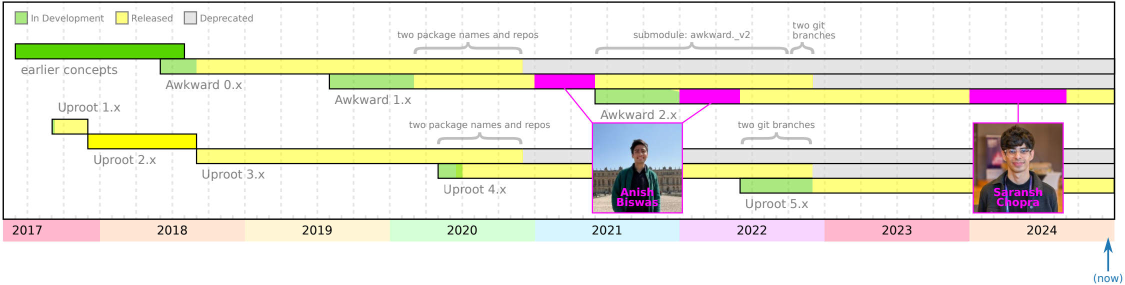


JAX backend development



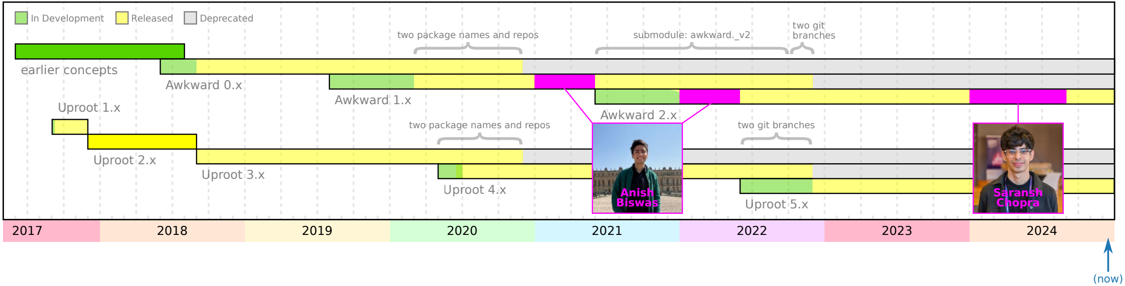
- Jan 2021–Apr 2021: Anish Biswas as IRIS-HEP Fellow
  - wasn't possible at all, motivated Awkward v2 C++ $\rightarrow$ Python reimplementation

JAX backend development



- Jan 2021–Apr 2021: Anish Biswas as IRIS-HEP Fellow
  - wasn't possible at all, motivated Awkward v2 C++ $\rightarrow$ Python reimplementation
- Jan 2022–Jun 2022: Anish Biswas as CERN Contractor
  - implemented map-like functions in PyTrees, reduce-like functions with LAX

JAX backend development



- Jan 2021–Apr 2021: Anish Biswas as IRIS-HEP Fellow
  - wasn't possible at all, motivated Awkward v2 C++ → Python reimplementation
- Jan 2022–Jun 2022: Anish Biswas as CERN Contractor
  - implemented map-like functions in PyTrees, reduce-like functions with LAX
- Jan 2024–Sep 2024: Saransh Chopra as IRIS-HEP Gap Year Fellow
  - "on call" for feedback from autograd users; received very little

Limitations on the JAX backend

- JAX's JIT-compilation is a non-starter: XLA requires array sizes to not be dependent on data values at compile-time, and that rule is broken throughout Awkward's codebase.

Limitations on the JAX backend

- JAX's JIT-compilation is a non-starter: XLA requires array sizes to not be dependent on data values at compile-time, and that rule is broken throughout Awkward's codebase.
- JAX's PyTree extension mechanism exclusively works on map-like operations: $f(\vec{x}) = f(x_i)$ for all components x_i of array $\vec{x}$.

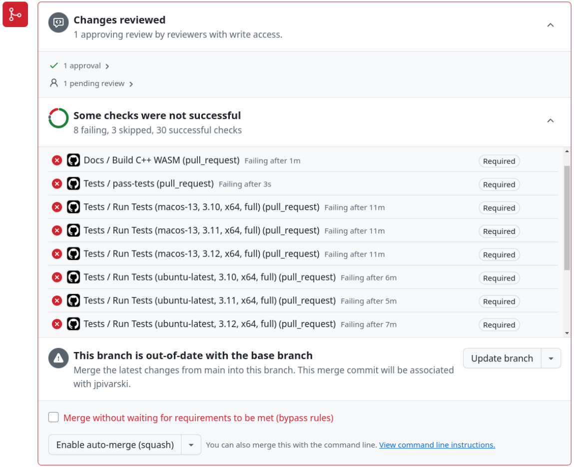
Limitations on the JAX backend

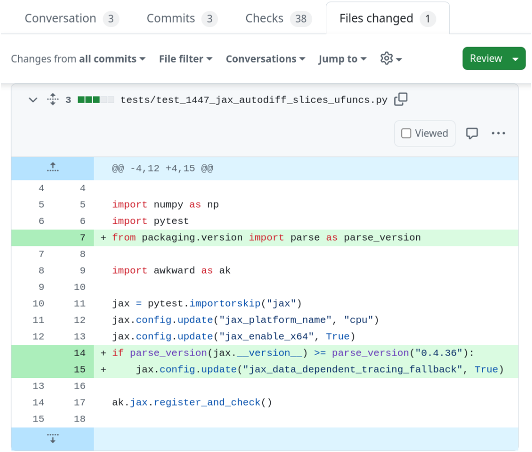
- JAX's JIT-compilation is a non-starter: XLA requires array sizes to not be dependent on data values at compile-time, and that rule is broken throughout Awkward's codebase.
- JAX's PyTree extension mechanism exclusively works on map-like operations: $f(\vec{x}) = f(x_i)$ for all components x_i of array $\vec{x}$.
- We were lucky enough that JAX's LAX library has segmented reductions.

Limitations on the JAX backend

- JAX's JIT-compilation is a non-starter: XLA requires array sizes to not be dependent on data values at compile-time, and that rule is broken throughout Awkward's codebase.
- JAX's PyTree extension mechanism exclusively works on map-like operations: $f(\vec{x}) = f(x_i)$ for all components x_i of array $\vec{x}$.
- We were lucky enough that JAX's LAX library has segmented reductions.
- I'm not completely sure all of the above works: Anish was fighting corner cases to the end and we don't have much user feedback.

Occasionally, the JAX tests fail and have to be patched





Can we get autodiff without JAX?

- HIPS/autograd is a simpler library; Awkward v1's PR #120 wraps `elementwise_grad` (forward autodiff of mappers as a dectorator).

Can we get autodiff without JAX?

- HIPS/autograd is a simpler library; Awkward v1's PR #120 wraps `elementwise_grad` (forward autodiff of mappers as a dectorator).
- If forward autodiff is acceptable, a backend based on an eager primal + tangent array would let us control program flow and get 100% coverage.

Can we get autodiff without JAX?

- HIPS/autograd is a simpler library; Awkward v1's PR #120 wraps `elementwise_grad` (forward autodiff of mappers as a dectorator).
- If forward autodiff is acceptable, a backend based on an eager primal + tangent array would let us control program flow and get 100% coverage.
- If backpropagation is necessary, we could at least get dask-awkward's coverage by using our own typetracer or Dask DAGs.

Can we get autodiff without JAX?

- HIPS/autograd is a simpler library; Awkward v1's PR #120 wraps `elementwise_grad` (forward autodiff of mappers as a dectorator).
- If forward autodiff is acceptable, a backend based on an eager primal + tangent array would let us control program flow and get 100% coverage.
- If backpropagation is necessary, we could at least get dask-awkward's coverage by using our own typetracer or Dask DAGs.
- How hard is it to implement autodiff, anyway?

Theory of autodiff: complex numbers

https://www.hedonisticlearning.com/posts/complex-step-differentiation.html

https://researchrepository.wvu.edu/faculty_publications/426

Calculating a derivative of $f : \mathbb{R} \to \mathbb{R}$ to $\mathcal{O}(h^2)$ by finite differences:

$$f'(x) \approx \frac{1}{2h}\Big(f(x+h) - f(x-h)\Big)$$

but too-small h has large cancellations and too-large h steps over bumps in f.

Theory of autodiff: complex numbers

https://www.hedonisticlearning.com/posts/complex-step-differentiation.html

https://researchrepository.wvu.edu/faculty_publications/426

Calculating a derivative of $f : \mathbb{R} \to \mathbb{R}$ to $\mathcal{O}(h^2)$ by finite differences:

$$f'(x) \approx \frac{1}{2h}\Big(f(x+h) - f(x-h)\Big)$$

but too-small h has large cancellations and too-large h steps over bumps in f.

Instead, take a step $h = i\varepsilon$ on a function $F(z) = f(z)$ for $z \in \mathbb{R}$ with $F(\overline{z}) = \overline{F(z)}$:

$$F'(x) \approx \frac{1}{2i\varepsilon}\Big(F(x+i\varepsilon) - F(x-i\varepsilon)\Big) = \frac{1}{\varepsilon}\mathrm{Im}\Big(F(x+i\varepsilon)\Big)$$

Theory of autodiff: complex numbers

https://www.hedonisticlearning.com/posts/complex-step-differentiation.html

https://researchrepository.wvu.edu/faculty_publications/426

Calculating a derivative of $f : \mathbb{R} \to \mathbb{R}$ to $\mathcal{O}(h^2)$ by finite differences:

$$f'(x) \approx \frac{1}{2h}\Big(f(x+h) - f(x-h)\Big)$$

but too-small h has large cancellations and too-large h steps over bumps in f.

Instead, take a step $h = i\varepsilon$ on a function $F(z) = f(z)$ for $z \in \mathbb{R}$ with $F(\bar{z}) = \overline{F(z)}$:

$$F'(x) \approx \frac{1}{2i\varepsilon}\Big(F(x+i\varepsilon) - F(x-i\varepsilon)\Big) = \frac{1}{\varepsilon}\mathrm{Im}\Big(F(x+i\varepsilon)\Big)$$

No more additive cancellations and the step is perpendicular to bumps in f.

Theory of autodiff: dual numbers

The derivative of f from its complex extension F,

$$f'(x) \approx \mathrm{Im}\Big(F(x + i\varepsilon) \Big) / \varepsilon$$

is exact if $\varepsilon > 0$ and $\varepsilon^2 = 0$. The complex numbers don't have this property, but imagine an abstract algebra in which this is true.

Theory of autodiff: dual numbers

The derivative of f from its complex extension F,

$$f'(x) \approx \operatorname{Im}\Big(F(x + i\varepsilon) \Big) / \varepsilon$$

is exact if $\varepsilon > 0$ and $\varepsilon^2 = 0$. The complex numbers don't have this property, but imagine an abstract algebra in which this is true.

This algebra is called "dual numbers," and it's the space-time extension used in SUSY. The Taylor expansion of a quantum field in superspace has exactly 2 terms, identified as a fermion field and a boson field.

Theory of autodiff: dual numbers

The derivative of f from its complex extension F,

$$f'(x) \approx \mathrm{Im}\Big(F(x + i\varepsilon)\Big)/\varepsilon$$

is exact if $\varepsilon > 0$ and $\varepsilon^2 = 0$. The complex numbers don't have this property, but imagine an abstract algebra in which this is true.

This algebra is called "dual numbers," and it's the space-time extension used in SUSY. The Taylor expansion of a quantum field in superspace has exactly 2 terms, identified as a fermion field and a boson field.

For autodiff, the two components are the primal array and tangent array. Implementing autodiff is as hard as implementing complex extensions for every real function.

Theory of autodiff: dual numbers

The derivative of f from its complex extension F,

$$f'(x) \approx \mathrm{Im}\Big(F(x + i\varepsilon)\Big)/\varepsilon$$

is exact if $\varepsilon > 0$ and $\varepsilon^2 = 0$. The complex numbers don't have this property, but imagine an abstract algebra in which this is true.

This algebra is called "dual numbers," and it's the space-time extension used in SUSY. The Taylor expansion of a quantum field in superspace has exactly 2 terms, identified as a fermion field and a boson field.

For autodiff, the two components are the primal array and tangent array. Implementing autodiff is as hard as implementing complex extensions for every real function.

But if we already have a complex implementation of all of our functions, we can set $\varepsilon = 10^{-8}$ to get 10^{-16} errors (typical errors in double-precision floating-point).

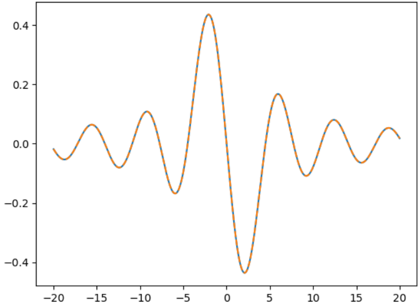
A complete autodiff library on one slide (NumPy & CuPy)

```python
import numpy as np
from numpy.lib.mixins import NDArrayOperatorsMixin

class diffarray(NDArrayOperatorsMixin):
    @classmethod
    def _build(cls, complex_array):
        self = cls.__new__(cls); self._array = complex_array
        return self

    def __init__(self, primal, tangent=None):
        if issubclass(primal.dtype.type, np.float32):
            self._array = primal.astype(np.complex64)
        elif issubclass(primal.dtype.type, np.float64):
            self._array = primal.astype(np.complex128)
        else:
            raise TypeError("array must be float32 or float64")
        if tangent is None:
            self._array += 1j * self._step_scale
        else:
            self._array += tangent * 1j * self._step_scale
    @property
    def _step_scale(self):
        return 1e-4 if issubclass(
            self._array.dtype.type, np.complex128) else 1e-8

    @property
    def primal(self):
        return np.real(self._array)
    @property
    def tangent(self):
        return np.imag(self._array) / self._step_scale
```

```python
def __array_ufunc__(self, ufunc, method, *args, **kwargs):
    return self.__array_function__(ufunc, None, args, kwargs)

def __array_function__(self, func, types, args, kwargs):
    # functions that would be misinterpreted on complex
    if func.__name__ == "abs":
        out = args[0].copy()
        out[out.real < 0] *= -1
        return type(self)._build(out)
    if func.__name__ == "real":
        return type(self)._build(args[0]._array)
    if func.__name__ == "imag":
        return type(self)._build(args[0]._array * 0)
    if func.__name__ in ("less", "less_equal", "equal",
                         "not_equal", "greater", "greater_equal"):
        args = [getattr(x, "_array", x).real for x in args]
        return func(*args, **kwargs)

    # all other functions
    args = [getattr(x, "_array", x) for x in args]
    kwargs = {
        k: getattr(v, "_array", v) for k, v in kwargs.items()
    }
    out = func(*args, **kwargs)
    return type(self)._build(out) if issubclass(
        out.dtype.type, np.complexfloating) else out

def __getitem__(self, where):
    out = self._array[where]
    return type(self)._build(np.asarray(out)) if isinstance(
        out, np.complexfloating) else out
```

A complete autodiff library on one slide (NumPy & CuPy)

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(-20, 20, 10000)
>>> da_x = diffarray(x)
>>> da_y = np.sin(da_x) / da_x
>>> abs(da_y.tangent - ((x*np.cos(x) - np.sin(x)) / x**2)).max()
3.9683650809863025e-10

>>> plt.plot(x, da_y.tangent)
>>> plt.plot(x, (x*np.cos(x) - np.sin(x)) / x**2, ls="--")
```



So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).

So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).

- Can we do second (or higher) order derivatives? No.
  (https://dl.acm.org/doi/10.1145/2168773.2168774)

So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).
- Can we do second (or higher) order derivatives? No.
  (https://dl.acm.org/doi/10.1145/2168773.2168774)
- Differentiate with respect to multiple arguments? Probably!

So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).
- Can we do second (or higher) order derivatives? No.
  (https://dl.acm.org/doi/10.1145/2168773.2168774)
- Differentiate with respect to multiple arguments? Probably!
- Backpropagation: we might need to use our typetracer and/or Dask DAGs.

So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).
- Can we do second (or higher) order derivatives? No.
  (https://dl.acm.org/doi/10.1145/2168773.2168774)
- Differentiate with respect to multiple arguments? Probably!
- Backpropagation: we might need to use our typetracer and/or Dask DAGs.
- Usable in Numba: in principle, and that's an interesting possibility, considering that JAX can't and applying e.g. Enzyme is complicated.

So, what's wrong with it?

- Bad stuff happens at points where the function isn't infinitely differentiable (same for other autodiff libraries: consider the derivative of ReLU).
- Can we do second (or higher) order derivatives? No.
  (https://dl.acm.org/doi/10.1145/2168773.2168774)
- Differentiate with respect to multiple arguments? Probably!
- Backpropagation: we might need to use our typetracer and/or Dask DAGs.
- Usable in Numba: in principle, and that's an interesting possibility, considering that JAX can't and applying e.g. Enzyme is complicated.

What else? Can anyone find a reason why this is not sufficient?

What are our options?

1. Keep the JAX backend in Awkward Array, tweaking as necessary.
2. Drop the JAX backend and...
    2.1 give up on autodiff.
    2.2 make a new autodiff mini-library that is Awkward-friendly that...
        2.2.1 just uses the complex-valued implementations we already have.
        2.2.2 implements a conventional autodiff.
    2.3 hide an autodiff implementation inside Awkward that...
        2.3.1 just uses the complex-valued implementations we already have.
        2.3.2 implements a conventional autodiff.