Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem

Aaron Meurer[‡†*], Athan Reines[‡†], Ralf Gommers[‡†], Yao-Lung L. Fang[§†], John Kirkham[§†], Matthew Barber[‡†], Stephan Hoyer[¶], Andreas Müller[‖], Sheng Zha[**], Saul Shanabrook, Stephannie Jiménez Gacha[‡], Mario Lezcano-Casado[‡], Thomas J. Fan[‡], Tyler Reddy[††], Alexandre Passos, Hyukjin Kwon[‡‡], Travis Oliphant[‡], Consortium for Python Data API Standards

Abstract—The Python array API standard specifies standardized application programming interfaces (APIs) and behaviors for array and tensor objects and operations as commonly found in libraries such as NumPy [1], CuPy [2], PyTorch [3], JAX [4], TensorFlow [5], Dask [6], and MXNet [7]. The establishment and subsequent adoption of the standard aims to reduce ecosystem fragmentation and facilitate array library interoperability in user code and among array-consuming libraries, such as scikit-learn [8] and SciPy [9]. A key benefit of array interoperability for downstream consumers of the standard is device agnosticism, whereby previously CPU-bound implementations can more readily leverage hardware acceleration via graphics processing units (GPUs), tensor processing units (TPUs), and other accelerator devices.

In this paper, we first introduce the Consortium for Python Data API Standards and define the scope of the array API standard. We then discuss the current status of standardization and associated tooling (including a test suite and compatibility layer). We conclude by outlining plans for future work.

Index Terms—Python, Arrays, Tensors, NumPy, CuPy, PyTorch, JAX, Tensorflow, Dask, MXNet

Introduction

Python users have a wealth of choices for libraries and frameworks for numerical computing [10][1][9][2][6][11][12][13], data science [14][15][16][17], machine learning [8], and deep learning [7][3][5][18]. New frameworks pushing forward the state of the art appear every year. One consequence of all this activity has been fragmentation in the fundamental building blocks—multidimensional arrays [19] (also known as tensors)—that underpin the scientific Python ecosystem (hereafter referred to as "the ecosystem").

This fragmentation comes with significant costs, from reinvention and re-implementation of arrays and associated application programming interfaces (APIs) to siloed technical stacks targeting only one array library to the proliferation of user guides providing guidance on how to convert between libraries. The APIs of each library are largely similar, but each have enough differences that end users have to relearn and rewrite code in order to work with multiple libraries. This process can be very painful as the transition is far from seamless and creates barriers for libraries wanting to support multiple array library backends.

The Consortium for Python Data API Standards (hereafter referred to as "the Consortium" and "we") aims to address this problem by standardizing a fundamental array data structure and an associated set of common APIs for working with arrays, thus facilitating interchange and interoperability.

Paper Overview

This paper is written as an introduction to the Consortium and the array API standard. The aim is to provide a high-level overview of the standard and its continued evolution and to solicit further engagement from the Python community.

After providing an overview of the Consortium, we first discuss standardization methodology. We then discuss the current status of the array API standard and highlight the main standardization areas. Next, we introduce tooling associated with the standard for testing compliance and shimming incompatible array library behavior. We conclude by outlining open questions and opportunities for further standardization. Links to the specification and all current repository artifacts, including associated tooling, can be found in the bibliography.

Consortium Overview

History

While the Python programming language was not explicitly designed for numerical computing, the language gained popularity in scientific and engineering communities soon after its release. The first array computing library for numerical and scientific computing in Python was Numeric, developed in the mid-1990s [20][1]. To better accommodate this library and its use cases, Python's syntax was extended to include indexing syntax [21].

In the early 2000s, Numarray introduced a more flexible data structure [22]. Numarray had faster operations for large arrays, but

† *These authors contributed equally.*
∗ *Corresponding author: asmeurer@quansight.com*
‡ *Quansight*
§ *NVIDIA Corporation*
¶ *Google*
‖ *Microsoft*
∗∗ *Amazon*
†† *LANL*
‡‡ *Databricks*

slower operations for small arrays. Subsequently, both Numeric and Numarray coexisted to satisfy different use cases.

In early 2005, the NumPy library unified Numeric and Numarray as a single array package by porting Numarray's features to Numeric [1]. This effort was largely successful and resolved the fragmentation at the time. For roughly a decade, NumPy was the only widely used array library. Building on NumPy, pandas was subsequently introduced in 2008 in order to address the need for a high performance, flexible tool for performing quantitative analysis on labeled tabular data [23].

Over the past 10 years, the rise of deep learning and the emergence of new hardware has led to a proliferation of new libraries and a corresponding fragmentation within the PyData array and dataframe ecosystem. These libraries often borrowed concepts from, or entirely copied, the APIs of older libraries, such as NumPy, and then modified and evolved those APIs to address new needs and use cases. Although the communities of each library individually discussed interchange and interoperability, no general coordination arose among libraries to avoid further fragmentation and to arrive at a common set of API standards until the founding of the Consortium.

The genesis for the Consortium grew out of many conversations among maintainers during 2019–2020. During those conversations, it quickly became clear that any attempt to create a new reference library to address fragmentation was infeasible. Unlike in 2005, too many different use cases and varying stakeholders now exist. Furthermore, the speed of innovation of both hardware and software is simply too great.

In May 2020, an initial group of maintainers and industry stakeholders[1] assembled to form the Consortium for Python Data API Standards and began drafting a specification for array APIs, which could then be adopted by existing array libraries and their dependents and by any new libraries which arise.

Objectives

Standardization efforts must maintain a balance between codifying what already exists and maintaining relevance with respect to future innovation. The latter aspect is particularly fraught, as relevance requires anticipating future needs, technological advances, and emerging use cases. Accordingly, if a standard is to remain relevant, the standardization process must be conservative in its scope, thorough in its consideration of current and prior art, and have clearly defined objectives against which success is measured.

To this end, we established four objectives for the array API standard. 1) Allow array-consuming libraries to accept and operate on arrays from multiple different array libraries. 2) Establish a common set of standardized APIs and behaviors, enabling more sharing and code reuse. 3) For new array libraries, offer a concrete API that can be adopted as-is. 4) Minimize the learning curve and friction for users as they switch between array libraries.

We explicitly omitted three notable possible objectives. 1) Make array libraries identical for the purpose of merging them. Different array libraries have different strengths (e.g., performance characteristics, hardware support, and tailored use cases, such as deep learning), and merging them into a single array library is neither practical nor realistic. 2) Implement a backend or runtime switching system in order to switch from one array library to another via a single setting or line of code. While potentially feasible, array consumers are likely to need to modify code in order to ensure optimal performance and behavior. 3) Support mixing multiple array libraries in a single function call. Mixing array libraries requires defining hierarchies and specifying rules for device synchronization and data localization. Such rules are likely to be specific to individual use cases.

Design Principles

In order to define the contours of the standardization process, we established the following design principles:

Functions. The standardized API should consist primarily of standalone functions. Function-based API design is the dominant pattern among array libraries, both in Python and in other frequently used programming languages supporting array computation, such as MATLAB [24] and Julia [25]. While method chaining and the fluent interface design pattern are also relatively common, especially among array libraries supporting deferred execution and operator fusion, function-based APIs are generally preferred. This mirrors design patterns used in underlying implementations, such as those written in C/C++ and Fortran, and more closely matches written mathematical notation.

Minimal array object. The standard should not require that an array object have any attributes or methods beyond what is necessary for inspection (e.g., shape, data type, and device) or for supporting operator overloading (i.e., magic methods).[2]

No dependencies. The standard and its implementations should not require any dependencies outside of Python itself.

Accelerator support. Standardized APIs and behaviors should be possible to implement for both central processing units (CPUs) and hardware-accelerated devices, such as graphics processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs).

Compiler support. Standardized APIs and behaviors should be amenable to just-in-time (JIT) and ahead-of-time (AOT) compilation and graph-based optimization techniques, such as those used by PyTorch [3], JAX [4], and TensorFlow [5]. APIs and behaviors not amenable to compilation, such as APIs returning arrays having data-dependent output shapes or polymorphic return types, should either be omitted or specified as optional.[3] In general, the shape, data type, and device of the return value from any function should be predictable from its input arguments.

Distributed support. Standardized APIs and behaviors should be amenable to implementation in array libraries supporting distributed computing (e.g., Dask [6]).

Consistency. Except in scenarios involving backward compatibility concerns, naming conventions and design patterns should be consistent across standardized APIs.

Extensibility. Conforming array libraries may implement functionality which is not included in the array API standard. Array consumers thus bear responsibility for ensuring that their API usage is portable across specification-conforming array libraries.

Deference. Where possible, the array API standard should defer to existing, widely-used standards. For example, the accu-

1. Direct stakeholders include the maintainers of Python array and dataframe libraries and organizations which sponsor library development. Indirect stakeholders include maintainers of libraries which consume array and dataframe objects ("consuming libraries"), developers of compilers and runtimes with array- and dataframe-specific functionality, and end users, such as data scientists and application developers.

2. Notably, array strides should be considered an implementation detail and should not be required as a public Python attribute.

3. Copy-view mutation semantics, such as those currently supported by NumPy, should be considered an implementation detail and, thus, not suitable for standardization.

racy and precision of numerical functions should not be specified beyond the guidance included in IEEE 754 [26].

Universality. Standardized APIs and behaviors should reflect common usage among a wide range of existing array libraries. Accordingly, with rare exception, only APIs and behaviors having existing implementations and broad support within the ecosystem may be considered candidates for standardization.

Methodology

A foundational step in technical standardization is articulating a subset of established practices and defining those practices in unambiguous terms. To this end, the standardization process must approach the problem from two directions: design and usage.

The former direction seeks to understand both current implementation design (APIs, names, signatures, classes, and objects) and semantics (calling conventions and behavior). The latter direction seeks to quantify API consumers (who are the downstream users of a given API?), usage frequency (how often is an API consumed?), and consumption patterns (which optional arguments are provided and in what context?). By analyzing both design and usage, we sought to ground the standardization process and specification decisions in empirical data and analysis.

Design

To understand API design of array libraries within the ecosystem, we first identified a representative sample of commonly used array libraries. This sample included NumPy, CuPy, PyTorch, JAX, TensorFlow, Dask, and MXNet. Next, we extracted public APIs for each library by analyzing module exports and scraping public web documentation. The following APIs for computing the arithmetic mean provide an example of extracted API data:

```
numpy.mean(a, axis=None, dtype=None, out=None,
    keepdims=<no value>)
cupy.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
torch.mean(input, dim, keepdim=False, out=None)
jax.numpy.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
tf.math.reduce_mean(input_tensor, axis=None,
    keepdims=False, name=None)
dask.array.mean(a, axis=None, dtype=None, out=None,
    keepdims=False, split_every=None)
mxnet.np.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
```

We determined commonalities and differences by analyzing the intersection, and its complement, of available APIs across each array library. From the intersection, we derived a subset of common APIs suitable for standardization based on prevalence and ease of implementation. The common API subset included function names, method names, attribute names, and positional and keyword arguments. As an example of a derived API, consider the common API for computing the arithmetic mean:

```
mean(a, axis=None, keepdims=False)
```

To assist in determining standardization prioritization, we leveraged usage data (discussed below) to confirm API need and to inform naming conventions, supported data types, and optional arguments. We have summarized findings and published tooling [27] for additional analysis and exploration, including Jupyter notebooks [17], as public artifacts available on GitHub.

Usage

To understand usage patterns of array libraries within the ecosystem, we first identified a representative sample of commonly used Python libraries ("downstream libraries") which consume the aforementioned array libraries. The sample of downstream libraries included SciPy [9], pandas [23], Matplotlib [14], xarray [12], scikit-learn [8], statsmodels [16], and scikit-image [11], among others. Next, we ran downstream library test suites with runtime instrumentation enabled. We recorded input arguments and return values for each API invocation by inspecting the bytecode stack at call time [28]. From the recorded data, we generated inferred signatures for each function based on provided arguments and associated types, noting which downstream library called which empirical API and at what frequency. We organized the API results in human-readable form as type definition files and compared the inferred API to the publicly documented APIs obtained during design analysis.

The following is an example of two inferred API signatures for `numpy.mean`, with the docstring indicating the number of lines of code which invoked the function for each downstream library when running library test suites. Based on the example, we can infer that invoking the function with an array input argument is a more common usage pattern among downstream libraries than invoking the function with a list of floats.

```
@overload
def mean(a: numpy.ndarray):
    """
    usage.dask: 21
    usage.matplotlib: 7
    usage.scipy: 26
    usage.skimage: 36
    usage.sklearn: 130
    usage.statsmodels: 45
    usage.xarray: 1
    """

@overload
def mean(a: List[float]):
    """
    usage.networkx: 6
    usage.sklearn: 3
    usage.statsmodels: 9
    """
```

As a final step, we ranked each API according to relative usage using the Dowdall positional voting system [29] (a variant of the Borda count [30] that favors candidate APIs having high relative usage). From the rankings, we assigned standardization priorities, with higher ranking APIs taking precedence over lower ranking APIs, and extended the analysis to aggregated API categories (e.g., array creation, manipulation, statistics, etc.). All source code, usage data, and analysis are publicly available on GitHub [28][27].

Array API Standard

The Python array API standard specifies standardized APIs and behaviors for array and tensor objects and operations. The scope of the standard includes defining, but is not limited to, the following: 1) a minimal array object; 2) semantics governing array interaction, including type promotion and broadcasting; 3) an interchange protocol for converting array objects originating from different array libraries; 4) a set of required array-aware functions; and 5) optional extensions for specialized APIs and array behaviors. We discuss each of these standardization areas in turn.

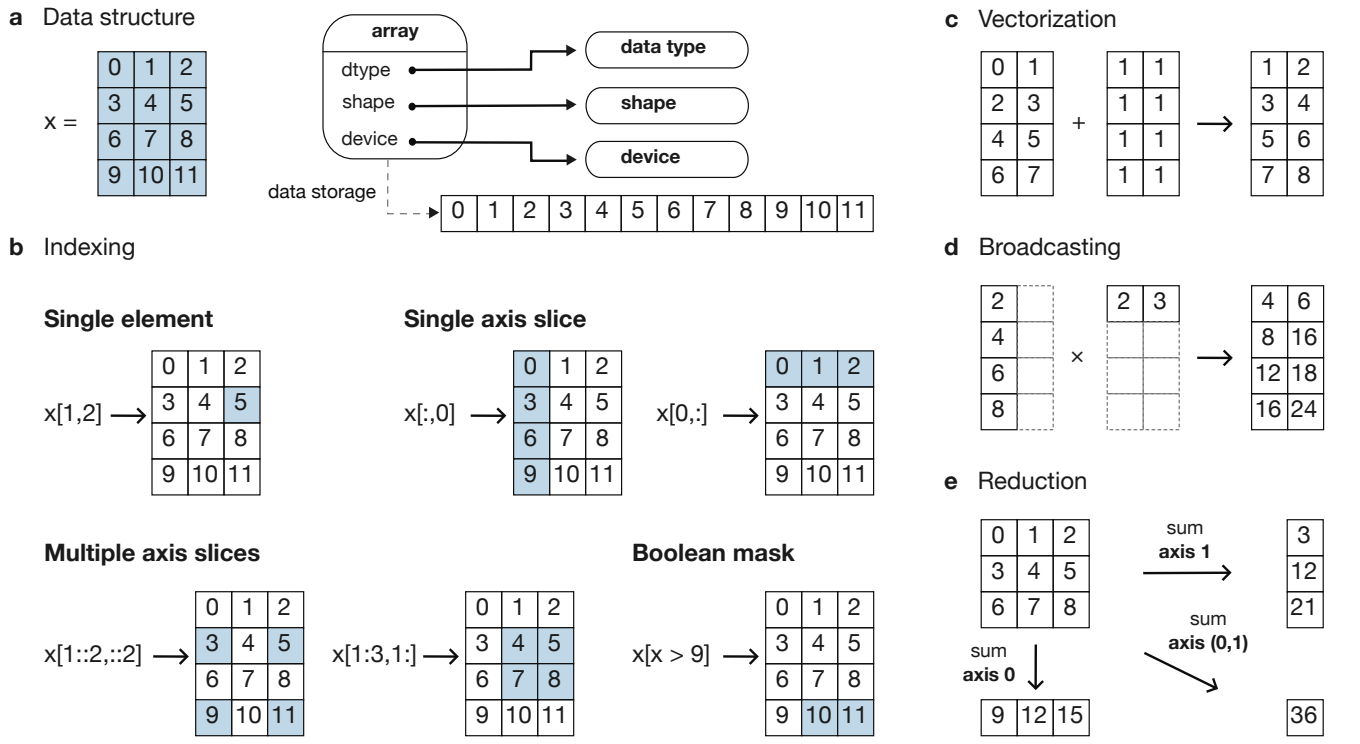

Fig. 1: *The array data structure and fundamental concepts.* ***a)*** *An array data structure and its associated metadata fields.* ***b)*** *Indexing an array. Indexing operations may access individual elements or sub-arrays. Applying a boolean mask is an optional indexing behavior and may not be supported by all conforming libraries.* ***c)*** *Vectorization obviates the need for explicit looping in user code by applying operations to multiple array elements.* ***d)*** *Broadcasting enables efficient computation by implicitly expanding the dimensions of array operands to equal sizes.* ***e)*** *Reduction operations act along one or more axes. In the example, summation along a single axis produces a one-dimensional array, while summation along two axes produces a zero-dimensional array containing the sum of all array elements.*

Array Object

An array object is a data structure for efficiently storing and accessing multidimensional arrays [19]. Within the context of the array API standard, the data structure is opaque—libraries may or may not grant direct access to raw memory—and includes metadata for interpreting the underlying data, notably "data type", "shape", and "device" (Fig. 1a).

An array data type ("dtype") describes how to interpret a single array element (e.g., integer, real- or complex-valued floating-point, boolean, or other). A conforming array object has a single dtype. To facilitate interoperability, conforming libraries must support and provide a minimal set of dtype objects (e.g., `int8`, `int16`, `int32`, `float32`, and `float64`). To ensure portability, data type objects must be provided by name in the array library namespace (e.g., `xp.bool`).

An array shape specifies the number of elements along each array axis (also referred to as "dimension"). The number of axes corresponds to the dimensionality (or "rank") of an array. For example, the shape `(3, 5)` corresponds to a two-dimensional array whose inner dimension contains five elements and whose outer dimension contains three elements. The shape `()` corresponds to a zero-dimensional array containing a single element.

An array device specifies the location of array memory allocation. A conforming array object is assigned to a single logical device. To support array libraries supporting execution on different device types (e.g., CPUs, GPUs, TPUs, etc.), conforming libraries must provide standardized device APIs in order to coordinate execution location. The following example demonstrates how an array-consuming library might use standardized device APIs to ensure execution occurs on the same device as the input.

```
def some_function(x):
    # Retrieve a standard-compliant namespace
    xp = x.__array_namespace__()

    # Allocate a new array on the same device as x
    y = xp.linspace(0, 2*xp.pi, 100, device=x.device)

    # Perform computation (on device)
    return xp.sin(y) * x
```

To interact with array objects, one uses "indexing" to access sub-arrays and individual elements, "operators" to perform logical and arithmetic operations (e.g., `+`, `-`, `*`, `/`, and `@`), and array-aware functions (e.g., for linear algebra, statistical reductions, and element-wise computation). Array indexing semantics extend built-in Python sequence `__getitem__()` indexing semantics to support element access across multiple dimensions (Fig. 1b).[4] Indexing an array using a boolean array (also known as "masking") is an optional standardized behavior.[5] The result of a mask operation is data-dependent and thus difficult to implement in array libraries relying on static analysis for graph-based optimization.

4. The array API standard includes support for in-place operations via `__setitem__()`; however, behavior is undefined if an in-place operation would affect arrays other than the target array (e.g., in array libraries supporting multiple "views" of the same underlying memory).

5. While not currently supported, integer array indexing may be included in a future revision of the array API standard.

Array Interaction

The Python array API standard further specifies rules governing expected behavior when an operation involves two or more array operands. For operations in which the data type of a resulting array object is resolved from operand data types, the resolved data type must follow type promotion semantics. Importantly, type promotion semantics are independent of array shape or contained values (including when an operand is a zero-dimensional array). For example, when adding one array having a `float32` data type to another array having a `float64` data type, the data type of the resulting array should be the promoted data type `float64`.

```
>>> x1 = xp.ones((2, 2), dtype=xp.float32)
>>> x2 = xp.ones((2, 2), dtype=xp.float64)
>>> y = x1 + x2
>>> y.dtype == xp.float64
True
```

In addition to type promotion, the array API standard specifies rules governing the automatic (and implicit) expansion of array dimensions to be of equal sizes (Fig. 1d). Standardized broadcasting semantics are the same as those popularized by NumPy [1].

Interchange Protocol

We expect that array library consumers will generally prefer to use a single array "type" (e.g., a NumPy `ndarray`, PyTorch `Tensor`, or Dask `array`) and will thus need a standardized mechanism for array object conversion. For example, suppose a data visualization library prefers to use NumPy internally but would like to extend API support to any conforming array object type. In such a scenario, the library would benefit from a reliable mechanism for accessing and reinterpreting the memory of externally provided array objects without triggering potential performance cliffs due to unnecessary copying of array data. To this end, the Python array API standard specifies an interchange protocol describing the memory layout of a strided, n-dimensional array in an implementation-independent manner.

The basis of this protocol is DLPack, an open in-memory structure for sharing tensors among frameworks [31]. DLPack is a standalone protocol with an ABI stable, header-only C implementation with cross hardware support. The array API standard builds on DLPack by specifying Python APIs for array object data interchange [32]. Conforming array objects must support `__dlpack__` and `__dlpack_device__` magic methods for accessing array data and querying the array device. A standardized `from_dlpack()` API calls these methods to construct a new array object of the desired type using zero-copy semantics when possible. The combination of DLPack and standardized Python APIs thus provides a stable, widely adopted, and efficient means for array object interchange.

Array Functions

To complement the minimal array object, the Python array API standard specifies a set of required array-aware functions for arithmetic, statistical, algebraic, and general computation. Where applicable, required functions must support vectorization (Fig. 1d), which obviates the need for explicit looping in user code by applying operations to multiple array elements. Vectorized abstractions confer two primary benefits: 1) implementation-dependent optimizations leading to increased performance and 2) concise expression of mathematical operations. For example, one can express element-wise computation of z-scores in a single line.

```
def z_score(x):
    return (x - xp.mean(x)) / xp.stdev(x)
```

In addition to vectorized operations, the array API standard includes, but is not limited to, functions for creating new arrays, with support for explicit device allocation, reshaping and manipulating existing arrays, performing statistical reductions across one, multiple, or all array axes (Fig. 1e), and sorting array elements. Altogether, these APIs provide a robust and portable foundation for higher-order array operations and general array computation.

Optional Extensions

While a set of commonly used array-aware functions is sufficient for many array computation use cases, additional, more specialized, functionality may be warranted. For example, while most data visualization libraries are unlikely to explicitly rely on APIs for computing Fourier transforms, signal analysis libraries supporting spectral analysis of time series are likely to require Fourier transform APIs. To accommodate specialized APIs, the Python array API standard includes standardized optional extensions.

An extension is a sub-namespace of a main namespace and is defined as a coherent set of standardized functionality which is commonly implemented across many, but not all, array libraries. Due to implementation difficulty (or impracticality), limited general applicability, a desire to avoid significantly expanding API surface area beyond what is essential, or some combination of the above, requiring conforming array libraries to implement and maintain extended functionality beyond their target domain is not desirable. Extensions provide a means for conforming array libraries to opt-in to supporting standardized API subsets according to need and target audience.

Specification Status

Following formation of the Consortium in 2020, we released an initial draft of the Python array API standard for community review in 2021. We have released two subsequent revisions:

v2021.12: The first full release of the specification, detailing purpose and scope, standardization methodology, future standard evolution, a minimal array object, an interchange protocol, required data types, type promotion and broadcasting semantics, an optional linear algebra extension, and array-aware functions for array creation, manipulation, statistical reduction, and vectorization, among others.

v2022.12: This revision includes errata for the v2021.12 release and adds support for single- and double-precision complex floating-type data types, additional array-aware APIs, an optional extension for computing fast Fourier transforms.

For future revisions, we expect annual release cadences; however, array API standard consumers should not assume a fixed release schedule.

Implementation Status

Reference Implementation

To supplement the Python array API standard, we developed a standalone reference implementation. The implementation is strictly compliant (i.e., any non-portable usage triggers an exception) and is available as the `numpy.array_api` submodule (discussed in [33]). In general, we do not expect for users to rely on the reference implementation for production use cases. Instead, the reference implementation is primarily useful for array-consuming libraries as a means for testing whether array library usage is guaranteed to be portable.

Ecosystem Adoption

Arrays are fundamental to scientific computing, data science, and machine learning. As a consequence, the Python array API standard has many stakeholders within the ecosystem. When establishing the Consortium, we thus sought participation from a diverse and representative cross-section of industry partners and maintainers of array and array-consuming libraries. To satisfy stakeholder needs, array library maintainers worked in close partnership with maintainers of array-consuming libraries throughout the array API standardization process to identify key use cases and achieve consensus on standardized APIs and behaviors.

Direct participation in the Consortium by array and array-consuming library maintainers has facilitated coordination across the ecosystem. In addition to the numpy.array_api reference implementation [34], several commonly used array libraries, including NumPy [35], CuPy [36], PyTorch [37], JAX [38], Dask [39], and MXNet [40], have either adopted or are in the process of adopting the array API standard. Increased array library adoption has increased array interoperability, which, in turn, has encouraged array-consuming libraries, such as SciPy [41] and scikit-learn [42] (discussed below), to begin adopting the standard by decoupling their implementations from specific array libraries. As array library adoption of the standard matures, we expect ecosystem adoption to accelerate.

Tooling

Test Suite

To facilitate adoption of the Python array API standard by libraries within the ecosystem, we developed a test suite to measure specification compliance [43]. The test suite covers all major aspects of the specification, such as broadcasting, type promotion, function signatures, special case handling, and expected return values.

Underpinning the test suite is Hypothesis, a Python library for creating unit tests [44]. Hypothesis uses property-based testing, a technique for generating arbitrary data satisfying provided specifications and asserting the truth of some "property" that should be true. Property-based testing is particularly convenient when authoring compliance tests, as the technique enables the direct translation of specification guidance into test code.

The test suite is the first example known to these authors of a Python test suite capable of running against multiple different libraries. As part of our work, we upstreamed strategies to Hypothesis for generating arbitrary arrays from any conforming array library, thus allowing downstream array consumers to test against multiple array libraries and their associated hardware devices.

Compatibility Layer

While we expect that maintainers of conforming array libraries will co-evolve library APIs and behaviors with those specified in the Python array API standard, we recognize that co-evolution is not likely to always proceed in unison due to varying release cycles and competing priorities. Varying timelines for adoption and full-compliance present obstacles for array-consuming libraries hoping to use the most recent standardized behavior, as such libraries are effectively blocked by the slowest array library release schedule.

To address this problem and facilitate adoption of the standard by array-consuming libraries, we developed a compatibility layer (named `array-api-compat`), which provides a thin wrapper around common array libraries [45]. The layer transparently intercepts API calls for any API which is not fully-compliant and polyfills non-compliant specification-defined behavior. For compliant APIs, it exposes the APIs directly, without interception, thus mitigating performance degradation risks due to redirection. To reduce barriers to adoption, the layer supports vendoring and has a small, pure Python codebase with no hard dependencies.

While the Python array API standard facilitates array interoperability in theory, the compatibility layer does so in practice, helping array-consuming libraries decouple adoption of the standard from the release cycles of upstream array libraries. Currently, the layer provides shims for NumPy, CuPy, and PyTorch and aims to support additional array libraries in the future. By ensuring specification-compliant behavior, we expect the compatibility layer to have a significant impact in accelerating adoption among array-consuming libraries.

Discussion

The principle aim of the Python array API standard is to facilitate interoperability of array libraries within the ecosystem. In achieving this aim, array-consuming libraries, such as those for statistical computing, data science, and machine learning, can decouple their implementations from specific array libraries. Decoupling subsequently allows end users to use the array library that is most applicable to their use case and to no longer be limited by the set of array libraries a particular array-consuming library supports.

In addition to improved developer ergonomics afforded by standardized APIs and increased interoperability, standardization allows end users and the authors of array-consuming libraries to use a declarative, rather than imperative, programming paradigm. This paradigm change has a key benefit in enabling users to opt into performance improvements based on their constraints and hardware capabilities. To assess the impact of this change, we worked with maintainers of scikit-learn and SciPy to measure the performance implications of specification adoption (Fig. 2).

scikit-learn

scikit-learn is a widely-used machine learning library. Its current implementation relies heavily on NumPy and SciPy and is a mixture of Python and Cython. Due to its dependence on NumPy for array computation, scikit-learn is CPU-bound, and the library is unable to capture the benefits of GPU- and TPU-based execution models. By adopting the Python array API standard, scikit-learn can decouple its implementation from NumPy and support non-CPU-based execution, potentially enabling increased performance.

To test this hypothesis, we examined the scikit-learn codebase to identify APIs which rely primarily on NumPy for their implementation. scikit-learn estimators are one such set of APIs, having methods for model fitting, classification prediction, and data projection, which are amenable to input arrays supporting alternative execution models. Having identified potential API candidates, we selected the estimator class for linear discriminant analysis (LDA) as a representative test case. Refactoring the LDA implementation was illustrative in several respects, as demonstrated in the following code snippet showing source code modifications[6]:

```
1   Xc = []
2   for idx, group in enumerate(self.classes_):
3 -     Xg = X[y == group, :]
4 -     Xc.append(Xg - self.means_[idx])
```

6. Source code modifications reflect those required for NumPy version 1.24.3 and Python array API standard version 2022.12.

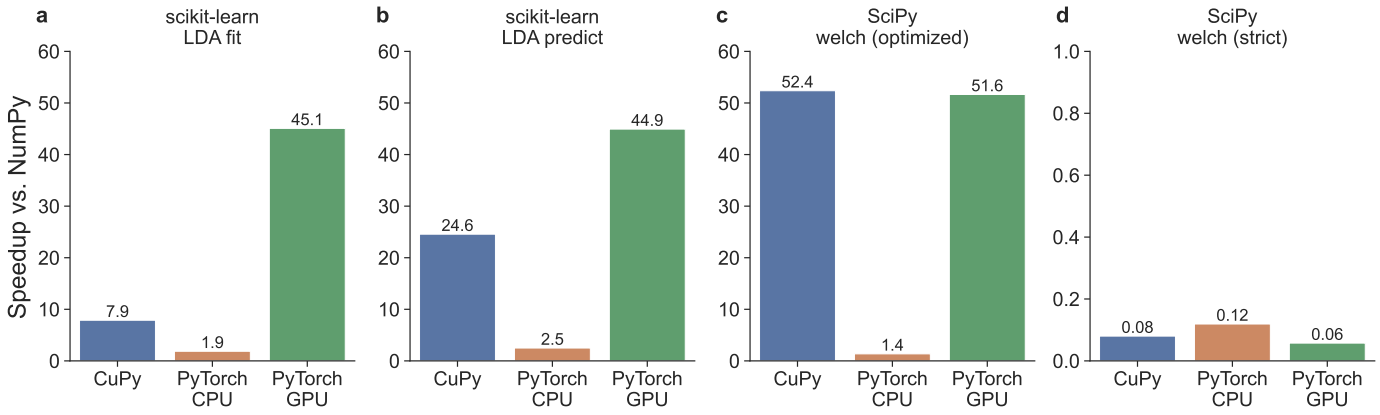

Fig. 2: *Benchmarks measuring performance implications of adoption in array-consuming libraries. Displayed timings are relative to NumPy. All benchmarks were run on Intel i9-9900K and NVIDIA RTX 2080 hardware.* ***a)*** *Fitting a linear discriminant analysis (LDA) model.* ***b)*** *Predicting class labels using LDA.* ***c)*** *Estimating power spectral density using Welch's method and library-specific optimizations.* ***d)*** *Same as* ***c****, but using a strictly portable implementation. Note that* ***d*** *has different vertical axis limits than* ***a-c****.*

```
5 +      Xg = X[y == group]
6 +      Xc.append(Xg - self.means_[idx, :])
7
8 - self.xbar_ = np.dot(self.priors_, self.means_)
9 + self.xbar_ = self.priors_ @ self.means_
10
11 - Xc = np.concatenate(Xc, axis=0)
12 + Xc = xp.concat(Xc, axis=0)
13
14 - std = Xc.std(axis=0)
15 + std = xp.std(Xc, axis=0)
16
17   std[std == 0] = 1.0
18 - fac = 1.0 / (n_samples - n_classes)
19 + fac = xp.asarray(1.0 / (n_samples - n_classes))
20
21 - X = np.sqrt(fac) * (Xc / std)
22 + X = xp.sqrt(fac) * (Xc / std)
23
24   U, S, Vt = svd(X, full_matrices=False)
25
26 - rank = np.sum(S > self.tol)
27 + rank = xp.sum(xp.astype(S > self.tol, xp.int32))
```

Indexing: *(lines 3-6)* NumPy supports non-standardized indexing semantics. To be compliant with the standard, 1) boolean masks must be the sole index and cannot be combined with other indexing expressions, and 2) the number of provided single-axis indexing expressions must equal the number of dimensions.

Non-standardized APIs: *(lines 8-9)* NumPy supports several APIs having no equivalent in the array API standard; `np.dot()` is one such API. For two-dimensional arrays, `np.dot()` is equivalent to matrix multiplication and was updated accordingly.

Naming conventions: *(lines 11-12)* NumPy contains several standard-compliant APIs whose naming conventions differ from those in the array API standard. In this and similar cases, adoption requires conforming to the standardized conventions.

Functions: *(lines 14-15)* NumPy supports several array object methods having no equivalent in the array API standard. To ensure portability, we refactored use of non-standardized methods in terms of standardized function-based APIs.

Scalars: *(lines 18-22)* NumPy often supports non-array input arguments, such as scalars, Python lists, and other objects, as "array-like" arguments in its array-aware APIs. While the array API standard does not prohibit such polymorphism, the standard does not require array-like support. In this case, we explicitly convert a scalar expression to a zero-dimensional array in order to ensure portability when calling `xp.sqrt()`.

Data types: *(lines 26-27)* NumPy often supports implicit type conversion of non-numeric data types in numerical APIs. The array API standard does not require such support, and, more generally, mixed-kind type promotion semantics (e.g., boolean to integer, integer to floating-point, etc.) are not specified. To ensure portability, we must explicitly convert a boolean array to an integer array before calling `xp.sum()`.

To test the performance implications of refactoring scikit-learn's LDA implementation, we generated a random two-class classification problem having 400,000 samples and 300 features.[7] We next devised two benchmarks, one for fitting an LDA model and the second for predicting class labels for each simulated sample. We ran the benchmarks and measured execution time for NumPy, PyTorch, and CuPy backends on Intel i9-9900K and NVIDIA RTX 2080 hardware. For PyTorch, we collected timings for both CPU and GPU execution models. To ensure timing reproducibility and reduce timing noise, we repeated each benchmark ten times and computed the average execution time.

Fig. 2a and Fig. 2b display results, showing average execution time relative to NumPy. When fitting an LDA model (Fig. 2a), we observe 1.9× higher throughput for PyTorch CPU, 7.9× for CuPy, and 45.1× for PyTorch GPU. When predicting class labels (Fig. 2b), we observe 2.5× higher throughput for PyTorch CPU, 24.6× for CuPy, and 44.9× for PyTorch GPU. In both benchmarks, using GPU execution models corresponded to significantly increased performance, thus supporting our hypothesis that scikit-learn can benefit from non-CPU-based execution models.

SciPy

SciPy is a collection of mathematical algorithms and convenience functions for numerical integration, optimization, interpolation, statistics, linear algebra, signal processing, and image processing, among others. Similar to scikit-learn, its current implementation relies heavily on NumPy. We thus sought to test whether SciPy could benefit from adopting the Python array API standard.

7. To ensure that observed performance is not an artifact of the generated dataset, we tested performance across multiple random datasets and did not observe a measurable difference across benchmark runs.

Following a similar approach to the scikit-learn benchmarks, we identified SciPy's signal processing APIs as being amenable to input arrays supporting alternative execution models and selected an API for estimating the power spectral density using Welch's method [46] as a representative test case. We then generated a representative synthetic test signal (a 2 Vrms sine wave at 1234 Hz, corrupted by 0.001 V^2/Hz of white noise sampled at 10 kHz) having 50,000,000 data points. We next devised two benchmarks, one using library-specific optimizations and a second strictly using APIs in the array API standard. We ran the benchmarks for the same backends, on the same hardware, and using the same analysis approach as the scikit-learn benchmarks discussed above.

Fig. 2c and Fig. 2d display results, showing average execution time relative to NumPy. When using library-specific optimizations (Fig. 2c), we observe 1.4× higher throughput for PyTorch CPU, 51.6× for PyTorch GPU, and 52.4× for CuPy. When omitting library-specific optimizations (Fig. 2d), we observe a 12-25× **decreased** throughput across all non-NumPy backends.

The source of the performance disparity is due to use of strided views in the optimized implementation. NumPy, CuPy, and PyTorch support the concept of strides, where a stride describes the number of bytes to move forward in memory to progress to the next position along an axis, and provide similar, non-standardized APIs for manipulating the internal data structure of an array. While one can use standardized APIs to achieve the same result, using stride "tricks" enables increased performance. This finding raises an important point. Namely, while the array API standard aims to reduce the need for library-specific code, it will never fully eliminate that need. Users of the standard may need to maintain similar library-specific performance optimizations to achieve maximal performance. We expect, however, that the maintenance burden should only apply for those scenarios in which the performance benefits significantly outweigh the maintenance costs.

Future Work

Consortium work is comprised of three focus areas: standardization, adoption, and coordination.

Standardization: Standardization is the core of Consortium efforts. The Python array API standard is a living standard, which should evolve to reflect the needs and evolution of array libraries within the ecosystem. As such, we expect to continue working with array and array-consuming library maintainers to codify APIs and behaviors suitable for standardization.

Adoption: To ensure the success of the Python array API standard, we work closely with maintainers of array and array-consuming libraries to facilitate adoption by soliciting feedback, addressing pain points, and resolving specification ambiguities. In the immediate future, we plan to release additional tooling for tracking adoption and measuring specification compliance. For the former, we are collecting static compliance data and will publish compatibility tables as part of the array API standard publicly available on-line. For the latter, we are developing an automated test suite reporting system to gather array API test suite results from array libraries as part of continuous integration. We expect these tools to be particularly valuable to array-consuming libraries in order to quickly assess API portability.

Coordination: Providing a forum for coordination among array libraries (and their consumers) was the primary motivating factor behind Consortium formation and is the most important byproduct of Consortium efforts. By facilitating knowledge exchange among array library communities, the Consortium serves as a critical bulwark against further fragmentation and siloed technical stacks. Preventing such fragmentation is to the ultimate benefit of array library consumers and their communities. Additionally, coordination allows for orienting around a shared long-term vision regarding future needs and possible solutions. We are particularly keen to explore the following areas and open questions: device standardization, extended data type support (including strings and datetimes), input-output (IO) APIs, support for mixing array libraries, parallelization, and optional extensions for deep learning, statistical computing, and, more generally, functionality which is out-of-scope, but needed in specific contexts.[8]

We should also note that array API standardization is not the only standardization effort spearheaded by the Consortium. We are also working to standardize APIs and behaviors for Python dataframe libraries, including an interchange protocol and a library-author focused dataframe object and associated set of APIs. This work will be discussed in a future paper.

Conclusion

We introduced the Consortium and the Python array API standard, which specifies standardized APIs and behaviors for array and tensor objects and operations. In developing an initial specification draft, we analyzed common array libraries in the ecosystem and determined a set of common APIs suitable for standardization. In consultation with array and array-consuming library maintainers, we published two specification revisions codifying APIs and behaviors for array objects and their interaction, array interchange, and array-aware functions for array creation and manipulation, statistical reduction, and linear algebra. In addition, we released tooling to facilitate adoption of the array API standard: 1) a test suite for measuring specification compliance and 2) a compatibility layer to allow array-consuming libraries to adopt the standard without having to wait on upstream release cycles.

We further explored performance implications of adopting the array API standard in two commonly-used array-consuming libraries: scikit-learn and SciPy. For the former, we found that adoption enabled scikit-learn to use GPU-based execution models, resulting in significantly increased performance. For the latter, we found similar performance gains; however, in order to realize the performance gains, we needed to use library-specific optimizations. This finding highlights a limitation of the standard. Namely, while the array API standard aims to reduce the need for library-specific code, it will never fully eliminate that need. Users of the standard may need to maintain similar library-specific performance optimizations to achieve maximal performance.

Our work demonstrates the usefulness of the Consortium and the array API standard in facilitating array interoperability within the ecosystem. In addition to shepherding standardization and promoting adoption of the array API standard, the Consortium provides a critical forum for coordinating efforts among array and array-consuming library maintainers. Such coordination is critical to the long-term success and viability of the ecosystem and its communities. Having established a blueprint for standardization methodology and process, the Consortium is also leading a similar effort to standardize Python dataframe APIs and behaviors, thus working to reduce fragmentation for the two fundamental data structures underpinning the ecosystem—arrays and dataframes.

8. To participate in Consortium efforts, consult the Python array API standard public issue tracker [47].

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, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2

[2] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: https://www.semanticscholar.org/paper/CuPy-%3A-A-NumPy-Compatible-Library-for-NVIDIA-GPU-Okuta-Unno/a59da4639436f582e483347a4833e7659fd3e598

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: an imperative style, high-performance deep learning library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., Dec. 2019, no. 721, pp. 8026–8037.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," http://github.com/google/jax, 2018. [Online]. Available: http://github.com/google/jax

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, Nov. 2016, pp. 265–283.

[6] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings for the Annual Scientific Computing with Python Conference*, Austin, Texas, 2015, pp. 126–132, https://doi.org/10.25080/Majora-7b98e3ed-013. [Online]. Available: https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," Dec. 2015, https://doi.org/10.48550/arXiv.1512.01274. [Online]. Available: http://arxiv.org/abs/1512.01274

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research (JMLR)*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: www.jmlr.org/papers/v12/pedregosa11a.html

[9] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, and P. van Mulbregt, "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, https://doi.org/10.1038/s41592-019-0686-2. [Online]. Available: https://www.nature.com/articles/s41592-019-0686-2

[10] K. J. Millman and M. Aivazis, "Python for Scientists and Engineers," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 9–12, Mar. 2011, https://doi.org/10.1109/MCSE.2011.36.

[11] S. J. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. c. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, Jun. 2014, https://doi.org/10.7717/peerj.453. [Online]. Available: https://peerj.com/articles/453

[12] S. Hoyer and J. Hamman, "xarray: N-D labeled Arrays and Datasets in Python," vol. 5, no. 1, p. 10, Apr. 2017, https://doi.org/10.5334/jors.148. [Online]. Available: https://openresearchsoftware.metajnl.com/articles/10.5334/jors.148

[13] H. Abbasi, "Sparse: A more modern sparse array library," in *Proceedings of the 17th Python in Science Conference*, Austin, Texas, 2018, pp. 65–68, https://doi.org/10.25080/Majora-4af1f417-00a. [Online]. Available: https://conference.scipy.org/proceedings/scipy2018/hameer_abbasi.html

[14] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, May 2007, https://doi.org/10.1109/MCSE.2007.55.

[15] F. Pérez, B. E. Granger, and J. D. Hunter, "Python: An Ecosystem for Scientific Computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 13–21, Mar. 2011, https://doi.org/10.1109/MCSE.2010.119.

[16] S. Seabold and J. Perktold, "Statsmodels: Econometric and Statistical Modeling with Python," in *Proceedings for the Annual Scientific Computing with Python Conference*, Austin, Texas, 2010, pp. 92–96, https://doi.org/10.25080/Majora-92bf1922-011. [Online]. Available: https://conference.scipy.org/proceedings/scipy2010/seabold.html

[17] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. D. Team, "Jupyter notebooks - a publishing format for reproducible computational workflows," in *International Conference on Electronic Publishing*, 2016, https://doi.org/10.3233/978-1-61499-649-1-87.

[18] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," in *Proceedings of SysML Conference*, 2018. [Online]. Available: https://mlsys.org/Conferences/doc/2018/146.pdf

[19] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011, https://doi.org/10.1109/MCSE.2011.37.

[20] P. F. Dubois, K. Hinsen, and J. Hugunin, "Numerical Python," *Computer in Physics*, vol. 10, no. 3, pp. 262–267, May 1996, https://doi.org/10.1063/1.4822400. [Online]. Available: https://doi.org/10.1063/1.4822400

[21] J. Hugunin, "Extending Python for Numerical Computation," http://hugunin.net/papers/hugunin95numpy.html, 1995. [Online]. Available: http://hugunin.net/papers/hugunin95numpy.html

[22] P. Greenfield, J. T. Miller, J.-c. Hsu, and R. L. White, "numarray: A New Scientific Array Package for Python," in *PyCon DC*, 2003. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.9899

[23] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," 2011. [Online]. Available: https://www.semanticscholar.org/paper/pandas:-a-Foundational-Python-Library-for-Data-and-McKinney/1a62eb61b2663f8135347171e30cb9dc0a8931b5

[24] C. Moler and J. Little, "A history of MATLAB," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 81:1–81:67, Jun. 2020, https://doi.org/10.1145/3386331. [Online]. Available: https://dl.acm.org/doi/10.1145/3386331

[25] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017, https://doi.org/10.1137/141000671. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671

[26] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019, https://doi.org/10.1109/IEEESTD.2019.8766229.

[27] C. for Python Data API Standards, "Array API Comparison," https://github.com/data-apis/array-api-comparison, 2022. [Online]. Available: https://github.com/data-apis/array-api-comparison

[28] ——, "Python Record API," https://github.com/data-apis/python-record-api, 2020. [Online]. Available: https://github.com/data-apis/python-record-api

[29] J. Fraenkel and B. Grofman, "The Borda Count and its real-world alternatives: Comparing scoring rules in Nauru and Slovenia," *Australian Journal of Political Science*, vol. 49, no. 2, pp. 186–205, Apr. 2014, https://doi.org/10.1080/10361146.2014.900530. [Online]. Available: https://doi.org/10.1080/10361146.2014.900530

[30] P. Emerson, "The original Borda count and partial voting," *Social Choice and Welfare*, vol. 40, no. 2, pp. 353–358, Feb. 2013, https://doi.org/10.1007/s00355-011-0603-9. [Online]. Available: https://doi.org/10.1007/s00355-011-0603-9

[31] DLPack, "Open In Memory Tensor Structure," https://github.com/dmlc/dlpack, 2023. [Online]. Available: https://github.com/dmlc/dlpack

[32] ——, "Python Specification for DLPack," https://dmlc.github.io/dlpack/latest/python_spec.html, 2023. [Online]. Available: https://dmlc.github.io/dlpack/latest/python_spec.html

[33] R. Gommers, S. Hoyer, and A. Meurer, "NEP 47 — Adopting the array API standard — NumPy Enhancement Proposals," https://numpy.org/

neps/nep-0047-array-api-standard.html, Jan. 2021. [Online]. Available: https://numpy.org/neps/nep-0047-array-api-standard.html

[34] A. Meurer, "Implementation of the NEP 47 (adopting the array API standard) by asmeurer · Pull Request #18585 · numpy/numpy," https://github.com/numpy/numpy/pull/18585, Mar. 2021. [Online]. Available: https://github.com/numpy/numpy/pull/18585

[35] S. Berg, "Road to NumPy 2.0," https://mail.python.org/archives/list/numpy-discussion@python.org/thread/XYA5KZNL362Q5KWLKS5QFBQNRT5N2ZJO/#XCJU55EXSQPN5W7UWHDKURBU7EKBBTD2, Jan. 2023. [Online]. Available: https://mail.python.org/archives/list/numpy-discussion@python.org/thread/XYA5KZNL362Q5KWLKS5QFBQNRT5N2ZJO/#XCJU55EXSQPN5W7UWHDKURBU7EKBBTD2

[36] Y.-L. L. Fang, "Adopt Python Array API standard · Issue #4789 · cupy/cupy," https://github.com/cupy/cupy/issues/4789, Mar. 2021. [Online]. Available: https://github.com/cupy/cupy/issues/4789

[37] P. Meier, "Python Array API Compatibility Tracker · Issue #58743 · pytorch/pytorch," https://github.com/pytorch/pytorch/issues/58743, May 2021. [Online]. Available: https://github.com/pytorch/pytorch/issues/58743

[38] J. VanderPlas, "Initial implementation of the Python Array API standard · Pull Request #16099 · google/jax," https://github.com/google/jax/pull/16099, May 2023. [Online]. Available: https://github.com/google/jax/pull/16099

[39] T. White, "Python Array API in Dask issue tracking · Issue #8917 · dask/dask," https://github.com/dask/dask/issues/8917, Apr. 2022. [Online]. Available: https://github.com/dask/dask/issues/8917

[40] N. Yyc, "Python Array API standardization · Issue #20501 · apache/mxnet," https://github.com/apache/mxnet/issues/20501, Aug. 2021. [Online]. Available: https://github.com/apache/mxnet/issues/20501

[41] I. Yashchuk, "Using Array API standard for functions implemented using pure Python and NumPy API · Issue #15354 · scipy/scipy," https://github.com/scipy/scipy/issues/15354, Jan. 2022. [Online]. Available: https://github.com/scipy/scipy/issues/15354

[42] T. Fan, "Path for Adopting the Array API spec · Issue #22352 · scikit-learn/scikit-learn," https://github.com/scikit-learn/scikit-learn/issues/22352, Jan. 2022. [Online]. Available: https://github.com/scikit-learn/scikit-learn/issues/22352

[43] C. for Python Data API Standards, "Test Suite for Array API Compliance," https://github.com/data-apis/array-api-tests, 2022. [Online]. Available: https://github.com/data-apis/array-api-tests

[44] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 11 2019, https://doi.org/10.21105/joss.01891. [Online]. Available: http://dx.doi.org/10.21105/joss.01891

[45] C. for Python Data API Standards, "Array API compatibility library," https://github.com/data-apis/array-api-compat, 2023. [Online]. Available: https://github.com/data-apis/array-api-compat

[46] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 70–73, Jun. 1967, https://doi.org/10.1109/TAU.1967.1161901.

[47] C. for Python Data API Standards, "Array API standard," https://github.com/data-apis/array-api, 2022. [Online]. Available: https://github.com/data-apis/array-api