# Contents

1 What’s new in unumpy? 3

2 Relation to the NumPy duck-array ecosystem 5
   2.1 unumpy ................................................. 5
       2.1.1 Writing Backends .................................. 6

3 Indices and tables 9

Python Module Index 11

Index 13
**Note:** This page describes the overall philosophy behind *unumpy*. If you are interested in a general dispatch mechanism, see *uarray*.

*unumpy* builds on top of *uarray*. It is an effort to specify the core NumPy API, and provide backends for the API.
unumpy is the first approach to leverage uarray in order to build a generic backend system for (what we hope will be) the core NumPy API specification. It’s possible to create the backend object, and use that to perform operations. In addition, it’s possible to change the used backend via a context manager.
CHAPTER 2

Relation to the NumPy duck-array ecosystem

There are three main NumPy enhancement proposals (NEPs) inside NumPy itself that relate to the duck-array ecosystem. There is NEP-22, which is a high-level overview of the duck-array ecosystem, and the direction NumPy intends to move towards. Two main protocols were introduced to fill this gap, the __array_function__ protocol defined in NEP-18, and the older __array_ufunc__ protocol defined in NEP-13.

unumpy provides an alternate framework based on uarray, bypassing the __array_function__ and __array_ufunc__ protocols entirely. It provides a clear separation of concerns. It defines callables which can be overridden, and expresses everything else in terms of these callables. See the uarray documentation for more details.

2.1 unumpy

Note: If you are interested in writing backends or multimethods for unumpy, please look at the documentation for uarray, which explains how to do this.

unumpy is meant for three groups of individuals:

- Those who write array-like objects, like developers of Dask, Xnd, PyData/Sparse, CuPy, and others.
- Library authors or programmers that hope to target multiple array backends, listed above.
- Users who wish to target their code to other backends.

For example, the following is currently possible:

```python
>>> import uarray as ua
>>> import unumpy as np
>>> import unumpy.dask_backend as dask_backend
>>> import unumpy.sparse_backend as sparse_backend
>>> import sparse, dask.array as da
>>> def main():
...    x = np.zeros(5)
```
...    return np.exp(x)

>>> with ua.set_backend(dask_backend):
...    isinstance(main(), da.core.Array)
True

>>> with ua.set_backend(sparse_backend):
...    isinstance(main(), sparse.SparseArray)
True

Now imagine some arbitrarily nested code, all for which the implementations can be switched out using a simple context manager.

unumpy is an in-progress mirror of the NumPy API which allows the user to dynamically switch out the backend that is used. It also allows auto-selection of the backend based on the arguments passed into a function. It does this by defining a collection of uarray multimethods that support dispatch. Although it currently provides a number of backends, the aspiration is that, with time, these back-ends will move into the respective libraries and it will be possible to use the library modules directly as backends.

Note that currently, our coverage is very incomplete. However, we have attempted to provide at least one of each kind of object in unumpy for reference. There are ufunc s and ndarray s, which are classes, methods on ufunc such as __call__, and reduce and also functions such as sum.

Where possible, we attempt to provide default implementations so that the whole API does not have to be reimplemented, however, it might be useful to gain speed or to re-implement it in terms of other functions which already exist in your library.

The idea is that once things are more mature, it will be possible to switch out your backend with a simple import statement switch:

```python
import numpy as np  # Old method
import unumpy as np  # Once this project is mature
```

Currently, the following functions are supported:

- All NumPy universal functions.
  - ufunc reductions

For the full range of functions, use dir(unumpy).

You can use the uarray.set_backend decorator to set a backend and use the desired backend. Note that not every backend supports every method. For example, PyTorch does not have an exact ufunc equivalent, so we dispatch to actual methods using a dictionary lookup. The following backends are supported:

- numpy_backend
- torch_backend
- xnd_backend
- dask_backend
- cupy_backend
- sparse_backend

### 2.1.1 Writing Backends

Since unumpy is based on uarray, all overrides are done via the __ua_*__ protocols. We strongly recommend you read the uarray documentation for context.
All functions/methods in unumpy are uarray multimethods. This means you can override them using the __ua_function__ protocol.

In addition, unumpy allows dispatch on numpy.ndarray, numpy.ufunc and numpy.dtype via the __ua_convert__ protocol.

Dispatching on objects means one can intercept these, convert to an equivalent native format, or dispatch on their methods, including __call__.

We suggest you browse the source for example backends.

Differences between overriding numpy.ufunc objects and other multimethods

Of note here is that there are certain callable objects within NumPy, most prominently numpy.ufunc objects, that are not typical functions/methods, and so cannot be directly overridden, the key word here being directly.

In Python, when a method is called, i.e. `x.method(*a, **kw)` it is the same as writing `type(x).method(x, *a, **kw)` assuming that method was a regular method defined on the type. This allows some very interesting things to happen.

For instance, if we make method a multimethod, it allows us to override methods, provided we know that the first argument passed in will be `x`.

One other thing that is possible (and done in unumpy) is to override the __call__ method on a callable object. This is, in fact, exactly how to override a ufunc.

Other interesting things that can be done (but as of now, are not) are to replace ufunc objects entirely by native equivalents overriding the __get__ method. This technique can also be applied to dtype objects.

Meta-array support

Meta-arrays are arrays that can hold other arrays, such as Dask arrays and XArray datasets.

If meta-arrays and libraries depend on unumpy instead of NumPy, they can benefit from containerization and hold arbitrary arrays; not just numpy.ndarray objects.

Inside their __ua_function__ implementation, they might need to do something like the following:

```python
>>> class Backend: pass
>>> meta_backend = Backend()
>>> meta_backend.__ua_domain__ = "numpy"
>>> def ua_func(f, a, kw):
...     # We do this to avoid infinite recursion
...     with ua.skip_backend(meta_backend):
...         # Actual implementation here
...         pass
>>> meta_backend.__ua_function__ = ua_func
```

In this form, one could do something like the following to use the meta-backend:

```python
>>> with ua.set_backend(sparse_backend), ua.set_backend(dask_backend):
...     x = np.zeros((2000, 2000))
...     isinstance(x, da.Array)
...     isinstance(x.compute(), sparse.SparseArray)
True
True
```

2.1. unumpy
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
U
unumpy (module), 5