ndarray

Daniël de Kok

Introduction

The `ndarray` crate provides n-dimensional array data types. With many useful features:

- Generic over the value type.
- Efficient implementation of many linalg operations.
- Can use a BLAS library for certain linalg operations.
- Array views and subviews.

Resources

- GitHub: https://github.com/bluss/ndarray
- Documentation: https://docs.rs/ndarray
- Crate: https://crates.io/crates/ndarray
- ndarray for NumPy users (**Highly recommended**)

A quick tour

Creating an owned array

```
use ndarray::{Array, Ix1};

// Create a vector with 10 components.
let vector: Array<f32, Ix1> = Array::zeros((10,));
```

Creating an owned array

```rust
use ndarray::{Array, Ix1};

// Create a vector with 10 components.
let vector: Array<f32, Ix1> = Array::zeros((10,));

use ndarray::Ix2;

// Create a matrix with 2 rows and 5 columns.
let matrix: Array<f32, Ix2> = Array::zeros((2, 5));
```

Handy type aliases

```
type Array0<A> = Array<A, Ix0>;
type Array1<A> = Array<A, Ix1>;
type Array2<A> = Array<A, Ix2>;
```

Handy type aliases

```
type Array0<A> = Array<A, Ix0>;
type Array1<A> = Array<A, Ix1>;
type Array2<A> = Array<A, Ix2>;

use ndarray::{Array1, Array2};

let vector: Array1<f32> = Array1::zeros((10,));
let matrix: Array2<f32> = Array2::zeros((2, 5));
```

Views

- A *view* represents an array or a part of it.
- The most basic view is a view of the complete array.

Immutable view

```rust
use ndarray::ArrayView2;

let matrix: Array2<f32> = Array2::zeros((2, 5));

// Create an immutable view
let matrix_view: ArrayView2<f32> = matrix.view();
```

Mutable view

```rust
use ndarray::ArrayViewMut2;

let mut matrix: Array2<f32> = Array2::zeros((2, 5));

// Create a mutable view
let mut matrix_view: ArrayViewMut2<f32> =
    matrix.view_mut();
```

Mutable view types

`ArrayView2` and `ArrayViewMut` are again type aliases:

```
type ArrayView2<'a, A> = ArrayView<'a, A, Ix2>;

type ArrayViewMut2<'a, A> = ArrayViewMut<'a, A, Ix2>;
```

Mutable view types

`ArrayView2` and `ArrayViewMut` are again type aliases:

```
type ArrayView2<'a, A> = ArrayView<'a, A, Ix2>;

type ArrayViewMut2<'a, A> = ArrayViewMut<'a, A, Ix2>;
```

- The lifetime of a view is bound the data struture that owns the data.

Mutable view types

`ArrayView2` and `ArrayViewMut` are again type aliases:

```
type ArrayView2<'a, A> = ArrayView<'a, A, Ix2>;

type ArrayViewMut2<'a, A> = ArrayViewMut<'a, A, Ix2>;
```

- The lifetime of a view is bound the data struture that owns the data.
- Such view types are also defined for different dimensionalities.

Subviews

- A **subview** is a view on a part of an array.

Subviews

- A **subview** is a view on a part of an array.
- A subview is always created along an axis.

Subviews

- A **subview** is a view on a part of an array.
- A subview is always created along an axis.
- For example, for a matrix:
  - `Axis(0)` signifies the rows of the matrix.

Subviews

- A **subview** is a view on a part of an array.
- A subview is always created along an axis.
- For example, for a matrix:
  - `Axis(0)` signifies the rows of the matrix.
  - `Axis(1)` signifies the columns of the matrix.

Subviews

- A **subview** is a view on a part of an array.
- A subview is always created along an axis.
- For example, for a matrix:
  - `Axis(0)` signifies the rows of the matrix.
  - `Axis(1)` signifies the columns of the matrix.
- Examples:
  - Get a view of row number 1 of a matrix:
    ```
    matrix.subview(Axis(0), 1);
    ```
  - Get a view of column number 1 of a matrix:
    ```
    matrix.subview(Axis(1), 1);
    ```

Example

```
use ndarray::{arr2, ArrayView, Axis};

let a = arr2(&[[1., 2. ],
               [3., 4. ],
               [5., 6. ]]);
```

Example

```
use ndarray::{arr2, ArrayView, Axis};

let a = arr2(&[[1., 2. ],
               [3., 4. ],
               [5., 6. ]]);

assert!(a.subview(Axis(0), 1) ==
  ArrayView::from(&[3., 4.]));
```

Example

```
use ndarray::{arr2, ArrayView, Axis};

let a = arr2(&[[1., 2. ],
               [3., 4. ],
               [5., 6. ]]);

assert!(a.subview(Axis(0), 1) ==
  ArrayView::from(&[3., 4.]));

assert!(a.subview(Axis(1), 1) ==
  ArrayView::from(&[2., 4., 6.]));
```

Mutable subviews

`subview_mut` also works following the previous examples to get mutable subviews.

Constructing arrays

- Construct an N-dimensional array filled with ones:

```
assert_eq!(Array::ones((2, 3)),
    arr2(&[[1, 1, 1], [1, 1, 1]]));
```

Constructing arrays

- Construct an N-dimensional array filled with ones:

```
assert_eq!(Array::ones((2, 3)),
    arr2(&[[1, 1, 1], [1, 1, 1]]));
```

- Construct a 1D array from a range:

```
assert_eq!(Array::range(0., 2., 0.5),
    arr1(&[0.0, 0.5, 1.0, 1.5]));
```

Constructing arrays

- Construct an N-dimensional array filled with ones:

```
assert_eq!(Array::ones((2, 3)),
    arr2(&[[1, 1, 1], [1, 1, 1]]));
```

- Construct a 1D array from a range:

```
assert_eq!(Array::range(0., 2., 0.5),
    arr1(&[0.0, 0.5, 1.0, 1.5]));
```

- Construct a 1D array with the interval $[0 \ldots 1]$ with 5 elements:

```
assert_eq!(Array::linspace(0., 1., 5),
    arr1(&[0.0, 0.25, 0.5, 0.75, 1.0]));
```

Constructing arrays (2)

- Construct an array filled with 9:

```
assert_eq!(Array::from_elem((2, 3), 9),
    arr2(&[[9, 9, 9], [9, 9, 9]]));
```

- Create a 4×4 identity matrix:

```
assert_eq!(Array::eye(4), arr2(
    &[[1, 0, 0, 0], [0, 1, 0, 0],
      [0, 0, 1, 0], [0, 0, 0, 1]]));
```

- Create a 3×2 matrix from the elements in a `Vec`:

```
assert_eq!(Array::from_shape_vec((3, 2),
    vec![1, 2, 3, 4, 5, 6]).unwrap(),
    arr2(&[[1, 2], [3, 4], [5, 6]]));
```

Indexing

`Index` is implemented on `ndarray` arrays. Indexes are tuples or fixed-size arrays:

```
assert_eq!(arr2(&[[1, 2], [3, 4], [5, 6]])[(2, 0)], 5);

assert_eq!(arr2(&[[1, 2], [3, 4], [5, 6]])[[1, 1]], 4);
```

Indexing (2)

`IndexMut` is also implemented, so we can mutate cells as well:

```
let mut a = arr2(&[[1, 2], [3, 4], [5, 6]]);
a[(2, 0)] = 7;
assert_eq!(a, arr2(&[[1, 2], [3, 4], [7, 6]]));
```

Operators

The arithmetic operators work elementwise:

```
assert_eq!(arr1(&[1, 2, 3, 4, 5]) * arr1(&[1, 2, 3, 4, 5]),
    arr1(&[1, 4, 9, 16, 25]));

assert_eq!(arr1(&[1, 2, 3, 4, 5]) + arr1(&[1, 2, 3, 4, 5]),
    arr1(&[2, 4, 6, 8, 10]));


assert_eq!(arr1(&[1, 2, 3, 4, 5]) - arr1(&[1, 2, 3, 4, 5]),
    arr1(&[0, 0, 0, 0, 0]));
```

Shapes

- `shape()` gets the shape of an array:

```
assert_eq!(arr2(&[[1, 2], [3, 4], [5, 6]]).shape(),
    &[3, 2]);
```

- An existing array can also be reshaped:

```
assert_eq!(arr2(&[[1, 2], [3, 4], [5, 6]])
    .into_shape((2, 3))
    .unwrap(),
    arr2(&[[1, 2, 3], [4, 5, 6]]));
```

Dot product

- `dot()` computes the dot product between two vectors:

```
assert_eq!(arr1(&[1, 2, 3, 4, 5])
  .dot(&arr1(&[1, 2, 3, 4, 5])),
  55);
```

- And the matrix multiplication of two matrices:

```
assert_eq!(arr2(&[[1, 2, 3], [4, 5, 6]])
  .dot(&arr2(&[[1, 2], [3, 4], [5, 6]])),
  arr2(&[[22, 28], [49, 64]]));
```

How does it work?

ArrayBase

Every array type is a direct or indirect type alias to `ArrayBase`:

```
pub struct ArrayBase<S, D>
    where S: Data
{
    data: S,
    ptr: *mut S::Elem,
    dim: D,
    strides: D,
}
```

ArrayBase

Every array type is a direct or indirect type alias to `ArrayBase`:

```
pub struct ArrayBase<S, D>
    where S: Data
{
    data: S,
    ptr: *mut S::Elem,
    dim: D,
    strides: D,
}
```

- `data`: the actual data
- `ptr`: a pointer to an element in the data.
- `dims`: The dimensionality (shape) of the array.
- `strides`: the array's stride

Array

- The owned array `Array` is an `ArrayBase` that uses `OwnedRepr` as the type for data.

```
type Array<A, D> = ArrayBase<OwnedRepr<A>, D>;
```

Array

- The owned array `Array` is an `ArrayBase` that uses `OwnedRepr` as the type for data.

```
type Array<A, D> = ArrayBase<OwnedRepr<A>, D>;
```

- The owned representation uses a `Vec`:

```
pub struct OwnedRepr<A>(Vec<A>);
```

ArrayView

- The view `ArrayView` is an `ArrayBase` that uses `ViewRepr` as the type for data.

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;
```

ArrayView

- The view `ArrayView` is an `ArrayBase` that uses `ViewRepr` as the type for data.

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;
```

- `ViewRepr` may be a bit surprising:

ArrayView

- The view `ArrayView` is an `ArrayBase` that uses `ViewRepr` as the type for data.

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;
```

- `ViewRepr` may be a bit surprising:

```
pub struct ViewRepr<A> {
    life: PhantomData<A>,
}
```

ArrayView

- The view `ArrayView` is an `ArrayBase` that uses `ViewRepr` as the type for data.

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;
```

- `ViewRepr` may be a bit surprising:

```
pub struct ViewRepr<A> {
    life: PhantomData<A>,
}
```

- Used to carry the lifetime of the data.

ArrayView (2)

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;

pub struct ViewRepr<A> {
    life: PhantomData<A>,
}
```

Where is the actual view stored?

ArrayView (2)

```
type ArrayView<'a, A, D> =
  ArrayBase<ViewRepr<&'a A>, D>;

pub struct ViewRepr<A> {
    life: PhantomData<A>,
}
```

Where is the actual view stored? The `ArrayView` uses is initialized with the appropriate values for `ptr` and `dim`.

`ArrayView` overhead

- Liberally create views
- Memory cost:
  - Pointer
  - Shape
  - Stride