ANCB

Release 0.1.1

Drason "Emmy" Chow

Feb 27, 2021

CONTENTS:

1 Installation **3**

1.1 Basic Usage 3

1.2 About ANCB 6

1.3 ancb package 7

Python Module Index **13**

Index **15**

Another Numpy Circular Buffer is an efficient, fast, and powerful NumPy compatible circular buffers for use in data processing, especially real-time data processing.

CHAPTER

ONE

INSTALLATION

At the command line:

```
$ pip3 install ancb
```

1.1 Basic Usage

A great feature of *ancb.NumpyCircularBuffer* is that it inherits from numpy.ndarray. Many features of *ndarray* are also true of *NumpyCircularBuffer*.

1.1.1 Instantiation

NumpyCircularBuffer requires an ndarray to use for storage of data elements.

```
import numpy as np
from ancb import NumpyCircularBuffer

data = np.empty(3)
buffer = NumpyCircularBuffer(data)
```

The first dimension of the ndarray used to declare a NumpyCircularBuffer is always the length of the buffer. For example if the buffer is (N, a, b) it would be an N length buffer of array elements of shape (a, b).

1.1.2 Buffer operations

Say we have predeclared a buffer that stores 3D vectors as rows.

```
import numpy as np
from ancb import NumpyCircularBuffer
data = np.empty((3, 3))
buffer = NumpyCircularBuffer(data)
```

Appending and popping elements are done through the *ancb.NumpyCircularBuffer.append()* and the *ancb.NumpyCircularBuffer.pop()* functions. *ancb.NumpyCircularBuffer.peek()* can be used if you don't want to consume the element at the beginnning of the buffer.

```
>>> buffer.append([0, 1, 2])
>>> buffer.append([3, 4, 5])
>>> buffer.append([6, 7, 8])
```

(continues on next page)

(continued from previous page)

```
>>> buffer
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Note, while popping an element does **not** fill the space it used to fill with zeros or anything, it simply just marks the space as available to be filled for another element.

```
>>> buffer.pop()
array([0, 1, 2])
>>> buffer.pop()
array([3, 4, 5])
>>> buffer.pop()
array([6, 7, 8])
>>> buffer
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

You can check if a buffer is full or empty using the useful properties *ancb.NumpyCircularBuffer.full()* and *ancb.NumpyCircularBuffer.empty()*. These are convience O(1) operations that check the number of elements in the buffer and do a comparison to see if it's full or empty.

```
>>> buffer.empty
True
buffer.full
False
>>> buffer.append([0, 1, 2])
>>> buffer.empty
False
>>> buffer.full
False
>>> buffer.append([3, 4, 5])
>>> buffer.append([6, 7, 8])
>>> buffer.full
True
>>> buffer.empty
False
```

As a quick explaination of circular buffers, when you write to a full buffer, the oldest element is overwritten.

```
>>> buffer.append([9, 10, 11])
>>> buffer
array([[9, 10, 11],  <- end (append will write to the next element)
       [3, 4, 5],  <- start (popping will give you this element)
       [6, 7, 8]])
```

Another useful property to test if you're intending on making your own wrapper functions is fragmentation. Roughly speaking, when the elements are no longer contigously placed (when the end of the buffer occurs in the data before the beginning as above), the buffer is said to be fragmented.

There is another O(1) operation that checks the position of the beginning and end of the buffer along with its current size to determine if it's fragmented.

```
>>> buffer.fragmented
True
```

(continues on next page)

(continued from previous page)

```
>>> buffer.append([12, 13, 14])
>>> buffer.append([15, 16, 17])
>>> buffer
array([[9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
>>> buffer.fragmented
False
>>> buffer.pop()
array([9, 10, 11])
>>> buffer.fragmented
False
```

1.1.3 Overloaded Operations

While all of this is useful, perhaps what is more interesting is the idea of using such a buffer for data processing. Let's imagine a scenario where you want to weight the data by a vector such as [1, 0.5, 0.25] so that each element is weighted half as much as the one before it.

If the data was coming in live, we would have no choice but to use `numpy.roll()` on the data so that it aligns with our weights array. Even if we try to use a circular buffer, it turns out that the gains in performance by using it are lost when we are forced to roll the array for our algorithm since `numpy.roll()` has to every element in the array and move it to a new location.

Fortunately, NumpyCircularBuffer recognizes that you shouldn't need to reorder elements before you do the operation. Since we know where the buffer fragments, we can simply add the end of the buffer to the end of the array and the start of the buffer to the start of the array at no extra cost.

All this shuffling takes place behind the scenes, so you can do:

```
>>> buffer.append([18, 19, 20])
>>> buffer
array([[18, 19, 20],
       [12, 13, 14],
       [15, 16, 17]])
>>> buffer * np.array([0.25, 0.5, 1]).reshape(3, 1)
array([[ 3.  ,  3.25,  3.5],
       [ 7.5 ,  8.  ,  8.5],
       [18.  , 19.  , 20.]])
```

1.1.4 A Caveat: Matrix Multiplication

Most of the library has no overhead; however, an exception to this are certain kinds of matrix multiplication. I will outline the cases below.

Right matrix multiplication (x @ buffer) if the buffer is fragmented:

- x.ndim == 1 and buffer.ndim > 1 or
- x.ndim > 1 and buffer.ndim == 1 or
- buffer.ndim == 2

Left matrix multiplication (buffer @ x) if the buffer is fragmented:

- buffer.ndim == 1

In all of these cases, the overhead is a memory allocation of an ndarray equal to the size of the output. For all functions in ANCB, the specified operation takes place in two seperate parts; however, for these kinds of matrix multiplication, the parts overlap and must be added together for the final result unlike other functions.

The functions `ancb.NumpyCircularBuffer.matmul()` and `ancb.NumpyCircularBuffer.rmatmul()` have been provided to combat this overhead. They allow you to use preallocated space to reduce the overhead of the allocations for repeated operations such as in a loop.

```
import numpy as np
from ancb import NumpyCircularBuffer

data = np.empty(3)
buffer = NumpyCircularBuffer(data)

buffer.append(0)
buffer.append(1)
buffer.append(2)
buffer.append(3)

A = np.arange(9).reshape(3, 3)
work_buffer = empty(3)

# Same as A @ buffer
print(buffer.rmatmul(A, work_buffer))
```

```
[8 26 44]
```

1.2 About ANCB

Another NumPy Circular Buffer is another attempt to leverage Python's Numpy library to implement the circular buffer (ring buffer) data structure. While it is relatively easy to implement a circular buffer (ring buffer) data structure, it is relatively hard to make NumPy ndarray operations work with them.

Most implementations are a class that expose append and pop methods and use NumPy for allocating the data that backs the buffer; however, trying to use such a buffer with NumPy requires the same amount of overhead as other common solutions to using a buffer such as `collections.deque` or `numpy.roll()`, which reallocates new arrays when (for the case of `collections.deque`) converted to a `numpy.ndarray` or rolled into order (in the case of more common NumPy circular buffer implementations).

This means for most other implementations that want to use NumPy ufuncs to process data, first the data must always be copied into a newly allocated ndarray.

This is where ANCB comes in. ANCB implements ufunc operations on a circular buffer. Unlike other implementations, ANCB guarantees that all supported operations will never perform extra copying or rearranging of array elements unless explicitly mentioned. An example of addition is shown below.

[insert image here]

This reduces the overhead of allocating a new array and copying elements, which can be significant for very large buffers or frequently used buffers in loops.

ANCB was developed primarily by Drason "Emmy" Chow during their time at IU: Bloomington working as a Undergraduate Research Assistant. Inefficient control loops of various motion control algorithms required continuous buffers of data that were rolled, creating large performance bottlenecks. Work was done on ANCB to resolve such bottlenecks.

1.3 ancb package

1.3.1 Module contents

class ancb.**NumpyCircularBuffer**(*data*, *bounds: Tuple[int, int] = (0, 0)*)
Bases: numpy.ndarray

Implements a circular (ring) buffer using a numpy array. This implmentation uses an internal size count and capacity count so that the data region is fully utilized.

all(**args*, ***kwargs*)
Returns True if all elements evaluate to True.

Returns True if all elements evaluate to True, False otherwise.

See also:

ndarray.all()

any(**args*, ***kwargs*)
Returns True if any elements evaluate to True.

Returns True if any elements evaluate to True, False otherwise.

See also:

ndarray.any()

append(*value*)
Append a value to the buffer on the right. If the buffer is full, the buffer will advance forward (wrapping around at the ends) and overwrite an element.

Time complexity: O(1)

See also:

NumpyCircularBuffer.pop(), *NumpyCircularBuffer.peek()*

argmax(**args*, ***kwargs*)

Raises **NotImplementedError** – This function will be implemented in the future

argmin(**args*, ***kwargs*)

Raises **NotImplementedError** – This function will be implemented in the future

byteswap(*inplace=False*)
Swap the bytes of the array elements over the valid range of the buffer

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

See also:

ndarray.byteswap()

choose(*choices*, *out=None*, *mode='raise'*)

Raises **NotImplementedError** – This function is being considered for implementation in the future

clip(*min=None*, *max=None*, *out=None*, ***kwargs*)
Return an array whose values are limited to [min, max] over the valid range of the buffer. One of max or min must be given.

See also:

`numpy.clip()`

conj()
Complex-conjugate all elements over the valid range of the buffer.

See also:

`numpy.conjugate()`

conjugate()
Complex-conjugate all elements over the valid range of the buffer.

See also:

`numpy.conjugate()`

copy(*order='C'*, *defrag=False*)
Return a copy of the array over the valid range of the buffer.

See also:

`ndarray.copy()`

cumprod(*axis=None*, *dtype=None*, *out=None*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

cumsum(*axis=None*, *dtype=None*, *out=None*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

diagonal(*offset=0*, *axis1=0*, *axis2=1*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

dot(*b*, *out=None*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

dump(*b*, *out=None*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

dumps(*b*, *out=None*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

property empty
Property that returns True if the buffer is empty, False otherwise.

Time complexity: O(1)

Returns True if buffer is empty, False otherwise.

Return type bool

See also:

NumpyCircularBuffer.full()

fill(*value*)
Fill the valid region of the buffer with a scalar value.

Parameters **value** (*(scalar)*) – All elements of *a* will be assigned this value.

See also:

`ndarray.fill()`

flatten(*order='C'*, *defrag=False*)
Return a copy of the array collapsed into one dimension.

See also:

`ndarray.flatten()`

property fragmented
Property that returns True if the buffer is fragmented (the beginning index is greater than the end index), False otherwise.

Time complexity: O(1)

Returns True if buffer is fragmented, False otherwise.

Return type bool

property full
Property that returns True if the buffer is full, False otherwise.

Time complexity: O(1)

Returns True if buffer is full, False otherwise.

Return type bool

See also:

NumpyCircularBuffer.empty()

get_partions() → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]
Gets a slice of the buffer between the beginning and end indices. If the buffer is fragmented, a tuple of two slices of the two fragments sequentially. Concatenating the slices in the order they are in the tuple will return a list of elements in the correct order.

Time complexity: O(1)

Returns slice or tuple of slices of the array elements in order

Return type Union[ndarray, Tuple[ndarray, ndarray]]

getfield(*offset=0*) → NoReturn

Raises **NotImplementedError** – This function is being considered for implementation in the future

item(**args*) → NoReturn

Raises **NotImplementedError** – This function is being considered for implementation in the future

itemset(**args*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

max(*axis=None*, *out=None*, *keepdims=False*, *initial=None*, *where=True*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

min(*axis=None*, *out=None*, *keepdims=False*, *initial=None*, *where=True*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

newbyteorder(*new_order='S'*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

nonzero() → NoReturn

Raises **NotImplementedError** – This function has no plan for implementation as of this version.

peek()
Gets the element at the start of the buffer without advancing the start of the buffer.

Time complexity: O(1)

Raises ValueError if buffer is empty

Returns element at the start of the buffer

See also:

NumpyCircularBuffer.pop(), *NumpyCircularBuffer.append()*

pop()
Gets the element at the start of the buffer and advances the start of the buffer by one, consuming the element returned.

Time complexity: O(1)

Raises ValueError if buffer is empty

Returns element at the start of the buffer

See also:

NumpyCircularBuffer.peek(), *NumpyCircularBuffer.append()*

prod(*dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

ptp(*out=None*, *keepdims=False*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

put(*values*, *mode='raise'*) → NoReturn

Raises **NotImplementedError** – This function has no plan for implementation as of this version.

ravel() → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

repeat() → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

reset()
Empties all elements from the buffer.

Time complexity: O(1)

Returns True if buffer is empty, False otherwise.

Return type bool

round(*decimals=0*, *out=None*)
Return a copy of the valid region of the buffer with each element rounded to the given number of decimals.

See also:

numpy.around()

searchsorted(*v*, *side='left'*, *sorter=None*) → NoReturn

Raises **NotImplementedError** – This function has no plan for implementation as of this version.

setfield(*val*, *dtype*, *offset=0*) → NoReturn

Raises **NotImplementedError** – This function has no plan for implementation as of this version.

squeeze(*axis=- 1*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

std(*axis=- 1*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

sum(*axis=- 1*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

swapaxes(*axis1*, *axis2*) → NoReturn

Raises **NotImplementedError** – This function is being considered for implementation in the future

take(*indices*, *axis=None*, *out=None*, *mode='raise'*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

tobytes(*order='C'*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

tofile(*fid*, *sep=''*, *format='%s'*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

tolist() → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

tostring(*order='C'*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

transpose(**axes*) → NoReturn

Raises **NotImplementedError** – This function has no plan for implementation as of this version.

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*) → NoReturn

Raises **NotImplementedError** – This function will be implemented in the future

ancb.**can_broadcast**(*shape1*, *shape2*) → bool

Check if shapes shape1 and shape2 can be broadcast together.

Parameters

- **shape1** (*Tuple*) – first shape to parse
- **shape2** (*Tuple*) – second shape to parse

Returns True if arr1 and arr2 can be broadcast together, False otherwise

Return type bool

ancb.**star_can_broadcast**(*starexpr*) → bool

Check if shapes shape1 and shape2 can be broadcast together from a tuple of zip_longest(shape1, shape2, fillvalue=1) called the "starexpr"

Parameters **starexpr** (*Tuple*) – starexpr to parse

Returns True if shape1 and shape2 can be broadcast together, False otherwise

Return type bool

PYTHON MODULE INDEX

a

ancb, 7

INDEX

A

all() (*ancb.NumpyCircularBuffer method*), 7
ancb
    module, 7
any() (*ancb.NumpyCircularBuffer method*), 7
append() (*ancb.NumpyCircularBuffer method*), 7
argmax() (*ancb.NumpyCircularBuffer method*), 7
argmin() (*ancb.NumpyCircularBuffer method*), 7

B

byteswap() (*ancb.NumpyCircularBuffer method*), 7

C

can_broadcast() (*in module ancb*), 11
choose() (*ancb.NumpyCircularBuffer method*), 7
clip() (*ancb.NumpyCircularBuffer method*), 7
conj() (*ancb.NumpyCircularBuffer method*), 8
conjugate() (*ancb.NumpyCircularBuffer method*), 8
copy() (*ancb.NumpyCircularBuffer method*), 8
cumprod() (*ancb.NumpyCircularBuffer method*), 8
cumsum() (*ancb.NumpyCircularBuffer method*), 8

D

diagonal() (*ancb.NumpyCircularBuffer method*), 8
dot() (*ancb.NumpyCircularBuffer method*), 8
dump() (*ancb.NumpyCircularBuffer method*), 8
dumps() (*ancb.NumpyCircularBuffer method*), 8

E

empty() (*ancb.NumpyCircularBuffer property*), 8

F

fill() (*ancb.NumpyCircularBuffer method*), 8
flatten() (*ancb.NumpyCircularBuffer method*), 8
fragmented() (*ancb.NumpyCircularBuffer property*), 9
full() (*ancb.NumpyCircularBuffer property*), 9

G

get_partions() (*ancb.NumpyCircularBuffer method*), 9
getfield() (*ancb.NumpyCircularBuffer method*), 9

I

item() (*ancb.NumpyCircularBuffer method*), 9
itemset() (*ancb.NumpyCircularBuffer method*), 9

M

max() (*ancb.NumpyCircularBuffer method*), 9
min() (*ancb.NumpyCircularBuffer method*), 9
module
    ancb, 7

N

newbyteorder() (*ancb.NumpyCircularBuffer method*), 9
nonzero() (*ancb.NumpyCircularBuffer method*), 9
NumpyCircularBuffer (*class in ancb*), 7

P

peek() (*ancb.NumpyCircularBuffer method*), 10
pop() (*ancb.NumpyCircularBuffer method*), 10
prod() (*ancb.NumpyCircularBuffer method*), 10
ptp() (*ancb.NumpyCircularBuffer method*), 10
put() (*ancb.NumpyCircularBuffer method*), 10

R

ravel() (*ancb.NumpyCircularBuffer method*), 10
repeat() (*ancb.NumpyCircularBuffer method*), 10
reset() (*ancb.NumpyCircularBuffer method*), 10
round() (*ancb.NumpyCircularBuffer method*), 10

S

searchsorted() (*ancb.NumpyCircularBuffer method*), 10
setfield() (*ancb.NumpyCircularBuffer method*), 11
squeeze() (*ancb.NumpyCircularBuffer method*), 11
star_can_broadcast() (*in module ancb*), 11
std() (*ancb.NumpyCircularBuffer method*), 11
sum() (*ancb.NumpyCircularBuffer method*), 11
swapaxes() (*ancb.NumpyCircularBuffer method*), 11

T

take() (*ancb.NumpyCircularBuffer method*), 11
tobytes() (*ancb.NumpyCircularBuffer method*), 11
tofile() (*ancb.NumpyCircularBuffer method*), 11
tolist() (*ancb.NumpyCircularBuffer method*), 11
tostring() (*ancb.NumpyCircularBuffer method*), 11
transpose() (*ancb.NumpyCircularBuffer method*), 11

V

var() (*ancb.NumpyCircularBuffer method*), 11