bottleneck Documentation

Release 1.3.0

Keith Goodman and Bottleneck Developers

Nov 13, 2019

CONTENTS

Fast NumPy array functions written in C.

CHAPTER

ONE

BOTTLENECK

Bottleneck is a collection of fast NumPy array functions written in C.

Let's give it a try. Create a NumPy array:

```
>>> import numpy as np
>>> a = np.array([1, 2, np.nan, 4, 5])
```

Find the nanmean:

```
>>> import bottleneck as bn
>>> bn.nanmean(a)
3.0
```

Moving window mean:

```
>>> bn.move_mean(a, window=2, min_count=1)
array([ 1. ,  1.5,  2. ,  4. ,  4.5])
```

1.1 Benchmark

Bottleneck comes with a benchmark suite:

```
>>> bn.bench()
Bottleneck performance benchmark
    Bottleneck 1.3.0.dev0+122.gb1615d7; Numpy 1.16.4
    Speed is NumPy time divided by Bottleneck time
    NaN means approx one-fifth NaNs; float64 used

               no NaN      no NaN       NaN        no NaN       NaN
                (100,)  (1000,1000)(1000,1000)(1000,1000)(1000,1000)
                axis=0      axis=0     axis=0     axis=1      axis=1
nansum           29.7         1.4        1.6        2.0         2.1
nanmean          99.0         2.0        1.8        3.2         2.5
nanstd          145.6         1.8        1.8        2.7         2.5
nanvar          138.4         1.8        1.8        2.8         2.5
nanmin           27.6         0.5        1.7        0.7         2.4
nanmax           26.6         0.6        1.6        0.7         2.5
median          120.6         1.3        4.9        1.1         5.7
nanmedian       117.8         5.0        5.7        4.8         5.5
ss               13.2         1.2        1.3        1.5         1.5
nanargmin        66.8         5.5        4.8        3.5         7.1
nanargmax        57.6         2.9        5.1        2.5         5.3
```

(continues on next page)

(continued from previous page)

```
anynan          10.2        0.3       52.3        0.8       41.6
allnan          15.1      196.0      156.3      135.8      111.2
rankdata        45.9        1.2        1.2        2.1        2.1
nanrankdata     50.5        1.4        1.3        2.4        2.3
partition        3.3        1.1        1.6        1.0        1.5
argpartition     3.4        1.2        1.5        1.1        1.6
replace          9.0        1.5        1.5        1.5        1.5
push          1565.6        5.9        7.0       13.0       10.9
move_sum      2159.3       31.1       83.6      186.9      182.5
move_mean     6264.3       66.2      111.9      361.1      246.5
move_std      8653.6       86.5      163.7      232.0      317.7
move_var      8856.0       96.3      171.6      267.9      332.9
move_min      1186.6       13.4       30.9       23.5       45.0
move_max      1188.0       14.6       29.9       23.5       46.0
move_argmin   2568.3       33.3       61.0       49.2       86.8
move_argmax   2475.8       30.9       58.6       45.0       82.8
move_median   2236.9      153.9      151.4      171.3      166.9
move_rank      847.1        1.2        1.4        2.3        2.6
```

You can also run a detailed benchmark for a single function using, for example, the command:

```
>>> bn.bench_detailed("move_median", fraction_nan=0.3)
```

Only arrays with data type (dtype) int32, int64, float32, and float64 are accelerated. All other dtypes result in calls to slower, unaccelerated functions. In the rare case of a byte-swapped input array (e.g. a big-endian array on a little-endian operating system) the function will not be accelerated regardless of dtype.

1.2 Where

download	https://pypi.python.org/pypi/Bottleneck
docs	https://bottleneck.readthedocs.io
code	https://github.com/pydata/bottleneck
mailing list	https://groups.google.com/group/bottle-neck

1.3 License

Bottleneck is distributed under a Simplified BSD license. See the LICENSE file and LICENSES directory for details.

1.4 Install

Requirements:

Bottleneck	Python 2.7, 3.5, 3.6, 3.7, 3.8; NumPy 1.16.0+
Compile	gcc, clang, MinGW or MSVC
Unit tests	pytest
Documentation	sphinx, numpydoc

To install Bottleneck on Linux, Mac OS X, et al.:

```
$ pip install .
```

To install bottleneck on Windows, first install MinGW and add it to your system path. Then install Bottleneck with the command:

```
python setup.py install --compiler=mingw32
```

Alternatively, you can use the Windows binaries created by Christoph Gohlke: http://www.lfd.uci.edu/~gohlke/pythonlibs/#bottleneck

1.5 Unit tests

After you have installed Bottleneck, run the suite of unit tests:

```
In [1]: import bottleneck as bn

In [2]: bn.test()
============================= test session starts ==============================
platform linux -- Python 3.7.4, pytest-4.3.1, py-1.8.0, pluggy-0.12.0
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/chris/
↪code/bottleneck/.hypothesis/examples')
rootdir: /home/chris/code/bottleneck, inifile: setup.cfg
plugins: openfiles-0.3.2, remotedata-0.3.2, doctestplus-0.3.0, mock-1.10.4, forked-1.
↪0.2, cov-2.7.1, hypothesis-4.32.2, xdist-1.26.1, arraydiff-0.3
collected 190 items

bottleneck/tests/input_modification_test.py ........................... [ 14%]
..                                                                     [ 15%]
bottleneck/tests/list_input_test.py ...............................    [ 30%]
bottleneck/tests/move_test.py .....................................    [ 47%]
bottleneck/tests/nonreduce_axis_test.py ......................         [ 58%]
bottleneck/tests/nonreduce_test.py ..........                          [ 63%]
bottleneck/tests/reduce_test.py ....................................... [ 84%]
.............                                                          [ 90%]
bottleneck/tests/scalar_input_test.py ...................              [100%]

========================== 190 passed in 46.42 seconds ==========================
Out[2]: True
```

If developing in the git repo, simply run `py.test`

CHAPTER

TWO

FUNCTION REFERENCE

Bottleneck provides the following functions:

reduce	*nansum*, *nanmean*, *nanstd*, *nanvar*, *nanmin*, *nanmax*, *median*, *nanmedian*, *ss*, *nanargmin*, *nanargmax*, *anynan*, *allnan*
non-reduce	*replace*
non-reduce with axis	*rankdata*, *nanrankdata*, *partition*, *argpartition*, *push*
moving window	*move_sum*, *move_mean*, *move_std*, *move_var*, *move_min*, *move_max*, *move_argmin*, *move_argmax*, *move_median*, *move_rank*

2.1 Reduce

Functions the reduce the input array along the specified axis.

---

bottleneck.**nansum**(*a*, *axis=None*)

Sum of array elements along given axis treating NaNs as zero.

The data type (dtype) of the output is the same as the input. On 64-bit operating systems, 32-bit input is NOT upcast to 64-bit accumulator and return values.

Parameters

a [array_like] Array containing numbers whose sum is desired. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the sum is computed. The default (axis=None) is to compute the sum of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned.

Notes

No error is raised on overflow.

If positive or negative infinity are present the result is positive or negative infinity. But if both positive and negative infinity are present, the result is Not A Number (NaN).

Examples

```
>>> bn.nansum(1)
1
>>> bn.nansum([1])
1
>>> bn.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> bn.nansum(a)
3.0
>>> bn.nansum(a, axis=0)
array([ 2.,  1.])
```

When positive infinity and negative infinity are present:

```
>>> bn.nansum([1, np.nan, np.inf])
inf
>>> bn.nansum([1, np.nan, np.NINF])
-inf
>>> bn.nansum([1, np.nan, np.inf, np.NINF])
nan
```

---

bottleneck.**nanmean**(*a*, *axis=None*)

Mean of array elements along given axis ignoring NaNs.

float64 intermediate and return values are used for integer inputs.

Parameters

a [array_like] Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the means are computed. The default (axis=None) is to compute the mean of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned. *float64* intermediate and return values are used for integer inputs.

See also:

bottleneck.nanmedian Median along specified axis, ignoring NaNs.

Notes

No error is raised on overflow. (The sum is computed and then the result is divided by the number of non-NaN elements.)

If positive or negative infinity are present the result is positive or negative infinity. But if both positive and negative infinity are present, the result is Not A Number (NaN).

Examples

```
>>> bn.nanmean(1)
1.0
>>> bn.nanmean([1])
1.0
>>> bn.nanmean([1, np.nan])
1.0
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.nanmean(a)
2.0
>>> bn.nanmean(a, axis=0)
array([ 1.,  4.])
```

When positive infinity and negative infinity are present:

```
>>> bn.nanmean([1, np.nan, np.inf])
inf
>>> bn.nanmean([1, np.nan, np.NINF])
-inf
>>> bn.nanmean([1, np.nan, np.inf, np.NINF])
nan
```

---

bottleneck.**nanstd**(*a*, *axis=None*, *ddof=0*)

Standard deviation along the specified axis, ignoring NaNs.

float64 intermediate and return values are used for integer inputs.

Instead of a faster one-pass algorithm, a more stable two-pass algorithm is used.

An example of a one-pass algorithm:

```
>>> np.sqrt((a*a).mean() - a.mean()**2)
```

An example of a two-pass algorithm:

```
>>> np.sqrt(((a - a.mean())**2).mean())
```

Note in the two-pass algorithm the mean must be found (first pass) before the squared deviation (second pass) can be found.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the standard deviation is computed. The default (axis=None) is to compute the standard deviation of the flattened array.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned. *float64* intermediate and return values are used for integer inputs. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See also:

bottleneck.nanvar Variance along specified axis ignoring NaNs

Notes

If positive or negative infinity are present the result is Not A Number (NaN).

Examples

```
>>> bn.nanstd(1)
0.0
>>> bn.nanstd([1])
0.0
>>> bn.nanstd([1, np.nan])
0.0
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.nanstd(a)
1.4142135623730951
>>> bn.nanstd(a, axis=0)
array([ 0.,  0.])
```

When positive infinity or negative infinity are present NaN is returned:

```
>>> bn.nanstd([1, np.nan, np.inf])
nan
```

---

bottleneck.**nanvar**(*a*, *axis=None*, *ddof=0*)

Variance along the specified axis, ignoring NaNs.

float64 intermediate and return values are used for integer inputs.

Instead of a faster one-pass algorithm, a more stable two-pass algorithm is used.

An example of a one-pass algorithm:

```
>>> (a*a).mean() - a.mean()**2
```

An example of a two-pass algorithm:

```
>>> ((a - a.mean())**2).mean()
```

Note in the two-pass algorithm the mean must be found (first pass) before the squared deviation (second pass) can be found.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the variance is computed. The default (axis=None) is to compute the variance of the flattened array.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of non_NaN elements. By default *ddof* is zero.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned. *float64* intermediate and return values are used for integer inputs. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See also:

bottleneck.nanstd Standard deviation along specified axis ignoring NaNs.

Notes

If positive or negative infinity are present the result is Not A Number (NaN).

Examples

```
>>> bn.nanvar(1)
0.0
>>> bn.nanvar([1])
0.0
>>> bn.nanvar([1, np.nan])
0.0
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.nanvar(a)
2.0
>>> bn.nanvar(a, axis=0)
array([ 0.,  0.])
```

When positive infinity or negative infinity are present NaN is returned:

```
>>> bn.nanvar([1, np.nan, np.inf])
nan
```

---

bottleneck.**nanmin**(*a*, *axis=None*)

Minimum values along specified axis, ignoring NaNs.

When all-NaN slices are encountered, NaN is returned for that slice.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the minimum is computed. The default (axis=None) is to compute the minimum of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned. The same dtype as *a* is returned.

See also:

bottleneck.nanmax Maximum along specified axis, ignoring NaNs.

bottleneck.nanargmin Indices of minimum values along axis, ignoring NaNs.

Examples

```
>>> bn.nanmin(1)
1
>>> bn.nanmin([1])
1
>>> bn.nanmin([1, np.nan])
1.0
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.nanmin(a)
1.0
>>> bn.nanmin(a, axis=0)
array([ 1.,  4.])
```

---

bottleneck.**nanmax**(*a*, *axis=None*)

Maximum values along specified axis, ignoring NaNs.

When all-NaN slices are encountered, NaN is returned for that slice.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the maximum is computed. The default (axis=None) is to compute the maximum of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, a scalar is returned. The same dtype as *a* is returned.

See also:

bottleneck.nanmin Minimum along specified axis, ignoring NaNs.

bottleneck.nanargmax Indices of maximum values along axis, ignoring NaNs.

Examples

```
>>> bn.nanmax(1)
1
>>> bn.nanmax([1])
1
>>> bn.nanmax([1, np.nan])
1.0
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.nanmax(a)
4.0
>>> bn.nanmax(a, axis=0)
array([ 1.,  4.])
```

---

bottleneck.**median**(*a*, *axis=None*)

Median of array elements along given axis.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the median is computed. The default (axis=None) is to compute the median of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, except that the specified axis has been removed. If *a* is a 0d array, or if axis is None, a scalar is returned. *float64* return values are used for integer inputs. NaN is returned for a slice that contains one or more NaNs.

See also:

bottleneck.nanmedian Median along specified axis ignoring NaNs.

Examples

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> bn.median(a)
    3.5
>>> bn.median(a, axis=0)
    array([ 6.5,  4.5,  2.5])
>>> bn.median(a, axis=1)
    array([ 7.,  2.])
```

`bottleneck.`**`nanmedian`**(*a*, *axis=None*)

Median of array elements along given axis ignoring NaNs.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the median is computed. The default (axis=None) is to compute the median of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*, except that the specified axis has been removed. If *a* is a 0d array, or if axis is None, a scalar is returned. *float64* return values are used for integer inputs.

See also:

bottleneck.median Median along specified axis.

Examples

```
>>> a = np.array([[np.nan, 7, 4], [3, 2, 1]])
>>> a
array([[ nan,   7.,   4.],
       [  3.,   2.,   1.]])
>>> bn.nanmedian(a)
3.0
>> bn.nanmedian(a, axis=0)
array([ 3. ,  4.5,  2.5])
>> bn.nanmedian(a, axis=1)
array([ 5.5,  2. ])
```

bottleneck.**ss**(*a*, *axis=None*)

Sum of the square of each element along the specified axis.

Parameters

a [array_like] Array whose sum of squares is desired. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the sum of squares is computed. The default (axis=None) is to sum the squares of the flattened array.

Returns

y [ndarray] The sum of a**2 along the given axis.

Examples

```
>>> a = np.array([1., 2., 5.])
>>> bn.ss(a)
30.0
```

And calculating along an axis:

```
>>> b = np.array([[1., 2., 5.], [2., 5., 6.]])
>>> bn.ss(b, axis=1)
array([ 30., 65.])
```

---

bottleneck.**nanargmin**(*a*, *axis=None*)

Indices of the minimum values along an axis, ignoring NaNs.

For all-NaN slices `ValueError` is raised. Unlike NumPy, the results can be trusted if a slice contains only NaNs and Infs.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which to operate. By default (axis=None) flattened input is used.

Returns

index_array [ndarray] An array of indices or a single index value.

See also:

bottleneck.nanargmax Indices of the maximum values along an axis.

bottleneck.nanmin Minimum values along specified axis, ignoring NaNs.

Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> bn.nanargmin(a)
2
>>> a.flat[2]
2.0
>>> bn.nanargmin(a, axis=0)
array([1, 1])
>>> bn.nanargmin(a, axis=1)
array([1, 0])
```

---

bottleneck.**nanargmax**(*a*, *axis=None*)

Indices of the maximum values along an axis, ignoring NaNs.

For all-NaN slices `ValueError` is raised. Unlike NumPy, the results can be trusted if a slice contains only NaNs and Infs.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which to operate. By default (axis=None) flattened input is used.

Returns

index_array [ndarray] An array of indices or a single index value.

See also:

bottleneck.nanargmin Indices of the minimum values along an axis.

bottleneck.nanmax Maximum values along specified axis, ignoring NaNs.

Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> bn.nanargmax(a)
1
>>> a.flat[1]
4.0
>>> bn.nanargmax(a, axis=0)
array([1, 0])
>>> bn.nanargmax(a, axis=1)
array([1, 1])
```

---

bottleneck.**anynan**(*a*, *axis=None*)

Test whether any array element along a given axis is NaN.

Returns the same output as np.isnan(a).any(axis)

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which NaNs are searched. The default (*axis* = `None`) is to search for NaNs over a flattened input array.

Returns

y [bool or ndarray] A boolean or new *ndarray* is returned.

See also:

bottleneck.allnan Test if all array elements along given axis are NaN

Examples

```
>>> bn.anynan(1)
False
>>> bn.anynan(np.nan)
True
>>> bn.anynan([1, np.nan])
True
>>> a = np.array([[1, 4], [1, np.nan]])
>>> bn.anynan(a)
True
>>> bn.anynan(a, axis=0)
array([False,  True], dtype=bool)
```

---

bottleneck.**allnan**(*a*, *axis=None*)

Test whether all array elements along a given axis are NaN.

Returns the same output as np.isnan(a).all(axis)

Note that allnan([]) is True to match np.isnan([]).all() and all([])

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which NaNs are searched. The default (*axis* = `None`) is to search for NaNs over a flattened input array.

Returns

y [bool or ndarray] A boolean or new *ndarray* is returned.

See also:

bottleneck.anynan Test if any array element along given axis is NaN

Examples

```
>>> bn.allnan(1)
False
>>> bn.allnan(np.nan)
True
>>> bn.allnan([1, np.nan])
False
>>> a = np.array([[1, np.nan], [1, np.nan]])
>>> bn.allnan(a)
False
>>> bn.allnan(a, axis=0)
array([False,  True], dtype=bool)
```

An empty array returns True:

```
>>> bn.allnan([])
True
```

which is similar to:

```
>>> all([])
True
>>> np.isnan([]).all()
True
```

2.2 Non-reduce

Functions that do not reduce the input array and do not take *axis* as input.

---

bottleneck.**replace**(*a*, *old*, *new*)

Replace (inplace) given scalar values of an array with new values.

The equivalent numpy function:

a[a==old] = new

Or in the case where old=np.nan:

a[np.isnan(old)] = new

Parameters

a [numpy.ndarray] The input array, which is also the output array since this functions works inplace.

old [scalar] All elements in *a* with this value will be replaced by *new*.

new [scalar] All elements in *a* with a value of *old* will be replaced by *new*.

Returns

Returns a view of the input array after performing the replacements,

if any.

Examples

Replace zero with 3 (note that the input array is modified):

```
>>> a = np.array([1, 2, 0])
>>> bn.replace(a, 0, 3)
>>> a
array([1, 2, 3])
```

Replace np.nan with 0:

```
>>> a = np.array([1, 2, np.nan])
>>> bn.replace(a, np.nan, 0)
>>> a
array([ 1.,  2.,  0.])
```

2.3 Non-reduce with axis

Functions that do not reduce the input array but operate along a specified axis.

---

bottleneck.**rankdata**(*a*, *axis=None*)

Ranks the data, dealing with ties appropriately.

Equal values are assigned a rank that is the average of the ranks that would have been otherwise assigned to all of the values within that set. Ranks begin at 1, not 0.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the elements of the array are ranked. The default (axis=None) is to rank the elements of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*. The dtype is 'float64'.

See also:

bottleneck.nanrankdata Ranks the data dealing with ties and NaNs.

Examples

```
>>> bn.rankdata([0, 2, 2, 3])
array([ 1. ,  2.5,  2.5,  4. ])
>>> bn.rankdata([[0, 2], [2, 3]])
array([ 1. ,  2.5,  2.5,  4. ])
>>> bn.rankdata([[0, 2], [2, 3]], axis=0)
array([[ 1.,  1.],
       [ 2.,  2.]])
>>> bn.rankdata([[0, 2], [2, 3]], axis=1)
array([[ 1.,  2.],
       [ 1.,  2.]])
```

---

bottleneck.**nanrankdata**(*a*, *axis=None*)

Ranks the data, dealing with ties and NaNs appropriately.

Equal values are assigned a rank that is the average of the ranks that would have been otherwise assigned to all of the values within that set. Ranks begin at 1, not 0.

NaNs in the input array are returned as NaNs.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

axis [{int, None}, optional] Axis along which the elements of the array are ranked. The default (axis=None) is to rank the elements of the flattened array.

Returns

y [ndarray] An array with the same shape as *a*. The dtype is 'float64'.

See also:

bottleneck.rankdata Ranks the data, dealing with ties and appropriately.

Examples

```
>>> bn.nanrankdata([np.nan, 2, 2, 3])
array([ nan,  1.5,  1.5,  3. ])
>>> bn.nanrankdata([[np.nan, 2], [2, 3]])
array([ nan,  1.5,  1.5,  3. ])
>>> bn.nanrankdata([[np.nan, 2], [2, 3]], axis=0)
array([[ nan,   1.],
       [  1.,   2.]])
>>> bn.nanrankdata([[np.nan, 2], [2, 3]], axis=1)
array([[ nan,   1.],
       [  1.,   2.]])
```

---

bottleneck.**partition**(*a*, *kth*, *axis=-1*)

Partition array elements along given axis.

A 1d array B is partitioned at array index *kth* if three conditions are met: (1) B[kth] is in its sorted position, (2) all elements to the left of *kth* are less than or equal to B[kth], and (3) all elements to the right of *kth* are greater than or equal to B[kth]. Note that the array elements in conditions (2) and (3) are in general unordered.

Shuffling the input array may change the output. The only guarantee is given by the three conditions above.

This functions is not protected against NaN. Therefore, you may get unexpected results if the input contains NaN.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

kth [int] The value of the element at index *kth* will be in its sorted position. Smaller (larger) or equal values will be to the left (right) of index *kth*.

axis [{int, None}, optional] Axis along which the partition is performed. The default (axis=-1) is to partition along the last axis.

Returns

y [ndarray] A partitioned copy of the input array with the same shape and type of *a*.

See also:

bottleneck.argpartition Indices that would partition an array

Notes

Unexpected results may occur if the input array contains NaN.

Examples

Create a numpy array:

```
>>> a = np.array([1, 0, 3, 4, 2])
```

Partition array so that the first 3 elements (indices 0, 1, 2) are the smallest 3 elements (note, as in this example, that the smallest 3 elements may not be sorted):

```
>>> bn.partition(a, kth=2)
array([1, 0, 2, 4, 3])
```

Now Partition array so that the last 2 elements are the largest 2 elements:

```
>>> bn.partition(a, kth=3)
array([1, 0, 2, 3, 4])
```

---

bottleneck.**argpartition**(*a*, *kth*, *axis=-1*)

Return indices that would partition array along the given axis.

A 1d array B is partitioned at array index *kth* if three conditions are met: (1) B[kth] is in its sorted position, (2) all elements to the left of *kth* are less than or equal to B[kth], and (3) all elements to the right of *kth* are greater than or equal to B[kth]. Note that the array elements in conditions (2) and (3) are in general unordered.

Shuffling the input array may change the output. The only guarantee is given by the three conditions above.

This functions is not protected against NaN. Therefore, you may get unexpected results if the input contains NaN.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

kth [int] The value of the element at index *kth* will be in its sorted position. Smaller (larger) or equal values will be to the left (right) of index *kth*.

axis [{int, None}, optional] Axis along which the partition is performed. The default (axis=-1) is to partition along the last axis.

Returns

y [ndarray] An array the same shape as the input array containing the indices that partition *a*. The dtype of the indices is numpy.intp.

See also:

bottleneck.partition Partition array elements along given axis.

Notes

Unexpected results may occur if the input array contains NaN.

Examples

Create a numpy array:

```
>>> a = np.array([10, 0, 30, 40, 20])
```

Find the indices that partition the array so that the first 3 elements are the smallest 3 elements:

```
>>> index = bn.argpartition(a, kth=2)
>>> index
array([0, 1, 4, 3, 2])
```

Let's use the indices to partition the array (note, as in this example, that the smallest 3 elements may not be in order):

```
>>> a[index]
array([10, 0, 20, 40, 30])
```

---

bottleneck.**push**(*a*, *n=None*, *axis=-1*)

Fill missing values (NaNs) with most recent non-missing values.

Filling proceeds along the specified axis from small index values to large index values.

Parameters

a [array_like] Input array. If *a* is not an array, a conversion is attempted.

n [{int, None}, optional] How far to push values. If the most recent non-NaN array element is more than *n* index positions away, than a NaN is returned. The default (n = None) is to push the entire length of the slice. If *n* is an integer it must be nonnegative.

axis [int, optional] Axis along which the elements of the array are pushed. The default (axis=-1) is to push along the last axis of the input array.

Returns

y [ndarray] An array with the same shape and dtype as *a*.

See also:

bottleneck.replace Replace specified value of an array with new value.

Examples

```
>>> a = np.array([5, np.nan, np.nan, 6, np.nan])
>>> bn.push(a)
    array([ 5.,  5.,  5.,  6.,  6.])
>>> bn.push(a, n=1)
    array([  5.,   5.,  nan,   6.,   6.])
>>> bn.push(a, n=2)
    array([ 5.,  5.,  5.,  6.,  6.])
```

2.4 Moving window functions

Functions that operate along a (1d) moving window.

---

bottleneck.**move_sum**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window sum along the specified axis, optionally ignoring NaNs.

This function cannot handle input arrays that contain Inf. When the window contains Inf, the output will correctly be Inf. However, when Inf moves out of the window, the remaining output values in the slice will incorrectly be NaN.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving sum of the input array along the specified axis. The output has the same shape as the input.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_sum(a, window=2)
array([ nan,   3.,   5.,  nan,  nan])
>>> bn.move_sum(a, window=2, min_count=1)
array([ 1.,  3.,  5.,  3.,  5.])
```

---

bottleneck.**move_mean**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window mean along the specified axis, optionally ignoring NaNs.

This function cannot handle input arrays that contain Inf. When the window contains Inf, the output will correctly be Inf. However, when Inf moves out of the window, the remaining output values in the slice will incorrectly be NaN.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving mean of the input array along the specified axis. The output has the same shape as the input.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_mean(a, window=2)
array([ nan,  1.5,  2.5,  nan,  nan])
>>> bn.move_mean(a, window=2, min_count=1)
array([ 1. ,  1.5,  2.5,  3. ,  5. ])
```

---

bottleneck.**move_std**(*a*, *window*, *min_count=None*, *axis=-1*, *ddof=0*)

Moving window standard deviation along the specified axis, optionally ignoring NaNs.

This function cannot handle input arrays that contain Inf. When Inf enters the moving window, the outout becomes NaN and will continue to be NaN for the remainer of the slice.

Unlike bn.nanstd, which uses a two-pass algorithm, move_nanstd uses a one-pass algorithm called Welford's method. The algorithm is slow but numerically stable for cases where the mean is large compared to the standard deviation.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

Returns

y [ndarray] The moving standard deviation of the input array along the specified axis. The output has the same shape as the input.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_std(a, window=2)
array([ nan,  0.5,  0.5,  nan,  nan])
>>> bn.move_std(a, window=2, min_count=1)
array([ 0. ,  0.5,  0.5,  0. ,  0. ])
```

---

bottleneck.**move_var**(*a*, *window*, *min_count=None*, *axis=-1*, *ddof=0*)

Moving window variance along the specified axis, optionally ignoring NaNs.

This function cannot handle input arrays that contain Inf. When Inf enters the moving window, the outout becomes NaN and will continue to be NaN for the remainer of the slice.

Unlike bn.nanvar, which uses a two-pass algorithm, move_nanvar uses a one-pass algorithm called Welford's method. The algorithm is slow but numerically stable for cases where the mean is large compared to the standard deviation.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

Returns

y [ndarray] The moving variance of the input array along the specified axis. The output has the same shape as the input.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_var(a, window=2)
array([ nan,  0.25,  0.25,  nan,  nan])
>>> bn.move_var(a, window=2, min_count=1)
array([ 0. ,  0.25,  0.25,  0. ,  0. ])
```

`bottleneck.`**`move_min`**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window minimum along the specified axis, optionally ignoring NaNs.

float64 output is returned for all input data types.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving minimum of the input array along the specified axis. The output has the same shape as the input. The dtype of the output is always float64.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_min(a, window=2)
array([ nan,  1.,  2.,  nan,  nan])
>>> bn.move_min(a, window=2, min_count=1)
array([ 1.,  1.,  2.,  3.,  5.])
```

`bottleneck.`**`move_max`**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window maximum along the specified axis, optionally ignoring NaNs.

float64 output is returned for all input data types.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving maximum of the input array along the specified axis. The output has the same shape as the input. The dtype of the output is always float64.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, np.nan, 5.0])
>>> bn.move_max(a, window=2)
array([ nan,  2.,  3.,  nan,  nan])
>>> bn.move_max(a, window=2, min_count=1)
array([ 1.,  2.,  3.,  3.,  5.])
```

---

bottleneck.**move_argmin**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window index of minimum along the specified axis, optionally ignoring NaNs.

Index 0 is at the rightmost edge of the window. For example, if the array is monotonically decreasing (increasing) along the specified axis then the output array will contain zeros (window-1).

If there is a tie in input values within a window, then the rightmost index is returned.

float64 output is returned for all input data types.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving index of minimum values of the input array along the specified axis. The output has the same shape as the input. The dtype of the output is always float64.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
>>> bn.move_argmin(a, window=2)
array([ nan,  1.,  1.,  1.,  1.])
```

```
>>> a = np.array([5.0, 4.0, 3.0, 2.0, 1.0])
>>> bn.move_argmin(a, window=2)
array([ nan,  0.,  0.,  0.,  0.])
```

```
>>> a = np.array([2.0, 3.0, 4.0, 1.0, 7.0, 5.0, 6.0])
>>> bn.move_argmin(a, window=3)
array([ nan,  nan,   2.,   0.,   1.,   2.,   1.])
```

---

bottleneck.**move_argmax**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window index of maximum along the specified axis, optionally ignoring NaNs.

Index 0 is at the rightmost edge of the window. For example, if the array is monotonically increasing (decreasing) along the specified axis then the output array will contain zeros (window-1).

If there is a tie in input values within a window, then the rightmost index is returned.

float64 output is returned for all input data types.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving index of maximum values of the input array along the specified axis. The output has the same shape as the input. The dtype of the output is always float64.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
>>> bn.move_argmax(a, window=2)
array([ nan,   0.,   0.,   0.,   0.])
```

```
>>> a = np.array([5.0, 4.0, 3.0, 2.0, 1.0])
>>> bn.move_argmax(a, window=2)
array([ nan,   1.,   1.,   1.,   1.])
```

```
>>> a = np.array([2.0, 3.0, 4.0, 1.0, 7.0, 5.0, 6.0])
>>> bn.move_argmax(a, window=3)
array([ nan,  nan,   0.,   1.,   0.,   1.,   2.])
```

---

bottleneck.**move_median**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window median along the specified axis, optionally ignoring NaNs.

float64 output is returned for all input data types.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving median of the input array along the specified axis. The output has the same shape as the input.

Examples

```
>>> a = np.array([1.0, 2.0, 3.0, 4.0])
>>> bn.move_median(a, window=2)
array([ nan,  1.5,  2.5,  3.5])
>>> bn.move_median(a, window=2, min_count=1)
array([ 1. ,  1.5,  2.5,  3.5])
```

---

bottleneck.**move_rank**(*a*, *window*, *min_count=None*, *axis=-1*)

Moving window ranking along the specified axis, optionally ignoring NaNs.

The output is normalized to be between -1 and 1. For example, with a window width of 3 (and with no ties), the possible output values are -1, 0, 1.

Ties are broken by averaging the rankings. See the examples below.

The runtime depends almost linearly on *window*. The more NaNs there are in the input array, the shorter the runtime.

Parameters

a [ndarray] Input array. If *a* is not an array, a conversion is attempted.

window [int] The number of elements in the moving window.

min_count: {int, None}, optional If the number of non-NaN values in a window is less than *min_count*, then a value of NaN is assigned to the window. By default *min_count* is None, which is equivalent to setting *min_count* equal to *window*.

axis [int, optional] The axis over which the window is moved. By default the last axis (axis=-1) is used. An axis of None is not allowed.

Returns

y [ndarray] The moving ranking along the specified axis. The output has the same shape as the input. For integer input arrays, the dtype of the output is float64.

Examples

With window=3 and no ties, there are 3 possible output values, i.e. [-1., 0., 1.]:

```
>>> a = np.array([1, 2, 3, 9, 8, 7, 5, 6, 4])
>>> bn.move_rank(a, window=3)
    array([ nan,  nan,   1.,   1.,   0.,  -1.,  -1.,   0.,  -1.])
```

Ties are broken by averaging the rankings of the tied elements:

```
>>> a = np.array([1, 2, 3, 3, 3, 4])
>>> bn.move_rank(a, window=3)
    array([ nan,  nan,  1. ,  0.5,  0. ,  1. ])
```

In an increasing sequence, the moving window ranking is always equal to 1:

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> bn.move_rank(a, window=2)
    array([ nan,   1.,   1.,   1.,   1.])
```

CHAPTER

THREE

RELEASE NOTES

These are the major changes made in each release. For details of the changes see the commit log at https://github.com/pydata/bottleneck

3.1 Bottleneck 1.3.0

Release date: 2019-11-12

3.1.1 Project Updates

- Bottleneck has a new maintainer, Christopher Whelan (`@qwhelan` on GitHub).
- Documentation now hosted at https://bottleneck.readthedocs.io
- 1.3.x will be the last release to support Python 2.7
- Bottleneck now supports and is tested against Python 3.7 and 3.8. (#211, #268)
- The `LICENSE` file has been restructured to only include the license for the Bottleneck project to aid license audit tools. There has been no change to the licensing of Bottleneck.
  - Licenses for other projects incorporated by Bottleneck are now reproduced in full in separate files in the `LICENSES/` directory (eg, `LICENSES/NUMPY_LICENSE`)
  - All licenses have been updated. Notably, setuptools is now MIT licensed and no longer under the ambiguous dual PSF/Zope license.
- Bottleneck now uses **PEP 518** for specifying build dependencies, with per Python version specifications (#247)

3.1.2 Enhancements

- Remove `numpydoc` package from Bottleneck source distribution
- `bottleneck.slow.reduce.nansum()` and `bottleneck.slow.reduce.ss()` now longer coerce output to have the same dtype as input
- Test (tox, travis, appveyor) against latest `numpy` (in conda)
- Performance benchmarking also available via `asv`
- `versioneer` now used for versioning (#213)
- Test suite now uses `pytest` as `nose` is deprecated (#222)
- `python setup.py build_ext --inplace` is now incremental (#224)

- `python setup.py clean` now cleans all artifacts (#226)
- Compiler feature support now identified by testing rather than hardcoding (#227)
- The `BN_OPT_3` macro allows selective use of `-O3` at the function level (#223)
- Contributors are now automatically cited in the release notes (#244)

3.1.3 Performance

- Speed up `bottleneck.reduce.anynan()` and `bottleneck.reduce.allnan()` by 2x via `BN_OPT_3` (#223)
- All functions covered by `asv` benchmarks
- `bottleneck.nonreduce.replace()` speedup of 4x via more explicit typing (#239)
- `bottleneck.reduce.median()` up to 2x faster for Fortran-ordered arrays (#248)

3.1.4 Bug Fixes

- Documentation fails to build on Python 3 (#170)
- `bottleneck.benchmark.bench()` crashes on python 3.6.3, numpy 1.13.3 (#175)
- `bottleneck.nonreduce_axis.push()` raises when `n=None` is explicitly passed (#178)
- `bottleneck.reduce.nansum()` wrong output when `a = np.ones((2, 2))[..., np.newaxis]` same issue of other reduce functions (#183)
- Silenced FutureWarning from NumPy in the slow version of move functions (#194)
- Installing bottleneck onto a system that does not already have Numpy (#195)
- Memory leaked when input was not a NumPy array (#201)
- Tautological comparison in `bottleneck.move.move_rank()` removed (#207, #212)

3.1.5 Cleanup

- The `ez_setup.py` module is no longer packaged (#211)
- Building documentation is now self-contained in `make doc` (#214)
- Codebase now `flake8` compliant and run on every commit
- Codebase now uses `black` for autoformatting (#253)

3.1.6 Contributors

A total of 9 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Ales Erjavec +
- Christoph Gohlke
- Christopher Whelan +
- Daniel Hakimi +

- Ghislain Antony Vaillant +
- Keith Goodman
- Stephan Hoyer
- Thomas Robitaille +
- kwgoodman

3.1.7 Older Releases

Bottleneck 1.2.1

Release date: 2017-05-15

This release adds support for NumPy's relaxed strides checking and fixes a few bugs.

Bug Fixes

- Installing bottleneck when two versions of NumPy are present (#156)
- Compiling on Ubuntu 14.04 inside a Windows 7 WMware (#157)
- Occasional segmentation fault in `bn.nanargmin()`, `nanargmax()`, `median()`, and `nanmedian()` when all of the following conditions are met: axis is None, input array is 2d or greater, and input array is not C contiguous. (#159)
- Reducing np.array([2**31], dtype=np.int64) overflows on Windows (#163)

Contributors

A total of 1 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Keith Goodman

Bottleneck 1.2.0

Release date: 2016-10-20

This release is a complete rewrite of Bottleneck.

Port to C

- Bottleneck is now written in C
- Cython is no longer a dependency
- Source tarball size reduced by 80%
- Build time reduced by 66%
- Install size reduced by 45%

Redesign

- Besides porting to C, much of bottleneck has been redesigned to be simpler and faster. For example, bottleneck now uses its own N-dimensional array iterators, reducing function call overhead.

New features

- The new function bench_detailed runs a detailed performance benchmark on a single bottleneck function.

- Bottleneck can be installed on systems that do not yet have NumPy installed. Previously that only worked on some systems.

Beware

- Functions partsort and argpartsort have been renamed to partition and argpartition to match NumPy. Additionally the meaning of the input arguments have changed: `bn.partsort(a, n)()` is now equivalent to `bn.partition(a, kth=n-1)()`. Similarly for bn.argpartition.
- The keyword for array input has been changed from *arr* to *a* in all functions. It now matches NumPy.

Thanks

- Moritz E. Beber: continuous integration with AppVeyor
- Christoph Gohlke: Windows compatibility
- Jennifer Olsen: comments and suggestions
- A special thanks to the Cython developers. The quickest way to appreciate their work is to remove Cython from your project. It is not easy.

Contributors

A total of 3 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Keith Goodman
- Moritz E. Beber +
- kwgoodman

Bottleneck 1.1.0

Release date: 2016-06-22

This release makes Bottleneck more robust, releases GIL, adds new functions.

More Robust

- `bn.move_median()` can now handle NaNs and *min_count* parameter
- `bn.move_std()` is slower but numerically more stable
- Bottleneck no longer crashes on byte-swapped input arrays

Faster

- All Bottleneck functions release the GIL
- median is faster if the input array contains NaN
- move_median is faster for input arrays that contain lots of NaNs
- No speed penalty for median, nanmedian, nanargmin, nanargmax for Fortran ordered input arrays when axis is None
- Function call overhead cut in half for reduction along all axes (axis=None) if the input array satisfies at least one of the following properties: 1d, C contiguous, F contiguous
- Reduction along all axes (axis=None) is more than twice as fast for long, narrow input arrays such as a (1000000, 2) C contiguous array and a (2, 1000000) F contiguous array

New Functions

- move_var

- move_argmin
- move_argmax
- move_rank
- push

Beware

- `bn.median()` now returns NaN for a slice that contains one or more NaNs
- Instead of using the distutils default, the '-O2' C compiler flag is forced
- `bn.move_std()` output changed when mean is large compared to standard deviation
- Fixed: Non-accelerated moving window functions used min_count incorrectly
- `bn.move_median()` is a bit slower for float input arrays that do not contain NaN

Thanks

Alphabeticaly by last name

- Alessandro Amici worked on setup.py
- Pietro Battiston modernized bottleneck installation
- Moritz E. Beber set up continuous integration with Travis CI
- Jaime Frio improved the numerical stability of move_std
- Christoph Gohlke revived Windows compatibility
- Jennifer Olsen added NaN support to move_median

Contributors

A total of 10 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Alessandro Amici +
- Christoph Gohlke
- Jaime Fernandez +
- Jenn Olsen +
- Keith Goodman
- Midnighter +
- Pietro Battiston +
- jaimefrio +
- jennolsen84 +
- kwgoodman

Bottleneck 1.0.0

Release date: 2015-02-06

This release is a complete rewrite of Bottleneck.

Faster

- "python setup.py build" is 18.7 times faster
- Function-call overhead cut in half—a big speed up for small input arrays
- Arbitrary ndim input arrays accelerated; previously only 1d, 2d, and 3d
- bn.nanrankdata is twice as fast for float input arrays
- bn.move_max, bn.move_min are faster for int input arrays
- No speed penalty for reducing along all axes when input is Fortran ordered

Smaller

- Compiled binaries 14.1 times smaller
- Source tarball 4.7 times smaller
- 9.8 times less C code
- 4.3 times less Cython code
- 3.7 times less Python code

Beware

- Requires numpy 1.9.1
- Single API, e.g.: bn.nansum instead of bn.nansum and nansum_2d_float64_axis0
- On 64-bit systems bn.nansum(int32) returns int32 instead of int64
- bn.nansum now returns 0 for all NaN slices (as does numpy 1.9.1)
- Reducing over all axes returns, e.g., 6.0; previously np.float64(6.0)
- bn.ss() now has default axis=None instead of axis=0
- bn.nn() is no longer in bottleneck

min_count

- Previous releases had moving window function pairs: move_sum, move_nansum
- This release only has half of the pairs: move_sum
- Instead a new input parameter, min_count, has been added
- min_count=None same as old move_sum; min_count=1 same as old move_nansum
- If # non-NaN values in window < min_count, then NaN assigned to the window
- Exception: move_median does not take min_count as input

Bug Fixes

- Can now install bottleneck with pip even if numpy is not already installed
- bn.move_max, bn.move_min now return float32 for float32 input

Contributors

A total of 4 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Keith Goodman
- Lev Givon +
- Stephan Hoyer

- kwgoodman

Bottleneck 0.8.0

Release date: 2014-01-21

This version of Bottleneck requires NumPy 1.8.

Breaks from 0.7.0

- This version of Bottleneck requires NumPy 1.8
- nanargmin and nanargmax behave like the corresponding functions in NumPy 1.8

Bug fixes

- nanargmax/nanargmin wrong for redundant max/min values in 1d int arrays

Contributors

A total of 4 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Christoph Gohlke +
- Keith Goodman
- Stephan Hoyer +
- kwgoodman

Bottleneck 0.7.0

Release date: 2013-09-10

Enhancements

- bn.rankdata() is twice as fast (with input a = np.random.rand(1000000))
- C files now included in github repo; cython not needed to try latest
- C files are now generated with Cython 0.19.1 instead of 0.16
- Test bottleneck across multiple python/numpy versions using tox
- Source tarball size cut in half

Bug fixes

- move_std, move_nanstd return inappropriate NaNs (sqrt of negative #) (#50)
- *make test* fails on some computers (#52)
- scipy optional yet some unit tests depend on scipy (#57)
- now works on Mac OS X 10.8 using clang compiler (#49, #55)
- nanstd([1.0], ddof=1) and nanvar([1.0], ddof=1) crash (#60)

Contributors

A total of 5 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Jens Hedegaard Nielsen +

- John Benediktsson +
- Keith Goodman
- jmcloughlin +
- kwgoodman

Bottleneck 0.6.0

Release date: 2012-06-04

Thanks to Dougal Sutherland, Bottleneck now runs on Python 3.2.

New functions

- replace(arr, old, new), e.g, replace(arr, np.nan, 0)
- nn(arr, arr0, axis) nearest neighbor and its index of 1d arr0 in 2d arr
- anynan(arr, axis) faster alternative to np.isnan(arr).any(axis)
- allnan(arr, axis) faster alternative to np.isnan(arr).all(axis)

Enhancements

- Python 3.2 support (may work on earlier versions of Python 3)
- C files are now generated with Cython 0.16 instead of 0.14.1
- Upgrade numpydoc from 0.3.1 to 0.4 to support Sphinx 1.0.1

Breaks from 0.5.0

- Support for Python 2.5 dropped
- Default axis for benchmark suite is now axis=1 (was 0)

Bug fixes

- Confusing error message in partsort and argpartsort (#31)
- Update path in MANIFEST.in (#32)
- Wrong output for very large (2**31) input arrays (#35)

Contributors

A total of 4 people contributed patches to this release. People with a "+" by their names contributed a patch for the first time.

- Ben Root +
- Dougal Sutherland +
- Keith Goodman
- kwgoodman +

Bottleneck 0.5.0

Release date: 2011-06-13

The fifth release of bottleneck adds four new functions, comes in a single source distribution instead of separate 32 and 64 bit versions, and contains bug fixes.

J. David Lee wrote the C-code implementation of the double heap moving window median.

New functions

- move_median(), moving window median
- partsort(), partial sort
- argpartsort()
- ss(), sum of squares, faster version of scipy.stats.ss

Changes

- Single source distribution instead of separate 32 and 64 bit versions
- nanmax and nanmin now follow Numpy 1.6 (not 1.5.1) when input is all NaN

Bug fixes

- Support python 2.5 by importing *with* statement (#14)
- nanmedian wrong for particular ordering of NaN and non-NaN elements (#22)
- argpartsort, nanargmin, nanargmax returned wrong dtype on 64-bit Windows (#26)
- rankdata and nanrankdata crashed on 64-bit Windows (#29)

Bottleneck 0.4.3

Release date: 2011-03-17

This is a bug fix release.

Bug fixes

- median and nanmedian modified (partial sort) input array (#11)
- nanmedian wrong when odd number of elements with all but last a NaN (#12)

Enhancement

- Lazy import of SciPy (rarely used) speeds Bottleneck import 3x

Bottleneck 0.4.2

Release date: 2011-03-08

This is a bug fix release.

Same bug fixed in Bottleneck 0.4.1 for nanstd() was fixed for nanvar() in this release. Thanks again to Christoph Gohlke for finding the bug.

Bottleneck 0.4.1

Release date: 2011-03-08

This is a bug fix release.

The low-level functions nanstd_3d_int32_axis1 and nanstd_3d_int64_axis1, called by bottleneck.nanstd(), wrote beyond the memory owned by the output array if arr.shape[1] == 0 and arr.shape[0] > arr.shape[2], where arr is the input array.

Thanks to Christoph Gohlke for finding an example to demonstrate the bug.

Bottleneck 0.4.0

Release date: 2011-03-08

The fourth release of Bottleneck contains new functions and bug fixes. Separate source code distributions are now made for 32 bit and 64 bit operating systems.

New functions

- rankdata()
- nanrankdata()

Enhancements

- Optionally specify the shapes of the arrays used in benchmark
- Can specify which input arrays to fill with one-third NaNs in benchmark

Breaks from 0.3.0

- Removed group_nanmean() function
- Bump dependency from NumPy 1.4.1 to NumPy 1.5.1
- C files are now generated with Cython 0.14.1 instead of 0.13

Bug fixes

- Some functions gave wrong output dtype for some input dtypes on 32 bit OS (#6)
- Some functions choked on size zero input arrays (#7)
- Segmentation fault with Cython 0.14.1 (but not 0.13) (#8)

Bottleneck 0.3.0

Release date: 2010-01-19

The third release of Bottleneck is twice as fast for small input arrays and contains 10 new functions.

Faster

- All functions are faster (less overhead in selector functions)

New functions

- nansum()
- move_sum()
- move_nansum()
- move_mean()
- move_std()
- move_nanstd()
- move_min()
- move_nanmin()
- move_max()
- move_nanmax()

Enhancements

- You can now specify the dtype and axis to use in the benchmark timings
- Improved documentation and more unit tests

Breaks from 0.2.0

- Moving window functions now default to axis=-1 instead of axis=0
- Low-level moving window selector functions no longer take window as input

Bug fix

- int input array resulted in call to slow, non-cython version of move_nanmean

Bottleneck 0.2.0

Release date: 2010-12-27

The second release of Bottleneck is faster, contains more functions, and supports more dtypes.

Faster

- All functions faster (less overhead) when output is not a scalar
- Faster nanmean() for 2d, 3d arrays containing NaNs when axis is not None

New functions

- nanargmin()
- nanargmax()
- nanmedian()

Enhancements

- Added support for float32
- Fallback to slower, non-Cython functions for unaccelerated ndim/dtype
- Scipy is no longer a dependency
- Added support for older versions of NumPy (1.4.1)
- All functions are now templated for dtype and axis
- Added a sandbox for prototyping of new Bottleneck functions
- Rewrote benchmarking code

Bottleneck 0.1.0

Release date: 2010-12-10

Initial release. The three categories of Bottleneck functions:

- Faster replacement for NumPy and SciPy functions
- Moving window functions
- Group functions that bin calculations by like-labeled elements

CHAPTER

FOUR

LICENSES

Bottleneck is distributed under a Simplified BSD license. Parts of NumPy and SciPy, which have BSD licenses, are included in Bottleneck. The setuptools project has a MIT license and is used for configuration and installation.

4.1 Bottleneck License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

4.2 Other licenses

4.2.1 NumPy License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- **Redistributions of source code must retain the above copyright** notice, this list of conditions and the following disclaimer.
- **Redistributions in binary form must reproduce the above** copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- **Neither the name of the NumPy Developers nor the names of any** contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

4.2.2 SciPy License

Copyright (c) 2001-2002 Enthought, Inc. 2003-2019, SciPy Developers. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

4.2.3 Setuptools License

Copyright (C) 2016 Jason R Coombs <jaraco@jaraco.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX