Bohrium Documentation

Release 0.11.0

eScience Group @ NBI

Nov 16, 2020

Contents

Bohrium provides automatic acceleration of array operations in Python/NumPy, C, and C++ targeting multi-core CPUs and GP-GPUs. Forget handcrafting CUDA/OpenCL to utilize your GPU and forget threading, mutexes and locks to utilize your multi-core CPU, just use Bohrium!

CHAPTER 1

Features

	Architecture Support		Frontends			
	Multi-Core CPU	Many-Core GPU	Python2/NumPy	Python3/NumPy	C	C++
Linux	✓	✓	✓	✓	✓	✓
Mac OS	✓	✓	✓		✓	✓

- **Lazy Evaluation**, Bohrium will lazy evaluate all Python/NumPy operations until it encounters a "Python Read" such a printing an array or having a if-statement testing the value of an array.
- **Views** Bohrium supports NumPy views fully thus operating on array slices does not involve data copying.
- **Loop Fusion**, Bohrium uses a fusion algorithm that fuses (or merges) array operations into the same computation kernel that are then JIT-compiled and executed. However, Bohrium can only fuse operations that have some common sized dimension and no horizontal data conflicts.
- **Lazy CPU/GPU Communication**, Bohrium only moves data between the host and the GPU when the data is accessed directly by Python or a Python C-extension.
- **python -m bohrium**, automatically makes `import numpy` use Bohrium.
- Jupyter Support, you can use the magic command `%%bohrium` to automatically use Bohrium as NumPy.
- **Zero-copy interoperability with:**
  - NumPy
  - Cython
  - PyOpenCL
  - PyCUDA

Please note:

- Bohrium is a 64-bit project exclusively.
- Source code is available here: https://github.com/bh107/bohrium

CHAPTER 2

Get Started!

2.1 Installation

Bohrium supports Linux and Mac OS.

2.1.1 Linux

PyPI Package

If you use Bohrium through Python, we strongly recommend to install Bohrium through pypi, which will include BLAS, LAPACK, OpenCV, and OpenCL support:

```
pip install bohrium
# and / or
pip install bh107
```

Note: Bohrium requires `gcc` in `$PATH`.

Ubuntu

In order to install Bohrium on Ubuntu, you need to install the `python-pip` package **AND** its recommends:

```
apt install --install-recommends python-pip
```

Anaconda

To use Anaconda, simply install the Bohrium PyPI package in an environment:

```
# Activate the environment where you want to install Bohrium:
source activate my_env
# Install Bohrium using pip
pip install bohrium
```

Note: Bohrium requires `gcc` in `$PATH`. E.g. on Ubuntu install the build-essential package: `sudo apt install build-essential`.

Install From Source Package

Visit Bohrium on github.com and download the latest release: https://github.com/bh107/bohrium/releases/latest. Then build and install Bohrium as described in the following subsections.

Install dependencies, which on Ubuntu is:

```
sudo apt install build-essential python-pip python-virtualenv cmake git unzip␣
↪libboost-filesystem-dev libboost-serialization-dev libboost-regex-dev zlib1g-dev␣
↪libsigsegv-dev
```

And some additional packages for visualization:

```
sudo apt-get install freeglut3 freeglut3-dev libxmu-dev libxi-dev
```

Build and install:

```
wget https://github.com/bh107/bohrium/archive/master.zip
unzip master.zip
cd bohrium-master
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=<path to install directory>
make
make install
```

Note: The default install directory is `~/.local`

Note: To compile to a custom Python (with valgrind debug support for example), set `-DPYTHON_EXECUTABLE=<custom python binary>`.

Finally, you need to set the `LD_LIBRARY_PATH` environment variables and if you didn't install Bohrium in `$HOME/.local/lib` your need to set `PYTHONPATH` as well.

The `LD_LIBRARY_PATH` should include the path to the installation directory:

```
export LD_LIBRARY_PATH="<install dir>:$LD_LIBRARY_PATH"
```

The `PYTHONPATH` should include the path to the newly installed Bohrium Python module:

```
export PYTHONPATH="<install dir>/lib/python<python version>/site-packages:$PYTHONPATH"
```

Check Your Installation

Check installation by printing the current runtime stack:

```
python -m bohrium_api --info
```

2.1.2 Mac OS

The following explains how to get going on Mac OS.

You need to install the Xcode Developer Tools package, which is found in the App Store.

Note: You might have to manually install some extra header files by running `sudo installer -pkg /Library/Developer/CommandLineTools/Package/macOS_SDK_headers_for_macOS_10.14.pkg -target /` where `10.14` is your current version (more info).

PyPI Package

If you use Bohrium through Python, we strongly recommend to install Bohrium through pypi, which will include BLAS, LAPACK, OpenCV, and OpenCL support:

```
python -m pip install --user bohrium
# and / or
python -m pip install --user bh107
```

Note: If you get an error message saying that no package match your criteria it is properly because you are using a Python version for which **'no package exist https://pypi.org/project/bohrium-api/#files'_** . Please contact us and we will build a package using your specific Python version.

Install From Source Package

Start by installing Homebrew as explained on their website

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

Install dependencies:

```
brew install python
brew install cmake
brew install boost --with-icu4c
brew install libsigsegv
python3 -m pip install --user numpy cython twine gcc7
```

Visit Bohrium on github.com, download the latest release: https://github.com/bh107/bohrium/releases/latest or download *master*, and then build it:

```
wget https://github.com/bh107/bohrium/archive/master.zip
unzip master.zip
cd bohrium-master
mkdir build
cd build
export PATH="$(brew --prefix)/bin:/usr/local/opt/llvm/bin:/usr/local/opt/opencv3/bin:
↪$PATH"
export CC="clang"
export CXX="clang++"
export C_INCLUDE_PATH=$(llvm-config --includedir)
export CPLUS_INCLUDE_PATH=$(llvm-config --includedir)
export LIBRARY_PATH=$(llvm-config --libdir):$LIBRARY_PATH
cmake .. -DCMAKE_INSTALL_PREFIX=<path to install directory>
make
make install
```

Note: The default install directory is `~/.local`

Note: To compile to a custom Python (with valgrind debug support for example), set `-DPYTHON_EXECUTABLE=<custom python binary>`.

Finally, you need to set the `DYLD_LIBRARY_PATH` and `LIBRARY_PATH` environment variables and if you didn't install Bohrium in `$HOME/.local/lib` your need to set `PYTHONPATH` as well.

The `DYLD_LIBRARY_PATH` and `LIBRARY_PATH` should include the path to the installation directory:

```
export DYLD_LIBRARY_PATH="<install dir>:$DYLD_LIBRARY_PATH"
export LIBRARY_PATH="<install dir>:$LIBRARY_PATH"
```

The `PYTHONPATH` should include the path to the newly installed Bohrium Python module:

```
export PYTHONPATH="<install dir>/lib/python<python version>/site-packages:$PYTHONPATH"
```

Check Your Installation

Check installation by printing the current runtime stack:

```
python -m bohrium_api --info
```

2.1.3 Installation using Spack

This guide will install Bohrium using the Spack package manager.

Why use Spack?

Spack is a package management tool tailored specifically for supercomputers with a rather dated software stack. It allows to install and maintain packages, starting only from very few dependencies: Pretty much just python2.6, git, curl and some c++ compiler are all that's needed for the bootstrap.

Needless to say that the request for installing a particular package automatically yields the installation of all dependencies with exactly the right version and configurations. If this causes multiple versions/configurations of the same package to be required, this is no problem and gets resolved automatically, too. As a bonus on top, using an installed package later is super easy as well due to an automatic generation of module files, which set the required environment up.

Installation overview

First step is to clone and setup Spack:

```
export SPACK_ROOT="$PWD/spack"
git clone https://github.com/llnl/spack.git
. $SPACK_ROOT/share/spack/setup-env.sh
```

Afterwards the installation of Bohrium is instructed:

```
spack install bohrium
```

This step will take a while, since Spack will download the sources of all dependencies, unpack, configure and compile them. But since everything happens in the right order automatically, you could easily do this over night.

That's it. If you want to use Bohrium, setup up Spack as above, then load the required modules:

```
spack module loads -r bohrium > /tmp/bohrium.modules
. /tmp/bohrium.modules
```

and you are ready to go as the shell environment now contains all required variables (*LD_LIBRARY_PATH*, *PATH*, *CPATH*, *PYTHONPATH*, ...) to get going.

If you get some errors about the command *module* not being found, you need to install the Spack package *environment-modules* beforehand. Again, just a plain:

```
spack install environment-modules
```

is enough to achieve this.

Tuning the installation procedure

Spack offers countless ways to influence how things are installed and what is installed. See the Documentation and especially the Getting Started section for a good overview.

Most importantly the so-called *spec* allows to specify features or requirements with respect to versions and dependencies, that should be enabled or disabled when building the package. For example:

```
spec install bohrium~cuda~opencl
```

Will install Bohrium *without* CUDA or OpenCL support, which has a dramatic impact on the install time due to the reduced amount of dependencies to be installed. On the other hand:

```
spec install bohrium@develop
```

will install specifically the development version of Bohrium. This the current *HEAD* of the *master* branch in the github repository. One may also influence the versions of the dependencies by themselves. For example:

```
spec install bohrium+python^python@3:
```

will specifically compile Bohrium with a python version larger than 3.

The current list of features the Bohrium package has to offer can be listed by the command:

```
spack info bohrium
```

and the list of dependencies which will be installed by a particlar *spec* can be easily reviewed by something like:

```
spack spec bohrium@develop~cuda~opencl
```

2.2 User Guide

2.2.1 Python/NumPy

Three Python packages of Bohrium exist:

- **bohrium**: is a package that integrate into NumPy and accelerate NumPy operations seamlessly. Everything is completely automatic, which is great when it works but it also makes it hard to know why code does perform as expected.
- **bh107**: is a package that provide a similar interface and similar semantic as NumPy but everything is explicit. However, it is very easy to convert a **bh107** array into a NumPy array without any data copying.
- **bohrium_api**: as the name suggest, this packages implements the core Bohrium API, which **bohrium** and *bh107** uses. It is not targeting the end-user.

Bohrium (NumPy Integration)

Getting Started

Bohrium implements a new python module `bohrium` that introduces a new array class *bohrium._bh.ndarray()* which inherits from `numpy.ndarray()`. The two array classes are fully compatible thus you only has to replace `numpy.ndarray()` with *bohrium._bh.ndarray()* in order to utilize the Bohrium runtime system. Alternatively, in order to have Bohrium replacing NumPy automatically, you can use the `-m bohrium` argument when running Python:

```
$ python -m bohrium my_numpy_app.py
```

In order to choose which Bohrium backend to use, you can define the `BH_STACK` environment variable. Currently, three backends exist: `openmp`, `opencl`, and `cuda`.

Before using Bohrium, you can check the current runtime configuration using:

```
$ BH_STACK=opencl python -m bohrium --info

----
Bohrium version: 0.10.2.post8
----
Bohrium API version: 0.10.2.post8
Installed through PyPI: False
Config file: ~/.bohrium/config.ini
Header dir: ~/.local/lib/python3.7/site-packages/bohrium_api/include
Backend stack:
----
```

(continues on next page)

(continued from previous page)

```
OpenCL:
  Device[0]: AMD Accelerated Parallel Processing / Intel(R) Core(TM) i7-5600U CPU @ 2.
↪60GHz (OpenCL C 1.2 )
  Memory:         7676 MB
  Malloc cache limit: 767 MB (90%)
  Cache dir: "~/.local/var/bohrium/cache"
  Temp dir: "/tmp/bh_75cf_314f5"
  Codegen flags:
    Index-as-var: true
    Strides-as-var: true
    const-as-var: true
----
OpenMP:
  Main memory: 7676 MB
  Hardware threads: 4
  Malloc cache limit: 2190 MB (80% of unused memory)
  Cache dir: "~/.local/var/bohrium/cache"
  Temp dir: "/tmp/bh_75a5_c6368"
  Codegen flags:
    OpenMP: true
    OpenMP+SIMD: true
    Index-as-var: true
    Strides-as-var: true
    Const-as-var: true
  JIT Command: "/usr/bin/cc -x c -fPIC -shared  -std=gnu99  -O3 -march=native -Werror␣
↪-fopenmp -fopenmp-simd -I~/.local/share/bohrium/include {IN} -o {OUT}"
----
```

Notice, since `BH_STACK=opencl` is defined, the runtime stack consist of both the OpenCL and the OpenMP backend. In this case, OpenMP only handles operations unsupported by OpenCL.

Heat Equation Example

The following example is a heat-equation solver that uses Bohrium. Note that the only difference between Bohrium code and NumPy code is the first line where we import bohrium as np instead of numpy as np:

```
import bohrium as np
def heat2d(height, width, epsilon=42):
  G = np.zeros((height+2,width+2),dtype=np.float64)
  G[:,0]  = -273.15
  G[:,-1] = -273.15
  G[-1,:] = -273.15
  G[0,:]  = 40.0
  center = G[1:-1,1:-1]
  north  = G[:-2,1:-1]
  south  = G[2:,1:-1]
  east   = G[1:-1,:-2]
  west   = G[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.sum(np.abs(tmp-center))
    center[:] = tmp
  return center
heat2d(100, 100)
```

Alternatively, you can import Bohrium as NumPy through the command line argument `-m bohrium`:

```
$ python -m bohrium heat2d.py
```

In this case, all instances of `import numpy` is converted to `import bohrium` seamlessly. If you need to access the real numpy module use `import numpy_force`.

Acceleration

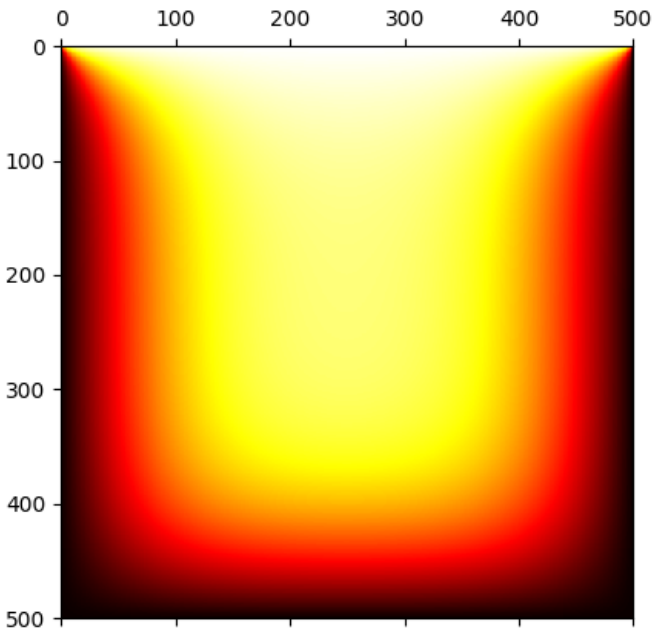
The approach of Bohrium is to accelerate all element-wise functions in NumPy (aka universal functions) as well as the reductions and accumulations of element-wise functions. This approach makes it possible to accelerate the heat-equation solver on both multi-core CPUs and GPUs.

Beside element-wise functions, Bohrium also accelerates a selection of common NumPy functions such as `dot()` and `solve()`. But the number of functions in NumPy and related projects such as SciPy is enormous thus we cannot hope to accelerate every single function in Bohrium. Instead, Bohrium will automatically convert `bohrium.ndarray` to `numpy.ndarray` when encountering a function that Bohrium cannot accelerate. When running on the CPU, this conversion is very cheap but when running on the GPU, this conversion requires the array data to be copied from the GPU to the CPU.

Matplotlib's `matshow()` function is example of a function Bohrium cannot accelerate. Say we want to visualize the result of the heat-equation solver, we could use `matshow()`:

```
from matplotlib import pyplot as plt

res = heat2d(100, 100)
plt.matshow(res, cmap='hot')
plt.show()
```



Beside producing the image (after approx. 1 min), the execution will raise a Python warning informing you that matplotlib function is handled like a regular NumPy:

```
/usr/lib/python2.7/site-packages/matplotlib/cbook.py:1506: RuntimeWarning:
Encountering an operation not supported by Bohrium. It will be handled by the␣
↪original NumPy.
x = np.array(x, subok=True, copy=copy)
```

Note: Increasing the problem size will improve the performance of Bohrium significantly!

Convert between Bohrium and NumPy

It is possible to convert between Bohrium and NumPy explicitly and thus avoid Python warnings. Let's walk through an example:

Create a new NumPy array with ones:

```
np_ary = numpy.ones(42)
```

Convert any type of array to Bohrium:

```
bh_ary = bohrium.array(np_ary)
```

Copy a bohrium array into a new NumPy array:

```
npy2 = bh_ary.copy2numpy()
```

Accelerate Loops

As we all know, having for and while loops in Python is bad for performance but is sometimes necessary. E.g. in the case of the `heat2d()` code, we have to evaluate `delta > epsilon` in order to know when to stop iterating. To address this issue, Bohrium introduces the function *bohrium.loop.do_while()*, which takes a function and calls it repeatedly until either a maximum number of calls has been reached or until the function return False.

The function signature:

```
def do_while(func, niters, *args, **kwargs):
    """Repeatedly calls the `func` with the `*args` and `**kwargs` as argument.

    The `func` is called while `func` returns True or None and the maximum number
    of iterations, `niters`, hasn't been reached.

    Parameters
    ----------
    func : function
        The function to run in each iterations. `func` can take any argument and may␣
↪return
        a boolean `bharray` with one element.
    niters: int or None
        Maximum number of iterations in the loop (number of times `func` is called).␣
↪If None, there is no maximum.
    *args, **kwargs : list and dict
        The arguments to `func`

    Notes
```

(continues on next page)

(continued from previous page)

```
    -----
    `func` can only use operations supported natively in Bohrium.
    """
```

An example where the function doesn't return anything:

```
>>> def loop_body(a):
...     a += 1
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, 5, a)
>>> a
array([5, 5, 5, 5])
```

An example where the function returns a `bharray` with one element and of type `bh.bool`:

```
>>> def loop_body(a):
...     a += 1
...     return bh.sum(a) < 10
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, None, a)
>>> a
array([3, 3, 3, 3])
```

Sliding Views Between Iterations

It can be useful to increase/decrease the beginning of certain array views between iterations of a loop. This can be achieved using *bohrium.loop.get_iterator()*, which returns a special bohrium iterator. The iterator can be given an optional start value (0 by default). The iterator is increased by one for each iteration, but can be changed increase or decrease by multiplying any constant (see example 2).

Iterators only supports addition, subtraction and multiplication. *bohrium.loop.get_iterator()* can only be used within Bohrium loops. Views using iterators cannot change shape between iterations. Therefore, views such as `a[i:2*i]` are not supported.

Example 1. Using iterators to create a loop-based function for calculating the triangular numbers (from 1 to 10). The loop in numpy looks the following:

```
>>> a = np.arange(1,11)
>>> for i in range(0,9):
...     a[i+1] += a[i]
>>> a
array([1 3 6 10 15 21 28 36 45 55])
```

The same can be written in Bohrium as:

```
>>> def loop_body(a):
...     i = get_iterator()
...     a[i+1] += a[i]
>>> a = bh.arange(1,11)
>>> bh.do_while(loop_body, 9, a)
>>> a
array([1 3 6 10 15 21 28 36 45 55])
```

Example 2. Increasing every second element by one, starting at both ends, in the same loop. As it can be seen: i is increased by 2, while j is descreased by 2 for each iteration:

```
>>> def loop_body(a):
...     i = get_iterator(1)
...     a[2*i] += a[2*(i-1)]
...     j = i+1
...     a[1-2*j] += a[1-2*(j-1)]
>>> a = bh.ones(10)
>>> bh.for_loop(loop_body, 4, a)
>>> a
array([1 5 2 4 3 3 4 2 5 1])
```

Nested loops is also available in *bohrium.loop.do_while()* by using grids. A grid is a set of iterators that depend on each other, just as with nested loops. A grid can have arbitrary size and is available via. the function *bohrium.loop.get_grid()*, which is only usable within a *bohrium.loop.do_while()* loop body. The function takes an amount of integers as parameters, corresponding to the range of the loops (from outer to inner). It returns the same amount of iterators, which functions as a grid. An example of this can be seen in Example 3 below. Example 3. Creating a range in an array with multiple dimensions. In Numpy it can be written as:

```
>>> a = bh.zeros((3,3))
>>> counter = bh.zeros(1)
>>> for i in range(3):
...     for j in range(3):
...         counter += 1
...         a[i,j] += counter
>>> a
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

The same can done within a `do_while` loop by using a grid:

```
>>> def kernel(a, counter):
...     i, j = get_grid(3,3)
...     counter += 1
...     a[i,j] += counter
>>> a = bh.zeros((3,3))
>>> counter = bh.zeros(1)
>>> bh.do_while(kernel, 3*3, a, counter)
>>> a
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

UserKernel

Bohrium supports user kernel, which makes it possible to implement a specialized handwritten kernel. The idea is that if you encounter a problem that you cannot implement using array programming and Bohrium cannot accelerate, you can write a kernel in C99 that calls other libraries or do the calculation itself.

OpenMP Example

In order to write and run your own kernel use *bohrium.user_kernel.execute()*:

```
import bohrium as bh

def fftn(ary):
    # Making sure that `ary` is complex, contiguous, and uses no offset
    ary = bh.user_kernel.make_behaving(ary, dtype=bh.complex128)
    res = bh.empty_like(a)

    # Indicates the direction of the transform you are interested in;
    # technically, it is the sign of the exponent in the transform.
    sign = ["FFTW_FORWARD", "FFTW_BACKWARD"]

    kernel = """
    #include <stdint.h>
    #include <stdlib.h>
    #include <complex.h>
    #include <fftw3.h>

    #if defined(_OPENMP)
        #include <omp.h>
    #else
        static inline int omp_get_max_threads() { return 1; }
        static inline int omp_get_thread_num()  { return 0; }
        static inline int omp_get_num_threads() { return 1; }
    #endif

    void execute(double complex *in, double complex *out) {
        const int ndim = %(ndim)d;
        const int shape[] = {%(shape)s};
        const int sign = %(sign)s;

        fftw_init_threads();
        fftw_plan_with_nthreads(omp_get_max_threads());

        fftw_plan p = fftw_plan_dft(ndim, shape, in, out, sign, FFTW_ESTIMATE);
        if(p == NULL) {
            printf("fftw plan fail!\\n");
            exit(-1);
        }
        fftw_execute(p);
        fftw_destroy_plan(p);
        fftw_cleanup_threads();
    }
    """ % {'ndim': a.ndim, 'shape': str(a.shape)[1:-1], 'sign': sign[0]}

    # Adding some extra link options to the compiler command
    cmd = bh.user_kernel.get_default_compiler_command() + " -lfftw3 -lfftw3_threads"
    bh.user_kernel.execute(kernel, [ary, res], compiler_command=cmd)
    return res
```

Two useful help functions when writing user kernels is *bohrium.user_kernel.make_behaving()*, which makes that an array is of a specific data type, is contiguous, and uses no offset and *bohrium.user_kernel.dtype_to_c99()*, which converts a Bohrium/NumPy array data type into a C99 data type.

OpenCL Example

In order to use the OpenCL backend, use the *tag* and *param* of *bohrium.user_kernel.execute()*:

```
import bohrium as bh

kernel = """
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

kernel void execute(global double *a, global double *b) {
    int i0 = get_global_id(0);
    int i1 = get_global_id(1);
    int gid = i0 * 5 + i1;
    b[gid] = a[gid] + gid;
}
"""
a = bh.ones(10*5, bh.double).reshape(10,5)
res = bh.empty_like(a)
# Notice, the OpenCL backend requires global_work_size and local_work_size
bh.user_kernel.execute(kernel, [a, res],
                       tag="opencl",
                       param={"global_work_size": [10, 5], "local_work_size": [1, 1]})
print(res)
```

Note: Remember to use the OpenCL backend by setting *BH_STACK=opencl*.

Interoperability

Bohrium is interoperable with other popular Python projects such as Cython and PyOpenCL. The idea is that if you encounter a problem that you cannot implement using array programming and Bohrium cannot accelerate, you can manually accelerate that problem using Cython or PyOpenCL.

NumPy

One example of such a problem is bincount() from NumPy. `bincount()` computes a histogram of an array, which isn't possible to implement efficiently through array programming. One approach is simply to use the implementation of NumPy:

```
import numpy
import bohrium

def bincount_numpy(ary):
    # Make a NumPy copy of the Bohrium array
    np_ary = ary.copy2numpy()
    # Let NumPy handle the calculation
    result = numpy.bincount(np_ary)
    # Copy the result back into a new Bohrium array
    return bohrium.array(result)
```

In this case, we use *bohrium._bh.ndarray.copy2numpy()* and *bohrium.array()* to copy the Bohrium to NumPy and back again.

Cython

In order to parallelize `bincount()` for a multi-core CPU, one can use Cython:

```
import numpy as np
import bohrium
import cython
from cython.parallel import prange, parallel
from libc.stdlib cimport abort, malloc, free
cimport numpy as cnp
cimport openmp
ctypedef cnp.uint64_t uint64

@cython.boundscheck(False) # turn off bounds-checking
@cython.cdivision(True) # turn off division-by-zero checking
cdef _count(uint64[:] x, uint64[:] out):
    cdef int num_threads, thds_id
    cdef uint64 i, start, end
    cdef uint64* local_histo

    with nogil, parallel():
        num_threads = openmp.omp_get_num_threads()
        thds_id = openmp.omp_get_thread_num()
        start = (x.shape[0] / num_threads) * thds_id
        if thds_id == num_threads-1:
            end = x.shape[0]
        else:
            end = start + (x.shape[0] / num_threads)

        if not(thds_id < num_threads-1 and x.shape[0] < num_threads):
            local_histo = <uint64 *> malloc(sizeof(uint64) * out.shape[0])
            if local_histo == NULL:
                abort()
            for i in range(out.shape[0]):
                local_histo[i] = 0

            for i in range(start, end):
                local_histo[x[i]] += 1

            with gil:
                for i in range(out.shape[0]):
                    out[i] += local_histo[i]
            free(local_histo)


def bincount_cython(x, minlength=None):
    # The output `ret` has the size of the max element plus one
    ret = bohrium.zeros(x.max()+1, dtype=x.dtype)

    # To reduce overhead, we use `interop_numpy.get_array()` instead of `copy2numpy()`
    # This approach means that `x_buf` and `ret_buf` points to the same memory as `x`␣
↪and `ret`.
    # Therefore, only change or deallocate `x` and `ret` when you are finished using␣
↪`x_buf` and `ret_buf`.
    x_buf = bohrium.interop_numpy.get_array(x)
    ret_buf = bohrium.interop_numpy.get_array(ret))

    # Now, we can run the Cython function
    _count(x_buf, ret_buf))

    # Since `ret_buf` points to the memory of `ret`, we can simply return `ret`.
    return ret
```

The function `_count()` is a regular Cython function that performs the histogram calculation. The function `bincount_cython()` uses *bohrium.interop_numpy.get_array()* to retrieve data pointers from the Bohrium arrays without any data copying.

Note: Changing or deallocating the Bohrium array given to *bohrium.interop_numpy.get_array()* invalidates the returned NumPy array!

PyOpenCL

In order to parallelize `bincount()` for a GPGPU, one can use PyOpenCL:

```
import bohrium
import pyopencl as cl

def bincount_pyopencl(x):
    # Check that PyOpenCL is installed and that the Bohrium runtime uses the OpenCL␣
↪backend
    if not interop_pyopencl.available():
        raise NotImplementedError("OpenCL not available")

    # Get the OpenCL context from Bohrium
    ctx = bohrium.interop_pyopencl.get_context()
    queue = cl.CommandQueue(ctx)

    x_max = int(x.max())

    # Check that the size of histogram doesn't exceeds the memory capacity of the GPU
    if x_max >= interop_pyopencl.max_local_memory(queue.device) // x.itemsize:
        raise NotImplementedError("OpenCL: max element is too large for the GPU")

    # Let's create the output array and retrieve the in-/output OpenCL buffers
    # NB: we always return uint32 array
    ret = bohrium.empty((x_max+1, ), dtype=np.uint32)
    x_buf = bohrium.interop_pyopencl.get_buffer(x)
    ret_buf = bohrium.interop_pyopencl.get_buffer(ret)

    # The OpenCL kernel is based on the book "OpenCL Programming Guide" by Aaftab␣
↪Munshi at al.
    source = """
    kernel void histogram_partial(
        global DTYPE *input,
        global uint *partial_histo,
        uint input_size
    ){
        int local_size = (int)get_local_size(0);
        int group_indx = get_group_id(0) * HISTO_SIZE;
        int gid = get_global_id(0);
        int tid = get_local_id(0);

        local uint tmp_histogram[HISTO_SIZE];

        int j = HISTO_SIZE;
        int indx = 0;

        // clear the local buffer that will generate the partial histogram
```

(continues on next page)

(continued from previous page)

```
        do {
            if (tid < j)
                tmp_histogram[indx+tid] = 0;
            j -= local_size;
            indx += local_size;
        } while (j > 0);

        barrier(CLK_LOCAL_MEM_FENCE);

        if (gid < input_size) {
            atomic_inc(&tmp_histogram[input[gid]]);
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        // copy the partial histogram to appropriate location in
        // histogram given by group_indx
        if (local_size >= HISTO_SIZE){
            if (tid < HISTO_SIZE)
                partial_histo[group_indx + tid] = tmp_histogram[tid];
        }else{
            j = HISTO_SIZE;
            indx = 0;
            do {
                if (tid < j)
                    partial_histo[group_indx + indx + tid] = tmp_histogram[indx +
↪tid];

                j -= local_size;
                indx += local_size;
            } while (j > 0);
        }
    }

    kernel void histogram_sum_partial_results(
        global uint *partial_histogram,
        int num_groups,
        global uint *histogram
    ){
        int gid = (int)get_global_id(0);
        int group_indx;
        int n = num_groups;
        local uint tmp_histogram[HISTO_SIZE];

        tmp_histogram[gid] = partial_histogram[gid];
        group_indx = HISTO_SIZE;
        while (--n > 0) {
            tmp_histogram[gid] += partial_histogram[group_indx + gid];
            group_indx += HISTO_SIZE;
        }
        histogram[gid] = tmp_histogram[gid];
    }
    """
    source = source.replace("HISTO_SIZE", "%d" % ret.shape[0])
    source = source.replace("DTYPE", interop_pyopencl.type_np2opencl_str(x.dtype))
    prg = cl.Program(ctx, source).build()
```

(continues on next page)

(continued from previous page)

```
    # Calculate sizes for the kernel execution
    local_size = interop_pyopencl.kernel_info(prg.histogram_partial, queue)[0]  # Max␣
↪work-group size
    num_groups = int(math.ceil(x.shape[0] / float(local_size)))
    global_size = local_size * num_groups

    # First we compute the partial histograms
    partial_res_g = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, num_groups * ret.nbytes)
    prg.histogram_partial(queue, (global_size,), (local_size,), x_buf, partial_res_g,␣
↪np.uint32(x.shape[0]))

    # Then we sum the partial histograms into the final histogram
    prg.histogram_sum_partial_results(queue, ret.shape, None, partial_res_g, np.
↪uint32(num_groups), ret_buf)
    return ret
```

The implementation is regular PyOpenCL and the OpenCL kernel is based on the book "OpenCL Programming Guide" by Aaftab Munshi et al. However, notice that we use *bohrium.interop_pyopencl.get_context()* to get the PyOpenCL context rather than pyopencl.create_some_context(). In order to avoid copying data between host and device memory, we use *bohrium.interop_pyopencl.get_buffer()* to create a OpenCL buffer that points to the device memory of the Bohrium arrays.

PyCUDA

The PyCUDA implementation is very similar to the PyOpenCL. Besides some minor difference in the kernel source code, we use *bohrium.interop_pycuda.init()* to initiate PyCUDA and use *bohrium.interop_pycuda.get_gpuarray()* to get the CUDA buffers from the Bohrium arrays:

```
def bincount_pycuda(x, minlength=None):
    """PyCUDA implementation of `bincount()`"""

    if not interop_pycuda.available():
        raise NotImplementedError("CUDA not available")

    import pycuda
    from pycuda.compiler import SourceModule

    interop_pycuda.init()

    x_max = int(x.max())
    if x_max < 0:
        raise RuntimeError("bincount(): first argument must be a 1 dimensional, non-
↪negative int array")
    if x_max > np.iinfo(np.uint32).max:
        raise NotImplementedError("CUDA: the elements in the first argument must fit␣
↪in a 32bit integer")
    if minlength is not None:
        x_max = max(x_max, minlength)

    # TODO: handle large max element by running multiple bincount() on a range
    if x_max >= interop_pycuda.max_local_memory() // x.itemsize:
        raise NotImplementedError("CUDA: max element is too large for the GPU")

    # Let's create the output array and retrieve the in-/output CUDA buffers
    # NB: we always return uint32 array
```

(continues on next page)

(continued from previous page)

```
    ret = array_create.ones((x_max+1, ), dtype=np.uint32)
    x_buf = interop_pycuda.get_gpuarray(x)
    ret_buf = interop_pycuda.get_gpuarray(ret)

    # CUDA kernel is based on the book "OpenCL Programming Guide" by Aaftab Munshi et␣
↪al.
    source = """
    __global__ void histogram_partial(
        DTYPE *input,
        uint *partial_histo,
        uint input_size
    ){
        int local_size = blockDim.x;
        int group_indx = blockIdx.x * HISTO_SIZE;
        int gid = (blockIdx.x * blockDim.x + threadIdx.x);
        int tid = threadIdx.x;

        __shared__ uint tmp_histogram[HISTO_SIZE];

        int j = HISTO_SIZE;
        int indx = 0;

        // clear the local buffer that will generate the partial histogram
        do {
            if (tid < j)
                tmp_histogram[indx+tid] = 0;
            j -= local_size;
            indx += local_size;
        } while (j > 0);

        __syncthreads();

        if (gid < input_size) {
            atomicAdd(&tmp_histogram[input[gid]], 1);
        }

        __syncthreads();

        // copy the partial histogram to appropriate location in
        // histogram given by group_indx
        if (local_size >= HISTO_SIZE){
            if (tid < HISTO_SIZE)
                partial_histo[group_indx + tid] = tmp_histogram[tid];
        }else{
            j = HISTO_SIZE;
            indx = 0;
            do {
                if (tid < j)
                    partial_histo[group_indx + indx + tid] = tmp_histogram[indx +␣
↪tid];

                j -= local_size;
                indx += local_size;
            } while (j > 0);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    __global__ void histogram_sum_partial_results(
        uint *partial_histogram,
        int num_groups,
        uint *histogram
    ){
        int gid = (blockIdx.x * blockDim.x + threadIdx.x);
        int group_indx;
        int n = num_groups;
        __shared__ uint tmp_histogram[HISTO_SIZE];

        tmp_histogram[gid] = partial_histogram[gid];
        group_indx = HISTO_SIZE;
        while (--n > 0) {
            tmp_histogram[gid] += partial_histogram[group_indx + gid];
            group_indx += HISTO_SIZE;
        }
        histogram[gid] = tmp_histogram[gid];
    }
    """
    source = source.replace("HISTO_SIZE", "%d" % ret.shape[0])
    source = source.replace("DTYPE", interop_pycuda.type_np2cuda_str(x.dtype))
    prg = SourceModule(source)

    # Calculate sizes for the kernel execution
    kernel = prg.get_function("histogram_partial")
    local_size = kernel.get_attribute(pycuda.driver.function_attribute.MAX_THREADS_
↪PER_BLOCK)  # Max work-group size
    num_groups = int(math.ceil(x.shape[0] / float(local_size)))
    global_size = local_size * num_groups

    # First we compute the partial histograms
    partial_res_g = pycuda.driver.mem_alloc(num_groups * ret.nbytes)
    kernel(x_buf, partial_res_g, np.uint32(x.shape[0]), block=(local_size, 1, 1),␣
↪grid=(num_groups, 1))

    # Then we sum the partial histograms into the final histogram
    kernel = prg.get_function("histogram_sum_partial_results")
    kernel(partial_res_g, np.uint32(num_groups), ret_buf, block=(1, 1, 1), grid=(ret.
↪shape[0], 1))
    return ret
```

Performance Comparison

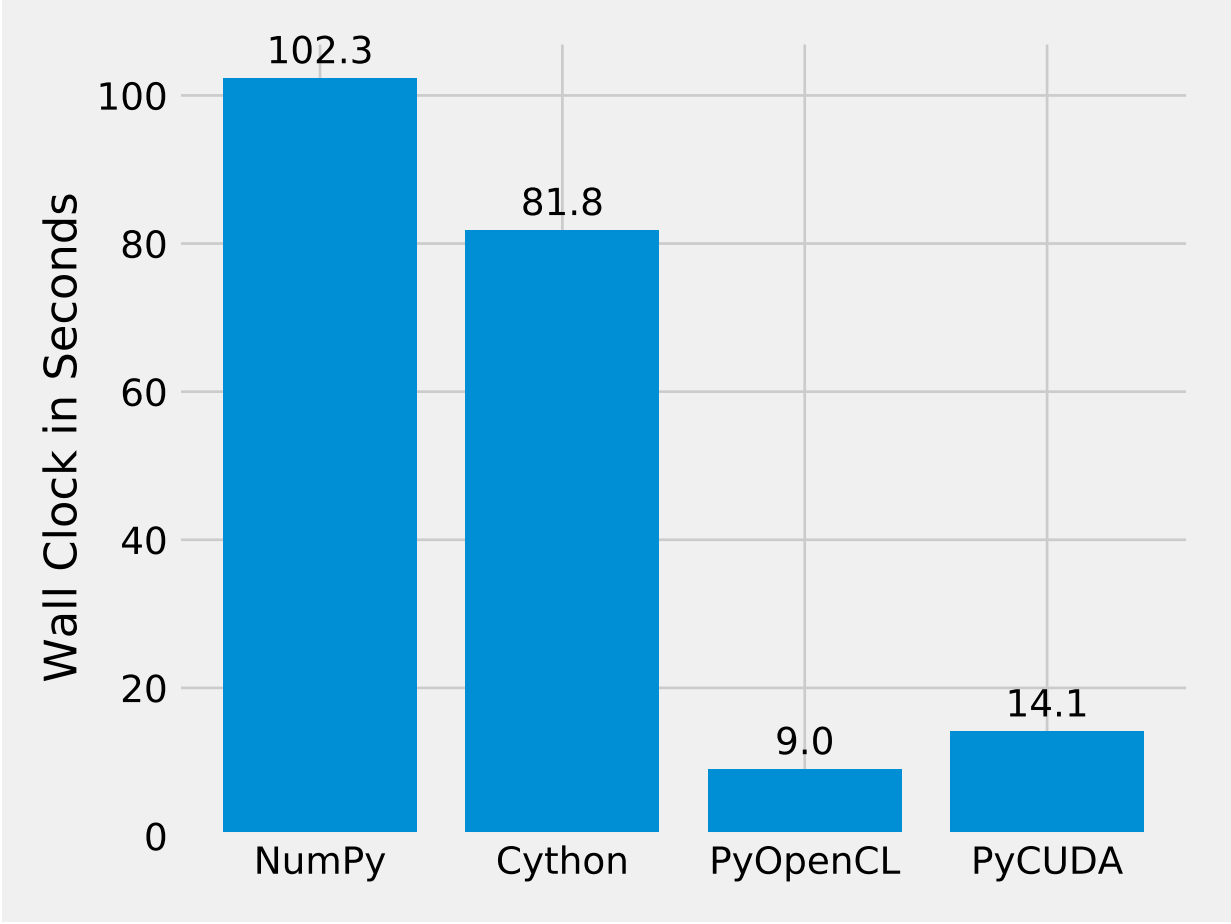
Finally, let's compare the performance of the difference approaches. We run on a *Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz* with 4 CPU-cores and a *GeForce GTX Titan X (maxwell)*. The timing is wall-clock time including everything, in particular the host/device communication overhead.

The timing code:

```
import numpy as np
import time

SIZE = 500000000
ITER = 100
```

(continues on next page)


102.3
81.8
9.0
14.1
100
80
60
40
20
0
Wall Clock in Seconds
NumPy
Cython
PyOpenCL
PyCUDA

(continued from previous page)

```
t1 = time.time()
a = np.minimum(np.arange(SIZE, dtype=np.int64), 64)
for _ in range(ITER):
    b = np.bincount(a)
t2 = time.time()
s = b.sum()
print ("Sum: %d, time: %f sec" % (s, t2 - t1))
```

Conclusion

Interoperability makes it possible to accelerate code that Bohrium doesn't accelerate automatically. The Bohrium team constantly works on improving the performance and increase the number of NumPy operations automatically accelerated but in some cases we simply have to give the user full control.

Python API

Bohrium inherit must of the NumPy API – it is not all functions that are accelerated but they are all usable and can be found under the same name as in NumPy. The following is the part of the Bohrium API that is special to Bohrium.

Bohrium's ndarray

class bohrium._bh.**ndarray**

all(*axis=None*, *out=None*)
Test whether all array elements along a given axis evaluate to True.

Refer to *numpy.all* for full documentation.

any(*axis=None*, *out=None*)
Test whether any array element along a given axis evaluates to True.

Refer to *numpy.any* for full documentation.

argmax(*axis=None*, *out=None*)
Returns the indices of the maximum values along an axis.

Refer to numpy.argmax for full documentation.

argmin(*axis=None*, *out=None*)
Returns the indices of the minimum values along an axis.

Refer to numpy.argmin for full documentation.

astype(*dtype*, *order='C'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

bhc_dynamic_view_info
The information regarding dynamic changes to a view within a do_while loop

bhc_mmap_allocated
Is the base data allocated with mmap?

conj(*x*[, *out*])
Return the complex conjugate, element-wise.

Refer to *numpy.conj* for full documentation.

conjugate(*x*[, *out*])
Return the complex conjugate, element-wise.

Refer to *numpy.conj* for full documentation.

copy(*order='C'*)
Return a copy of the array.

copy2numpy()
Copy the array in C-style memory layout to a regular NumPy array

cumprod(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the array elements over the given axis

Refer to *numpy.cumprod* for full documentation.

cumsum(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the array elements over the given axis.

Refer to *bohrium.cumsum* for full documentation.

dot(*b*, *out=None*)
Compute the dot product.

fill(*value*)
Fill the array with a scalar value.

flatten()
Return a copy of the array collapsed into one dimension.

max(*axis=None*, *out=None*)
Return the maximum along a given axis.

Refer to numpy.amax for full documentation.

mean(*axis=None*, *dtype=None*, *out=None*)
Compute the arithmetic mean along the specified axis.

min(*axis=None*, *out=None*)
Return the minimum along a given axis.

Refer to numpy.amin for full documentation.

prod(*axis=None*, *dtype=None*, *out=None*)
Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

put(*indices*, *values*, *mode='raise'*)
Set a.flat[n] = values[n] for all n in indices.

ravel()
Return a copy of the array collapsed into one dimension.

reshape(*shape*)
Returns an array containing the same data with a new shape.

Refer to *bohrium.reshape* for full documentation.

resize()
Change shape and size of array in-place

sum(*axis=None*, *dtype=None*, *out=None*)
Return the sum of the array elements over the given axis.

Refer to *bohrium.sum* for full documentation.

take()
a.take(indices, axis=None, out=None, mode='raise').

tofile(*fid*, *sep=""*, *format="%s"*)
Write array to a file as text or binary (default).

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Return the sum along diagonals of the array.

Module contents

The module initialization of npbackend/bohrium imports and exposes all methods required to become a drop-in replacement for numpy.

bohrium.**flush**()
Evaluate all delayed array operations

bohrium.**array**(*obj*, *dtype=None*, *copy=False*, *order=None*, *subok=False*, *ndmin=0*, *bohrium=True*)
Create an array – Bohrium or NumPy ndarray.

Parameters

obj [array_like] An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype [data-type, optional] The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy [bool, optional] If true, then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*dtype*, *order*, etc.).

order [{'C', 'F', 'A'}, optional] Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

subok [bool, optional] If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

ndmin [int, optional] Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

bohrium [boolean, optional] Determines whether it is a Bohrium array (bohrium.ndarray) or a regular NumPy array (numpy.ndarray)

Returns

out [ndarray] An array object satisfying the specified requirements.

See also:

empty, **empty_like**, **zeros**, **zeros_like**, **ones**, **ones_like**, **fill**

Examples

```
>>> np.array([1, 2, 3])
array([1, 2, 3])
```

Upcasting:

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

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

Minimum dimensions 2:

```
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])
```

Type provided:

```
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

bohrium.backend_messaging module

Send and receive a message through the Bohrium component stack

bohrium.backend_messaging.**cuda_use_current_context**()
Tell the CUDA backend to use the current CUDA context (useful for PyCUDA interop)

bohrium.backend_messaging.**gpu_disable**()
Disable the GPU backend in the current runtime stack

bohrium.backend_messaging.**gpu_enable**()
Enable the GPU backend in the current runtime stack

bohrium.backend_messaging.**runtime_info**()
Return a YAML string describing the current Bohrium runtime

bohrium.backend_messaging.**statistic**()
Return a YAML string of Bohrium statistic

bohrium.backend_messaging.**statistic_enable_and_reset**()
Reset and enable the Bohrium statistic

bohrium.bhary module

This module consist of bohrium.ndarray methods

The functions here serve as a means to determine whether a given array is a numpy.ndarray or a bohrium.ndarray as well as moving between the two "spaces".

— License — This file is part of Bohrium and copyright (c) 2012 the Bohrium http://bohrium.bitbucket.org

Bohrium is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Bohrium is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Bohrium.

If not, see <http://www.gnu.org/licenses/>.

bohrium.bhary.**check**()
Returns True if 'ary' is a Bohrium array

bohrium.bhary.**check_biclass_bh_over_np**()
Returns True if 'ary' is a Bohrium view with a NumPy base array

bohrium.bhary.**check_biclass_np_over_bh**()
Returns True if 'ary' is a NumPy view with a Bohrium base array

bohrium.bhary.**fix_biclass**()
Makes sure that when 'ary' or its base is a Bohrium array, both of them are.

bohrium.bhary.**fix_biclass_wrapper**()
Function decorator that makes sure that the function doesn't reads or writes biclass arrays

bohrium.bhary.**get_base**()
Get the final base array of 'ary'.

bohrium.bhary.**is_base**()
Return True when 'ary' is a base array.

bohrium.blas module

Basic Linear Algebra Subprograms (BLAS)

Utilize BLAS directly from Python

bohrium.blas.**gemm**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * B + beta * C

bohrium.blas.**gemmt**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A^T * B + beta * C

bohrium.blas.**hemm**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * B + beta * C

bohrium.blas.**her2k**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * B**H + conjg(alpha) * B * A**H + beta * C

bohrium.blas.**herk**(*a*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * A**H + beta * C

bohrium.blas.**symm**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * B + beta * C

bohrium.blas.**syr2k**(*a*, *b*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * B**T + alpha * B * A**T + beta * C

bohrium.blas.**syrk**(*a*, *alpha=1.0*, *c=None*, *beta=0.0*)
C := alpha * A * A**T + beta * C

bohrium.blas.**trmm**(*a*, *b*, *alpha=1.0*)
B := alpha * A * B

bohrium.blas.**trsm**(*a*, *b*)
Solves: A * X = B

bohrium.concatenate module

Array concatenate functions

bohrium.concatenate.**atleast_1d**(**arys*)
Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

Parameters

arys1, arys2, ... [array_like] One or more input arrays.

Returns

ret [ndarray] An array, or list of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

See also:

atleast_2d*, *atleast_3d

Examples

```
>>> np.atleast_1d(1.0)
array_create.array([ 1.])
```

```
>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array_create.array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
```

(continues on next page)

(continued from previous page)

```
       [ 6.,  7.,  8.]])
>>> np.atleast_1d(x) is x
True
```

```
>>> np.atleast_1d(1, [3, 4])
[array_create.array([1]), array_create.array([3, 4])]
```

bohrium.concatenate.**atleast_2d**(**arys*)

View inputs as arrays with at least two dimensions.

Parameters

arys1, arys2, ... [array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

Returns

res, res2, ... [ndarray] An array, or list of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

See also:

atleast_1d, ***atleast_3d***

Examples

```
>>> np.atleast_2d(3.0)
array_create.array([[ 3.]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array_create.array([[ 0.,  1.,  2.]])
>>> np.atleast_2d(x).base is x
True
```

```
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array_create.array([[1]]), array_create.array([[1, 2]]), array_create.array([[1,␣
↪2]])]
```

bohrium.concatenate.**atleast_3d**(**arys*)

View inputs as arrays with at least three dimensions.

Parameters

arys1, arys2, ... [array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

Returns

res1, res2, ... [ndarray] An array, or list of arrays, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

See also:

atleast_1d, ***atleast_2d***

Examples

```
>>> np.atleast_3d(3.0)
array_create.array([[[ 3.]]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x.base  # x is a reshape, so not base itself
True
```

```
>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print(arr, arr.shape)
...
[[[1]
  [2]]] (1, 2, 1)
[[[1]
  [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

bohrium.concatenate.**concatenate**(*(a1, a2, ...), axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

a1, a2, ... [sequence of array_like] The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis [int, optional] The axis along which the arrays will be joined. Default is 0.

Returns

res [ndarray] The concatenated array.

See also:

ma.concatenate Concatenate function that preserves input masks.

array_split Split an array into multiple sub-arrays of equal or near-equal size.

split Split array into a list of multiple sub-arrays of equal size.

hsplit Split array into multiple sub-arrays horizontally (column wise)

vsplit Split array into multiple sub-arrays vertically (row wise)

dsplit Split array into multiple sub-arrays along the 3rd axis (depth).

stack Stack a sequence of arrays along a new axis.

hstack Stack arrays in sequence horizontally (column wise)

vstack Stack arrays in sequence vertically (row wise)

dstack Stack arrays in sequence depth wise (along third dimension)

Notes

When one or more of the arrays to be concatenated is a MaskedArray, this function will return a MaskedArray object instead of an ndarray, but the input masks are *not* preserved. In cases where a MaskedArray is expected as input, use the ma.concatenate function from the masked array module instead.

Examples

```
>>> a = np.array_create.array([[1, 2], [3, 4]])
>>> b = np.array_create.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array_create.array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array_create.array([[1, 2, 5],
       [3, 4, 6]])
```

This function will not preserve masking of MaskedArray inputs.

```
>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False  True False],
       fill_value = 999999)
>>> b
array_create.array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data = [0 1 2 2 3 4],
             mask = False,
       fill_value = 999999)
>>> np.ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False  True False False False False],
       fill_value = 999999)
```

bohrium.concatenate.**hstack**(*tup*)

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by *hsplit*.

This function continues to be supported for backward compatibility, but you should prefer `np.concatenate` or `np.stack`. The `np.stack` function was added in NumPy 1.10.

Parameters

tup [sequence of ndarrays] All arrays must have the same shape along all but the second axis.

Returns

stacked [ndarray] The array formed by stacking the given arrays.

See also:

stack Join a sequence of arrays along a new axis.

vstack Stack arrays in sequence vertically (row wise).

dstack Stack arrays in sequence depth wise (along third axis).

concatenate Join a sequence of arrays along an existing axis.

hsplit Split array along second axis.

Notes

Equivalent to `np.concatenate(tup, axis=1)`

Examples

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

`bohrium.concatenate.`**stack**(*arrays*, *axis=0*)

Join a sequence of arrays along a new axis.

The *axis* parameter specifies the index of the new axis in the dimensions of the result. For example, if `axis=0` it will be the first dimension and if `axis=-1` it will be the last dimension.

New in version 1.10.0.

Parameters

arrays [sequence of array_like] Each array must have the same shape.

axis [int, optional] The axis in the result array along which the input arrays are stacked.

Returns

stacked [ndarray] The stacked array has one more dimension than the input arrays.

See also:

concatenate Join a sequence of arrays along an existing axis.

split Split array into a list of multiple sub-arrays of equal size.

Examples

```
>>> arrays = [np.random.randn(3, 4) for _ in range(10)]
>>> np.stack(arrays, axis=0).shape
(10, 3, 4)
```

```
>>> np.stack(arrays, axis=1).shape
(3, 10, 4)
```

```
>>> np.stack(arrays, axis=2).shape
(3, 4, 10)
```

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

```
>>> np.stack((a, b), axis=-1)
array_create.array([[1, 2],
       [2, 3],
       [3, 4]])
```

bohrium.concatenate.**vstack**(*tup*)

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by *vsplit*.

This function continues to be supported for backward compatibility, but you should prefer `np.concatenate` or `np.stack`. The `np.stack` function was added in NumPy 1.10.

Parameters

tup [sequence of ndarrays] Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

stacked [ndarray] The array formed by stacking the given arrays.

See also:

stack Join a sequence of arrays along a new axis.

hstack Stack arrays in sequence horizontally (column wise).

dstack Stack arrays in sequence depth wise (along third dimension).

concatenate Join a sequence of arrays along an existing axis.

vsplit Split array into a list of multiple sub-arrays vertically.

Notes

Equivalent to `np.concatenate(tup, axis=0)` if *tup* contains arrays that are at least 2-dimensional.

Examples

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

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

bohrium.contexts module

Bohrium Contexts

class bohrium.contexts.**DisableBohrium**
Disable Bohrium within the context

class bohrium.contexts.**DisableGPU**
Disable the GPU backend within the context.

class bohrium.contexts.**EnableBohrium**
Enable Bohrium within the context

class bohrium.contexts.**Profiling**
Profiling the Bohrium backends within the context.

bohrium.interop_numpy module

Interop NumPy

bohrium.interop_numpy.**get_array**(*bh_ary*)
Return a NumPy array wrapping the memory of the Bohrium array *ary*.

Parameters

bh_ary [ndarray (Bohrium array)] Must be a Bohrium base array

Returns

out [ndarray (regular NumPy array)]

Notes

Changing or deallocating *bh_ary* invalidates the returned NumPy array!

bohrium.interop_pycuda module

Interop PyCUDA

bohrium.interop_pycuda.**available**()
Is CUDA available?

bohrium.interop_pycuda.**get_gpuarray**(*bh_ary*)
Return a PyCUDA GPUArray object that points to the same device memory as *bh_ary*.

Parameters

bh_ary [ndarray (Bohrium array)] Must be a Bohrium base array

Returns

out [GPUArray]

Notes

Changing or deallocating *bh_ary* invalidates the returned GPUArray array!

bohrium.interop_pycuda.**init**()
Initiate the PyCUDA module. Must be called before any other PyCUDA calls and preferable also before any Bohrium operations

bohrium.interop_pycuda.**max_local_memory**(*cuda_device=None*)
Returns the maximum allowed local memory (memory per block) on *cuda_device*. If *cuda_device* is None, use current device

bohrium.interop_pycuda.**type_np2cuda_str**(*np_type*)
Converts a NumPy type to a CUDA type string

bohrium.interop_pyopencl module

Interop PyOpenCL

bohrium.interop_pyopencl.**available**()
Is PyOpenCL available?

bohrium.interop_pyopencl.**get_buffer**(*bh_ary*)
Return a OpenCL Buffer object wrapping the Bohrium array *ary*.

Parameters

bh_ary [ndarray (Bohrium array)] Must be a Bohrium base array

Returns

out [pyopencl.Buffer]

Notes

Changing or deallocating *bh_ary* invalidates the returned pyopencl.Buffer!

bohrium.interop_pyopencl.**get_context**()
Return a PyOpenCL context

bohrium.interop_pyopencl.**kernel_info**(*opencl_kernel*, *queue*)
Info about the *opencl_kernel* Returns 4-tuple:

- Max work-group size
- Recommended work-group multiple
- Local mem used by kernel

- Private mem used by kernel

bohrium.interop_pyopencl.**max_local_memory**(*opencl_device*)
Returns the maximum allowed local memory on *opencl_device*

bohrium.interop_pyopencl.**set_buffer**(*bh_ary*, *buffer*)
Assign a OpenCL Buffer object to a Bohrium array *ary*.

Parameters

bh_ary [ndarray (Bohrium array)] Must be a Bohrium base array

buffer [pyopencl.Buffer] The PyOpenCL device buffer

bohrium.interop_pyopencl.**type_np2opencl_str**(*np_type*)
Converts a NumPy type to a OpenCL type string

bohrium.linalg module

LinAlg

Common linear algebra functions

bohrium.linalg.**gauss**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**lu**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**solve**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**jacobi**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**matmul**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**dot**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**norm**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**tensordot**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**solve_tridiagonal**
Invokes 'func' and strips "biclass" from the result.

bohrium.linalg.**cg**
Invokes 'func' and strips "biclass" from the result.

bohrium.loop module

Bohrium Loop

class bohrium.loop.**DynamicViewInfo**(*dynamic_changes*, *shape*, *stride*, *resets={}*)
Bases: object

Object for storing information about dynamic changes to the view

Methods

add_dynamic_change(self, dim, slide, ...[, ...])	Add dynamic changes to the dynamic changes information of the view.
changes_in_dim(self, dim)	Returns a list of all dynamic changes in a dimension.
dim_shape_change(self, dim)	Returns the summed shape change in the given dimension.
dims_with_changes(self)	Returns a list of all dimensions with dynamic changes.
get_shape_changes(self)	Returns a dictionary of all changes to the shape.
has_changes(self)	Returns whether the object contains any dynamic changes.
has_changes_in_dim(self, dim)	Check whether there are any dynamic changes in the given dimension.
index_into(self, dvi)	Modifies the dynamic change such that is reflects being indexed into another view with dynamic changes.

add_dynamic_change(*self*, *dim*, *slide*, *shape_change*, *step_delay*, *shape=None*, *stride=None*)
Add dynamic changes to the dynamic changes information of the view.

Parameters

dim [int] The relevant dimension

slide [int] The change to the offset in the given dimension (can be both positive and negative)

shape_change [int] The amount the shape changes in the given dimension (can also be both positive and negative)

step_delay [int] If the change is based on an iterator in a grid, it is the changes can be delayed until the inner iterators have been updated *step_delay* times.

shape [int] The shape that the view can slide within. If not given, self.shape[dim] is used instead

stride [int] The stride that the view can slide within. If not given, self.stride[dim] is used instead

changes_in_dim(*self*, *dim*)
Returns a list of all dynamic changes in a dimension. If the dimension does not contain any dynamic changes, an empty list is returned.

Parameters

dim [int] The relevant dimension

dim_shape_change(*self*, *dim*)
Returns the summed shape change in the given dimension.

Parameters

dim [int] The relevant dimension

dims_with_changes(*self*)
Returns a list of all dimensions with dynamic changes.

get_shape_changes(*self*)
Returns a dictionary of all changes to the shape. The dimension is the key and the shape change in the dimension is the value.

has_changes(*self*)
Returns whether the object contains any dynamic changes.

has_changes_in_dim(*self*, *dim*)
Check whether there are any dynamic changes in the given dimension.

Parameters

dim [int] The relevant dimension

index_into(*self*, *dvi*)
Modifies the dynamic change such that is reflects being indexed into another view with dynamic changes.

Parameters

dim [DynamicViewInfo] The infomation about dynamic changes within the view which is indexed into

class bohrium.loop.**Iterator**(*max_iter*, *value*, *step_delay=1*, *reset=None*)
Bases: object

Iterator used for sliding views within loops.

Notes

Supports addition, subtraction and multiplication.

exception bohrium.loop.**IteratorIllegalBroadcast**(*dim*, *a_shape*, *a_shape_change*, *bcast_shape*, *bcast_shape_change*)
Bases: exceptions.Exception

Exception thrown when a view consists of a mix of iterator depths.

exception bohrium.loop.**IteratorIllegalDepth**
Bases: exceptions.Exception

Exception thrown when a view consists of a mix of iterator depths.

exception bohrium.loop.**IteratorOutOfBounds**(*dim*, *shape*, *first_index*, *last_index*)
Bases: exceptions.Exception

Exception thrown when a view goes out of bounds after the maximum iterations.

bohrium.loop.**add_slide_info**(*a*)

Checks whether a view is dynamic and adds the relevant information to the view structure within BXX if it is.

Parameters

a [array view] A view into an array

bohrium.loop.**do_while**(*func*, *niters*, **args*, ***kwargs*)
Repeatedly calls the *func* with the **args* and ***kwargs* as argument.

The *func* is called while *func* returns True or None and the maximum number of iterations, *niters*, hasn't been reached.

Parameters

func [function] The function to run in each iterations. *func* can take any argument and may return a boolean *bharray* with one element.

niters: int or None Maximum number of iterations in the loop (number of times *func* is called). If None, there is no maximum.

***args, **kwargs** [list and dict] The arguments to *func*

Notes

func can only use operations supported natively in Bohrium.

Examples

```
>>> def loop_body(a):
...     a += 1
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, 5, a)
>>> a
array([5, 5, 5, 5])
```

```
>>> def loop_body(a):
...     a += 1
...     return bh.sum(a) < 10
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, None, a)
>>> a
array([3, 3, 3, 3])
```

bohrium.loop.**get_grid**(*max_iter*, **args*)

Returns n iterators in a grid, corresponding to nested loops.

Parameters

args [pointer to two or more integers] The first integer is the maximum iterations of the loop, used for checking boundaries. The rest are the shape of the grid.

Notes

get_grid can only be used within a bohrium loop function. Within the loop *max_iter* is set by a lambda function. There are no upper bound on the amount of grid values.

Examples

```
>>> def kernel(a):
...     i, j, k = get_grid(3,3,3)
...     a[i,j,k] += 1
```

correspondes to

```
>>> for i in range(3):
...     for j in range(3):
...         for k in range(3):
...             a[i,j,k] += 1
```

bohrium.loop.**get_iterator**(*max_iter*, *val*, *step_delay=1*)

Returns an iterator with a given starting value. An iterator behaves like an integer, but is used to slide view between loop iterations.

Parameters

max_iter [int] The maximum amount of iterations of the loop. Used for checking boundaries.

val [int] The initial value of the iterator.

Notes

get_iterator can only be used within a bohrium loop function. Within the loop *max_iter* is set by a lambda function. This is also the case in the example.

Examples

```
>>> def kernel(a):
...     i = get_iterator(1)
...     a[i] *= a[i-1]
>>> a = bh.arange(1,6)
>>> bh.do_while(kernel, 4, a)
array([1, 2, 6, 24, 120])
```

bohrium.loop.**has_iterator**(**s*)

Checks whether a (multidimensional) slice contains an iterator

Parameters

s [pointer to an integer, iterator or a tuple of integers/iterators]

Notes

Only called from __getitem__ and __setitem__ in bohrium arrays (see _bh.c).

bohrium.loop.**inherit_dynamic_changes**(*a*, *sliced*)

Creates a view into another view which has dynamic changes. The new view inherits the dynamic changes.

bohrium.random123 module

Random

Random functions

bohrium.random123.**seed**(*seed=None*)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

Parameters

seed [int or array_like, optional] Seed for *RandomState*.

See also:

RandomState

bohrium.random123.**random_sample**()
Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random* by *(b-a)* and add a:

```
(b - a) * random() + a
```

Parameters

shape [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

Returns

out [float or ndarray of floats] Array of random floats of shape *shape* (unless `shape=None`, in which case a single float is returned).

Examples

```
>>> np.random.random()
0.47108547995356098
>>> type(np.random.random())
<type 'float'>
>>> np.random.random((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

bohrium.random123.**uniform**(*low=0.0*, *high=1.0*, *size=None*, *dtype=float*, *bohrium=True*)
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [ndarray] Drawn samples, with shape *size*.

See also:

randint Discrete uniform distribution, yielding integers.

random_integers Discrete uniform distribution over the closed interval `[low, high]`.

random_sample Floats uniformly distributed over `[0, 1)`.

random Alias for *random_sample*.

rand Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

same as: random_sample(size) * (high - low) + low

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

`bohrium.random123.`**rand**(*d0*, *d1*, ..., *dn*, *dtype=float*, *bohrium=True*)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

Parameters

d0, d1, . . . , dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

Returns

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

See also:

random

Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

bohrium.random123.**randn**(*d0*, *d1*, ..., *dn*, *dtype=float*, *bohrium=True*)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

Parameters

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Returns

Z [ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

See also:

`random.standard_normal` Similar, but takes a tuple as its argument.

Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

Examples

```
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

bohrium.random123.**random_integers**(*low*, *high=None*, *size=None*, *dtype=int*, *bohrium=True*)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*].

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

See also:

random.randint Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

Examples

```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3.,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

bohrium.random123.**standard_normal**(*size=None*, *dtype=float*, *bohrium=True*)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

Parameters

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [float or ndarray] Drawn samples.

Examples

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311, #random
       -0.38672696, -0.4685006 ])                                #random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

bohrium.random123.**normal**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=float*, *bohrium=True*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

Parameters

loc [float] Mean ("centre") of the distribution.

scale [float] Standard deviation (spread or "width") of the distribution.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

See also:

scipy.stats.distributions.norm probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

[1], [2]

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
```

```
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

bohrium.random123.**standard_exponential**(*size=None*, *dtype=float*, *bohrium=True*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [float or ndarray] Drawn samples.

Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

bohrium.random123.**exponential**(*scale=1.0*, *size=None*, *dtype=float*, *bohrium=True*)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1], or the time between page requests to Wikipedia [2].

Parameters

scale [float] The scale parameter, $\beta = 1/\lambda$.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

References

[1], [2], [3]

bohrium.signal module

Signal Processing

Common signal processing functions, which often handle multiple dimension

bohrium.summations module

Summations and products

bohrium.summations.**average**(*a*, *axis=None*, *dtype=None*, *out=None*)

Compute the arithmetic mean along the specified axis. Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *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 [None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array. .. versionadded:: 1.7.0 If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out [ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

Returns

m [ndarray, see dtype parameter above] If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also

average [Weighted average]

std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

By default, *float16* results are computed using *float32* intermediates for extra precision.

Examples

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

```
>>> np.mean(a)
    ..
```

2.5

```
>>> np.mean(a, axis=0)
    ..
```

array([2., 3.])

```
>>> np.mean(a, axis=1)
    ..
```

array([1.5, 3.5])

In single precision, *mean* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
    ..
```

```
>>> a[0, :] = 1.0
    ..
```

```
>>> a[1, :] = 0.1
    ..
```

```
>>> np.mean(a)
    ..
```

0.54999924

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
    ..
```

0.55000000074505806

bohrium.summations.**mean**(*a*, *axis=None*, *dtype=None*, *out=None*)

Compute the arithmetic mean along the specified axis. Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *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 [None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array. .. versionadded:: 1.7.0 If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out [ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

Returns

m [ndarray, see dtype parameter above] If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also

average [Weighted average]

std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

By default, *float16* results are computed using *float32* intermediates for extra precision.

Examples

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

```
>>> np.mean(a)
    ..
```

2.5

```
>>> np.mean(a, axis=0)
    ..
```

array([2., 3.])

```
>>> np.mean(a, axis=1)
    ..
```

array([1.5, 3.5])

In single precision, *mean* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
    ..
```

```
>>> a[0, :] = 1.0
    ..
```

```
>>> a[1, :] = 0.1
    ..
```

```
>>> np.mean(a)
    ..
```

0.54999924

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
    ..
```

0.55000000074505806

bohrium.user_kernel module

bohrium.user_kernel.**dtype_to_c99**(*dtype*)
Returns the C99 name of *dtype*

bohrium.user_kernel.**execute**(*kernel_source*, *operand_list*, *compiler_command=None*, *tag='openmp'*, *param=None*, *only_behaving_operands=True*)
Compile and execute the function *execute()* with the arguments in *operand_list*

Parameters

kernel_source [str] The kernel source code that most define the function *execute()* that should take arguments corresponding to the *operand_list*

operand_list [list of bohrium arrays] The arrays given to the *execute()* function defined in *kernel_source*

compiler_command [str, optional] The compiler command to use when comping the kernel. *{OUT}* and *{IN}* in the command are replaced with the name of the binary and source path. When this options isn't specified, the default command are used see *get_default_compiler_command()*.

tag [str, optional] Name of the backend that should handle this kernel.

param [dict, optional] Backend specific parameters (e.g. OpenCL needs *global_work_size* and *local_work_size*).

only_behaving_operands [bool, optional]

Set to False in order to allow non-behaving operands. Requirements for a behaving array:

- Is a bohrium array
- Is C-style contiguous
- Points to the first element in the underlying base array (no offset)
- Has the same total length as its base

See *make_behaving()*

Examples

Simple addition kernel import bohrium as bh kernel = r"' #include <stdint.h> void execute(double *a, double *b, double *c) {

for(uint64_t i=0; i<100; ++i) { c[i] = a[i] + b[i] + i;

}

}"' a = bh.ones(100, bh.double) b = bh.ones(100, bh.double) res = bh.empty_like(a) bh.user_kernel.execute(kernel, [a, b, res])

bohrium.user_kernel.**gen_function_prototype**(*operand_list*, *operand_name_list=None*)
Returns the *execute() definition based on the arrays in 'operand_list*

bohrium.user_kernel.**get_default_compiler_command**()
Returns the default compiler command, which is the one typically extended with extra link commands

bohrium.user_kernel.**make_behaving**(*ary*, *dtype=None*)
Make sure that *ary* is a "behaving" bohrium array of type *dtype*. Requirements for a behaving array:

- Is a bohrium array
- Is C-style contiguous
- Points to the first element in the underlying base array (no offset)
- Has the same total length as its base

Parameters

ary [array_like] The array to make behaving

dtype [boolean, optional] The return array is converted to *dtype* if not None

Returns

A behaving Bohrium array that might be a copy of *ary*

Bh107 (NumPy Imitation)

Getting Started

Bh107 implements a new python module `bh107` that introduces a new array class *bh107.BhArray()* which imitation `numpy.ndarray()`. The two array classes are zero-copy compatible thus you can convert a *bh107.BhArray()* to a `numpy.ndarray()` without any data copy.

In order to choose which Bohrium backend to use, you can define the `BH_STACK` environment variable. Currently, three backends exist: `openmp`, `opencl`, and `cuda`.

Before using Bohrium, you can check the current runtime configuration using:

```
$ BH_STACK=opencl python -m bohrium_api --info

----
Bohrium version: 0.10.2.post8
----
Bohrium API version: 0.10.2.post8
Installed through PyPI: False
Config file: ~/.bohrium/config.ini
Header dir: ~/.local/lib/python3.7/site-packages/bohrium_api/include
Backend stack:
----
OpenCL:
  Device[0]: AMD Accelerated Parallel Processing / Intel(R) Core(TM) i7-5600U CPU @ 2.
↪60GHz (OpenCL C 1.2 )
  Memory:         7676 MB
  Malloc cache limit: 767 MB (90%)
  Cache dir: "~/.local/var/bohrium/cache"
  Temp dir: "/tmp/bh_75cf_314f5"
  Codegen flags:
    Index-as-var: true
```

(continues on next page)

(continued from previous page)

```
    Strides-as-var: true
    const-as-var: true
----
OpenMP:
  Main memory: 7676 MB
  Hardware threads: 4
  Malloc cache limit: 2190 MB (80% of unused memory)
  Cache dir: "~/.local/var/bohrium/cache"
  Temp dir: "/tmp/bh_75a5_c6368"
  Codegen flags:
    OpenMP: true
    OpenMP+SIMD: true
    Index-as-var: true
    Strides-as-var: true
    Const-as-var: true
  JIT Command: "/usr/bin/cc -x c -fPIC -shared  -std=gnu99  -O3 -march=native -Werror␣
↪-fopenmp -fopenmp-simd -I~/.local/share/bohrium/include {IN} -o {OUT}"
----
```

Notice, since `BH_STACK=opencl` is defined, the runtime stack consist of both the OpenCL and the OpenMP backend. In this case, OpenMP only handles operations unsupported by OpenCL.

Heat Equation Example

The following example is a heat-equation solver that uses Bh107. Note that the only difference between Bohrium code and NumPy code is the first line where we import bohrium as np instead of numpy as np:

```
import bh107 as np
def heat2d(height, width, epsilon=42):
  G = np.zeros((height+2,width+2),dtype=np.float64)
  G[:,0]  = -273.15
  G[:,-1] = -273.15
  G[-1,:] = -273.15
  G[0,:]  = 40.0
  center = G[1:-1,1:-1]
  north  = G[:-2,1:-1]
  south  = G[2:,1:-1]
  east   = G[1:-1,:-2]
  west   = G[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.add.reduce(np.abs(tmp-center))
    center[:] = tmp
  return center
heat2d(100, 100)
```

Convert between Bh107 and NumPy

Create a new NumPy array with ones:

```
np_ary = numpy.ones(42)
```

Convert any type of array to Bh107:

```
bh_ary = bh107.array(np_ary)
```

Copy a Bh107 array into a new NumPy array:

```
npy2 = bh_ary.copy2numpy()
```

Zero-copy a Bh107 array into a NumPy array:

```
npy3 = bh_ary.asnumpy()
# At this point `bh_ary` and `npy3` points to the same data.
```

UserKernel

Bh107 supports user kernels, which makes it possible to implement a specialized handwritten kernel. The idea is that if you encounter a problem that you cannot implement using array programming and Bh107 cannot accelerate, you can write a kernel in C99 that calls other libraries or do the calculation itself.

OpenMP Example

In order to write and run your own kernel use *bh107.user_kernel.execute()*:

```
import bh107 as bh

def fftn(ary):
    # Making sure that `ary` is complex, contiguous, and uses no offset
    ary = bh.user_kernel.make_behaving(ary, dtype=bh.complex128)
    res = bh.empty_like(a)

    # Indicates the direction of the transform you are interested in;
    # technically, it is the sign of the exponent in the transform.
    sign = ["FFTW_FORWARD", "FFTW_BACKWARD"]

    kernel = """
    #include <stdint.h>
    #include <stdlib.h>
    #include <complex.h>
    #include <fftw3.h>

    #if defined(_OPENMP)
        #include <omp.h>
    #else
        static inline int omp_get_max_threads() { return 1; }
        static inline int omp_get_thread_num()  { return 0; }
        static inline int omp_get_num_threads() { return 1; }
    #endif

    void execute(double complex *in, double complex *out) {
        const int ndim = %(ndim)d;
        const int shape[] = {%(shape)s};
        const int sign = %(sign)s;

        fftw_init_threads();
        fftw_plan_with_nthreads(omp_get_max_threads());
```

(continues on next page)

(continued from previous page)

```
        fftw_plan p = fftw_plan_dft(ndim, shape, in, out, sign, FFTW_ESTIMATE);
        if(p == NULL) {
            printf("fftw plan fail!\\n");
            exit(-1);
        }
        fftw_execute(p);
        fftw_destroy_plan(p);
        fftw_cleanup_threads();
    }
    """ % {'ndim': a.ndim, 'shape': str(a.shape)[1:-1], 'sign': sign[0]}

    # Adding some extra link options to the compiler command
    cmd = bh.user_kernel.get_default_compiler_command() + " -lfftw3 -lfftw3_threads"
    bh.user_kernel.execute(kernel, [ary, res], compiler_command=cmd)
    return res
```

Two useful help functions when writing user kernels is *bh107.user_kernel.make_behaving()*, which makes that an array is of a specific data type, is contiguous, and uses no offset and *bh107.user_kernel.dtype_to_c99()*, which converts a Bh107/NumPy array data type into a C99 data type.

OpenCL Example

In order to use the OpenCL backend, use the *tag* and *param* of *bh107.user_kernel.execute()*:

```
import bh107 as bh

kernel = """
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

kernel void execute(global double *a, global double *b) {
    int i0 = get_global_id(0);
    int i1 = get_global_id(1);
    int gid = i0 * 5 + i1;
    b[gid] = a[gid] + gid;
}
"""
a = bh.ones(10*5, bh.double).reshape(10,5)
res = bh.empty_like(a)
# Notice, the OpenCL backend requires global_work_size and local_work_size
bh.user_kernel.execute(kernel, [a, res],
                       tag="opencl",
                       param={"global_work_size": [10, 5], "local_work_size": [1, 1]})
print(res)
```

Note: Remember to use the OpenCL backend by setting *BH_STACK=opencl*.

Python API

- *BhBase and BhArray*

BhBase and BhArray

class bh107.**BhBase**(*dtype*, *nelem*)
A base array that represent a block of memory. A base array is always the sole owner of a complete memory allocation.

dtype = None
The data type of the base array

itemsize = None
Size of an element in bytes

nbytes = None
Total size of the base array in bytes

nelem = None
Number of elements

class bh107.**BhArray**(*shape*, *dtype*, *strides=None*, *offset=0*, *base=None*, *is_scalar=False*)
A array that represent a *view* of a base array. Multiple array views can point to the same base array.

T

asnumpy(*self*)
Returns a NumPy array that points to the same memory as this BhArray

astype(*self*, *dtype*, *always_copy=True*)

base = None
The base array

copy(*self*)
Return a copy of the array.

Returns

out [BhArray] Copy of *self*

copy2numpy(*self*)
Returns a NumPy array that is a copy of this BhArray

dtype

empty(*self*)

fill(*self*, *value*)
Fill the array with a scalar value.

Parameters

value [scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = bh107.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = bh107.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

flatten (*self*, *always_copy=True*)

Return a copy of the array collapsed into one dimension.

Parameters

always_copy [boolean] When False, a copy is only made when necessary

Returns

y [ndarray] A copy of the array, flattened to one dimension.

Notes

The order of the data in memory is always row-major (C-style).

Examples

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

classmethod from_numpy (*cls*, *numpy_array*)

classmethod from_object (*cls*, *obj*)

classmethod from_scalar (*cls*, *scalar*)

isbehaving (*self*)

iscontiguous (*self*)

isscalar (*self*)

ndim

ravel (*self*)

Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Returns

y [ndarray] A copy or view of the array, flattened to one dimension.

reshape (*self*, *shape*)

shape

size

strides
Gets the strides in elements

strides_in_bytes
Gets the strides in bytes

transpose(*self*, *axes=None*)
Permute the dimensions of an array.

Parameters

axes [list of ints, optional] By default, reverse the dimensions, otherwise permute the axes according to the values given.

view(*self*)
Returns a new view that points to the same base as this BhArray

Array Creation

bh107.**array**(*obj*, *dtype=None*, *copy=False*)
Create an BhArray.

Parameters

obj [array_like] An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype [data-type, optional] The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy [bool, optional] If true, then the object is copied. Otherwise, a copy will only be made if obj isn't a BhArray of the correct dtype already

Returns

out [BhArray] An array of dtype.

See also:

empty, ***empty_like***, ***zeros***, ***zeros_like***, ***ones***, ***ones_like***, **fill**

Examples

```
>>> bh.array([1, 2, 3])
array([1, 2, 3])
```

Upcasting:

```
>>> bh.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> bh.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

Type provided:

```
>>> bh.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

bh107.**empty**(*shape*, *dtype=<type 'numpy.float64'>*)
Return a new matrix of given shape and type, without initializing entries.

Parameters

shape [int or tuple of int] Shape of the empty matrix.

dtype [data-type, optional] Desired output data-type.

See also:

empty_like, ***zeros***

Notes

The order of the data in memory is always row-major (C-style).

empty, unlike *zeros*, does not set the matrix values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

bh107.**zeros**(*shape*, *dtype=<type 'float'>*)
Array of zeros.

Return an array of given shape and type, filled with zeros.

Parameters

shape [{sequence of ints, int}] Shape of the array

dtype [data-type, optional] The desired data-type for the array, default is np.float64.

Returns

out [bharray] Array of zeros of given shape, dtype, and order.

bh107.**ones**(*shape*, *dtype=<type 'numpy.float64'>*)
Array of ones.

Return an array of given shape and type, filled with ones.

Parameters

shape [{sequence of ints, int}] Shape of the array

dtype [data-type, optional] The desired data-type for the array, default is np.float64.

Returns

out [bharray] Array of ones of given shape, dtype, and order.

bh107.**empty_like**(*a*, *dtype=None*)
Return a new array with the same shape and type as a given array.

Parameters

a [array_like] The shape and data-type of *a* define these same attributes of the returned array.

dtype [data-type, optional] Overrides the data type of the result.

Returns

out [ndarray] Array of uninitialized (arbitrary) data with the same shape and type as *a*.

See also:

ones_like Return an array of ones with shape and type of input.

zeros_like Return an array of zeros with shape and type of input.

empty Return a new uninitialized array.

ones Return a new array setting values to one.

zeros Return a new array setting values to zero.

Notes

The order of the data in memory is always row-major (C-style).

This function does *not* initialize the returned array; to do that use *zeros_like* or *ones_like* instead. It may be marginally faster than the functions that do set the array values.

Examples

```
>>> a = ([1,2,3], [4,5,6])                         # a is array-like
>>> bh.empty_like(a)
array([[-1073741821, -1073741821,           3],     #random
       [          0,           0, -1073741821]])
>>> a = np.array([[1., 2., 3.],[4.,5.,6.]])
>>> bh.empty_like(a)
array([[ -2.00000715e+000,   1.48219694e-323,  -2.00000572e+000],#random
       [  4.38791518e-305,  -2.00000715e+000,   4.17269252e-309]])
```

bh107.**zeros_like**(*a*, *dtype=None*)

Return an array of zeros with the same shape and type as a given array.

With default parameters, is equivalent to `a.copy().fill(0)`.

Parameters

a [array_like] The shape and data-type of *a* define these same attributes of the returned array.

dtype [data-type, optional] Overrides the data type of the result.

Returns

out [ndarray] Array of zeros with the same shape and type as *a*.

See also:

ones_like Return an array of ones with shape and type of input.

empty_like Return an empty array with shape and type of input.

zeros Return a new array setting values to zero.

ones Return a new array setting values to one.

empty Return a new uninitialized array.

Notes

The order of the data in memory is always row-major (C-style).

Examples

```
>>> x = bh.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> bh.zeros_like(x)
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y = bh.arange(3, dtype=bh.float)
>>> y
array([ 0.,  1.,  2.])
>>> bh.zeros_like(y)
array([ 0.,  0.,  0.])
```

bh107.**ones_like**(*a*, *dtype=None*)

Return an array of ones with the same shape and type as a given array.

With default parameters, is equivalent to `a.copy().fill(1)`.

Parameters

a [array_like] The shape and data-type of *a* define these same attributes of the returned array.

dtype [data-type, optional] Overrides the data type of the result.

Returns

out [ndarray] Array of zeros with the same shape and type as *a*.

See also:

zeros_like Return an array of zeros with shape and type of input.

empty_like Return an empty array with shape and type of input.

zeros Return a new array setting values to zero.

ones Return a new array setting values to one.

empty Return a new uninitialized array.

Notes

The order of the data in memory is always row-major (C-style).

Examples

```
>>> x = bh.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> bh.ones_like(x)
array([[1, 1, 1],
       [1, 1, 1]])
```

```
>>> y = bh.arange(3, dtype=bh.float)
>>> y
array([ 0.,  1.,  2.])
>>> bh.ones_like(y)
array([ 1.,  1.,  1.])
```

Random

class bh107.random.**RandomState**(*seed=None*)

Methods

exponential(self[, scale, shape])	Exponential distribution.
get_state(self)	Return a tuple representing the internal state of the generator.
rand(self, *shape)	Random values in a given shape.
randint(self, low[, high, shape])	Return random integers from *low* (inclusive) to *high* (exclusive).
random(self[, shape])	Return random floats in the half-open interval [0.0, 1.0).
random123(self, shape)	New array of uniform pseudo numbers based on the random123 philox2x32 algorithm.
random_integers(self, low[, high, shape])	Return random integers between *low* and *high*, inclusive.
random_of_dtype(self, dtype[, shape])	Return random array of *dtype*.
random_sample(self, shape)	Return random floats in the half-open interval [0.0, 1.0).
ranf(self[, shape])	Return random floats in the half-open interval [0.0, 1.0).
sample(self[, shape])	Return random floats in the half-open interval [0.0, 1.0).
seed(self[, seed])	Seed the generator.
set_state(self, state)	Set the internal state of the generator from a tuple.
standard_exponential(self[, shape])	Draw samples from the standard exponential distribution.
uniform(self[, low, high, shape])	Draw samples from a uniform distribution.

exponential(*self*, *scale=1.0*, *shape=None*)

Exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1], or the time between page requests to Wikipedia [2].

Parameters

scale [float] The scale parameter, $\beta = 1/\lambda$.

shape [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns

out [BhArray] Drawn samples.

References

[1], [2], [3]

get_state(*self*)

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

Returns

out [tuple(str, np.uint64, np.uint32)] The returned tuple has the following items:

1. the string 'Random123'.
2. an integer `index`.
3. an integer `key`.

See also:

set_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in Bohrium. If the internal state is manually altered, the user should know exactly what he/she is doing.

rand(*self*, **shape*)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

Parameters

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

Returns

out [BhArray, shape `(d0, d1, ..., dn)`] Random values.

See also:

random

Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

randint(*self*, *low*, *high=None*, *shape=None*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

shape [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [BhArray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

See also:

random.random_integers similar to *randint*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

random(*self*, *shape=None*)

Return random floats in the half-open interval [0.0, 1.0).

Alias for *random_sample*

random123(*self*, *shape*)

New array of uniform pseudo numbers based on the random123 philox2x32 algorithm. NB: dtype is np.uint64 always

Parameters

shape [int or tuple of ints]

Defines the shape of the returned array of random floats.

Returns

out [Array of uniform pseudo numbers]

random_integers(*self*, *low*, *high=None*, *shape=None*)

Return random integers between *low* and *high*, inclusive.

Return random integers from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*].

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

shape [tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [BhArray of ints] *size*-shaped array of random integers from the appropriate distribution.

See also:

random.randint Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (bh107.random.random_integers(N) - 1) / (N - 1.)
```

Examples

```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3.,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

random_of_dtype (*self*, *dtype*, *shape=None*)

Return random array of *dtype*. The values are in the interval of the *dtype*.

Parameters

dtype [data-type] The desired data-type for the array.

shape [int or tuple of ints] Defines the shape of the returned array of random floats.

Returns

out [BhArray of floats] Array of random floats of shape *shape*.

random_sample (*self*, *shape*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add a:

```
(b - a) * random() + a
```

Parameters

shape [int or tuple of ints] Defines the shape of the returned array of random floats.

Returns

out [BhArray of floats] Array of random floats of shape *shape*.

Examples

```
>>> np.random.random((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

ranf (*self*, *shape=None*)

Return random floats in the half-open interval [0.0, 1.0).

Alias for *random_sample*

sample (*self*, *shape=None*)

Return random floats in the half-open interval [0.0, 1.0).

Alias for *random_sample*

seed(*self*, *seed=None*)
Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

Parameters

seed [int or array_like, optional] Seed for *RandomState*.

See also:

RandomState

set_state(*self*, *state*)
Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

Parameters

state [tuple(str, np.uint64, np.uint32)] The returned tuple has the following items:

1. the string 'Random123'.
2. an integer `index`.
3. an integer `key`.

Returns

out [None] Returns 'None' on success.

See also:

get_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in Bohrium. If the internal state is manually altered, the user should know exactly what he/she is doing.

standard_exponential(*self*, *shape=None*)
Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters

shape [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [BhArray] Drawn samples.

Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

uniform(*self*, *low=0.0*, *high=1.0*, *shape=None*)
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

low [float, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

shape [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out [BhArray] Drawn samples, with shape *shape*.

See also:

randint Discrete uniform distribution, yielding integers.

random_integers Discrete uniform distribution over the closed interval `[low, high]`.

random_sample Floats uniformly distributed over `[0, 1)`.

random Alias for *random_sample*.

rand Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

same as: random_sample(size) * (high - low) + low

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

User Kernels

bh107.user_kernel.**dtype_to_c99**(*dtype*)
Returns the C99 name of *dtype*

bh107.user_kernel.**execute**(*kernel_source*, *operand_list*, *compiler_command=None*, *tag='openmp'*, *param=None*, *only_behaving_operands=True*)
Compile and execute the function *execute()* with the arguments in *operand_list*

Parameters

kernel_source [str] The kernel source code that most define the function *execute()* that should take arguments corresponding to the *operand_list*

operand_list [list of bohrium arrays] The arrays given to the *execute()* function defined in *kernel_source*

compiler_command [str, optional] The compiler command to use when comping the kernel. *{OUT}* and *{IN}* in the command are replaced with the name of the binary and source path. When this options isn't specified, the default command are used see *get_default_compiler_command()*.

tag [str, optional] Name of the backend that should handle this kernel.

param [dict, optional] Backend specific parameters (e.g. OpenCL needs *global_work_size* and *local_work_size*).

only_behaving_operands [bool, optional]

Set to False in order to allow non-behaving operands. Requirements for a behaving array:

- Is a bohrium array
- Is C-style contiguous
- Points to the first element in the underlying base array (no offset)
- Has the same total length as its base

See *make_behaving()*

Examples

Simple addition kernel import bohrium as bh kernel = r''' #include <stdint.h> void execute(double *a, double *b, double *c) {

for(uint64_t i=0; i<100; ++i) { c[i] = a[i] + b[i] + i;

}

}''' a = bh107.ones(100, np.double) b = bh107.ones(100, np.double) res = bh107.empty_like(a) bh107.user_kernel.execute(kernel, [a, b, res])

bh107.user_kernel.**gen_function_prototype**(*operand_list*, *operand_name_list=None*)
Returns the *execute() definition based on the arrays in 'operand_list*

bh107.user_kernel.**get_default_compiler_command**()
Returns the default compiler command, which is the one typically extended with extra link commands

bh107.user_kernel.**make_behaving**(*ary*, *dtype=None*)
Make sure that *ary* is a "behaving" bohrium array of type *dtype*.

Requirements for a behaving array:

- Is a bohrium array
- Points to the first element in the underlying base array (no offset)
- Has the same total length as its base

Parameters

ary [BhArray] The array to make behaving

dtype [boolean, optional] The return array is converted to *dtype* if not None

Returns

A behaving BhArray that might be a copy of *ary*

2.2.2 C++ library

The C++ interface of Bohrium is similar to NumPy but is still very basic.

Indexing / Slicing

Bohrium C++ only support single index indexing:

```
// Create a new empty array (4 by 5)
bhxx::BhArray<double> A = bhxx::empty<double>({4, 5});
// Create view of the third row of A
bhxx::BhArray<double> B = A[2];
```

If you need more flexible slicing, you can set the shape and stride manually:

```
// Create a new array (4 by 5) of ones
bhxx::BhArray<double> A = bhxx::ones<double>({4, 5});
// Create view of the complete A.
bhxx::BhArray<double> B = A;
// B is now a 2 by 5 view with a step of two in the first dimension.
// In NumPy, this corresponds to: `B = A[::2, :]`
B.setShapeAndStride({2, 5}, {10, 1});
```

Code Snippets

You can find some examples in the source tree and some code snippets here:

```
#include<bhxx/bhxx.hpp>

/** Return a new empty array */
bhxx::BhArray<double> A = bhxx::empty<double>({4, 5});

/** Return the rank (number of dimensions) of the array */
int rank = A.rank();

/** Return the offset of the array */
uint64_t offset = A.offset();

/** Return the shape of the array */
Shape shape = A.shape();

/** Return the stride of the array */
Stride stride = A.stride();

/** Return the total number of elements of the array */
uint64_t size = A.size();

/** Return a pointer to the base of the array */
std::shared_ptr<BhBase> base = A.base();

/** Return whether the view is contiguous and row-major */
bool is_contig = A.isContiguous();

/** Return a new copy of the array */
bhxx::BhArray<double> copy = A.copy();

/** Return a copy of the array as a standard vector */
std::vector<double> vec = A.vec();

/** Print the content of A */
std::cout << A << "\n";

// Return a new transposed view
bhxx::BhArray<double> A_T = A.transpose();

// Return a new reshaped view (the array must be contiguous)
bhxx::BhArray<double> A_reshaped = A.reshape(Shape shape);

/** Return a new view with a "new axis" inserted.
 *
 *  The "new axis" is inserted just before `axis`.
 *  If negative, the count is backwards
 */
bhxx::BhArray<double> A_new_axis = A.newAxis(1);

// Return a new empty array
auto A = bhxx::empty<float>({3,4});

// Return a new empty array that has the same shape as `ary`
auto B = bhxx::empty_like<float>(A);
```

(continues on next page)

(continued from previous page)

```
// Return a new array filled with zeros
auto A = bhxx::zeros<float>({3,4});

// Return evenly spaced values within a given interval.
auto A = bhxx::arange(1, 3, 2); // start, stop, step
auto A = bhxx::arange(1, 3); // start, stop, step=1
auto A = bhxx::arange(3); // start=0, stop, step=1

// Random array, interval [0.0, 1.0)
auto A = bhxx::random.randn<double>({3, 4});

// Element-wise `static_cast`.
bhxx::BhArray<int> B = bhxx::cast<int>(A);

// Alias, A and B points to the same underlying data.
bhxx::empty<float> A = bhxx::empty<float>({3,4});
bhxx::empty<float> B = A;

// a is an alias
void add_inplace(bhxx::BhArray<double> a,
                 bhxx::BhArray<double> b) {
    a += b;
}
add_inplace(A, B);

// Create the data of A into a new array B.
bhxx::empty<float> A = bhxx::empty<float>({3,4});
bhxx::empty<float> B = A.copy();

// Copy the data of B into the existing array A.
A = B;

// Copying and converting the data of A into C.
bhxx::empty<double> C = bhxx::cast<double>(A);

// Alias, A and B points to the same underlying data.
bhxx::empty<float> A = bhxx::empty<float>({3,4});
bhxx::empty<float> B = bhxx::empty<float>({4});
B.reset(A);

// Evaluation triggers:
bhxx::flush();
std::cout << A << "\n";
A.vec();
A.data();

// Operator overloads
A + B - C * E / G;

// Standard functions
bhxx::sin(A) + bhxx::cos(B) + bhxx::sqrt(C) + ...

// Reductions (sum, product, maximum, etc.)
bhxx::add_reduce(A, 0); // Sum of axis 0
bhxx::multiply_reduce(B, 1); // Product of axis 1
bhxx::maximum_reduce(C, 2); // Maximum of axis 2
```

The API

The following is the complete API as defined in the header file:

template <typename *T*>
class BhArray

#include <BhArray.hpp> Representation of a multidimensional array that point to a `BhBase` array.

Template Parameters

- `T`: The data type of the array and the underlying base array

Inherits from *bhxx::BhArrayUnTypedCore*

Public Types

typedef T **scalar_type**

The data type of each array element.

Public Functions

BhArray()

Default constructor that leave the instance completely uninitialized.

BhArray(*Shape shape*, *Stride stride*)

Create a new array. `Shape` and `Stride` must have the same length.

Parameters

- `shape`: Shape of the new array
- `stride`: Stride of the new array

BhArray(*Shape shape*)

Create a new array (contiguous stride, row-major)

BhArray(*std*::shared_ptr<*BhBase*> *base*, *Shape shape*, *Stride stride*, uint64_t *offset* = 0)

Create a array that points to the given base

Note The caller should make sure that the shared pointer uses the RuntimeDeleter as its deleter, since this is implicitly assumed throughout, i.e. if one wants to construct a *BhBase* object, use the make_base_ptr helper function.

BhArray(*std*::shared_ptr<*BhBase*> *base*, *Shape shape*)

Create a view that points to the given base (contiguous stride, row-major)

Note The caller should make sure that the shared pointer uses the RuntimeDeleter as its deleter, since this is implicitly assumed throughout, i.e. if one wants to construct a *BhBase* object, use the make_base_ptr helper function.

template <**typename** InType, **typename** std::enable_if< type_traits::is_safe_numeric_cast< *scalar_type*, InType >::value,
BhArray(**const** *BhArray*<InType> &*ary*)

Create a copy of `ary` using a Bohrium `identity` operation, which copies the underlying array data.

Note This function implements implicit type conversion for all widening type casts

BhArray(**const** *BhArray*&)
Copy constructor that only copies meta data. The underlying array data is untouched

BhArray(*BhArray*&&)
Move constructor that only moves meta data. The underlying array data is untouched

BhArray<T> &**operator=**(**const** *BhArray*<T> &*other*)
Copy the data of `other` into the array using a Bohrium `identity` operation

BhArray<T> &**operator=**(*BhArray*<T> &&*other*)
Copy the data of `other` into the array using a Bohrium `identity` operation

Note A move assignment is the same as a copy assignment.

template <**typename** InType, **typename** std::enable_if< type_traits::is_arithmetic< InType >::value, int >::type = 0>
BhArray<T> &**operator=**(**const** InType &*scalar_value*)
Copy the scalar of `scalar_value` into the array using a Bohrium `identity` operation

BhArray<T> **copy**() **const**
Return a new copy of the array using a Bohrium `identity` operation

void **reset**(*BhArray*<T> *ary*)
Reset the array to `ary`

void **reset**()
Reset the array by cleaning all meta data and leave the array uninitialized.

int **rank**() **const**
Return the rank (number of dimensions) of the array

uint64_t **size**() **const**
Return the total number of elements of the array

bool **isContiguous**() **const**
Return whether the view is contiguous and row-major

bool **isDataInitialised**() **const**
Is the data referenced by this view's base array already allocated, i.e. initialised

const T ***data**(bool *flush* = true) **const**
Obtain the data pointer of the array, not taking ownership of any kind.

Return The data pointer that might be a nullptr if the data in the base data is not initialised.

Parameters

- `flush`: Should we flush the runtime system before retrieving the data pointer

T ***data**(bool *flush* = true)
The non-const version of *.data()*

std::vector<T> **vec**() **const**
Return a copy of the array as a standard vector

Note The array must be contiguous

void **pprint**(*std*::ostream &*os*, int *current_nesting_level*, int *max_nesting_level*) **const**
Pretty printing the content of the array

Parameters

- `os`: The output stream to write to.
- `current_nesting_level`: The nesting level to print at (typically `0`).
- `max_nesting_level`: The maximum nesting level to print at (typically *rank()*`-1`).

BhArray<T> **operator[]**(int64_t *idx*) **const**
Returns a new view of the `idx` dimension. Negative index counts from the back.

BhArray<T> **transpose**() **const**
Return a new transposed view.

BhArray<T> **reshape**(*Shape* *shape*) **const**
Return a new reshaped view (the array must be contiguous)

BhArray<T> **newAxis**(int *axis*) **const**
Return a new view with a "new axis" inserted.

Return The new array

Parameters

- `axis`: The "new axis" is inserted just before `axis`. If negative, the count is backwards (e.g -1 insert a "new axis" at the end of the array)

class BhArrayUnTypedCore
#include <BhArray.hpp> Core class that represent the core attributes of a view that isn't typed by its dtype

Subclassed by *bhxx::BhArray< T >*

Public Functions

BhArrayUnTypedCore()
Default constructor that leave the instance completely uninitialized

BhArrayUnTypedCore(uint64_t *offset*, *Shape* *shape*, *Stride* *stride*, *std*::shared_ptr<*BhBase*> *base*)
Constructor to initiate all but the `_slides` attribute

bh_view **getBhView**() **const**
Return a `bh_view` of the array

uint64_t **offset**() **const**
Return the offset of the array

const *Shape* &**shape**() **const**
Return the shape of the array

const *Stride* &**stride**() **const**
Return the stride of the array

const *std*::shared_ptr<*BhBase*> &**base**() **const**
Return the base of the array

std::shared_ptr<*BhBase*> &**base**()
Return the base of the array

void **setShapeAndStride**(*Shape* *shape*, *Stride* *stride*)
Set the shape and stride of the array (both must have the same length)

const bh_slide &**slides**() **const**
Return the slides object of the array

bh_slide &**slides**()
Return the slides object of the array

Protected Attributes

uint64_t **_offset** = 0
The array offset (from the start of the base in number of elements)

Shape **_shape**
The array shape (size of each dimension in number of elements)

Stride **_stride**
The array stride (the absolute stride of each dimension in number of elements)

std::shared_ptr<*BhBase*> **_base**
Pointer to the base of this array.

bh_slide **_slides**
Metadata to support sliding views.

Friends

void **swap**(BhArrayUnTypedCore &*a*, BhArrayUnTypedCore &*b*)
Swapping `a` and `b`

class BhBase
#include <*BhBase.hpp*> The base underlying (multiple) arrays

Inherits from bh_base

Public Functions

bool **ownMemory**()
Is the memory managed referenced by bh_base's data pointer managed by Bohrium or is it owned externally

Note If this flag is false, the class will make sure that the memory is not deleted when going out of scope.

template <typename T>
BhBase(size_t *nelem*, T **memory*)
Construct a base array with nelem elements using externally managed storage.

The class will make sure, that the storage is not deleted when going out of scope. Needless to say that the memory should be large enough to incorporate nelem_ elements.

Template Parameters

- `T`: The type of each element

Parameters

- `nelem`: Number of elements
- `memory`: Pointer to the external memory

template <typename InputIterator, **typename** T = typename std::iterator_traits<InputIterator>::value_type>
BhBase(InputIterator *begin*, InputIterator *end*)

Construct a base array and initialise it with the elements provided by an iterator range.

The values are copied into the Bohrium storage. If you want to provide external storage to Bohrium use the constructor *BhBase(size_t nelem, T* memory)* instead.

template <typename T>
BhBase(T *dummy*, size_t *nelem*)

Construct a base array with `nelem` elements

Note The use of this particular constructor is discouraged. It is only needed from *BhArray* to construct base objects which are uninitialised and do not yet hold any deta. If you wish to construct an uninitialised *BhBase* object, do this via the *BhArray* interface and not using this constructor.

Parameters

- `dummy`: Dummy argument to fix the type of elements used. It may only have ever have the value 0 in the appropriate type.
- `nelem`: Number of elements

~BhBase()

Destructor

BhBase(**const** *BhBase*&)

Deleted copy constructor

BhBase &**operator=**(**const** *BhBase*&)

Deleted copy assignment

BhBase &**operator=**(*BhBase* &&*other*)

Delete move assignment

BhBase(*BhBase* &&*other*)

Move another *BhBase* object here

Private Members

bool **m_own_memory**

class Random

#include <random.hpp> *Random* class that maintain the state of the random number generation

Public Functions

Random(uint64_t *seed* = std::random_device{}())

Create a new random instance

Parameters

- `seed`: T he seed of the random number generation. If not set, `std::random_device` is used.

BhArray<uint64_t> **random123** (uint64_t *size*)
New 1D random array using the Random123 algorithm https://www.deshawresearch.com/resources_random123.html

Return The new random array

Parameters

- `size`: Size of the new 1D random array

void **reset** (uint64_t *seed* = std::random_device{}())
Reset the random instance

Parameters

- `seed`: The seed of the random number generation. If not set, `std::random_device` is used.

template <typename T>
BhArray<T> **randn** (*Shape shape*)
Return random floats in the half-open interval [0.0, 1.0) using Random123

Return Real array

Parameters

- `shape`: The shape of the returned array

Private Members

uint64_t **_seed**

uint64_t **_count** = 0

namespace bhxx

Typedefs

typedef BhStaticVector<uint64_t> **Shape**
Static allocated shape that is interchangeable with standard C++ vector as long as the vector is smaller than `BH_MAXDIM`.

typedef BhStaticVector<int64_t> **Stride**
Static allocated stride that is interchangeable with standard C++ vector as long as the vector is smaller than `BH_MAXDIM`.

Functions

template <typename T>
BhArray<T> **arange** (int64_t *start*, int64_t *stop*, int64_t *step*)
Return evenly spaced values within a given interval.

Return New 1D array

Template Parameters

- `T`: Data type of the returned array

Parameters

- `start`: Start of interval. The interval includes this value.
- `stop`: End of interval. The interval does not include this value.
- `step`: Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i].

void **flush**()
Force the execution of all lazy evaluated array operations

template <typename T>
BhArray<T> **empty**(*Shape shape*)
Return a new empty array

Return The new array

Template Parameters

- `T`: The data type of the new array

Parameters

- `shape`: The shape of the new array

template <typename OutType, **typename** InType>
BhArray<OutType> **empty_like**(**const** *bhxx*::*BhArray*<InType> &*ary*)
Return a new empty array that has the same shape as `ary`

Return The new array

Template Parameters

- `OutType`: The data type of the returned new array
- `InType`: The data type of the input array

Parameters

- `ary`: The array to take the shape from

template <typename T>
BhArray<T> **full**(*Shape shape*, T *value*)
Return a new array filled with `value`

Return The new array

Template Parameters

- `T`: The data type of the new array

Parameters

- `shape`: The shape of the new array
- `value`: The value to fill the new array with

template <typename T>
BhArray<T> **zeros**(*Shape shape*)
Return a new array filled with zeros

Return The new array

Template Parameters

- `T`: The data type of the new array

Parameters

- `shape`: The shape of the new array

template <typename T>
BhArray<T> **ones** (*Shape shape*)
Return a new array filled with ones

Return The new array

Template Parameters

- `T`: The data type of the new array

Parameters

- `shape`: The shape of the new array

template <typename T>
BhArray<T> **arange** (int64_t *start*, int64_t *stop*)
Return evenly spaced values within a given interval using steps of 1.

Return New 1D array

Template Parameters

- `T`: Data type of the returned array

Parameters

- `start`: Start of interval. The interval includes this value.
- `stop`: End of interval. The interval does not include this value.

template <typename T>
BhArray<T> **arange** (int64_t *stop*)
Return evenly spaced values from 0 to `stop` using steps of 1.

Return New 1D array

Template Parameters

- `T`: Data type of the returned array

Parameters

- `stop`: End of interval. The interval does not include this value.

template <typename OutType, **typename** InType>
BhArray<OutType> **cast** (**const** *bhxx*::*BhArray*<InType> &*ary*)
Element-wise `static_cast`.

Return New array

Template Parameters

- `OutType`: The data type of the returned array
- `InType`: The data type of the input array

Parameters

- `ary`: Input array to cast

Stride **contiguous_stride**(**const** *Shape* &*shape*)
Return a contiguous stride (row-major) based on `shape`

template <**typename** T>
std::ostream &**operator<<**(*std*::ostream &*os*, **const** *BhArray*<T> &*ary*)
Pretty printing the data of an array to a stream Example:

```
auto A = bhxx::arange<double>(3);
std::cout << A << std::endl;
```

Return A reference to `os`

Template Parameters

- `T`: The data of `ary`

Parameters

- `os`: The output stream to write to
- `ary`: The array to print

template <**typename** T>
BhArray<T> **as_contiguous**(*BhArray*<T> *ary*)
Create an contiguous view or a copy of an array. The array is only copied if it isn't already contiguous.

Return Either a view of `ary` or a new copy of `ary`.

Template Parameters

- `T`: The data type of `ary`.

Parameters

- `ary`: The array to make contiguous.

template <int *N*>
Shape **broadcasted_shape**(*std*::array<*Shape*, N> *shapes*)
Return the result of broadcasting `shapes` against each other

Return Broadcasted shape

Parameters

- `shapes`: Array of shapes

template <**typename** T>
BhArray<T> **broadcast_to**(*BhArray*<T> *ary*, **const** *Shape* &*shape*)
Return a new view of `ary` that is broadcasted to `shape` We use the term broadcast as defined by NumPy. Let `ret` be the broadcasted view of `ary`: 1) One-sized dimensions are prepended to `ret.shape()` until it has the same number of dimension as `ary`. 2) The stride of each one-sized dimension in `ret` is set to zero. 3) The shape of `ary` is set to `shape`

Note See: https://docs.scipy.org/doc/numpy-1.15.0/user/basics.broadcasting.html

Return The broadcasted array

Parameters

- `ary`: Input array
- `shape`: The new shape

template <**typename** T1, **typename** T2>
bool **is_same_array**(**const** *BhArray*<T1> &*a*, **const** *BhArray*<T2> &*b*)
Check whether `a` and `b` are the same view pointing to the same base

Return The boolean answer.

Template Parameters

- `T1`: The data type of `a`.
- `T2`: The data type of `b`.

Parameters

- `a`: The first array to compare.
- `b`: The second array to compare.

template <**typename** T1, **typename** T2>
bool **may_share_memory**(**const** *BhArray*<T1> &*a*, **const** *BhArray*<T2> &*b*)
Check whether `a` and `b` can share memory

Note A return of True does not necessarily mean that the two arrays share any element. It just means that they *might*.

Return The boolean answer.

Template Parameters

- `T1`: The data type of `a`.
- `T2`: The data type of `b`.

Parameters

- `a`: The first array to compare.
- `b`: The second array to compare.

BhArray<bool> **add**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **add**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **add**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<double>> **add**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<double>> **add**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)

Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<double>> **add**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<float>> **add**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<float>> **add**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)

Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<float>> **add**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<float> **add**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **add**(**const** *BhArray*<float> &*in1*, float *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **add**(float *in1*, **const** *BhArray*<float> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **add**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **add**(**const** *BhArray*<double> &*in1*, double *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<double> **add**(double *in1*, **const** *BhArray*<double> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **add**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **add**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **add**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **add**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **add**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **add**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **add**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **add**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **add**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **add**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **add**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **add**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **add**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **add**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **add**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **add**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint32_t> **add**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **add**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **add**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint64_t> **add**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint64_t> **add**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint8_t> **add**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint8_t> **add**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint8_t> **add**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Add arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.

- in2: Array input.

BhArray<*std*::complex<double>> **subtract**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<double>> **subtract**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<double>> **subtract**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<float>> **subtract**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<float>> **subtract**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<float>> **subtract**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<float> **subtract**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<float> **subtract**(**const** *BhArray*<float> &*in1*, float *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<float> **subtract**(float *in1*, **const** *BhArray*<float> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<double> **subtract**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<double> **subtract**(**const** *BhArray*<double> &*in1*, double *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<double> **subtract**(double *in1*, **const** *BhArray*<double> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **subtract**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **subtract**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **subtract**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **subtract**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **subtract**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int32_t> **subtract**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.

- in2: Array input.

BhArray<int64_t> **subtract** (**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **subtract** (**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **subtract** (int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **subtract** (**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **subtract** (**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **subtract** (int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **subtract**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **subtract**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **subtract**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **subtract**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **subtract**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **subtract**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **subtract**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **subtract**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **subtract**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **subtract**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **subtract**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **subtract**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Subtract arguments, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **multiply**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **multiply**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **multiply**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<double>> **multiply**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<double>> **multiply**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<double>> **multiply**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<float>> **multiply**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<*std*::complex<float>> **multiply**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<*std*::complex<float>> **multiply**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<float> **multiply**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **multiply**(**const** *BhArray*<float> &*in1*, float *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **multiply**(float *in1*, **const** *BhArray*<float> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **multiply**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **multiply**(**const** *BhArray*<double> &*in1*, double *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<double> **multiply**(double *in1*, **const** *BhArray*<double> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **multiply**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **multiply**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **multiply**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **multiply**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **multiply**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int32_t> **multiply**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int64_t> **multiply**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **multiply**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **multiply**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **multiply**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **multiply**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **multiply**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **multiply**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **multiply**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **multiply**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **multiply**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.

- in2: Array input.

BhArray<uint32_t> **multiply**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **multiply**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **multiply**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint64_t> **multiply**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint64_t> **multiply**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint8_t> **multiply**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint8_t> **multiply**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **multiply**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Multiply arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<*std*::complex<double>> **divide**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<*std*::complex<double>> **divide**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<*std*::complex<double>> **divide**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<*std*::complex<float>> **divide**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.

- in2: Array input.

BhArray<*std*::complex<float>> **divide**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<float>> **divide**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<float> **divide**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<float> **divide**(**const** *BhArray*<float> &*in1*, float *in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<float> **divide**(float *in1*, **const** *BhArray*<float> &*in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<double> **divide**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)

Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.

- in2: Array input.

BhArray<double> **divide**(**const** *BhArray*<double> &*in1*, double *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<double> **divide**(double *in1*, **const** *BhArray*<double> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **divide**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **divide**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **divide**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **divide**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **divide** (**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **divide** (int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **divide** (**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **divide** (**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **divide** (int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **divide** (**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **divide** (**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **divide** (int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **divide** (**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **divide** (**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **divide** (uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **divide** (**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **divide** (**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **divide**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **divide**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint64_t> **divide**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint64_t> **divide**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint8_t> **divide**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint8_t> **divide**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint8_t> **divide**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Divide arguments element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<double>> **power**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<double>> **power**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<*std*::complex<double>> **power**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<*std*::complex<float>> **power**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<float>> **power**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<*std*::complex<float>> **power**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<float> **power**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **power**(**const** *BhArray*<float> &*in1*, float *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **power**(float *in1*, **const** *BhArray*<float> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **power**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **power**(**const** *BhArray*<double> &*in1*, double *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<double> **power**(double *in1*, **const** *BhArray*<double> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **power**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **power**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **power**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **power**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **power**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int32_t> **power** (int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int64_t> **power** (**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **power** (**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **power** (int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **power** (**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **power** (**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.

- in2: Scalar input.

BhArray<int8_t> **power** (int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **power** (**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **power** (**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **power** (uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **power** (**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint32_t> **power** (**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **power**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **power**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **power**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **power**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **power**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **power**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **power**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
First array elements raised to powers from second array, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **absolute**(**const** *BhArray*<bool> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **absolute**(**const** *BhArray*<float> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **absolute**(**const** *BhArray*<double> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **absolute**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **absolute**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<int16_t> **absolute**(**const** *BhArray*<int16_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<int32_t> **absolute**(**const** *BhArray*<int32_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int64_t> **absolute**(**const** *BhArray*<int64_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int8_t> **absolute**(**const** *BhArray*<int8_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint16_t> **absolute**(**const** *BhArray*<uint16_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint32_t> **absolute**(**const** *BhArray*<uint32_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint64_t> **absolute**(**const** *BhArray*<uint64_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint8_t> **absolute**(**const** *BhArray*<uint8_t> &*in1*)
Calculate the absolute value element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **greater**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.

- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<double> &*in1*, double *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(double *in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Array input.

- in2: Scalar input.

BhArray<bool> **greater**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 > in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater_equal**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater_equal**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<double> &*in1*, double *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(double *in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **greater_equal**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater_equal**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater_equal**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **greater_equal**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **greater_equal**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 >= in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.

- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<double> &*in1*, double *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(double *in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 < in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.

- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<double> &*in1*, double *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(double *in1*, **const** *BhArray*<double> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **less_equal**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less_equal**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less_equal**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less_equal**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **less_equal**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **less_equal**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- in1: Array input.

- `in2`: Scalar input.

BhArray<bool> **less_equal**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the truth value of (in1 =< in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<float> &*in1*, float *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(float *in1*, **const** *BhArray*<float> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<double> &*in1*, double *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(double *in1*, **const** *BhArray*<double> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal** (**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal** (**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal** (int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **equal** (**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **equal** (**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **equal** (int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **equal**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **equal**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **equal**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **equal**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **equal**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.

- in2: Array input.

BhArray<bool> **equal** (**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **equal** (**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **equal** (uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return (in1 == in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal** (**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal** (**const** *BhArray*<bool> &*in1*, bool *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal** (bool *in1*, **const** *BhArray*<bool> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<*std*::complex<double>> &*in1*, *std*::complex<double> *in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(*std*::complex<double> *in1*, **const** *BhArray*<*std*::complex<double>> &*in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<*std*::complex<float>> &*in1*, *std*::complex<float> *in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(*std*::complex<float> *in1*, **const** *BhArray*<*std*::complex<float>> &*in2*)

Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.

- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<float> &*in1*, float *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(float *in1*, **const** *BhArray*<float> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<double> &*in1*, double *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(double *in1*, **const** *BhArray*<double> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **not_equal**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **not_equal**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **not_equal**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return (in1 != in2) element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **logical_and**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 AND in2 elementwise.

Return Output array.

Parameters

- in1: Array input.

- in2: Array input.

BhArray<bool> **logical_and**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the truth value of in1 AND in2 elementwise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **logical_and**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 AND in2 elementwise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **logical_or**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 OR in2 elementwise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **logical_or**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the truth value of in1 OR in2 elementwise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **logical_or**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 OR in2 elementwise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **logical_xor**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 XOR in2, element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **logical_xor**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the truth value of in1 XOR in2, element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **logical_xor**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the truth value of in1 XOR in2, element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **logical_not**(**const** *BhArray*<bool> &*in1*)
Compute the truth value of NOT elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **maximum**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **maximum**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **maximum**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<float> **maximum**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<float> **maximum**(**const** *BhArray*<float> &*in1*, float *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<float> **maximum**(float *in1*, **const** *BhArray*<float> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<double> **maximum**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<double> **maximum**(**const** *BhArray*<double> &*in1*, double *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<double> **maximum**(double *in1*, **const** *BhArray*<double> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **maximum**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **maximum**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **maximum**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **maximum**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **maximum**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int32_t> **maximum**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int64_t> **maximum**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.

- in2: Array input.

BhArray<int64_t> **maximum**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **maximum**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **maximum**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **maximum**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **maximum**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **maximum**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **maximum**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **maximum**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **maximum**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **maximum**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **maximum**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **maximum**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **maximum**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **maximum**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **maximum**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **maximum**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **maximum**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Element-wise maximum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **minimum**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **minimum**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **minimum**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<float> **minimum**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<float> **minimum**(**const** *BhArray*<float> &*in1*, float *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<float> **minimum**(float *in1*, **const** *BhArray*<float> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<double> **minimum**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<double> **minimum**(**const** *BhArray*<double> &*in1*, double *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<double> **minimum**(double *in1*, **const** *BhArray*<double> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **minimum**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **minimum**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **minimum**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **minimum**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **minimum**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.

- in2: Scalar input.

BhArray<int32_t> **minimum**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int64_t> **minimum**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **minimum**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **minimum**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **minimum**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **minimum**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **minimum**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **minimum**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **minimum**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **minimum**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **minimum**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **minimum**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **minimum**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **minimum**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **minimum**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **minimum**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **minimum**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **minimum**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **minimum**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Element-wise minimum of array elements.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **bitwise_and**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<bool> **bitwise_and**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<bool> **bitwise_and**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **bitwise_and**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **bitwise_and**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **bitwise_and**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **bitwise_and**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **bitwise_and**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int32_t> **bitwise_and**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int64_t> **bitwise_and**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **bitwise_and**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int64_t> **bitwise_and**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.

- in2: Array input.

BhArray<int8_t> **bitwise_and**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **bitwise_and**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **bitwise_and**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **bitwise_and**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **bitwise_and**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **bitwise_and**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **bitwise_and**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **bitwise_and**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **bitwise_and**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **bitwise_and**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **bitwise_and**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **bitwise_and**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_and**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_and**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **bitwise_and**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise AND of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **bitwise_or**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **bitwise_or**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **bitwise_or**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **bitwise_or**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **bitwise_or**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **bitwise_or**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **bitwise_or**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **bitwise_or**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **bitwise_or**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **bitwise_or**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **bitwise_or**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **bitwise_or**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **bitwise_or**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **bitwise_or**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **bitwise_or**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **bitwise_or**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **bitwise_or**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **bitwise_or**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **bitwise_or**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint32_t> **bitwise_or**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **bitwise_or**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **bitwise_or**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **bitwise_or**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **bitwise_or**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_or**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_or**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **bitwise_or**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise OR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **bitwise_xor**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.

- `in2`: Array input.

BhArray<bool> **bitwise_xor**(**const** *BhArray*<bool> &*in1*, bool *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<bool> **bitwise_xor**(bool *in1*, **const** *BhArray*<bool> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **bitwise_xor**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **bitwise_xor**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **bitwise_xor**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **bitwise_xor**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **bitwise_xor**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **bitwise_xor**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **bitwise_xor**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **bitwise_xor**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **bitwise_xor**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **bitwise_xor**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **bitwise_xor**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **bitwise_xor**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **bitwise_xor**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **bitwise_xor**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **bitwise_xor**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **bitwise_xor**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **bitwise_xor**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **bitwise_xor**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **bitwise_xor**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **bitwise_xor**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **bitwise_xor**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_xor**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **bitwise_xor**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint8_t> **bitwise_xor**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Compute the bit-wise XOR of two arrays element-wise.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<bool> **invert**(**const** *BhArray*<bool> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int16_t> **invert**(**const** *BhArray*<int16_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int32_t> **invert**(**const** *BhArray*<int32_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int64_t> **invert**(**const** *BhArray*<int64_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int8_t> **invert**(**const** *BhArray*<int8_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint16_t> **invert**(**const** *BhArray*<uint16_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint32_t> **invert** (**const** *BhArray*<uint32_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint64_t> **invert** (**const** *BhArray*<uint64_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<uint8_t> **invert** (**const** *BhArray*<uint8_t> &*in1*)
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<int16_t> **left_shift** (**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **left_shift** (**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **left_shift** (int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **left_shift** (**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **left_shift** (**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **left_shift** (int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **left_shift** (**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **left_shift** (**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **left_shift** (int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **left_shift** (**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **left_shift**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **left_shift**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **left_shift**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **left_shift**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **left_shift**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint32_t> **left_shift**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint32_t> **left_shift**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **left_shift**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **left_shift**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint64_t> **left_shift**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint64_t> **left_shift**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint8_t> **left_shift**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint8_t> **left_shift**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint8_t> **left_shift**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Shift the bits of an integer to the left.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int16_t> **right_shift**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **right_shift**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int16_t> **right_shift**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int32_t> **right_shift**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.

- `in2`: Array input.

BhArray<int32_t> **right_shift**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **right_shift**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **right_shift**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **right_shift**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **right_shift**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **right_shift**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **right_shift**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **right_shift**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **right_shift**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **right_shift**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **right_shift**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **right_shift**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **right_shift**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint32_t> **right_shift**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint64_t> **right_shift**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint64_t> **right_shift**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint64_t> **right_shift**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint8_t> **right_shift**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint8_t> **right_shift**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **right_shift**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Shift the bits of an integer to the right.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<*std*::complex<double>> **cos**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Cosine elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **cos**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Cosine elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **cos**(**const** *BhArray*<float> &*in1*)
Cosine elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **cos**(**const** *BhArray*<double> &*in1*)
Cosine elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **sin**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Trigonometric sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **sin**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Trigonometric sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **sin**(**const** *BhArray*<float> &*in1*)
Trigonometric sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **sin**(**const** *BhArray*<double> &*in1*)
Trigonometric sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **tan**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Compute tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **tan**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Compute tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **tan**(**const** *BhArray*<float> &*in1*)
Compute tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **tan**(**const** *BhArray*<double> &*in1*)
Compute tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **cosh**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Hyperbolic cosine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<float>> **cosh**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Hyperbolic cosine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **cosh**(**const** *BhArray*<float> &*in1*)
Hyperbolic cosine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **cosh**(**const** *BhArray*<double> &*in1*)
Hyperbolic cosine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<double>> **sinh**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Hyperbolic sine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<float>> **sinh**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Hyperbolic sine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **sinh**(**const** *BhArray*<float> &*in1*)
Hyperbolic sine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **sinh**(**const** *BhArray*<double> &*in1*)
Hyperbolic sine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<double>> **tanh**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Compute hyperbolic tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **tanh**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Compute hyperbolic tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **tanh**(**const** *BhArray*<float> &*in1*)
Compute hyperbolic tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **tanh**(**const** *BhArray*<double> &*in1*)
Compute hyperbolic tangent element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **arcsin**(**const** *BhArray*<float> &*in1*)
Inverse sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **arcsin**(**const** *BhArray*<double> &*in1*)
Inverse sine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **arccos**(**const** *BhArray*<float> &*in1*)
Trigonometric inverse cosine, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **arccos**(**const** *BhArray*<double> &*in1*)
Trigonometric inverse cosine, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **arctan**(**const** *BhArray*<float> &*in1*)
Trigonometric inverse tangent, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **arctan**(**const** *BhArray*<double> &*in1*)
Trigonometric inverse tangent, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **arcsinh**(**const** *BhArray*<float> &*in1*)
Inverse hyperbolic sine elementwise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **arcsinh**(**const** *BhArray*<double> &*in1*)
Inverse hyperbolic sine elementwise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **arccosh**(**const** *BhArray*<float> &*in1*)
Inverse hyperbolic cosine, elementwise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **arccosh**(**const** *BhArray*<double> &*in1*)
Inverse hyperbolic cosine, elementwise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **arctanh**(**const** *BhArray*<float> &*in1*)
Inverse hyperbolic tangent elementwise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **arctanh**(**const** *BhArray*<double> &*in1*)
Inverse hyperbolic tangent elementwise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **arctan2**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **arctan2**(**const** *BhArray*<float> &*in1*, float *in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **arctan2**(float *in1*, **const** *BhArray*<float> &*in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **arctan2**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **arctan2**(**const** *BhArray*<double> &*in1*, double *in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<double> **arctan2**(double *in1*, **const** *BhArray*<double> &*in2*)
Element-wise arc tangent of `in1/in2` choosing the quadrant correctly.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<*std*::complex<double>> **exp**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Calculate the exponential of all elements in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **exp**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Calculate the exponential of all elements in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **exp**(**const** *BhArray*<float> &*in1*)
Calculate the exponential of all elements in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **exp**(**const** *BhArray*<double> &*in1*)
Calculate the exponential of all elements in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **exp2**(**const** *BhArray*<float> &*in1*)
Calculate `2**p` for all `p` in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **exp2**(**const** *BhArray*<double> &*in1*)
Calculate `2**p` for all `p` in the input array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **expm1**(**const** *BhArray*<float> &*in1*)
Calculate `exp(in1) - 1` for all elements in the array.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **expm1**(**const** *BhArray*<double> &*in1*)
Calculate exp(in1) - 1 for all elements in the array.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<double>> **log**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Natural logarithm, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<float>> **log**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Natural logarithm, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **log**(**const** *BhArray*<float> &*in1*)
Natural logarithm, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **log**(**const** *BhArray*<double> &*in1*)
Natural logarithm, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **log2**(**const** *BhArray*<float> &*in1*)
Base-2 logarithm of in1.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **log2**(**const** *BhArray*<double> &*in1*)
Base-2 logarithm of in1.

Return Output array.

Parameters

- in1: Array input.

BhArray<*std*::complex<double>> **log10**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Return the base 10 logarithm of the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **log10**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Return the base 10 logarithm of the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **log10**(**const** *BhArray*<float> &*in1*)
Return the base 10 logarithm of the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **log10**(**const** *BhArray*<double> &*in1*)
Return the base 10 logarithm of the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **log1p**(**const** *BhArray*<float> &*in1*)
Return the natural logarithm of one plus the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **log1p**(**const** *BhArray*<double> &*in1*)
Return the natural logarithm of one plus the input array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **sqrt**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Return the positive square-root of an array, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **sqrt**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Return the positive square-root of an array, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **sqrt**(**const** *BhArray*<float> &*in1*)
Return the positive square-root of an array, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **sqrt**(**const** *BhArray*<double> &*in1*)
Return the positive square-root of an array, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **ceil**(**const** *BhArray*<float> &*in1*)
Return the ceiling of the input, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **ceil**(**const** *BhArray*<double> &*in1*)
Return the ceiling of the input, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **trunc**(**const** *BhArray*<float> &*in1*)
Return the truncated value of the input, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **trunc**(**const** *BhArray*<double> &*in1*)
Return the truncated value of the input, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **floor**(**const** *BhArray*<float> &*in1*)
Return the floor of the input, element-wise.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **floor**(**const** *BhArray*<double> &*in1*)
Return the floor of the input, element-wise.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **rint**(**const** *BhArray*<float> &*in1*)
Round elements of the array to the nearest integer.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **rint**(**const** *BhArray*<double> &*in1*)
Round elements of the array to the nearest integer.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **mod**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **mod**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **mod**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **mod**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **mod**(**const** *BhArray*<double> &*in1*, double *in2*)

Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<double> **mod**(double *in1*, **const** *BhArray*<double> &*in2*)

Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **mod**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)

Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **mod**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)

Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **mod**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)

Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **mod**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **mod**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **mod**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **mod**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **mod**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **mod**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<int8_t> **mod**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the element-wise modulo, which is in1 % in2 in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **mod**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the element-wise modulo, which is in1 % in2 in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<int8_t> **mod**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the element-wise modulo, which is in1 % in2 in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Scalar input.
- in2: Array input.

BhArray<uint16_t> **mod**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the element-wise modulo, which is in1 % in2 in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **mod**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the element-wise modulo, which is in1 % in2 in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- in1: Array input.
- in2: Scalar input.

BhArray<uint16_t> **mod**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **mod**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **mod**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **mod**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **mod**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **mod**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **mod**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **mod**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **mod**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **mod**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the element-wise modulo, which is `in1 % in2` in Python and has the same sign as the divisor in2.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<bool> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<int8_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<int16_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<int32_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<int64_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<uint8_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<uint16_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<uint32_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<uint64_t> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<float> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isnan**(**const** *BhArray*<double> &*in1*)
Test for NaN values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<bool> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<int8_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<int16_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<int32_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<int64_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<uint8_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<uint16_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<uint32_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<uint64_t> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<float> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isinf**(**const** *BhArray*<double> &*in1*)
Test for infinity values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **add_reduce**(**const** *BhArray*<*std*::complex<double>> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<*std*::complex<float>> **add_reduce**(**const** *BhArray*<*std*::complex<float>> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<float> **add_reduce**(**const** *BhArray*<float> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<double> **add_reduce**(**const** *BhArray*<double> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int16_t> **add_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int32_t> **add_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int64_t> **add_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **add_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **add_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **add_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **add_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint8_t> **add_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Sums all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<*std*::complex<double>> **multiply_reduce**(**const** *BhArray*<*std*::complex<double>> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<*std*::complex<float>> **multiply_reduce**(**const** *BhArray*<*std*::complex<float>> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<float> **multiply_reduce**(**const** *BhArray*<float> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<double> **multiply_reduce**(**const** *BhArray*<double> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int16_t> **multiply_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int32_t> **multiply_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int64_t> **multiply_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int8_t> **multiply_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint16_t> **multiply_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint32_t> **multiply_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint64_t> **multiply_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.

- in2: The axis to run over.

BhArray<uint8_t> **multiply_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Multiplies all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<bool> **minimum_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<float> **minimum_reduce**(**const** *BhArray*<float> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<double> **minimum_reduce**(**const** *BhArray*<double> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int16_t> **minimum_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int32_t> **minimum_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int64_t> **minimum_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **minimum_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **minimum_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **minimum_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **minimum_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint8_t> **minimum_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Finds the smallest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<bool> **maximum_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<float> **maximum_reduce**(**const** *BhArray*<float> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<double> **maximum_reduce**(**const** *BhArray*<double> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int16_t> **maximum_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int32_t> **maximum_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int64_t> **maximum_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **maximum_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **`maximum_reduce`**(**`const`** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **`maximum_reduce`**(**`const`** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **`maximum_reduce`**(**`const`** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint8_t> **`maximum_reduce`**(**`const`** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Finds the largest elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<bool> **`logical_and_reduce`**(**`const`** *BhArray*<bool> &*in1*, int64_t *in2*)
Logical AND of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<bool> **`bitwise_and_reduce`**(**`const`** *BhArray*<bool> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int16_t> **bitwise_and_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int32_t> **bitwise_and_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int64_t> **bitwise_and_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int8_t> **bitwise_and_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint16_t> **bitwise_and_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint32_t> **bitwise_and_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.

- in2: The axis to run over.

BhArray<uint64_t> **bitwise_and_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint8_t> **bitwise_and_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Bitwise AND of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<bool> **logical_or_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Logical OR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<bool> **bitwise_or_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int16_t> **bitwise_or_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int32_t> **bitwise_or_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<int64_t> **bitwise_or_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **bitwise_or_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **bitwise_or_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **bitwise_or_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **bitwise_or_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint8_t> **bitwise_or_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Bitwise OR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<bool> **logical_xor_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Logical XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<bool> **bitwise_xor_reduce**(**const** *BhArray*<bool> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int16_t> **bitwise_xor_reduce**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int32_t> **bitwise_xor_reduce**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int64_t> **bitwise_xor_reduce**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **bitwise_xor_reduce**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **bitwise_xor_reduce**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint32_t> **bitwise_xor_reduce**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint64_t> **bitwise_xor_reduce**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<uint8_t> **bitwise_xor_reduce**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Bitwise XOR of all elements in the specified dimension.

Return Output array.

Parameters

- in1: Array input.
- in2: The axis to run over.

BhArray<double> **real**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Return the real part of the elements of the array.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **real**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Return the real part of the elements of the array.

Return Output array.

Parameters

- in1: Array input.

BhArray<double> **imag**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Return the imaginary part of the elements of the array.

Return Output array.

Parameters

- in1: Array input.

BhArray<float> **imag**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Return the imaginary part of the elements of the array.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **add_accumulate**(**const** *BhArray*<*std*::complex<double>> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<*std*::complex<float>> **add_accumulate**(**const** *BhArray*<*std*::complex<float>> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<float> **add_accumulate**(**const** *BhArray*<float> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<double> **add_accumulate**(**const** *BhArray*<double> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int16_t> **add_accumulate**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int32_t> **add_accumulate**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int64_t> **add_accumulate**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **add_accumulate**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **add_accumulate**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **add_accumulate**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **add_accumulate**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint8_t> **add_accumulate**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Computes the prefix sum.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<*std*::complex<double>> **multiply_accumulate**(**const** *BhArray*<*std*::complex<double>> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<*std*::complex<float>> **multiply_accumulate**(**const** *BhArray*<*std*::complex<float>> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<float> **multiply_accumulate**(**const** *BhArray*<float> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<double> **multiply_accumulate**(**const** *BhArray*<double> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int16_t> **multiply_accumulate**(**const** *BhArray*<int16_t> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int32_t> **multiply_accumulate**(**const** *BhArray*<int32_t> &*in1*, int64_t *in2*)

Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int64_t> **multiply_accumulate**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<int8_t> **multiply_accumulate**(**const** *BhArray*<int8_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint16_t> **multiply_accumulate**(**const** *BhArray*<uint16_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint32_t> **multiply_accumulate**(**const** *BhArray*<uint32_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint64_t> **multiply_accumulate**(**const** *BhArray*<uint64_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<uint8_t> **multiply_accumulate**(**const** *BhArray*<uint8_t> &*in1*, int64_t *in2*)
Computes the prefix product.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: The axis to run over.

BhArray<*std*::complex<double>> **sign**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **sign**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<float> **sign**(**const** *BhArray*<float> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<double> **sign**(**const** *BhArray*<double> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<int16_t> **sign**(**const** *BhArray*<int16_t> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<int32_t> **sign**(**const** *BhArray*<int32_t> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<int64_t> **sign**(**const** *BhArray*<int64_t> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- in1: Array input.

BhArray<int8_t> **sign**(**const** *BhArray*<int8_t> &*in1*)
Computes the SIGN of elements. -1 = negative, 1=positive. 0 = 0.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **gather**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<double>> **gather**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<*std*::complex<float>> **gather**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<float> **gather**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<double> **gather**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int16_t> **gather**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int32_t> **gather**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int64_t> **gather**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<int8_t> **gather**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint16_t> **gather**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.

BhArray<uint32_t> **gather**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **gather**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **gather**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Gather elements from IN selected by INDEX into OUT. NB: OUT.shape == INDEX.shape and IN can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<bool> **scatter**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<*std*::complex<double>> **scatter**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<*std*::complex<float>> **scatter**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **scatter**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **scatter**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **scatter**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **scatter**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **scatter**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **scatter**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **scatter**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **scatter**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **scatter**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **scatter**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Scatter all elements of IN into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **remainder**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<float> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<float> **remainder**(**const** *BhArray*<float> &*in1*, float *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<float> **remainder**(float *in1*, **const** *BhArray*<float> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<double> **remainder**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<double> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<double> **remainder**(**const** *BhArray*<double> &*in1*, double *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<double> **remainder**(double *in1*, **const** *BhArray*<double> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int16_t> **remainder**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<int16_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int16_t> **remainder**(**const** *BhArray*<int16_t> &*in1*, int16_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int16_t> **remainder**(int16_t *in1*, **const** *BhArray*<int16_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int32_t> **remainder**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<int32_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int32_t> **remainder**(**const** *BhArray*<int32_t> &*in1*, int32_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int32_t> **remainder**(int32_t *in1*, **const** *BhArray*<int32_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int64_t> **remainder**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<int64_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int64_t> **remainder**(**const** *BhArray*<int64_t> &*in1*, int64_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int64_t> **remainder**(int64_t *in1*, **const** *BhArray*<int64_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<int8_t> **remainder**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<int8_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<int8_t> **remainder**(**const** *BhArray*<int8_t> &*in1*, int8_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<int8_t> **remainder**(int8_t *in1*, **const** *BhArray*<int8_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint16_t> **remainder**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint16_t> **remainder**(**const** *BhArray*<uint16_t> &*in1*, uint16_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint16_t> **remainder**(uint16_t *in1*, **const** *BhArray*<uint16_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint32_t> **remainder**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint32_t> **remainder**(**const** *BhArray*<uint32_t> &*in1*, uint32_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint32_t> **remainder**(uint32_t *in1*, **const** *BhArray*<uint32_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint64_t> **remainder**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint64_t> **remainder**(**const** *BhArray*<uint64_t> &*in1*, uint64_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint64_t> **remainder**(uint64_t *in1*, **const** *BhArray*<uint64_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<uint8_t> **remainder**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.

BhArray<uint8_t> **remainder**(**const** *BhArray*<uint8_t> &*in1*, uint8_t *in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Scalar input.

BhArray<uint8_t> **remainder**(uint8_t *in1*, **const** *BhArray*<uint8_t> &*in2*)
Return the element-wise remainder of division, which is `in1 % in2` in C99 and has the same sign as the divided in1.

Return Output array.

Parameters

- `in1`: Scalar input.
- `in2`: Array input.

BhArray<bool> **cond_scatter**(**const** *BhArray*<bool> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)
Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<*std*::complex<double>> **cond_scatter**(**const** *BhArray*<*std*::complex<double>> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)
Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<*std*::complex<float>> **cond_scatter**(**const** *BhArray*<*std*::complex<float>> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)
Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.

- in2: Array input.
- in3: Array input.

BhArray<float> **cond_scatter**(**const** *BhArray*<float> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<double> **cond_scatter**(**const** *BhArray*<double> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<int16_t> **cond_scatter**(**const** *BhArray*<int16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<int32_t> **cond_scatter**(**const** *BhArray*<int32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<int64_t> **cond_scatter**(**const** *BhArray*<int64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<int8_t> **cond_scatter**(**const** *BhArray*<int8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<uint16_t> **cond_scatter**(**const** *BhArray*<uint16_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<uint32_t> **cond_scatter**(**const** *BhArray*<uint32_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- `in1`: Array input.
- `in2`: Array input.
- `in3`: Array input.

BhArray<uint64_t> **cond_scatter**(**const** *BhArray*<uint64_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<uint8_t> **cond_scatter**(**const** *BhArray*<uint8_t> &*in1*, **const** *BhArray*<uint64_t> &*in2*, **const** *BhArray*<bool> &*in3*)

Conditional scatter elements of IN where COND is true into OUT selected by INDEX. NB: IN.shape == INDEX.shape and OUT can have any shape but must be contiguous.

Return Output array.

Parameters

- in1: Array input.
- in2: Array input.
- in3: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<bool> &*in1*)

Test for finite values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<*std*::complex<float>> &*in1*)

Test for finite values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<*std*::complex<double>> &*in1*)

Test for finite values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<int8_t> &*in1*)

Test for finite values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<int16_t> &*in1*)

Test for finite values.

Return Output array.

Parameters

- in1: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<int32_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<int64_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<uint8_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<uint16_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<uint32_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<uint64_t> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<float> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<bool> **isfinite**(**const** *BhArray*<double> &*in1*)
Test for finite values.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<float>> **conj**(**const** *BhArray*<*std*::complex<float>> &*in1*)
Complex conjugates.

Return Output array.

Parameters

- `in1`: Array input.

BhArray<*std*::complex<double>> **conj**(**const** *BhArray*<*std*::complex<double>> &*in1*)
Complex conjugates.

Return Output array.

Parameters

- `in1`: Array input.

void **random123**(*BhArray*<uint64_t> &*out*, uint64_t *seed*, uint64_t *key*)

Variables

Random **random**
Exposing the default instance of the random number generation

namespace [anonymous]

namespace std

file **array_create.hpp**
#include <cstdint>#include <bhxx/BhArray.hpp>#include <bhxx/array_operations.hpp>

file **BhArray.hpp**
#include <type_traits>#include <ostream>#include <bohrium/bh_static_vector.hpp>#include <bhxx/BhBase.hpp>#include <bhxx/type_traits_util.hpp>#include <bhxx/array_operations.hpp>

file **BhBase.hpp**
#include <cassert>#include <bohrium/bh_view.hpp>#include <bohrium/bh_main_memory.hpp>#include <memory>

file **BhInstruction.hpp**
#include "BhArray.hpp"#include <bohrium/bh_instruction.hpp>

file **bhxx.hpp**
#include <bhxx/BhArray.hpp>#include <bhxx/Runtime.hpp>#include <bhxx/array_operations.hpp>#include <bhxx/util.hpp>#include <bhxx/random.hpp>#include <bhxx/array_create.hpp>

file **random.hpp**
#include <cstdint>#include <random>#include <bhxx/BhArray.hpp>#include <bhxx/Runtime.hpp>

file **Runtime.hpp**
#include <iostream>#include <sstream>#include "BhInstruction.hpp"#include <bohrium/bh_component.hpp>

file **util.hpp**
#include <sstream>#include <algorithm>#include <bhxx/BhArray.hpp>

file **array_create.cpp**
#include <bhxx/Runtime.hpp>#include <bhxx/array_operations.hpp>#include <bhxx/util.hpp>#include <bhxx/array_create.hpp>#include <bhxx/random.hpp>

file **BhArray.cpp**
#include <bhxx/BhArray.hpp>#include <bhxx/Runtime.hpp>#include <bhxx/array_operations.hpp>#include <bhxx/util.hpp>#include <bhxx/array_create.hpp>

file **BhInstruction.cpp**
#include <bhxx/BhInstruction.hpp>

file **random.cpp**
#include <bhxx/random.hpp>#include <bhxx/type_traits_util.hpp>

file **Runtime.cpp**
#include <bhxx/Runtime.hpp>#include <iterator>

file **util.cpp**
#include <bhxx/util.hpp>#include <bhxx/Runtime.hpp>

file **array_operations.hpp**
#include <cstdint>#include <complex>

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/bridge/cxx/inc**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/doc/build/bhxx**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/bridge**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/doc/build**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/bridge/cxx**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/bridge/cxx/inc**

dir **/home/docs/checkouts/readthedocs.org/user_builds/bohrium/checkouts/latest/bridge/cxx/src**

2.2.3 C library

The C interface introduces two array concepts:

- A base array that has a *rank* (number of dimensions) and *shape* (array of dimension sizes). The memory of the base array is always a single contiguous block of memory.
- A view array that, beside a *rank* and a *shape*, has a *start* (start offset in number of elements) and a *stride* (array of dimension strides in number of elements). The view array refers to a (sub)set of a underlying base array where *start* is the offset into the base array and *stride* is number of elements to skip in order to iterate one step in a given dimension.

API

The C interface consists of a broad range of functions – in the following, we describe some of the important ones.

Create a new empty array with *rank* number of dimensions and with the shape *shape* and returns a handler/pointer to a *complete* view of this new array:

```
bh_multi_array_{TYPE}_p bh_multi_array_{TYPE}_new_empty(uint64_t rank, const int64_t*␣
↪shape);
```

Get pointer/handle to the base of a view:

```
bh_base_p bh_multi_array_{TYPE}_get_base(const bh_multi_array_{TYPE}_p self);
```

Destroy the base array and the associated memory:

```
void bh_multi_array_{TYPE}_destroy_base(bh_base_p base);
```

Destroy the view and base array (but not the associated memory):

```
void bh_multi_array_{TYPE}_free(const bh_multi_array_{TYPE}_p self);
```

Some meta-data access functions:

```
// Gets the number of elements in the array
uint64_t bh_multi_array_{TYPE}_get_length(bh_multi_array_{TYPE}_p self);

// Gets the number of dimensions in the array
uint64_t bh_multi_array_{TYPE}_get_rank(bh_multi_array_{TYPE}_p self);

// Gets the number of elements in the dimension
uint64_t bh_multi_array_{TYPE}_get_dimension_size(bh_multi_array_{TYPE}_p self, const␣
↪int64_t dimension);
```

Before accessing the memory of an array, one has to synchronize the array:

```
void bh_multi_array_{TYPE}_sync(const bh_multi_array_{TYPE}_p self);
```

Access the memory of an array (remember to synchronize):

```
bh_{TYPE}* bh_multi_array_{TYPE}_get_base_data(bh_base_p base);
```

Some of the element-wise operations:

```
//Addition
void bh_multi_array_{TYPE}_add(bh_multi_array_{TYPE}_p out, const bh_multi_array_
↪{TYPE}_p lhs, const bh_multi_array_{TYPE}_p rhs);

//Multiply
void bh_multi_array_{TYPE}_multiply(bh_multi_array_{TYPE}_p out, const bh_multi_array_
↪{TYPE}_p lhs, const bh_multi_array_{TYPE}_p rhs);

//Addition: scalar + array
void bh_multi_array_{TYPE}_add_scalar_lhs(bh_multi_array_{TYPE}_p out, bh_{TYPE} lhs,␣
↪const bh_multi_array_{TYPE}_p rhs);
```

Some of the reduction and accumulate (aka scan) functions where *axis* is the dimension to reduce/accumulate over:

```
//Sum
void bh_multi_array_{TYPE}_add_reduce(bh_multi_array_{TYPE}_p out, const bh_multi_
↪array_{TYPE}_p in, bh_int64 axis);

//Prefix sum
void bh_multi_array_{TYPE}_add_accumulate(bh_multi_array_{TYPE}_p out, const bh_multi_
↪array_{TYPE}_p in, bh_int64 axis);
```

2.2.4 Runtime Configuration

Bohrium supports a broad range of front and back-ends. The default backend is OpenMP. You can change which backend to use by defining the `BH_STACK` environment variable:

- The CPU backend that make use of OpenMP: `BH_STACK=openmp`
- The GPU backend that make use of OpenCL: `BH_STACK=opencl`
- The GPU backend that make use of CUDA: `BH_STACK=cude`

For debug information when running Bohrium, use the following environment variables:

```
BH_<backend>_PROF=true     -- Prints a performance profile at the end of execution.
BH_<backend>_VERBOSE=true  -- Prints a lot of information including the source of the␣
↪JIT compiled kernels. Enables per-kernel profiling when used together with BH_
↪OPENMP_PROF=true.
BH_SYNC_WARN=true          -- Show Python warnings in all instances when copying data␣
↪to Python.
BH_MEM_WARN=true           -- Show warnings when memory accesses are problematic.
BH_<backend>_GRAPH=true    -- Dump a dependency graph of the instructions send to the␣
↪back-ends (.dot file).
BH_<backend>_VOLATILE=true -- Declare temporary variables using `volatile`, which␣
↪avoid precision differences because of Intel's use of 80-bit floats internally.
```

Particularly, `BH_<backend>_PROF=true` is very useful to explore why Bohrium might not perform as expected:

```
BH_OPENMP_PROF=1 python -m bohrium heat_equation.py --size=4000*4000*100
heat_equation.py - target: bhc, bohrium: True, size: 4000*4000*100, elapsed-time: 6.
↪446084

[OpenMP] Profiling:
Fuse cache hits:                199/203 (98.0296%)
Codegen cache hits              299/304 (98.3553%)
Kernel cache hits               300/304 (98.6842%)
Array contractions:             700/1403 (49.8931%)
Outer-fusion ratio:             13/23 (56.5217%)

Max memory usage:               0 MB
Syncs to NumPy:                 99
Total Work:                     12800400099 operations
Throughput:                     1.9235e+09ops
Work below par-threshold (1000): 0%

Wall clock:                     6.65473s
Total Execution:                6.04354s
  Pre-fusion:                   0.000761211s
  Fusion:                       0.00411354s
  Codegen:                      0.00192224s
  Compile:                      0.285544s
  Exec:                         4.91214s
  Copy2dev:                     0s
  Copy2host:                    0s
  Ext-method:                   0s
  Offload:                      0s
  Other:                        0.839052s

Unaccounted for (wall - total):  0.611198s
```

Which tells us, among other things, that the execution of the compiled JIT kernels (`Exec`) takes 4.91 seconds, the JIT

compilation (`Compile`) takes 0.29 seconds, and the time spend outside of Bohrium (`Unaccounted for`) takes 0.61.

OpenCL Configuration

Bohrium sorts all available devices by type ('gpu', 'cpu', or 'accelerator'). Set the device number to the device Bohrium should use (0 means first):

```
BH_OPENCL_DEVICE_NUMBER=0
```

In order to see all available devices, run:

```
python -m bohrium_api --info
```

You can also set the options in the configure file under the `[opencl]` section.

Also under the `[opencl]` section, you can set the OpenCL work group sizes:

```
# OpenCL work group sizes
work_group_size_1dx = 128
work_group_size_2dx = 32
work_group_size_2dy = 4
work_group_size_3dx = 32
work_group_size_3dy = 2
work_group_size_3dz = 2
```

Advanced Configuration

In order to configure the runtime setup of Bohrium you must provide a configuration file to Bohrium. The installation of Bohrium installs a default configuration file in `/etc/bohrium/config.ini` when doing a system-wide installation, `~/.bohrium/config.ini` when doing a local installation, and `<python library>/bohrium/config.ini` when doing a pip installation.

At runtime Bohrium will search through the following prioritized list in order to find the configuration file:

- The environment variable `BH_CONFIG`
- The config within the Python package `bohrium/config.ini` (in the same directory as `__init__.py`)
- The home directory config `~/.bohrium/config.ini`
- The system-wide config `/usr/local/etc/bohrium/config.ini`
- The system-wide config `/usr/etc/bohrium/config.ini`
- The system-wide config `/etc/bohrium/config.ini`

The default configuration file looks similar to the config below:

```
#
# Stack configurations, which are a comma separated lists of components.
# NB: 'stacks' is a reserved section name and 'default'
#     is used when 'BH_STACK' is unset.
#     The bridge is never part of the list
#
[stacks]
default    = bcexp, bccon, node, openmp
openmp     = bcexp, bccon, node, openmp
```

(continues on next page)

(continued from previous page)

```
opencl     = bcexp, bccon, node, opencl, openmp

#
# Managers
#

[node]
impl = /usr/lib/libbh_vem_node.so
timing = false

[proxy]
address = localhost
port = 4200
impl = /usr/lib/libbh_vem_proxy.so


#
# Filters - Helpers / Tools
#
[pprint]
impl = /usr/lib/libbh_filter_pprint.so

#
# Filters - Bytecode transformers
#
[bccon]
impl = /usr/lib/libbh_filter_bccon.so
collect = true
stupidmath = true
muladd = true
reduction = false
find_repeats = false
timing = false
verbose = false

[bcexp]
impl = /usr/lib/libbh_filter_bcexp.so
powk = true
sign = false
repeat = false
reduce1d = 32000
timing = false
verbose = false

[noneremover]
impl = /usr/lib/libbh_filter_noneremover.so
timing = false
verbose = false

#
# Engines
#
[openmp]
impl = /usr/lib/libbh_ve_openmp.so
tmp_bin_dir = /usr/var/bohrium/object
tmp_src_dir = /usr/var/bohrium/source
dump_src = true
```

(continues on next page)

(continued from previous page)

```
verbose = false
prof = false #Profiling statistics
compiler_cmd = "/usr/bin/x86_64-linux-gnu-gcc"
compiler_inc = "-I/usr/share/bohrium/include"
compiler_lib = "-lm -L/usr/lib -lbh"
compiler_flg = "-x c -fPIC -shared  -std=gnu99  -O3 -march=native -Werror -fopenmp"
compiler_openmp = true
compiler_openmp_simd = false

[opencl]
impl = /usr/lib/libbh_ve_opencl.so
verbose = false
prof = false #Profiling statistics
# Additional options given to the opencl compiler. See documentation for␣
↪clBuildProgram
compiler_flg = "-I/usr/share/bohrium/include"
serial_fusion = false # Topological fusion is default
```

The configuration file consists of two things: `components` and orchestration of components in `stacks`.

Components marked with square brackets. For example `[node]`, `[openmp]`, `[opencl]` are all components available for the runtime system.

The `stacks` define different default configurations of the runtime environment and one can switch between them using the environment var `BH_STACK`.

The configuration of a component can be overwritten with environment variables using the naming convention `BH_[COMPONENT]_[OPTION]`, below are a couple of examples controlling the behavior of the CPU vector engine:

```
BH_OPENMP_PROF=true    -- Prints a performance profile at the end of execution.
BH_OPENMP_VERBOSE=true -- Prints a lot of information including the source of the JIT␣
↪compiled kernels. Enables per-kernel profiling when used together with BH_OPENMP_
↪PROF=true.
```

Useful environment variables:

```
BH_SYNC_WARN=true          -- Show Python warnings in all instances when copying data␣
↪to Python.
BH_MEM_WARN=true           -- Show warnings when memory accesses are problematic.
BH_UNSUP_WARN=false        -- Do not warn when when encountering unsupported NumPy␣
↪operations.
BH_<backend>_GRAPH=true    -- Dump a dependency graph of the instructions send to the␣
↪back-ends (.dot file).
BH_<backend>_VOLATILE=true -- Declare temporary variables using `volatile`, which␣
↪avoid precision differences because of Intel's use of 80-bit floats internally.
```

2.3 Developer Guide

Bohrium is hosted and made publicly available via a git-repository on github under the *LGPLv3 License*.

If you want to join / contribute then fork the repository on Github and get in touch with us.

If you just want read-access then simply clone the repository:

```
git clone git@github.com/bh107/bohrium.git
cd bohrium
```

Continue by taking a look at *Installation* on how to build / install Bohrium.

2.3.1 Further information

Tools

Valgrind, GDB, and Python

Valgrind is a great tool for memory debugging, memory leak detection, and profiling. However, both Python and NumPy floods the valgrind output with memory errors - it is therefore necessary to use a debug and valgrind friendly version of Python and NumPy:

```
sudo apt-get build-dep python
sudo apt-get install zlib1g-dev valgrind

mkdir python_debug_env
cd python_debug_env
export INSTALL_DIR=$PWD

# Build and install Python:
export VERSION=2.7.11
wget http://www.python.org/ftp/python/$VERSION/Python-$VERSION.tgz
tar -xzf Python-$VERSION.tgz
cd Python-$VERSION
./configure --with-pydebug --without-pymalloc --with-valgrind --prefix=$INSTALL_DIR
make install
sudo ln -s $PWD/python-gdb.py /usr/bin/python-gdb.py
sudo ln -s $INSTALL_DIR/bin/python /usr/bin/dython
cd ..
rm Python-$VERSION.tgz

# Build and install Cython
export VERSION=0.24
wget http://cython.org/release/Cython-$VERSION.tar.gz
tar -xzf Cython-$VERSION.tar.gz
cd Cython-$VERSION
dython setup.py install
cd ..
rm Cython-$VERSION.tar.gz

export VERSION=21.1.0
wget https://pypi.python.org/packages/f0/32/
↪99ead2d74cba43bd59aa213e9c6e8212a9d3ed07805bb66b8bf9affbb541/setuptools-$VERSION.
↪tar.gz#md5=8fd8bdbf05c286063e1052be20a5bd98
tar -xzf setuptools-$VERSION.tar.gz
cd setuptools-$VERSION
dython setup.py install
cd ..
rm setuptools-$VERSION.tar.gz

# Build and install NumPy
export VERSION=1.11.0
```

(continues on next page)

(continued from previous page)

```
wget  https://github.com/numpy/numpy/archive/v$VERSION.tar.gz
tar -xzf v$VERSION.tar.gz
cd numpy-$VERSION
dython setup.py install
cd ..
rm v$VERSION.tar.gz
```

Build Bohrium with custom Python

Build and install Bohrium (with some components deactivated):

```
unzip master.zip
cd bohrium-master
mkdir build
cd build
cmake .. -DPYTHON_EXECUTABLE=/usr/bin/dython -DEXT_FFTW=OFF -DEXT_VISUALIZER=OFF -
↪DVEM_PROXY=OFF -DVE_GPU=OFF  -DBRIDGE_NUMCIL=OFF -DTEST_CIL=OFF
make
make install
cd ..
rm master.zip
```

Most Used Commands

GDB

GDB supports some helpful Python commands (https://docs.python.org/devguide/gdb.html). To activate, `source` the `python-gdb.py` file within GDB:

```
source /usr/bin/python-gdb.py
```

Then you can use Python specific GDB commands such as `py-list` or `py-bt`.

Valgrind

Valgrind can be used to detect memory errors by invoking it with:

```
valgrind --suppressions=<path to bohrium>/misc/valgrind.supp dython <SCRIPT_NAME>
```

Narrowing the valgrind analysis, add the following to your source code:

```
#include <valgrind/callgrind.h>
... your code ...
CALLGRIND_START_INSTRUMENTATION;
... your code ...
CALLGRIND_STOP_INSTRUMENTATION;
CALLGRIND_DUMP_STATS;
```

Then run valgrind with the flag:

```
--instr-atstart=no
```

Invoking valgrind to determine cache-utilization:

```
--tool=callgrind --simulate-cache=yes <PROG> <PROG_PARAM>
```

Cluster VEM (MPI)

In order to use MPI with valgrind, the MPI implementation needs to be compiled with PIC and no-dlopen flag. E.g, OpenMPI could be installed as follows:

```
wget http://www.open-mpi.org/software/ompi/v1.6/downloads/openmpi-1.6.5.tar.gz
cd tar -xzf openmpi-1.6.5.tar.gz
cd openmpi-1.6.5
./configure --with-pic --disable-dlopen --prefix=/opt/openmpi
make
sudo make install
```

And then executed using valgrind:

```
export LD_LIBRARY_PATH=/opt/openmpi/lib/:$LD_LIBRARY_PATH
export PATH=/opt/openmpi/bin:$PATH
mpiexec -np 1 valgrind dython test/numpy/numpytest.py : -np 1 valgrind ~/.local/bh_
↪vem_cluster_slave
```

Writing Documentation

The documentation is written in Sphinx.

You will need the following to write/build the documentation:

```
sudo apt-get install doxygen python-sphinx python-docutils python-setuptools
```

As well as a python-packages **breathe** and **numpydoc** for integrating doxygen-docs with Sphinx:

```
sudo easy_install breathe numpydoc
```

Overview of the documentation files:

```
bohrium/doc                 # Root folder of the documentation.
bohrium/doc/source          # Write / Edit the documentation here.
bohrium/doc/build           # Documentation is "rendered" and stored here.
bohrium/doc/Makefile        # This file instructs Sphinx on how to "render" the␣
↪documentation.
bohrium/doc/make.bat        # ---- || ----, on Windows
bohrium/doc/deploy_doc.sh   # This script pushes the rendered docs to http://bohrium.
↪bitbucket.org.
```

Most used commands

These commands assume that your current working dir is **bohrium/doc**.

Initiate doxygen:

```
make doxy
```

Render a html version of the docs:

```
make html
```

Push the html-rendered docs to http://bohrium.bitbucket.org, this command assumes that you have write-access to the doc-repos on Bitbucket:

```
make deploy
```

The docs still needs a neat way to integrate a full API-documentation of the Bohrium core, managers and engines.

Continuous Integration

Currently we use both a privately hosted Jenkins server as well as Travis for our CI.

Setup jenkins:

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.
↪d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

Then configure it via the web interface.

- Open Student Projects
- Benchmark Suite

2.4 Frequently Asked Questions (FAQ)

Does it automatically support lazy evaluation (also called: late evaluation, expression templates)?

Yes, Bohrium will lazy evaluate all Python/NumPy operations until it encounters a "Python Read", such a printing an array or having an if-statement testing the value of an array.

Does it support "views" in the sense that a sub-slice is simply a view on the same array?

Yes, Bohrium supports NumPy views fully thus operating on array slices does not involve data copying.

Does it support generator functions (which only start calculating once the evaluation is forced)? Which ones are supported? Which conditions force evaluations? Presumably reduce operations?

Yes, Bohrium uses a fusion algorithm that fuses (or merges) array operations into the same computation kernel that are then JIT-compiled and executed. However, Bohrium can only fuse operations that have some common sized dimension and no horizontal data conflicts. Typically, reducing a vector to a scalar will force evaluate (but reducing a matrix to a vector will not force an evaluate on it own).

On GPUs, will Bohrium automatically keep all data (i.e. all Bohrium arrays) on the card?

Yes, we only move data back to the host when the data is accessed directly by Python or a Python C-extension.

Does it fully support operations on the complex datatype in Bohrium arrays?

Yes.

Will it lazily operate even over for-loops effectively unrolling them?

Yes, a for-loop in Python does not force evaluation. However, loops in Python with many iterations will hurt performance, just like it does in regular NumPy or Matlab

Is Bohrium using CUDA on Nvidia Cards or generic OpenCL for any GPU?

Bohrium can use both CUDA and OpenCL.

What is the disadvantage of Bohrium? I wonder why it exists as a separate project. From my point of view it looks like Bohrium is "just reimplementing" NumPy. That's probably extremely oversimplified, but is there a plan to feed the results of Bohrium into the NumPy project?

The only disadvantage of Bohrium is the extra dependencies e.g. Bohrium need a C99 compiler for JIT-complication. Thus, the idea of incorporating Bohrium into NumPy as an alternative "backend" is very appealing and we hope it could be realized some day.

I get the error: "Failed to open map segment shared object"

This is because `TMPDIR` is mounted using the `noexec` flag. Bohrium uses `TMPDIR` to write JIT-compiled kernels, which must be execuable. Please set `TMPDIR` to a location not mounted using `noexec` (thanks to Jonas Große Sundrup).

2.5 Reporting Bugs

Please help us make Bohrium even better by submitting bugs and/or feature requests to us via the issue tracker on https://github.com/bh107/bohrium/issues

When reporting problems please include the output from:

```
python -m bohrium --info
```

2.6 Publications

1) Mads R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In Python for High Performance and Scientific Computing (PyHPC 2013), 2013.

2) Simon A. F. Lund, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter. Doubling the Performance of Python/NumPy with less than 100 SLOC. In Python for High Performance and Scientific Computing (PyHPC 2013), 2013.

3) Troels Blum, Mads R. B. Kristensen, and Brian Vinter. Transparent gpu execution of numpy applications.. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International. IEEE, 2014.

4) Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a virtual machine approach to portable parallelism. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International. IEEE, 2014.

5) Simon A.F. Lund, Mads R.B. Kristensen, Brian Vinter, Dimitrios Katsaros. Bypassing the Conventional Software Stack Using Adaptable Runtime Systems. In Proceedings of the Euro-Par Workshops, 2014.

6) Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, James Avery, and Brian Vinter. Separating NumPy API from Implementation. In Proceedings of the Python for High Performance and Scientific Computing (PyHPC 2014), 2014.

7) Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, and James Avery. Fusion of Parallel Array Operations. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16), 2016.

8) Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, James Avery, and Brian Vinter. Battling Memory Requirements of Array Programming through Streaming. In Proceedings of the International Conference on High Performance Computing, 2016.

2.7 History and License

Bohrium is an active research project started by Mads R. B. Kristensen, Troels Blum, and Brian Vinter at the Niels Bohr Institute - University of Copenhagen. Contributors include those listed below in no particular order:

- Troels Blum <blum@nbi.dk>
- Brian Vinter <vinter@nbi.dk>
- Kenneth Skovhede <skovhede@nbi.dk>
- Simon Andreas Frimann Lund <safl@nbi.dk>
- Mads Ruben Burgdorff Kristensen <madsbk@nbi.dk>
- Mads Ohm Larsen <ohm@nbi.dk>

Contributors are welcome, do not hesitate to contact us!

Bohrium is distributed under the LGPLv3 license:

```
                   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.


This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

"The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
```

(continues on next page)

(continued from previous page)

```
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

 a) under this License, provided that you make a good faith effort to
 ensure that, in the event an Application does not supply the
 function or data, the facility still operates, and performs
 whatever part of its purpose remains meaningful, or

 b) under the GNU GPL, with none of the additional permissions of
 this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

 a) Give prominent notice with each copy of the object code that the
 Library is used in it and that the Library and its use are
 covered by this License.

 b) Accompany the object code with a copy of the GNU GPL and this license
 document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

 a) Give prominent notice with each copy of the Combined Work that
 the Library is used in it and that the Library and its use are
 covered by this License.
```

(continues on next page)

(continued from previous page)

```
 b) Accompany the Combined Work with a copy of the GNU GPL and this license
 document.

 c) For a Combined Work that displays copyright notices during
 execution, include the copyright notice for the Library among
 these notices, as well as a reference directing the user to the
 copies of the GNU GPL and this license document.

 d) Do one of the following:

     0) Convey the Minimal Corresponding Source under the terms of this
     License, and the Corresponding Application Code in a form
     suitable for, and under terms that permit, the user to
     recombine or relink the Application with a modified version of
     the Linked Version to produce a modified Combined Work, in the
     manner specified by section 6 of the GNU GPL for conveying
     Corresponding Source.

     1) Use a suitable shared library mechanism for linking with the
     Library.  A suitable mechanism is one that (a) uses at run time
     a copy of the Library already present on the user's computer
     system, and (b) will operate properly with a modified version
     of the Library that is interface-compatible with the Linked
     Version.

 e) Provide Installation Information, but only if you would otherwise
 be required to provide such information under section 6 of the
 GNU GPL, and only to the extent that such information is
 necessary to install and execute a modified version of the
 Combined Work produced by recombining or relinking the
 Application with a modified version of the Linked Version. (If
 you use option 4d0, the Installation Information must accompany
 the Minimal Corresponding Source and Corresponding Application
 Code. If you use option 4d1, you must provide the Installation
 Information in the manner specified by section 6 of the GNU GPL
 for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

 a) Accompany the combined library with a copy of the same work based
 on the Library, uncombined with any other library facilities,
 conveyed under the terms of this License.

 b) Give prominent notice with the combined library that part of it
 is a work based on the Library, and explaining where to find the
 accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
```

(continues on next page)

(continued from previous page)

```
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

Bibliography

[1] Wikipedia, “Normal distribution”, http://en.wikipedia.org/wiki/Normal_distribution

[2] P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.

[1] Peyton Z. Peebles Jr., “Probability, Random Variables and Random Signal Principles”, 4th ed, 2001, p. 57.

[2] “Poisson Process”, Wikipedia, http://en.wikipedia.org/wiki/Poisson_process

[3] “Exponential Distribution, Wikipedia, http://en.wikipedia.org/wiki/Exponential_distribution

[1] Peyton Z. Peebles Jr., “Probability, Random Variables and Random Signal Principles”, 4th ed, 2001, p. 57.

[2] “Poisson Process”, Wikipedia, http://en.wikipedia.org/wiki/Poisson_process

[3] “Exponential Distribution, Wikipedia, http://en.wikipedia.org/wiki/Exponential_distribution

Python Module Index

b

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Z