# npbackend Documentation

***Release 0.1***

**Bohrium Team**

April 06, 2016

Contents

The **npbackend** is a backend interface to Python/NumPy that translates NumPy array operations into a sequence of Python function calls. By implementing these Python functions, external libraries can accelerate NumPy array operations without any modifications to the user Python/NumPy code.

For now, **npbackend** is integrated with the Bohrium project and cannot be installed separately (the source code can be found here) but it is our intension to decouple **npbackend** out of Bohrium to become a standalone project.

Contents:

CHAPTER 1

# Installation

Since **npbackend** is integrated with the Bohrium, in order to install **npbackend** you simple has to install Bohrium. The installation guide can be found at http://www.bh107.org/installation/index.html.

CHAPTER 2

# Usage

Using **npbackend** requires enabling the module and choosing a `target`.

**Note:** Since **npbackend** is integrated with Bohrium, the module name is `bohrium` and not `npbackend`.

## 2.1 Enabling npbackend

The least intrusive method of enabling `bohrium` is invoking your Python program with the module loaded. Which means instead of invoking your Python/NumPy program like this:

```
python my_program.py
```

You will invoke it like this instead:

```
python -m bohrium my_program.py
```

Which will effectively overrule the `numpy` module and instead use `bohrium`. That is all it takes.

Another approach is replacing your `import numpy` statements with `import bohrium`. For example, replacing:

```
import numpy as np
```

With:

```
import bohrium as np
```

## 2.2 Choosing a npbackend target

**npbackend** will default to using Bohrium as the backend target. Changing backend target is done via the `NPBE_TARGET` environment variable. Valid values for `NPBE_TARGET` are:

1. bhc, the default targeting Bohrium
2. numexpr, targeting Numexpr
3. pygpu, targeting libgpuarray
4. numpy, Using NumPy itself through NumPy backend, this is most likely not something you would want to do, the value is only provided for testing purposes.

If you which to use something else than the default target, then you can invoke your Python/NumPy application with:

```
NPBE_TARGET="numexpr" python -m bohrium my_program.py
```

Or if your changed import statements:

```
NPBE_TARGET="numexpr" python my_program.py
```

That is all there is to it.

CHAPTER 3

# New Target

In order to implement a new target for **npbackend**, we need to implement the functions in interface.py:

```
"""
===================================
Interface for ``npbackend`` targets
===================================

Implementing a ``npbackend`` target, starts with fleshing out::

    import interface

    class View(interface.View):
        ...

    class Base(interface.Base):
        ...

And then implementing each of the methods described in ``interface.py``,
documented below:
"""

class Base(object):
    """
    Abstract base array handle (an array has only one base)
    Encapsulates memory allocated for an array.

    :param int size: Number of elements in the array
    :param numpy.dtype dtype: Data type of the elements
    """
    def __init__(self, size, dtype):
        self.size = size        # Total number of elements
        self.dtype = dtype      # Data type

class View(object):
    """
    Abstract array view handle.
    Encapsulates meta-data of an array.

    :param int ndim: Number of dimensions / rank of the view
    :param int start: Offset from base (in elements), converted to bytes upon construction.
    :param tuple(int*ndim) shape: Number of elements in each dimension of the array.
    :param tuple(int*ndim) strides: Stride for each dimension (in elements), converted to bytes upon
    :param interface.Base base: Base associated with array.
```

```
    """
    def __init__(self, ndim, start, shape, strides, base):
        self.ndim = ndim        # Number of dimensions
        self.shape = shape      # Tuple of dimension sizes
        self.base = base        # The base array this view refers to
        self.dtype = base.dtype
        self.start = start * base.dtype.itemsize # Offset from base (in bytes)
        self.strides = [x * base.dtype.itemsize for x in strides] #Tuple of strides (in bytes)

def runtime_flush():
    """Flush the runtime system"""
    pass

def tally():
    """Tally the runtime system"""
    pass

def get_data_pointer(ary, allocate=False, nullify=False):
    """
    Return a C-pointer to the array data (represented as a Python integer).

    .. note:: One way of implementing this would be to return a ndarray.ctypes.data.

    :param Mixed ary: The array to retrieve a data-pointer for.
    :param bool allocate: When true the target is expected to allocate the data prior to returning.
    :param bool nullify: TODO
    :returns: A pointer to memory associated with the given 'ary'
    :rtype: int
    """
    raise NotImplementedError()

def set_bhc_data_from_ary(self, ary):
    """
    Copy data from 'ary' into the array 'self'

    :param Mixed self: The array to copy data to.
    :param Mixed ary: The array to copy data from.
    :rtype: None
    """
    raise NotImplementedError()

def ufunc(op, *args):
    """
    Perform the ufunc 'op' on the 'args' arrays

    :param bohrium.ufunc.Ufunc op: The ufunc operation to apply to args.
    :param Mixed args: Args to the ufunc operation.
    :rtype: None
    """
    raise NotImplementedError()

def reduce(op, out, ary, axis):
    """
    Reduce 'axis' dimension of 'ary' and write the result to out

    :param op bohrium.ufunc.Ufunc: The ufunc operation to apply to args.
    :param out Mixed: The array to reduce "into".
    :param ary Mixed: The array to reduce.
```

```
    :param axis Mixed: The axis to apply the reduction over.
    :rtype: None
    """
    raise NotImplementedError()

def accumulate(op, out, ary, axis):
    """
    Accumulate/scan 'axis' dimension of 'ary' and write the result to 'out'.

    :param bohrium.ufunc.Ufunc op: The element-wise operator to accumulate.
    :param Mixed out: The array to accumulate/scan "into".
    :param Mixed ary: The array to accumulate/scan.
    :param Mixed axis: The axis to apply the accumulation/scan over.
    :rtype: None
    """
    raise NotImplementedError()

def extmethod(name, out, in1, in2):
    """
    Apply the extension method 'name'.

    :param Mixed out: The array to write results to.
    :param Mixed in1: First input array.
    :param Mixed in2: Second input array.
    :rtype: None
    """
    raise NotImplementedError()

def range(size, dtype):
    """
    Create a new array containing the values [0:size[.

    :param int size: Number of elements in the returned array.
    :param np.dtype dtype: Type of elements in the returned range.
    :rtype: Mixed
    """

    raise NotImplementedError()

def random123(size, start_index, key):
    """
    Create a new random array using the random123 algorithm.
    The dtype is uint64 always.

    :param int size: Number of elements in the returned array.
    :param int start_index: TODO
    :param int key: TODO
    """
    raise NotImplementedError()

def gather(out, ary, indexes):
    """
    Gather elements from 'ary' selected by 'indexes'.
    ary.shape == indexes.shape.

    :param Mixed out: The array to write results to.
    :param Mixed ary: Input array.
    :param Mixed indexes: Array of indexes (uint64).
```

```
    """
    raise NotImplementedError()
```

## 3.1 NumPy Example

An example of a target implementation is target_numpy.py that uses NumPy as a backend. Now, implementing a NumPy backend that targets NumPy does not make that much sense but it is a good example.

**Note:** In some cases using NumPy as a backend will output native NumPy because of memory allocation reuse.

```
"""
The Computation Backend

"""
from .. import bhc
from ._util import dtype_name
import numpy as np
import mmap
import time
import ctypes
from . import interface
import os

VCACHE = []
VCACHE_SIZE = int(os.environ.get("VCACHE_SIZE", 10))

class Base(interface.Base):
    """base array handle"""

    def __init__(self, size, dtype):
        super(Base, self).__init__(size, dtype)
        self.mmap_valid = True
        size *= dtype.itemsize
        for i, (vc_size, vc_mem) in enumerate(VCACHE):
            if vc_size == size:
                self.mmap = vc_mem
                VCACHE.pop(i)
                return

        self.mmap = mmap.mmap(-1, size, mmap.MAP_PRIVATE)

    def __str__(self):
        if self.mmap_valid:
            s = mmap
        else:
            s = "NULL"
        return "<base memory at %s>"%s

    def __del__(self):
        if self.mmap_valid:
            if len(VCACHE) < VCACHE_SIZE:
                VCACHE.append((self.size*self.dtype.itemsize, self.mmap))
                return
        self.mmap.close()
```

```
class View(interface.View):
    """array view handle"""

    def __init__(self, ndim, start, shape, strides, base):
        super(View, self).__init__(ndim, start, shape, strides, base)
        buf = np.frombuffer(self.base.mmap, dtype=self.dtype, offset=self.start)
        self.ndarray = np.lib.stride_tricks.as_strided(buf, shape, self.strides)

def views2numpy(views):
    """Extract the ndarray from the view."""

    ret = []
    for view in views:
        if isinstance(view, View):
            ret.append(view.ndarray)
        else:
            ret.append(view)
    return ret

def get_data_pointer(ary, allocate=False, nullify=False):
    """
    Extract the data-pointer from the given View (ary).

    :param target_numpy.View ary: The View to extract the ndarray form.
    :returns: Pointer to data associated with the 'ary'.
    :rtype: ctypes pointer
    """
    ret = ary.ndarray.ctypes.data
    if nullify:
        ary.base.mmap_valid = False
    return ret

def set_bhc_data_from_ary(self, ary):
    ptr = get_data_pointer(self, allocate=True, nullify=False)
    ctypes.memmove(ptr, ary.ctypes.data, ary.dtype.itemsize * ary.size)

def ufunc(op, *args):
    """Apply the 'op' on args, which is the output followed by one or two inputs"""

    args = views2numpy(args)
    if op.info['name'] == "identity":
        if np.isscalar(args[1]):
            exec("args[0][...] = args[1]")
        else:
            exec("args[0][...] = args[1][...]")
    else:
        func = eval("np.%s" % op.info['name'])
        func(*args[1:], out=args[0])

def reduce(op, out, ary, axis):
    """reduce 'axis' dimension of 'ary' and write the result to out"""

    func = eval("np.%s.reduce" % op.info['name'])
    (ary, out) = views2numpy((ary, out))
    if ary.ndim == 1:
        keepdims = True
    else:
        keepdims = False
```

```
    func(ary, axis=axis, out=out, keepdims=keepdims)

def accumulate(op, out, ary, axis):
    """accumulate 'axis' dimension of 'ary' and write the result to out"""

    func = eval("np.%s.accumulate" % op.info['name'])
    (ary, out) = views2numpy((ary, out))
    if ary.ndim == 1:
        keepdims = True
    else:
        keepdims = False
    func(ary, axis=axis, out=out, keepdims=keepdims)

def extmethod(name, out, in1, in2):
    """Apply the extended method 'name' """

    (out, in1, in2) = views2numpy((out, in1, in2))
    if name == "matmul":
        out[:] = np.dot(in1, in2)
    else:
        raise NotImplementedError("The current runtime system does not support "
                                  "the extension method '%s'" % name)

def range(size, dtype):
    """create a new array containing the values [0:size["""
    return np.arange((size,), dtype=dtype)

def random123(size, start_index, key):
    """Create a new random array using the random123 algorithm.
    The dtype is uint64 always."""
    return np.random.random(size)
```

Now, let's try to run a small Python/NumPy example:

```
import numpy as np

a = np.ones(100000000)
for _ in xrange(100):
    a = a + 42
```

First with native NumPy:

```
time -p python tt.py
real 26.69
user 10.20
sys 16.50
```

And then with **npbackend** using NumPy as backend:

```
NPBE_TARGET="numpy" time -p python -m bohrium tt.py
real 14.36
user 14.02
sys 0.34
```

Because of the memory allocation reuse, **npbackend** actually outperforms native NumPy. However, this is only the case because we allocate and free a significant number of large arrays.

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search