The NumPy Array: A Structure for Efficient Numerical Computation

In the Python world, NumPy arrays are the standard representation for numerical data and enable efficient implementation of numerical computations in a high-level language. As this effort shows, NumPy performance can be improved through three techniques: vectorizing calculations, avoiding copying data in memory, and minimizing operation counts.

The Python programming language provides a rich set of high-level data structures, such as lists for enumerating a collection of objects and dictionaries to build hash tables. However, these structures aren't ideally suited to high-performance numerical computation.

In the mid-90s, an international team of volunteers started to develop a data structure for efficient array computation. This structure evolved into what is now known as the *NumPy N-dimensional array*. The NumPy package, which comprises the NumPy array as well as a set of accompanying mathematical functions, has been widely adopted in academia, national laboratories, and industry, with applications ranging from gaming to space exploration.

A NumPy array is a multidimensional, uniform collection of elements (that is, all elements occupy the same number of bytes in memory). An array is characterized by both the type of elements it contains and its shape. For example, a matrix might be represented as an array of shape $M \times N$ that contains numbers, such as floating-point or complex numbers. Unlike matrices, NumPy arrays can have up to 32 dimensions; they might also contain other kinds of elements (or even combinations of elements), such as Booleans or dates.

Beneath the hood, a NumPy array is really just a convenient way of describing one or more blocks of computer memory so that the numbers represented can be easily manipulated. As we describe here, NumPy provides a high-level abstraction for numerical computation without compromising performance.

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Stéfan van der Walt
Stellenbosch University
S. Chris Colbert
Enthought, Inc.
Gaël Varoquaux
INRIA

Basic Usage

Throughout the code examples here, we assume that NumPy is imported as follows:

```
import numpy as np
```

We show code snippets as they appear inside an IPython[1] prompt, such as

```
In [3]: np.__version__
Out[3]: '1.4.1'
```

IPython is an interactive front end to Python with tab completion and easy documentation inspection. Lines of the form `In [n]:` represent the input prompt, equivalent to vanilla Python's `>>>`.

We can index an array's elements using the `[]` operator. In addition, we can retrieve parts of an array using standard Python slicing of the form

`start:stop:step`. For example, the first two rows of an array `x` are given by `x[:2, :]` or columns 1 through 3 by `x[:, 1:4]`. Similarly, every second row is given by `x[::2, :]`. Also, Python uses zero-based indexing and the slice `1:3` excludes the last element, 3.

More detailed information on the Python language is available at http://docs.python.org.

NumPy Array Structure: A View on Memory

A NumPy array—also called an "ndarray," short for *N*-dimensional array—describes memory using the following attributes:

- *data pointer:* the memory address of the first byte stored in the array;
- *data type description:* the kind of elements contained in the array, such as floating-point numbers or integers;
- *shape:* the array's shape, such as `(10, 10)` for a 10 × 10 array, or `(5, 5, 5)` for a block of data describing a mesh grid of *x*-, *y*-, and *z*-coordinates;
- *strides:* the number of bytes skipped in memory to proceed to the next element along a given dimension (for a `(10, 10)` array of bytes, for example, the strides might be `(10, 1)`, or proceed one byte to get to the next column and 10 bytes to locate the next row); and
- *flags:* define factors such as whether we're allowed to modify the array or whether memory layout is C- or Fortran-contiguous (in C, memory is laid out in "row major" order—that is, rows are stored one after another in memory—whereas in Fortran, columns are stored successively).

NumPy's strided memory model deserves particular attention, as it provides a powerful way of viewing the same memory in different ways without copying data. For example, consider the following integer array:

```
# Generate the integers from zero to
# eight and repack them into a 3x3
# array

In [1]: x = np.arange(9).reshape((3, 3))

In [2]: x
Out[2]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [3]: x.strides
Out[3]: (24, 8)
```

On our 64-bit system, the default integer data type occupies 64 bits—or 8 bytes—in memory. The strides therefore describe skipping three integers in memory to get to the next row and one to get to the next column. We can now generate a view on the same memory where we examine only every second element:

```
In [4]: y = x[::2, ::2]

In [5]: y
Out[5]:
array([[0, 2],
       [6, 8]])

In [6]: y.strides
Out[6]: (48, 16)
```

The arrays `x` and `y` point to the same memory—that is, if we modify the values in `y`, we also modify those in `x`—but the strides for `y` have been changed so that only every second element is seen along either axis. We call `y` a *view* on `x`:

```
In [7]: y[0, 0] = 100

In [8]: x
Out[8]:
array([[100,   1,   2],
       [  3,   4,   5],
       [  6,   7,   8]])
```

Views need not be created using slicing alone; by modifying strides, for example, an array can be transposed or reshaped at zero cost (no memory needs to be copied). Moreover, users can manually specify an array's strides, shape, and data type attributes (provided they're all compatible), enabling a plethora of ways in which to interpret the underlying data.

```
# Transpose the array, using the
# shorthand "T" property
In [9]: xT = x.T

In [10]: xT
Out[10]:
array([[100, 3, 6],
       [  1, 4, 7],
       [  2, 5, 8]])
```

```
In [11]: xT.strides
Out[11]: (8, 24)

# Change the shape of the array
In [12]: z = x.reshape((1, 9))

In [13]: z
Out[13]: array([[100, 1, 2, 3, 4,
                 5, 6, 7, 8]])

In [14]: z.strides
Out[14]: (72, 8)

# i.e., for the two-dimensional z,
# 9 * 8 bytes to skip over a row of
# 9 int64 elements, 8 bytes to skip
# a single element

# View data as bytes in memory rather
# than as 64bit integers
In [15]: z.view(np.uint8)

In [16]: z
Out[17]:
array([[100, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 2, 0,
        0, 0, 0, 0, 0, 0, 3, 0, 0,
        0, 0, 0, 0, 0, 4, 0, 0, 0,
        0, 0, 0, 0, 5, 0, 0, 0, 0,
        0, 0, 0, 6, 0, 0, 0, 0, 0,
        0, 0, 7, 0, 0, 0, 0, 0, 0,
        0, 8, 0, 0, 0, 0, 0, 0, 0]],
      dtype=uint8)

In [18]: z.shape
Out[19]: (1, 72)

In [20]: z.strides
Out[20]: (72, 1)
```

In each of these cases, the resulting array points to the same memory. The difference lies in the way the data is interpreted, based on shape, strides, and data type. Whenever possible, no data is copied in memory, making these operations extremely efficient.

Numerical Operations on Arrays

In any scripting language, injudicious use of `for` loops can lead to poor performance, particularly when a simple computation is applied to each element of a large data set. Grouping these element-wise operations together—a process known as *vectorization*—lets NumPy perform such computations much more rapidly.

Suppose we have a vector `a` and want to multiply its magnitude by three. A traditional Python `for` loop approach would be

```
In [21]: a = [1, 3, 5]

In [22]: b = [3*x for x in a]

In [23]: b
Out[23]: [3, 9, 15]
```

The vectorized approach applies this operation to all of an array's elements:

```
In [24]: a = np.array([1, 3, 5])

In [25]: b = 3 * a

In [26]: b
Out[26]: array([ 3,  9, 15])
```

Vectorized operations in NumPy are implemented in C, resulting in a significant speed improvement. Such operations aren't restricted to interactions between scalars and arrays. For example, NumPy can perform a fast element-wise subtraction of two arrays as follows:

```
In [27]: b - a
Out[27]: array([ 2,  6, 10])
```

When the shapes of the two arguments are different, but they share a common shape dimension, the operation is *broadcast* across the array. In other words, NumPy expands the arrays to make the operation viable:

```
In [28]: m = np.arange(6).reshape((2,3))

In [29]: m
Out[29]:
array([[0, 1, 2],
       [3, 4, 5]])

In [30]: b + m
Out[30]:
array([[ 3, 10, 17],
       [ 6, 13, 20]])
```

The "Broadcasting Rules" sidebar describes when such operations are viable. To save memory, broadcasted arrays are never physically constructed; NumPy simply uses the appropriate array elements during computation (under

Broadcasting Rules

Before broadcasting two arrays, NumPy verifies that all dimensions are suitably matched. Dimensions match when they are equal, or when either is `1` or `None`. In the latter case, the output array's dimension is expanded to the larger of the two.

For example, consider arrays `x` and `y` with shapes `(2, 4, 3)` and `(4, 1)`, respectively. These arrays are to be combined in a broadcasting operation such as `z = x + y`. We match their dimensions as follows:

```
x (2, 4, 3)
y (   4, 1)
-----------
z (2, 4, 3)
```

Therefore, the arrays' dimensions are compatible and yield an output of shape `(2, 4, 3)`.

the hood, this is achieved by using strides of zero).

Vectorization and Broadcasting Examples

The following four examples demonstrate how vectorization or broadcasting can be applied to improve computational or memory efficiency.

Evaluating Functions

Suppose we want to evaluate a function `f` over a large set of numbers, `x`, stored as an array. Using a `for` loop, the result is

```
In [31]: def f(x):
   ....:     return x**2 - 3*x + 4
   ....:

In [32]: x = np.arange(1e5)

In [33]: y = [f(i) for i in x]
```

On our machine, this loop executes in approximately 500 ms. Applying the function `f` on the NumPy array `x` engages the fast vectorized loop that operates on each element individually:

```
In [34]: y = f(x)

In [35]: y
Out[35]:
array([4.00000000e+00,
       2.00000000e+00,
       ...,
       9.99930001e+09,
       9.99950001e+09])
```

The vectorized computation executes in 1 ms. As the length of the input array `x` grows, however, execution speed decreases due to the construction of large temporary arrays. For example, the operation above roughly translates to

```
a = x**2
b = 3*x
c = a - b
fx = c + 4
```

Most array-based systems don't provide a way to circumvent the creation of these temporaries. With NumPy, the user can choose to perform operations "in place"—that is, in such a way that no new memory is allocated and all results are stored in the current array. To accomplish this, NumPy overrides the built-in in-place, or augmented, assignment operators:

```
def g(x):
    # Allocate output array, fill
    # with x^2
    fx = x**2

    # In-place addition/subtraction
    # no new memory allocated
    fx -= 3*x
    fx += 4

    return fx
```

Applying `g` to `x` takes 600 microseconds, which is almost twice as fast as the naive vectorization. We didn't compute `3*x` in place, as that would modify the original data in `x`. This example illustrates the ease with which NumPy handles vectorized array operations, without relinquishing control over performance-critical aspects such as memory allocation.

We can boost performance even further using tools such as Cython (http://www.cython.org), Theano,[2] or numexpr (http://code.google.com/p/numexpr), which lessen the load on the memory bus. Cython, a Python-to-C compiler that Stefan Behnel and his colleagues

discuss in their article, "Cython: The Best of Both Worlds" on page 31, is especially useful in cases where code can't be easily vectorized.

Finite Differencing

The derivative on a discrete sequence is often computed using finite differencing. Slicing makes this operation trivial. Suppose we have an $n + 1$ length vector and perform a forward divided difference:

```
In [36]: x = np.arange(0, 20, 2)

In [37]: x
Out[37]: array([ 0,  2,  4,  6,  8, 10, 12,
                14, 16, 18])

In [38]: y = x**2

In [39]: y
Out[39]: array([  0,   4,  16,  36,  64, 100,
                144, 196, 256, 324])

In [40]: dy_dx = ((y[1:] - y[:-1]) /
   ....:           (x[1:] - x[:-1]))

In [41]: dy_dx
Out[41]: array([ 2,  6, 10, 14, 18, 22, 26,
                30, 34])
```

In this example, `y[1:]` takes a slice of the `y` array starting at index 1 and continuing to the end of the array. Then, `y[:-1]` takes a slice that starts at index 0 and contains all elements apart from the last. Thus, `y[1:] - y[:-1]` has the effect of subtracting, from each element in the array, the element directly preceding it. Performing the same differencing on the `x` array and dividing the two resulting arrays yields the forward divided difference.

If we assume that the vectors are length $n + 2$, then calculating the central divided difference is simply a matter of modifying the slices:

```
In [42]: dy_dx_c = ((y[2:] - y[:-2]) /
   ....:             (x[2:] - x[:-2]))

In [43]: dy_dx_c
Out[43]: array([ 4,  8, 12, 16, 20, 24,
                28, 32])
```

In pure Python, these operation would be written using a `for` loop. For `x` containing 1,000 elements, the NumPy implementation is 100 times faster.

Using Broadcasting to Create a Grid

Suppose we want to produce a 3D grid of distances,

$$R_{ijk} = \sqrt{i^2 + j^2 + k^2},$$

with $i = -100 \ldots 99$, $j = -100 \ldots 99$, and $k = -100 \ldots 99$. In most vectorized programming languages, this would require forming three intermediate $200 \times 200 \times 200$ arrays, i, j, and k, as in:

```
In [44]: i, j, k = np.mgrid[-100:100,
   ....:                    -100:100,
   ....:                    -100:100]

In [45]: print(i.shape, j.shape, k.shape)
((200, 200, 200), (200, 200, 200),
 (200, 200, 200)

In [46]: R = np.sqrt(i**2 + j**2 + k**2)

In [47]: R.shape
Out[47]: (200, 200, 200)
```

where the special `mgrid` object produces a mesh grid when sliced.

In this example, we've allocated four named arrays—i, j, k, and R—and an additional five temporary arrays over the course of the operation. Each of these arrays contains roughly 64 Mbytes of data resulting in a total memory allocation of approximately 576 Mbytes. Altogether, it performs 48 million operations: 200^3 to square each array and 200^3 per addition.

In a non-vectorized language, we wouldn't need to allocate any temporary arrays when the output values are calculated in a nested `for` loop; in C, for example,

```
double R[200][200][200]; /* Assume
                            sufficiently
                            large stack */

int i, j, k;

for (i = -100; i < 100; i++)
  for (j = -100; j < 100; j++)
    for (k = -100; k < 100; k++)
      R[i + 100][j + 100][k + 100] = \
          sqrt(i*i + j*j + k*k);
```

We can achieve a similar effect using NumPy's broadcasting facilities. Instead of constructing large temporary arrays, we instruct NumPy to combine three 1D vectors (a row, a column, and a

depth vector) to form the 3D result. Broadcasting doesn't require large intermediate arrays.

First, construct the three coordinate vectors (*i* for the *x*-axis, *j* for the *y*-axis and *k* for the *z*-axis):

```
# Construct the row vector that runs
# from -100 to +100
i = np.arange(-100, 100).reshape(200, 1, 1)

# Construct the column vector
j = np.reshape(i, (1, 200, 1))

# Construct the depth vector
k = np.reshape(i, (1, 1, 200))
```

NumPy also provides a shorthand for the above construction, namely

```
i, j, k = \
   np.ogrid[-100:100, -100:100, -100:100]
```

Note how the arrays contain the same number of elements, but have different orientations. We now let NumPy broadcast *i*, *j* and *k* to form the 3D result, as Figure 1 shows.

```
In [48]: R = np.sqrt(i**2 + j**2 +
   ....:             k**2)

In [49]: R.shape
Out[49]: (200, 200, 200)
```

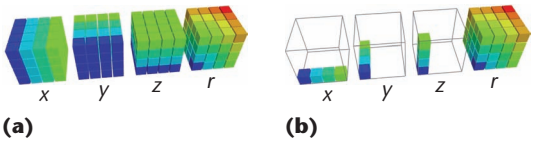
Here, the total memory allocation is only 128 Mbytes: four named arrays totaling approximately 64 Mbytes (1.6 Kbytes $\times$ 3 + 64 Mbytes) and five temporary arrays of approximately 64 Mbytes (1.6 Kbytes $\times$ 3 + 320 Kbytes + 64 Mbytes). A total of approximately 16 million operations are performed: 200 to square each array, 200^2 for the first addition, and 200^3 each for the second addition as well as for the square root.

When using naive vectorization, calculating *R* requires 410 milliseconds to compute. Broadcasting reduces this time to 182 ms—a factor two speed-up, along with a significant reduction in memory use.

Computer Vision

Consider an $n \times 3$ array of 3D point coordinates and a 3×3 camera matrix:

```
points = np.random.random((100000, 3))
camera = np.array([[500.,   0., 320.],
                   [  0., 500., 240.],
                   [  0.,   0.,   1.]])
```



(a) **(b)**

Figure 1. Memory use: broadcasting versus dense grid computation. (a) Computing the grid values without broadcasting. (b) Using broadcasting to reduce memory use.

Often, we want to transform the 3D coordinates into their 2D pixel locations on the image as viewed by the camera. This operation involves taking the matrix dot product of each point with the camera matrix, and then dividing the resulting vector by its third component. With NumPy, we write it as

```
# Perform the matrix product on
# all the coordinates
new_vecs = np.dot(camera, points.T).T

# Divide each resulting coordinate by
# its z-value
pixel_coords = (new_vecs /
                new_vecs[:, 2, np.newaxis])
```

The `dot` array method—which leverages accelerated basic linear algebra subroutines (BLAS) implementations, if available—implements the matrix product rather than the element-wise product `*`. It can be applied to 1D or 2D arrays. For our test arrays, this code executes in 9 ms—a 70 times speedup over a Python `for` loop version.

Aside from the optimized NumPy dot product, we use NumPy's array operations with element-by-element division and the broadcasting machinery. The code `new_vecs / new_vecs[:, 2, np.newaxis]` divides each column of `new_vecs` by its third column (in other words, each row is divided by its third element). The `np.newaxis` index is used to insert a new axis, thereby changing `new_vecs[:, 2]` into a column-vector so that broadcasting can occur.

The above examples show how vectorization provides a powerful and efficient means of operating on large arrays, without compromising clear and concise code or relinquishing control over aspects such as memory allocation. However, using vectorization and broadcasting is no panacea; for example, when repeated operations take place on large chunks of memory, it might be better to use an outer `for` loop combined with a vectorized inner loop to optimize use of the system cache.

Sharing Data

As we describe earlier, performance is often improved by preventing the repeated copying of data in memory. We now show how NumPy uses foreign memory—that is, memory it doesn't allocate or control—without copying data.

Efficient I/O with Memory Mapping

An array stored on disk can be addressed directly without copying it to memory in its entirety. This technique, known as *memory mapping*, is useful for addressing only a small portion of a large array (such as produced by an external instrument, for example).

NumPy supports memory mapped arrays with the same interface as any other NumPy array. First, we'll construct such an array and fill it with some data:

```
In [50]: a = np.memmap(
   ....:       '/tmp/myarray.memmap',
   ....:       mode='write',
   ....:       shape=(300, 300),
   ....:       dtype=np.int)

# Pretend "a" is a one-dimensional,
# 300*300 array and assign values
# into it
In [51]: a.flat = np.arange(300 * 300)

In [52]: a
Out[52]:
memmap
([[    0,     1,   ...,   298,   299],
  [  300,   301,   ...,   598,   599],
  [  600,   601,   ...,   898,   899],
  ...,
  [89100, 89101,   ..., 89398, 89399],
  [89400, 89401,   ..., 89698, 89699],
  [89700, 89701,   ..., 89998, 89999]])
```

When the "flush" method is called, its data is written to disk:

```
In [53]: a.flush()
```

The array can now be loaded and parts of it manipulated; calling "flush" writes the altered data back to disk:

```
# Load the memory mapped array
In [54]: b = np.memmap(
   ....:       '/tmp/myarray.memmap',
   ....:       mode='write',
   ....:       shape=(300, 300),
   ....:       dtype=np.int)

# Perform some operation on the
# elements of b
In [55]: b[100, :] *= 2

# Store the modifications to disk
In [56]: b.flush()
```

The Array Interface for Foreign Memory Blocks

Often, NumPy arrays must be created from memory constructed and populated by foreign code, such as a result produced by an external C++ or Fortran library. To facilitate such exchanges without copying the already allocated memory, NumPy defines an *array interface* that specifies how a given object exposes a block of memory. NumPy knows how to view any object with a valid `__array_interface__` dictionary attribute as an array. Its most important values are `data` (address of the data in memory), `shape`, and `typestr` (the kind of elements stored).

The example in Figure 2 defines a `MutableString` class that allocates a string `_s`. The `MutableString` represents a foreign block of memory, now made available to NumPy by defining the `__array_interface__` dictionary. We make use of the ctypes library, which lets Python execute code directly from dynamic C libraries. In this instance, its utility function, `create_string_buffer`, allocates a string and `addressof` establishes that string's position in memory.

Once the class is instantiated, NumPy is asked to interpret it as an array, which is possible because of its `__array_interface__` attribute.

```
# Create an instance of our mutable
# string class
In [57]: m = MutableString('abcde')

# View the character byte values
# as an array
In [58]: am = np.asarray(m)

In [59]: am
Out[59]: array([ 97,  98,  99, 100, 101],
               dtype=uint8)

# Modify the values of the array
In [60]: am += 2

In [61]: am
Out[61]: array([ 99, 100, 101, 102, 103],
               dtype=uint8)
```

```
# Since the underlying memory was
# updated, the string now has a
# different value
In [62]: print m
cdefg
```

As this example shows, NumPy can interpret any block of memory if the necessary information is provided via an `__array_interface__` dictionary.

Structured Data Types to Expose Complex Data

NumPy arrays are homogeneous—that is, each array element has the same data type. Traditionally, we think of fundamental data types, such as integers and floats. However, NumPy arrays can also store compound elements, such as the combination `(1, 0.5)`—an integer *and* a float. Arrays that store such compound elements are known as *structured arrays*.

Imagine an experiment that records measurements of two fields:

- a time stamp, in nanoseconds (a 64-bit unsigned integer); and
- a position (x and y coordinates, stored as floating-point numbers).

We can describe these fields using a single data type:

```
In [63]: dt = np.dtype([
   ....:     ('time', np.uint64),
   ....:     ('pos', [
   ....:         ('x', np.float),
   ....:         ('y', np.float),
   ....:             ],
   ....:         ),
   ....:     ])
```

We can then construct an array of measurements using this data type as

```
In [64]: x = np.array([
   ....:         (100, (  0,  0.5)),
   ....:         (200, (  0, 10.3)),
   ....:         (300, (5.5, 15.1)),
   ....:              ], dtype=dt)
```

We can also query the individual fields of such a structured array:

```
# Display all time-stamps
In [65]: x['time']
Out[65]: array([100, 200, 300],
               dtype=uint64)
```

```
import ctypes

class MutableString(object):

    def __init__(self, s):

        # Allocate string memory
        self._s = ctypes.create_string_bufer(s)

        self.__array_interface__ = {

            # Shape of the array is the
            # length of the string
            'shape': (len(s),),

            # Address of data, the memory is
            # not read-only
            'data': (ctypes.addressof(self._s),
                     False),

            # Stores 1-byte unsigned integers.
            # The "|" indicates that
            # Endianess is irrelevant for
            # this data-type. The "<" and
            # ">" symbols indicate little
            # and big endianess, respectively.
            'typestr': '|u1',

            }

    def __str__(self):
        """
        Convert the buffer to a string
        for printing.
        """
        return str(buffer(self._s))
```

Figure 2. Defining a `MutableString` class. The class allocates a string `_s`. The `MutableString` represents a foreign block of memory, now made available to NumPy by defining the `__array_interface__` dictionary.

```
# Display x-coordinates for all
# timestamps >= 200
In [66]: times = (x['time'] >= 200)

In [67]: print times
[False True True]

In [68]: x[times]['pos']['x']
Out[68]: array([ 0.,  5.5])
```

This example also shows how arrays are indexed using Boolean "masks." For example, when `x` is indexed by `times`, only the elements corresponding to a `True` value in `times` are returned.

Structured arrays are useful for reading complex binary files that contain sequential,

fixed-length records. Suppose we have a file `foo.dat` that contains binary data structured according to the data type `dt` introduced earlier. For each record, the first 8 bytes are a 64-bit unsigned integer time stamp and the next 16 bytes are a position comprised of two 64-bit floating-point numbers, *x* and *y*. Loading such data manually is cumbersome: for each field in the record, bytes are read from file and converted to the appropriate data type, taking into consideration factors such as endianess. In contrast, using a structured data type simplifies this operation to a single line of code:

```
data = np.fromfile('foo.dat',
                   dtype=dt)
```

When NumPy is skillfully applied, computation time is primarily spent on vectorized array operations instead of in Python `for` loops (which are often a bottleneck). Further speed improvements are achieved using optimizing compilers, such as Cython, which allow better control over cache effects.

In addition to low-level array operations, NumPy provides subpackages for linear algebra, FFTs, random number generation, and polynomial manipulation. Larger scientific packages, such as SciPy, are, in turn, built on this infrastructure.

NumPy and similar projects foster an environment in which users can easily describe numerical problems using high-level code, thereby opening the door to scientific code that is both transparent and easy to maintain.

NumPy is a volunteer effort, and its documentation is maintained using a Wikipedia-like community forum (http://docs.scipy.org). Discussions are held on the project mailing list at www.scipy.org/Mailing_Lists.

References

1. F. Perez and B.E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 21–29.
2. J. Bergstra, "Optimized Symbolic Expressions and GPU Metaprogramming with Theano," *Proc. 9th Python in Science Conf.* (SciPy2010); forthcoming; http://conference.scipy.org/proceedings.html.

S. Chris Colbert *is a scientific software developer at Enthought, Inc. His research interests include machine vision, digital image processing, and data visualization. Colbert has an MS in mechanical engineering from the University of South Florida. Contact him at chris.colbert@enthought.com.*

Stéfan van der Walt *is a researcher and lecturer in applied mathematics at Stellenbosch University, South Africa. His research interests include mathematical modeling in neuro-imaging, computer vision and scientific computation. Van der Walt has a PhD in electronics engineering from Stellenbosch University. Contact him at stefan@sun.ac.za.*

Gaël Varoquaux *is a computational science research fellow in the Neurospin Research Institute at the French National Institute for Research in Computer Science and Control (INRIA) in Paris. His research interests include statistical and computational techniques for probabilistic modeling of intrinsic brain activity using functional imaging, and in making advanced data-processing techniques available across new scientific fields. Varoquaux has a PhD in quantum physics from the Université Paris Sud Orsay, France, and is a member of IEEE. Contact him at gael.varoquaux@normalesup.org.*