numarray: A New Scientific Array Package for Python

Perry Greenfield, Jay Todd Miller, Jin-chung Hsu, & Richard L. White
Science Software Branch

Space Telescope Science Institute.
3700 San Martin Dr.
Baltimore, MD 21218
perry at stsci.edu

Python has long had an array module (Numeric) for science and engineering applications; why a replacement? We explain the motivations for developing numarray, which are primarily, though not entirely focused on enabling the use of larger arrays and data sets that Numeric has difficulty handling. We also describe the design issues in its development and its new features and capabilities. Numarray is highly compatible with Numeric, including the C-API, though there are some differences that are discussed. Numarray is sufficiently well developed that it is being used in production pipelines to reduce and calibrate Hubble Space Telescope (HST) data and is being distributed to HST users along with applications for data reduction. Finally, we outline planned enhancements and improvements. Numarray is available from the Sourceforge numpy project page.

Introduction

The Space Telescope Science Institute began using Python as a means of bridging an old legacy system with more modern software as part of an effort called PyRAF (White & Greenfield, 2000). Our success in doing so led us to believe that Python would also prove a productive language for writing astronomical data analysis applications. Whether Python is a suitable language for scientific or engineering development depends on various considerations. One of these is the ability to manipulate large sets of numbers efficiently without suffering too much penalty in execution time. Scientists and engineers may be willing to tolerate slower programs, but if the slowdown means that programs that took 5 minutes with a compiled language will take 2 hours in Python, it will not be accepted.

A common way for interpreted languages to deal with this issue is to adopt an array or matrix facility that allows actions or operations to be performed on whole arrays. Since the array operation is generally carried out in efficient compiled code, the overhead of interpretation becomes negligible if the arrays are sufficiently large. There are many such examples: APL, J, Matlab, IDL, and Octave are some of the better known.

Such tools may or may not be sufficient to allow development in an interpreted language; this generally depends on whether the problems to be solved can be efficiently cast into array operations. If they can't, then such facilities are primarily useful for visualization, and perhaps for scripting compiled codes (Beazley, 2000). But many other classes of

problems are well suited for array manipulations. In fact, IDL is popular in astronomical circles, and experience has shown that many useful astronomical applications can be developed with an array facility. We therefore had no doubts that such a facility in Python would be useful for our purposes.

The Numeric Module

To our benefit, an array manipulation package was already available in Python: Numeric (also known as Numpy). Numeric was developed primarily by Jim Hugunin, though many others contributed (see acknowledgements).

Before detailing the reasons why we were motivated to re-implement Numeric, a brief overview of the capabilities of Numeric is in order. The crux of array languages is the avoidance of explicit loops to carry out operations on all the array elements. Not only is it a matter of syntactic convenience, the fact that the loops over all elements are implicit means that they can be performed efficiently. This is because the loop is performed in the efficient, compiled implementation language rather than in Python. The following summarizes the kinds of operations that can be performed efficiently on arrays. Although this section references Numeric, the examples actually use numarray; for most of the examples the differences are slight.

Array Creation

Arrays may be created several different ways. In a real application they are likely to be obtained from data files or instrumentation, but there are more direct ways to create arrays, particularly for the purpose of examples.

```
>>> from numarray import *
>>> x = array([5,2,3,1,5]) # create an array from a list
>>> y = arange(5, type=Float32) # Create an array of single
                                # precision floats a la range()
>>> z = zeros((5,6)) # create a 5x6 array of zeros
>>> print x
[5 2 3 1 5]
>>> y
array([ 0.,  1.,  2.,  3.,  4.], type=Float32)
>>> print z
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

Arrays may also be created from binary data (either in strings or files). Most, if not all, numerical types from 8-bit unsigned integers to double precision complex numbers are supported.

Note that that Numeric and numarray adopt the C convention for array ordering; the last index represents the most rapidly varying element in the array. In other words, row-major ordering is used.

Universal Functions

Universal Functions (or more commonly, Ufuncs) are mathematical functions that can be applied to arrays element by element. Examples would be addition (in which case there would be two input arrays) and the trigonometric sine function. More specifically for the above examples:

```
>>> print x + y
[ 5.  3.  5.  4.  9.]
```

Note that the addition of the two array objects resulted in the addition of each corresponding array element.

```
>>> print sin(y)
array([ 0.         0.84147096  0.90929741  0.14112    -0.7568025],
type=Float32)
```

Here `sin()` is applied to each element. The standard Numeric also includes the basic mathematical operators and functions.

Array Structural and Conversion Operations

Array structural operations involve any creation of new array objects from existing ones by means of subset selection, rearrangement or combination. New arrays may be efficiently sliced (including use of strides, i.e., skipping *n* elements), reshaped, transposed, concatenated, sorted, or expanded by replication. Generally most such operations are functional, though a few (like flattening an array) can be performed as methods. Arrays may be converted from any supported type to another.

Array Element Selection and Manipulation

There are innumerable occasions that require treating a subset of array values specially rather than treating all elements identically. Such operations are performed by functions that can operate on a subset of elements by reason of value rather than their location. This includes being able to identify all elements satisfying some logical condition (e.g., greater than some threshold) and subsequently being able to modify such values. The following illustrates how this is typically done in numarray (Numeric does not provide the syntactic indexing convenience of the last example; the `put()` function must be used to accomplish the same action)

```
>>> print x
[5 2 3 1 5]
>>> print x < 3   # Results in Boolean array showing results for
[0 1 0 1 0]       # applying results to each element.
                  # (This uses “rich” comparisons that requires
                  # Python 2.1 or later)

>>> print nonzero(x < 3)  # Get indices where expression is true
(array([1, 3]),)
>>> x[nonzero(x < 3)] = 0 # Use previous expression to set values
>>> print x
[5 0 3 0 5]
```

The expression "x<3" (a Ufunc) produces an array where all array elements that satisfy the condition have true values, and those that don't have false values. The "nonzero" function produces an array of indices where the input array is nonzero. When an array is given as an index to another array, it uses those array values as indices to get or set (set in this case). The end result is that all array elements that have a value less than 3 will be set to 0. This example is both syntactically straightforward (the mathematical meaning is clear) and performed efficiently for large arrays.

Other Functions

There are many useful functions that are not applied element-by-element and so cannot be implemented as Ufuncs. This includes functions such as matrix multiplication, dot products, and sums over array elements. While a number of such functions (such as those just mentioned) are in the base Numeric, it generally is expected that most such functions will be in Numeric extension modules (such as FFT and Linear Algebra).

The numarray and Numeric manuals provide comprehensive descriptions of the capabilities available.

Motivation for a Numeric Replacement

Despite the tremendous functionality of Numeric, some capabilities are lacking that meant Numeric would be impractical for applications we need to write. The most important considerations were

1) Memory efficiency. We deal with large images, some larger than 4K by 4K pixels. Such arrays consume significant amounts of memory; it is important for us to minimize the memory usage to the greatest extent. To do so required two basic changes to Numeric capabilities, namely avoiding creation of temporary arrays when possible (such as when Numeric coerces an input array to a type expected by a function or operator) and the ability to use memory-mapped files. The latter allows accessing subsets of large datasets on disk in sections using array slicing thus reducing memory usage.
2) We deal with much data in tabular format. A means of efficiently accessing large binary tables was also important to us. Since such arrays of records may be accessed by column or row, copying the data from such data sets into Numeric arrays results in unnecessary copying as well as complicating the ability to update individual records.
3) The ability to easily extend arrays to other classes of objects without having to reproduce the common machinery of performing structural operations on arrays such as slicing or reshaping by use of class inheritance.
4) Many Numeric users' desire for such a library to be made available as part of the Standard Python distribution. Such inclusion would allow authors of other important Python modules to add the ability to access arrays in their code knowing that such objects are a standard Python format. It is for this primary reason that PIL, wxPython and other modules and packages do not handle arrays though Numeric arrays are an eminently sensible format for such packages. However, there was consensus that existing Numeric code was not suitable for inclusion into the Standard Library, at least without massive reorganization.

There are other less important reasons as well; these will be mentioned later primarily under new features or compatibility issues (in other words, most of the incompatibilities with Numeric are intentional).

Design Issues

The key design consideration is achieving both speed and memory efficiency. It is relatively simple to obtain one at the expense of the other. Speed has traditionally been achieved in array packages by making the array operations as efficient as possible when looping over all the elements in an array. This is invariably accomplished by having these loops coded (either explicitly, or implicitly) in a compiled language such as C. In this way, even though the interpreted language is relatively slow, the time to perform an operation on the array takes little longer than it would if coded in a compiled language. Most of the time, for large arrays, is spent in the loop that is written in such a compiled language. This is not a difficult problem for the specific case of dealing with fixed input and output types. But an array manipulation package must deal with many numeric types, from byte integers to double precision complex numbers. Devising a system for handling all the possible combinations of types is what adds all the complexity.

The first two motivations have profound impacts on the design of an array package in ways that are not immediately obvious. If memory-mapped arrays are to be supported, a mechanism for handling data in non-native processor ordering is imperative. It is common to store data arrays in a machine-independent format, and for some processors, this requires byte-swapping the data. But how can this be handled in a transparent way while still keeping the data on disk as the primary data object? The situation is complicated further by the desire to support arrays of records. With arbitrary records comprised of heterogeneous data types (e.g., 1-byte integers through 8-byte floats) one faces the prospect of some numeric quantities in the data buffer having address alignment issues, that is, it is necessary to copy the item to an aligned memory location or register before it can be accessed as that type. How is that to be handled transparently?

This requirement alone requires a complete reworking of how arrays are manipulated internally, compared to the design of Numeric that has no means of solving these problems. This issue was the single most difficult design problem.

Options Considered

We considered several approaches, but found only one practical. The main problem is limiting the combinatorial complexity of all possible array types and representations (i.e., byte-swapping and non-aligned data). In principle, one could create an addition function for all possible combination of types and representations. This, however, leads to an explosion of the number of C functions required (hundreds for each binary operation).

Numeric solves this problem by limiting the number of operator function signatures, and instead relying on the creation of temporary arrays to match the existing function. For example, if a Float32 array is added to an Int16 array, the Int16 array will be converted to a temporary Float32 array before calling a function that expects two Float32 array inputs. Likewise, if an array had to be byte-swapped or aligned, the conversion process would

also handle these issues. Unfortunately, this approach was unacceptable since it was wasteful of memory.

Yet another approach is to call conversion functions inside the C loop that iterates over an array. In this way, pointers to the appropriate conversion functions (for type, byte-swapping, or alignment) can be passed to the operator function, thus reducing the number of function versions tremendously. This approach is the most elegant, but suffers significant speed penalties. Our benchmarks found that regardless of optimizations, it generally suffered a factor of 3 or worse performance penalty over the fastest forms of processing loops.

The Blocking Approach

The approach we chose was a hybrid of the last two. Temporaries are used, but only for sub-blocks of the array, if it is large. In essence, the array is chunked through, conversions performed on each of these blocks, then the operator function called, until the whole array is completed. This leads to more complicated loop handling, but reduces memory usage without serious speed penalties. The block size is chosen to be as large as possible, but small enough to avoid memory-caching problems.

Another design choice included slicing as one of the transformation operations. Since a slice of a Numeric or numarray array does not result in a copy but rather a new view into the same data buffer, the resulting array may not have data that are contiguous in memory. The basic operator functions are designed to operate on simple contiguous arrays. This is to keep them very simple, and allow the compiler to optimize them as much as possible. Thus non-contiguous arrays, such as sliced arrays, are first copied to a contiguous temporary block (much like conversions are performed) before other operations can be performed.

The block size used for a particular case is determined from the dimensions of the array. The algorithm used for the current implementation of numarray will always use a blocksize at least half as large as the largest blocksize permitted (presuming the entire array is at least that large). The only exception is for the last block. That block may be smaller than half the largest size permitted. If the last dimension is larger than the maximum blocksize, that dimension will be partitioned into different blocks. If the last dimension is smaller than a blocksize, then the block will consist of as many whole dimensions that will fit within the maximum blocksize.

This is best demonstrated with examples. Suppose the maximum permitted blocksize is 10,000 bytes and we are dealing with a 20x20x20x20 array of Int32 values (total size of 640,000 bytes). The block used in this case will be a subarray of size 6x20x20 elements (size 9600 bytes). The iteration over blocks will be performed 20 x 4 times (20 for the size of the first dimension, and four times for the second dimension (3 whole subarrays for the second dimension plus a leftover subarray unit of size 2x20x20 elements since the first 3 iterations account for 18 of the 20 values possible leaving 2).

If instead the array had dimensions 20x9000 and the same maximum permitted blocksize, the last dimension would be subdivided into 3 2,500 element blocks with a leftover block of 1,500 elements. There would be 20 times 3 iterations over full-size blocks and 20 iterations over the leftover blocks.

The following schematically illustrates how a more complex case would work when adding a large (i.e., multi-block sized) byte-swapped Int32 array with a strided (i.e., non-contiguous) Unsigned Int32 array. To complicate the case, an existing Float64 array (of proper shape) is provided for the results. After the appropriate block size is determined a loop is set up to iterate over blocks. The output type must be Int64 to retain the full range of both inputs. For each iteration:

1) Copy a block from Int32 array to a temporary block buffer and byte-swap all elements in the process.
2) Copy the data from the temporary block buffer produced by step 1 to a second temporary block buffer, converting all elements to Int64
3) Copy a block from the Unsigned Int32 array to a temporary block buffer, thus producing a contiguous array from a noncontiguous one.
4) Copy the data from the temporary block buffer produced in step 3 to another temporary block buffer, converting all elements to Int64 in the process.
5) Add the elements in the output block buffers of Steps 2 and 4 and place the results in another temporary block buffer
6) Copy the results from the block buffer produced by step 5 to the appropriate location in the provided output array, converting all elements to Float64 in the process.

Each of these steps is performed by a very simple C function that iterates over all elements for the types expected in the input and output buffers. A considerable amount of logic is involved in determining what steps are necessary, and which C function is needed for that step. As an example, there is an addition function for every possible type of input, but all the addition functions require both inputs be the same type, hence the necessity for type conversion to be part of the block loop.

C Function Management

The C functions are handled as objects that can be passed into the block loop management code. The C functions are encapsulated Python types so that they may only be called in the proper context. This C function type is passed into the C code that sets up the call to the encapsulated C function. This C code checks the attributes associated with the C function to make sure that it is consistent with the array data passed to it. Otherwise it would be possible for Python code (mistakenly or maliciously) to call a C function mismatched with the data. For example, a C function that expects to iterate over Float32 data being given Int8 data would result in a buffer overrun and likely crash the process. The C function objects for Ufuncs are retrieved by using their input type signatures as keys in a dictionary of C function objects.

Array Class Hierarchy

In order for more complex array types, such as record arrays, to be created, the design of the array objects has been split into two classes. NDArray encapsulates all the structural aspects of arrays without regard for their contents. This includes all indexing, slicing, sorting, reshaping, concatenation and related operations on arrays. All numerical

functionality is implemented in a subclass of NDArray. In this way, it was possible to easily implement record arrays and character arrays as different subclasses of NDArray without having to reproduce the sizable machinery for structural operations.

Exception handling

Numeric currently has no support for IEEE-754 floating-point exception handling. There is no way to trap or otherwise be aware of such exceptions other than to check all results for new appearances of NaN and Inf values. We recognized that a numerical package needs to offer flexibility; no one approach to such exceptions is suitable for all problems. Numarray allows a user to select one of three modes of floating point exception handling. These modes can be independently set for four different kinds of exceptions: Invalid results (i.e., NaN), Overflows, Underflows, and Divide-by-zero). The three modes available are: Ignore (Numeric's behavior), Warn, or Raise. Warn results in a printed warning message and Raise results in a Python exception being raised. The latter two happen at the end of an array operation. There is no support for raising exceptions in the middle of an array operation nor is any planned.

The exception handling features are achieved by making use of the floating-point processor's sticky bits. The bits are cleared at the beginning of an operation and checked at the end. This is one of the few areas of numarray that has any platform-dependent code. It has been suggested that such code should eventually be migrated to the Python core for use by Python for scalar operations as well (Peters, 2001).

We have also used this mechanism for problems with integer operations as well. The C code that handles integer operations checks for overflow or divide-by-zero problems; if any are encountered, a floating point exception of the corresponding type is intentionally triggered so that the same mechanism may be used for all numeric types.

Buffer Objects

So that memory mapped files may be used as buffers for numarray objects, numarray uses the buffer object interface for its in memory buffers and memory mapped files. There are serious problems with the current buffer object that are driving an effort for their elimination. This will not be a problem so long as some mechanism is provided so that there is a means for using memory-mapped files as data buffers. There has been much discussion of the appropriate solutions, but little progress to date as to a final resolution of this issue. (See *Adding a bytes object type*, Python Enhancement Proposal (PEP) 296 & *The locked buffer interface*, PEP 298.)

Implementation Approach

The initial implementation was almost entirely in Python with only the very low level functions that performed mathematical operations written in C. Rather than use the template features of C++, we instead decided to use Python functions to generate the repetitive C code for the functions. The code generation is an intrinsic part of the building process (which uses distutils). It is our intent to keep as much of the code in Python as possible. Optimizations will mean some code will be migrated to C, but optimizations will be tried in Python whenever possible before migrating any code to C.

The goals for numarray functionality are more ambitious than that for Numeric; as a result one would expect numarray to be more complex. On the other hand numarray has most of its implementation in Python that makes it simpler. A comparison of the lines of code that each have illustrates that the use of Python offsets the extra complexity introduced by the new requirements.

	Numeric v22.0	numarray 0.4
C source code:	11.9K LOC	3.0K LOC (excluding program-generated code)
Python source:	2.1K LOC	11.7K LOC

C-API

Given that not all scientific and engineering algorithms can be efficiently handled with the base capabilities of Numeric/numarray it is essential that a means of extending numarray with compiled code be made available through a C-API. This allows building widely used extension modules such as an FFT module as well as allowing individuals to develop specialized modules for their own use.

As numarray matured, the attributes previously stored at the Python level were migrated to C structures. This allowed reuse of the existing Numeric array struct (with the addition of additional structure members). The C-API is now largely compatible with the Numeric API, the differences are limited to fairly little used aspects of the Numeric API. This has and will allow much easier migration of previous extension modules for Numeric to numarray.

The existence of more complex array representations means a richer API is necessary so that byte-swapping and alignment issues can be dealt with easily in C extensions. Unfortunately there does not exist a single, simple API that handles these issues in a manner that is simple to use, fast, and memory efficient in all cases. For that reason we provide four different approaches to suit the various needs that will arise.

C-API Examples

High-Level API

The simplest approach is to call an API function that will return a contiguous array that is of proper byte order and alignment that can easily be used by C code (as well as C++ and Fortran). The numarray High-Level API converts any array that isn't contiguous, aligned, or properly byte-ordered (such arrays hereafter will be referred to as "non-C-arrays") into one by copying it to a temporary array. This API can also handle any Python sequence of numbers so long as it can be sensibly converted to a numarray array.

Example: Obtaining a pointer to a C-array from any kind of numarray object or Python sequence of numbers

```
PyArrayObject *array = NA_InputArray(inputarray, tFloat64, C_ARRAY);
Float64 *data = (Float64 *) NA_OFFSETDATA(array);
```

In this example inputarray is a PyObject that itself is a numarray or any Python sequence of numbers (perhaps nested). The first line obtains a reference to double precision floating point C-array version of the input array (either a direct reference to inputarray if

it already is a double precision float C-array, or a temporary copy). The second line obtains a pointer to the beginning of the data array. C_ARRAY is a bit mask that indicates that the reference must be a contiguous, aligned, non-byte swapped array. It is possible, by use of other masks, to place fewer restrictions on the array (e.g., permit non-contiguous arrays).

Pros:
- Fast
- Simple code
- Direct access to data possible

Cons:
- Possible increased memory usage

Element-wise API

This API is used to access single elements of arrays that may be non-C-arrays or of incorrect type. The API consists of functions that obtain individual elements, handling any byte-swapping, alignment, non-contiguity, or type issues, and macros that only handle byte-swapping and alignment. Different versions of the functions support access for 1, 2, or 3 dimensional arrays.

Example: obtain value of *i,j* element from 2-d array as double precision float value.

```
Float64 alpha = NA_get2_Float64( inputarray, i, j);
```

Pros:
- Simple code
- Memory efficient

Cons:
- Slow

One-dimensional API

This API blends the good performance of the High-Level API with the low memory footprint of the Element-wise API by converting a series of array elements into C-Arrays with one function call. While the 1-D API makes it possible to efficiently access non-C-arrays without creating a temporary copy of the entire array, it does require more careful thought than either of the previous two APIs. In particular, whereas the previous APIs allow the C code to isolate the interface to numarray to the input or output stages thus isolating the API calls from the actual computational loops, this API requires use within the computational loop. It works best when one of the dimensions of the array is large, but not too large.

Example: Looping over large 2-d array

```
for(k=0; k<100; k++) {
      Float64 inner_band[256];
      long offset = NA_get_offset(arrayObject, 1, k);
           NA_get1D_Float64( arrayObject, offset, 256, inner_band);
      /* do something with inner_band */
}
```

Pros:
- Fast
- More memory efficient

Cons:
- More complex code

Numeric Emulation API

This API reproduces the interface and semantics of many of the Numeric C-API functions. In general, shallow wrapper functions around the High-Level API provide this capability. Existing code developed for Numeric can often simply be recompiled after simply changing the include file from Numeric to numarray. There are some exceptions described in the section on incompatibilities.

Example:

```
PyArrayObject *array = PyArray_ContiguousFromObject(
      objectFromArgTuple, tFloat64, 2, 2);
```

Pros:
- Backward compatible API
- Handles non-C-arrays
- Fast
- Direct access to data

Cons:
- Possible increased memory use

Features and Compatibility

Summary of New Features

1) **Ability to handle byte-swapped data transparently.** One may access data in arrays where the data is not in the processor's native byte order without having to produce a copy of the array or change the data in place. This means one may access memory-mapped data without having to copy (as a whole) it or change it on disk.
2) **Ability to handle arbitrary byte offsets and strides transparently (i.e., non-aligned data).** Regularly spaced data, even though it may have odd byte offsets between elements may be directly accessed. Such data may appear in arrays of binary records, for example.
3) **Memory mapped arrays.** Arrays may be created from memory-mapped files.
4) **New type objects.** In Numeric Float32 is an alias for the string 'f'. In numarray Float32 is an instance of a Floating Type object that is derived from a more general numeric type object. One can simply test for more general classes of type. For example:
   `>>> isinstance(arr.type(), ComplexType)`
5) **Faster, economical Ufunc code.** The C code used for Ufunc loops optimizes better.
6) **Improved coercion rules.** No longer does `x+1.` result in a double precision array when `x` is a single precision float array.
7) **Configurable handling of IEEE special values.** One can individually configure whether numarray ignores, warns, or raises an exception for underflows, overflows, divide-by-zeros, and invalid results. The same mechanism is available for Integer types.
8) **Arrays as indices.** One may use arrays as indices to other arrays. In such cases the array is interpreted as an array of indices used to create a new array. For example:

```
>>> ind = array([2,5,2,7])
>>> x = array([1,4,9,16,25,36,49,64,81,100])
>>> print x[ind]
[9 36  9 64]
```

Furthermore one may use an index array for each dimension so long as the index arrays are shape consistent. Unlike Numeric, `nonzero()` will work on multidimensional arrays and return an index array for each dimension.

9) **New array types** (Boolean, Int64)
10) **Record Arrays.** Records are similar to a struct containing supported numarray numeric types as well as fixed length character fields. One may access records by indexing elements of the record array, or fields of the record array that are either numarray or chararray objects without copying any data. For example

```
>>> r = recarray.array([(100, 2.5, 'abc'), (200, 3.5, 'xyz'),
                        (300, 4.1, 'pqr')], names = 'a,b,c')
>>> print r[0]   # first row
(100, 2.5, 'abc')
>>> columnb = r.field('b')  # Numeric array view of column b
>>> print columnb
[ 2.5   3.5   4.1]
>>> columna = r.field('a')  # likewise for column a
>>> print columna * columnb
[  250.  700.  1230.]
>>> columna[0] = 3000 # Change first element in column
                      # illustrating shared nature of
                      # record and column views.
>>>  print r[0]
(3000, 2.5, 'abc')
```

11) **Character Arrays.** These are arrays of fixed length strings that are contained in the same data buffer.
12) **Functions to set and test for IEEE-754 floating point special values.** This allows the use of IEEE-754 special values as an alternative to using mask arrays to track problem values. While it isn't suitable for all such cases, particularly when it is desired to retain the original value, it does provide a comparatively high-performance means to track bad values and is an alternative to the MA package in such cases.
13) **Array subclassing.** It is straightforward, if less than trivial, to subclass arrays. Most practical cases require defining all operators for the subclass, so it is rarely a short class (though much easier than trying to develop a new array module from scratch!).
14) **Broadcastable extension functions.** There are relatively simple ways to make an extension function handle looping over extra dimensions. Thus if one were to write a coordinate transformation function for 3-dimensional points, the overhead of having Python iterate over millions of points would make the advantage of such an extension minimal. But with the broadcasting mechanism, one may supply a million by 3 array of point coordinates and have the transformation function iterate in C over all million points making it run much faster.
15) **Ufunc templating.** If one wishes to add Ufuncs for many numeric types, the code generation mechanism used by numarray itself is available for users as well.
16) **Package organization.** The current release is not organized as a package, but the next release will be.

Compatibility with Numeric

At the outset of the project, compatibility was not accorded a high priority. We attempted to apply the principle that compatibility should be attempted in the absence of a compelling reason for divergence; we weren't seeking incompatibility for its own sake. But as the project has developed there has been a greater desire to seek compatibility unless there were strong reasons against it. This has even extended towards making as much of the C-API as compatible as possible. There remain some areas, however, where compatibility is not achieved, either because it wasn't technically feasible or desirable. Nevertheless, after doing much of the implementation and confronting many of the interface issues we are impressed at how many of the interface decisions made for Numeric were wise.

1) Numeric uses single character codes for array types. For example 's' represents short integers, 'f' single precision floats and so forth. Within the Numeric module variables are defined with these values to give more descriptive names to the types (e.g., Int16 = 's') but at their root, the types are represented as characters. Many find this representation hard to remember (what character should represent an unsigned Int16?). Numarray instead uses a numarray type class hierarchy. The types used are instances of these type classes. In this way much of the information about types is encapsulated with the type object. This includes information about the nature of the type (e.g., Integer or Floating), the size of the type, and the conversion functions for converting to other numarray types. Comparison methods are defined for these objects so that simple comparison to the traditional typecodes will work successfully and backward compatibility is mostly retained.
2) The type coercion model that Numeric uses for binary operations and functions has proved annoying to many when one of the binary operation arguments is a Python scalar. Python only has a subset of the available numeric types for its scalars. Thus if one follows traditional coercion rules when combining a single precision floating point array with a Python float, a double precision array is the result. This is often of little consequence, but when large data sets are involved, such unexpected coercion wastes memory. Numeric has no good solution to this issue. Either one is forced to wrap scalars as rank-0 arrays (in effect changing the scalar's type) within expressions, or to use a "savespace" attribute for an array which means its type overrides any coercion rules. The latter can also lead to quite unexpected behavior. We have chosen a more pragmatic approach for numarray. When scalars and arrays are combined, the array will only be coerced to the scalar's type if the scalar is considered a higher kind of number than the array.

   Thus a float scalar added to an Int16 array will produce a double precision float array, but if a float scalar is added to a Float32 array, the resulting array will still be Float32. In other words, if the types of the scalar and array are of the same kind, either int, float, or complex, then the precision of the array dominates. This is contrary to typical coercion rules, but the lack of richer Python scalar types prevents a better solution.
3) The only existing aspect of the existing Numeric C struct not mirrored by the numarray API is the type descriptor. Not all fields of the type descriptor are

supported. There appears to be little code that uses these members so this incompatibility should not have a great impact.

4) Because numarray uses a different mechanism for ufuncs than Numeric, the C-API for ufuncs is completely different.
5) The `nonzero()` function returns a tuple of arrays in all cases. This was to support the use of the function's return value directly as an index to arrays. In cases where the array that nonzero is being used on is multidimensional, an index array is returned for each dimension and thus these arrays are packaged in a tuple. For consistency the same is done even when there is only one dimension. Numeric's `nonzero()` only works with one-dimensional arrays.
6) Numeric currently is inconsistent with regard to whether it returns rank-0 arrays or scalars as a result of single element indexing. Numarray always returns a Python scalar in such cases. (Exceptions to this rule may be introduced if there is support for quad precision floats added if the Python floats are implemented as C doubles.)
7) Changes to Numeric behavior considered but rejected:
    a) Copy vs. view semantics for slicing. Numeric creates a view of the data buffer for array slices. This is contrary to usual Python behavior for lists, tuples and strings. Whether numarray should adopt copy semantics was a subject for much internal and external debate. In the end the desire for compatibility dominated and the view semantics were retained.
    b) Some argued that all array references including array reductions and single element indexing should produce rank-0 arrays rather than Python scalars as to promote more generic programming. Since most functions may see Python scalars as arguments, we decided that it would be hard to rely on all arguments being some form of array. Instead, it was decided to provide tools to make it easier to cast all inputs into arrays so that subsequent code would not have to have many special cases for scalars. There was also controversy about what the proper behavior for rank-0 arrays were and there appeared to be a consensus that Numeric does not handle rank-0 arrays consistently. It also appears that rank-1 len-1 arrays have the properties desired for more generic code. As a result, an alternate reduce method for Ufuncs, `areduce()`, is provided to return a rank-1, len-1 array in cases where the result would be a scalar for reduce().
    c) Complex comparisons; to allow or not. Python (as of v2.1) does not permit comparison operators, aside from `=, !=` to be used on complex scalars. Some argued that numarray should allow complex comparisons to make code more generic. Instead, we decided that most practical cases using comparisons should use the `.real` attribute for arrays in cases where real or complex numbers may be expected. This attribute is now available for Numeric and numarray for any numeric type, and not just complex types. Complex comparisons will not be allowed (they raise an exception).
    d) Default axis order. Some Numeric functions use the last axis by default and others use the first. Some argued that a more consistent approach should always use the last. There are reasonable arguments for both sides. Thus the

Numeric behavior was retained for compatibility's sake. The recommendation is to use the axis keyword explicitly whenever possible to avoid confusion.

e) Customized behavior. It would be a straightforward matter to make numarray have different personality variants. Some that have been suggested is having versions that used copy semantics for slicing, matrix operations in place of the existing Ufunc operations (e.g., * would represent matrix multiplication rather than element-by-element multiplication), or different coercion rules. Despite being quite feasible, we deem such variants as being very destructive to being able to develop a common community of 3rd party extensions. Since some modules would presume one set of behavior within its own Python code, passing arrays with other behaviors could well break such modules, or worse, lead to undetected errors. We strongly discourage such variants.

Progress to Date

Numarray has essentially all the functionality present in Numeric. Compatibility issues continue to surface as Numeric modules and packages are ported to use numarray. The remaining compatibility issues are expected to be minor. There may well be some differences remaining that are not yet known and that for reasons of consistency or simplicity will remain. The Numeric manual has been adapted for numarray and is available. The project is hosted at http://sourceforge.net/projects/numpy along with Numeric. The latest release at the time of writing is v0.4.

The standard modules distributed with Numeric have been ported to numarray (with the exception of MA). These include RandomArray, FFT, and LinearAlgebra, as well as a new one called Convolve.

The Space Telescope Science Institute has been using numarray for software that is used in pipelines that routinely reduce and calibrate Hubble Space Telescope data. We also distribute numarray as part of our PyRAF software distribution that is freely available to the public, but is primarily intended for those analyzing HST data. We have a module that performs I/O with astronomy data; the capabilities present in numarray have allowed us to write this module entirely in Python and yet have efficient access to standard astronomy data files.

Performance

The single largest remaining issue is performance. The existing design means that numarray is quite fast for large arrays. This is no surprise since it underlay the reasons that we decided to reimplement Numeric. In general numarray can handle large arrays much better than Numeric, both in speed, and memory usage (indeed, there are examples that work fine for numarray but crash Numeric because of memory issues).

But since much of the implementation is in Python, it has considerably more overhead in setting up an array operation than does Numeric. We have begun the process of optimizing small array performance, i.e., reducing array computation setup time. The approach we are taking is to cache the results of previous setups. Thus, subsequent operations that use the same combination of array types and functions or operators will need minimal setup time. The work is ongoing but has already yielded significant overhead reductions.

The setup time for a simple array operation (no type conversions, byte-swapping, alignment, or striding) for numarray version 0.4 was approximately 50 times longer than for Numeric. The latest work, not yet released, has reduced this to about 6 times that of Numeric. Work on the more complex cases is still underway but we believe that we will be able to make the overhead comparable for that of the simple case.

We expect that further work should bring the overhead time to within at least a factor of 2 or 3 of Numeric's overhead time.

Performance is perhaps the single largest issue that prevents some users from wholeheartedly adopting numarray in place of Numeric besides the usual inertia and porting hassles.

Planned Enhancements

Work is underway to port, or aid in porting, some of the more popular Numeric packages. This includes MA, and scipy. Discussions have been started to incorporate more flexible recarray field definitions to support PyTable requirements (Alted 2002) and to incorporate changes submitted by Francesc Alted to improve recarray performance.

Other improvements are possible though not on our immediate schedule. Allowing numarray to handle arrays larger than 2GB is one such enhancement, though doing it well will require changes to Python. Making numarray thread safe is another possible enhancement.

Acknowledgments

The existence of the original Numeric package was a tremendous help in defining how a re-implementation should be designed. Although little of its code was used, it would be fair to say that the design choices made for how Numeric would appear to the user probably made doing a re-implementation twice as easy as it would have been since many of the interface issues were already solved.

Edward Jones and Magnus Lie Hetland have provided invaluable feedback in noting documentation problems or bugs in the releases. Jochen Kupper has provided much help in translating the documentation into the standard Python documentation framework. We also appreciate all the feedback from the numpy and scipy mailing lists.

We would be greatly remiss if we did not acknowledge the discussions with, ideas from, and work contributed to the original Numeric of the following: Travis Oliphant, Paul Dubois, David Ascher, Tim Peters, Eric Jones, Jim Hugunin, Konrad Hinsen, Guido (you know who) and the past Numeric developers.

References

Alted, F., PyTables: http://sourceforge.net/projects/pytables/

Beazley, D. *Scientific Computing with Python*, Proceedings, Astronomical Data Analysis Software and Systems IX, Kona, Hawaii, October 2000.

Peters, T. Personal communication. May 2001.

White R. L. & Greenfield, P., *Using Python to Modernize Astronomical Software*, Proceedings of the 8th International Python Conference, Arlington, VA, January 2000.