[**Implementing Multidimensional Arrays in JavaScript**](https://0fps.net/2013/05/22/implementing-multidimensional-arrays-in-javascript/)

Posted on [May 22, 2013](https://0fps.net/2013/05/22/implementing-multidimensional-arrays-in-javascript/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

The past few months I’ve been working to move more of my work into JavaScript, and in particular the node.js ecosystem.  I hope that by doing this I will be able to create better demos and applications that are easier to use as well as develop.

Numerical Computing in a Hostile Environment

But despite the numerous practical advantages to working in JavaScript (especially from a user experience perspective) there are still many things that JS just sucks at.  For me the most painful of these is the weak support for numerical computing and binary data types.  These problems are due to fundamental limitations of the language, and not something that can be papered over in a library.  For example, in JS there is:

* No pass-by-value for non-primitive data types
* No way to control memory alignment for members of objects
* No way to [restrict](http://en.wikipedia.org/wiki/Restrict) references to pointers/prevent aliasing
* No way to ensure arrays of objects are allocated in place
* No [operator overloading](http://en.wikipedia.org/wiki/Operator_overloading)
* No [array operations](http://courses.physics.illinois.edu/phys466/comp_info/array.html)
* No [SIMD instructions](http://en.wikipedia.org/wiki/SIMD) and data types
* … etc.

This situation is bad, but the benefits of using JS from an accessibility perspective are just too big to ignore, and so it is at least worth it to try as much as can be done within the limitations of the system.  Moreover, these obstacles are by no means unprecedented in the world of computing.  As an instructive example, consider the [Java programming language](http://en.wikipedia.org/wiki/Java_(programming_language)):  Java suffers from exactly the same design flaws listed above, and yet it is a viable platform for numerical computing with an enthusiastic community (though it is still nowhere near as fast as more efficiently designed languages like Fortran/C99).  So, it is a plausible hypothesis that JavaScript could achieve similar results.

Now, reimplementing say LAPACK or SciPy is a huge project, and it is not reasonable to try doing it in a single article or even within a single library.  To help focus our inquiries, let us first restrict the scope of our search.    Instead, let us study today on the fundamental problem of implementing multidimensional arrays.

Multidimensional arrays

Multidimensional arrays are a natural first step towards building systems for solving numerical problems, and are a generally useful data type in their own right, with applications in graphics and vision.  In a language like JavaScript, there are multiple ways

Arrays-of-Arrays

Perhaps the most popular way to represent higher dimensional arrays in JavaScript is as arrays-of-arrays.  For example,

var A = [[1, 0, 0],

[0, 1, 0],

[0, 0, 1]]

This is the approach that [numeric.js](http://www.numericjs.com/) uses, and it has some merits.  For example, the syntax for accessing elements in an array-of-arrays is quite pleasingly simple:

var x = A[i][j]

And it does not require introducing any extra data structures or conventions into the language.

However, while arrays-of-arrays are by far the simplest solution, from a performance perspective they are disastrous.  The most obvious problem is that to allocate a d-dimensional array-of-arrays with O(n) elements per each element, you need to store O(n^{d-1}) extra bytes worth of memory in pointers and intermediate arrays.  Additionally, accessing an element in an array-of-arrays requires O(d) dependent memory dereferences.  These operations can’t be easily pipelined and are terrible from a cache performance perspective.  If performance is even remotely close to being an objective, then we need to try for something better.

Typed Arrays

Fortunately, in both JavaScript and Java, there is a better alternative to arrays-of-arrays: namely [typed arrays](https://developer.mozilla.org/en-US/docs/JavaScript/Typed_arrays).  Using a typed array lets us guarantee that the data for our array is arranged contiguously with no wasted space.  For example, here is how we would create a typed array for the same matrix described earlier:

var A = new Float32Array([1, 0, 0, 0, 1, 0, 0, 0, 1])

The downside is that accessing elements from a typed array is a bit more complicated.  To do this we need to agree on some convention up front about how the elements in the array are arranged in memory.  In languages like C, typically this is done in colexicographic (also known as [row major order](http://en.wikipedia.org/wiki/Row-major_order)), while other languages like Fortran use a lexicographic ordering (also called column major order).  The choice between the two is  totally arbitrary, but once you pick one you have to stick with it.  Assuming a row major ordering then, we can access an element from the array using the formula:

var x = A[3 \* i + j]

Typed arrays can be very fast, but they are a bit clumsy to work with.  You need to keep track of the number of both what convention you are using to store the array as well as what its dimensions are, which is an extra blob of data that needs to be passed around and slows things down substantially.

Strided Arrays

Strided arrays solve these problems by considering array storage in a more general form. Instead of keeping a fixed convention for indexing, they write the index of an element in array as a general linear function of its coordinate.  For example,

var x = A[ c0 \* i + c1 \* j + ... ]

The result is that they achieve the same performance as flattened typed arrays, but gain the advantage that they can be easily sliced, transposed or reversed just by changing the stride.  For example, if we wanted to switch the x/y axes of an image, all that we would need to do is swap the strides:

var x = A[ c1 \* i + c0 \* j + ... ]

Setting a stride to a negative value can be used to reverse an axis, and changing the shape of an array lets us easily crop out subarrays from higher dimensional arrays.

Introducing **ndarray**

At least theoretically, strided arrays sound like a great idea and so to see how well they worked in practice I created the following library:

<https://github.com/mikolalysenko/ndarray>

Essentially, an ndarray is an object that has 4 pieces of data:

* *data*– The underlying typed array
* *shape*– The shape of the ndarray
* *stride* – The stride of the ndarray
* *offset*– A pointer to the first element in the array

To look up an item in an ndarray you use the formula:

data[stride[0] \* i[0] + stride[1] \* i[1] + ... + offset]

You can download it and try it out yourself using npm, or check out some projects already built on top of [ndarray here](https://npmjs.org/search?q=ndarray).

But is it fast?

To see how each of these multidimensional array representations compare, I decided to compare their performance on a simple component-wise operation for various shapes of arrays.  In pseudocode, this operation looks something like:

function test(A, B) {

for(i in range) {

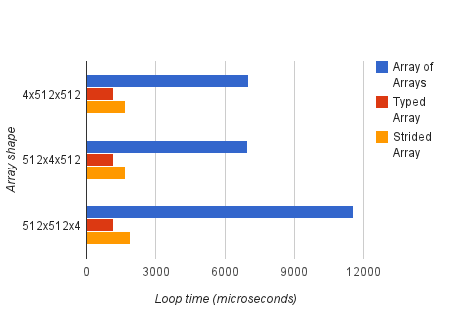
A[i] += B[i] + 0.1

B[i] -= A[i] \* 0.5

}

}

I ran this simple loop using different variations of multidimensional arrays with various shapes.  The following chart summarizes the results:

[](https://0fps.files.wordpress.com/2013/05/benchmark_overview.png)

Summary of array algorithm performance for various implementations.

You can find more detailed results and further experiments (including direct comparisons using numeric.js) at the github page for this project:

<https://github.com/mikolalysenko/ndarray-experiments>

These results are somewhat typical, indexing in an ndarray is a little slower than directly walking over a typed array, but not by much.  For arrays of arrays, the performance gets progressively worse as the first axes get larger, but even in the best case they trail far behind data structures built on typed arrays.

Next time

The natural question to ask is can we do even better?  And the answer is yes.  The key to getting further performance improvements is to batch operations together allowing for full array comprehensions.  I’ll discuss this in the next post, but for now you can take a look at my current implementation of bulk operations on ndarrays in the [cwise library](https://github.com/mikolalysenko/cwise).