Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by JDS on Tue, 28 Feb 2012 00:09:38 GMT
View Forum Message <> Reply to Message

On Thursday, February 23, 2012 3:38:35 PM UTC-5, Gianguido Cianci wrote:
> Hi all,
>
> I would like to write a generic version of the following, which is for a 3d movie:
>
> res=movie
> s=size(res, /dim)
> FOR i = 1, s[2]-1 DO BEGIN
> res[*, *, i] = max(dim = 3, res[*, *, 0:i])
> ENDFOR

Suppose you'd like to create a generic routine (any generic routine, ala HIST_ND, SORT_ND, etc.) which operates along any arbitrary dimension of multi-dimensional arrays. In this case, the *first* thing you'll need to do is let go of IDL's syntactic indexing conveniences ([*,*,i] and the like). These are simply IDL shorthand for creating index arrays, which is very useful, but also very limiting. Instead, you'll want to create your own index arrays, which is vastly more flexible, and not at all difficult once you get the hang of it.

Another useful rule of thumb: modifying arrays in place is somewhat more efficient if, on the left-hand-side, you specify a single index of the array, and have the right hand side simply fill in memory order. I.e.

```
a[off]=big_array
```

is faster than

```
a[off:off+n_elements(big_array)-1]=big_array
```

(and neater looking too). This obviously *only* works when the array you are filling is intended to be dumped in memory order, straight in.

What if your problem isn't so accommodating? What if, for example, you want to operate on an intermediate dimension of an array? The final and quite important trick in producing dimension-agnostic code is to force the input into submission by *rearranging* arrays to place the dimension of interest *last*. This means individuals "units" of comparison (planes of the 3D cube, in the example here) are accessible (and modifiable) *directly in memory order*. TRANSPOSE is the tool for this. This dramatically simplifies things. Also, in my experience, it's almost always faster to transpose the array, work along the now-final dimension of interest doing your possibly rather painful set of operations, and then transpose back, rather than employ an algorithm

that dances hither and yon plucking values from all around the array.

After that realization, it's a straightforward matter of converting between the linear indices of a blob of memory (which is all your fancy IDL arrays really are) and multi-dimensional array indices (which are a useful but arbitrary bookeeping construct maintained by the language). Since you now have your giant array properly ordered, you can simply operate on sequential blobs of data, which may represent some sub-array unit of arbitrary number of dimensions (e.g. two, for a plane).

In the given solution, you are overwriting the array, one plane at a time. This isn't a problem per se, but your version creates unnecessary work. This is because you already know the maximum up to the last index being considered. That is, the cumulative max at index 'i' is nothing more than the max between the prior cumulative max at index 'i-1' and the value(s) at index 'i'. No need to start all over at the beginning.

Here's a generic routine which puts all of these concepts together (and, doesn't even use MAX):

http://tir.astro.utoledo.edu/idl/max_cumulative.pro

Yes, it has a FOR loop, but notice that it only loops over the length of the dimension of interest. At each step of the loop, sub-arrays the size of the product of *all the rest* of the dimensions are operated on, which could represent rather substantial chunks for large multi-dimensional arrays. Thus, in reasonable cases, the looping overhead penalty is unimportant. In fact, I was very surprised to find that, when working along the final dimension of large arrays (of a few hundred million elements), MAX_CUMULATIVE is ~2x faster than its MAX(DIMENSION=) analog, which produces a subset of the information!

JD
