Subject: Cumulative max() in *arbitrary* dimension?
Posted by cgguido on Thu, 23 Feb 2012 20:38:35 GMT
View Forum Message <> Reply to Message

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
```

So how to I generalize this to any dimension? I could make something with execute, but there's gotta be a better way. I'm on IDL7.

Thanks,
Gianguido

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by cgguido on Mon, 27 Feb 2012 17:10:41 GMT
View Forum Message <> Reply to Message

On Monday, February 27, 2012 1:25:39 AM UTC-6, Craig Markwardt wrote:

> I didn't reply because the original poster appeared to be asking for a
> kind of "rolling" maximum, which CMAPPLY doesn't do. (i.e. MAX(res[*,
> *, 0:i]) for each i)
>
> Craig

That's exactly what I was looking for, sorry if my post wasn't clear enough.

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

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by cgguido on Tue, 28 Feb 2012 18:57:20 GMT
View Forum Message <> Reply to Message

On Monday, February 27, 2012 6:09:38 PM UTC-6, JDS wrote:
> 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

Mind. Blown.

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by JDS on Thu, 08 Mar 2012 18:33:39 GMT
View Forum Message <> Reply to Message

On Tuesday, February 28, 2012 1:57:20 PM UTC-5, Gianguido Cianci wrote:
> On Monday, February 27, 2012 6:09:38 PM UTC-6, JDS wrote:

>> 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!
>
> Mind. Blown.

I've since tuned this up a bit more, saving 1/2 of the index computation during each step of the loop by incrementing a running index array. It's now (rather remarkably) >5x faster than MAX(DIMENSION=3) for me with large 3D arrays. And of course it gives all the intermediate cumulative max values.

You can easily show how inefficient MAX is on the final dimension with a simple example:

```
IDL> a=byte(randomu(sd,300,400,3000)*256)
IDL> t=systime(1) & b=max(a,DIMENSION=3) & print,systime(1)-t
IDL> t=systime(1) & b2=a[*,*,0] & for i=1,3000-1 do b2>=a[*,*,i] & print,systime(1)-t
IDL> print,array_equal(b,b2)
```

I can only presume the built-in MAX has some design limitations for final dimension looping. I presume this all works the same way for MIN, BTW.

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by Lajos Foldy on Thu, 08 Mar 2012 21:15:19 GMT
View Forum Message <> Reply to Message

On Mar 8, 7:33 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:

>
> IDL> a=byte(randomu(sd,300,400,3000)*256)
> IDL> t=systime(1) & b=max(a,DIMENSION=3) & print,systime(1)-t
> IDL> t=systime(1) & b2=a[*,*,0] & for i=1,3000-1 do b2>=a[*,*,i] & print,systime(1)-t
> IDL> print,array_equal(b,b2)
>
> I can only presume the built-in MAX has some design limitations for final dimension looping. I presume this all works the same way for MIN, BTW.

b=max(a,DIMENSION=3) is equivalent to

```
b=a[*,*,0]
for j1=0,299 do begin
   for j2=0,399 do begin
      b[j1,j2]=a[j1,j2,0]
      for j3=1,2999 do b[j1,j2]>=a[j1,j2,j3]
   endfor
endfor
```

Your solution is equivalent to

```
b2=a[*,*,0]
for j3=1,2999 do begin
   for j2=0,399 do begin
      for j1=0,299 do b2[j1,j2]>=a[j1,j2,j3]
   endfor
endfor
```

The big difference comes from the memory access pattern.

regards,
Lajos

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by Heinz Stege on Fri, 09 Mar 2012 00:17:29 GMT
View Forum Message <> Reply to Message

On Thu, 8 Mar 2012 10:33:39 -0800 (PST), JDS wrote:

> I've since tuned this up a bit more, saving 1/2 of the index computation
> during each step of the loop by incrementing a running index array. It's
> now (rather remarkably) >5x faster than MAX(DIMENSION=3) for me with
> large 3D arrays. And of course it gives all the intermediate cumulative
> max values.
>
The loop can be tuned up even more. Replacing the array of indices by
two scalars for the subscript range makes the loop faster and also
saves memory. I replaced the following 2 lines of your code

```
  inds=lindgen(off)
  for i=1,s[d]-1 do a[i*off]=a[inds]>a[(inds+=off)]
```

by the following 3 lines:

```
  i1=0
  i2=off-1
  for i=1,s[d]-1 do a[i*off]=a[i1:i2]>a[(i1+=off):(i2+=off)]
```

In my examples max_cumulative is about 2 to 3 times faster than
before:
~2.5 times for a 60x400x3000 byte array
~3.1 times for a 60x400x300 byte array
~2.1 times for a 60x40x3000 byte array

Heinz

Subject: Re: Cumulative max() in *arbitrary* dimension?

Posted by JDS on Fri, 09 Mar 2012 15:50:24 GMT
View Forum Message <> Reply to Message

On Thursday, March 8, 2012 4:15:19 PM UTC-5, Lajos Foldy wrote:
> On Mar 8, 7:33 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:
>
>>
>> IDL> a=byte(randomu(sd,300,400,3000)*256)
>> IDL> t=systime(1) & b=max(a,DIMENSION=3) & print,systime(1)-t
>> IDL> t=systime(1) & b2=a[*,*,0] & for i=1,3000-1 do b2>=a[*,*,i] & print,systime(1)-t
>> IDL> print,array_equal(b,b2)
>>
>> I can only presume the built-in MAX has some design limitations for final dimension looping.
I presume this all works the same way for MIN, BTW.
>
> b=max(a,DIMENSION=3) is equivalent to
>
> b=a[*,*,0]
> for j1=0,299 do begin
> for j2=0,399 do begin
> b[j1,j2]=a[j1,j2,0]
> for j3=1,2999 do b[j1,j2]>=a[j1,j2,j3]
> endfor
> endfor
>
> Your solution is equivalent to
>
> b2=a[*,*,0]
> for j3=1,2999 do begin
> for j2=0,399 do begin
> for j1=0,299 do b2[j1,j2]>=a[j1,j2,j3]
> endfor
> endfor
>
> The big difference comes from the memory access pattern.

Seems sensible. I guess I'm surprised that MAX doesn't order the nested loops sensibly, given that it must of course have a large different loop sets for each of the possible total dimensions/target dimension combinations.

JD

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by JDS on Fri, 09 Mar 2012 16:58:44 GMT
View Forum Message <> Reply to Message

On Thursday, March 8, 2012 7:17:29 PM UTC-5, Heinz Stege wrote:

> On Thu, 8 Mar 2012 10:33:39 -0800 (PST), JDS wrote:
>
>> I've since tuned this up a bit more, saving 1/2 of the index computation
>> during each step of the loop by incrementing a running index array. It's
>> now (rather remarkably) >5x faster than MAX(DIMENSION=3) for me with
>> large 3D arrays. And of course it gives all the intermediate cumulative
>> max values.
>>
> The loop can be tuned up even more. Replacing the array of indices by
> two scalars for the subscript range makes the loop faster and also
> saves memory. I replaced the following 2 lines of your code
> inds=lindgen(off)
> for i=1,s[d]-1 do a[i*off]=a[inds]>a[(inds+=off)]
> by the following 3 lines:
> i1=0
> i2=off-1
> for i=1,s[d]-1 do a[i*off]=a[i1:i2]>a[(i1+=off):(i2+=off)]
>
> In my examples max_cumulative is about 2 to 3 times faster than
> before:
> ~2.5 times for a 60x400x3000 byte array
> ~3.1 times for a 60x400x300 byte array
> ~2.1 times for a 60x40x3000 byte array
>
> Heinz

Hey, Heinz... very cool. This flies in the face of the general notion that "having IDL compute index arrays in loops is wasteful." This is likely because you are required to update the index set during each iteration, so you may as well let IDL do this internally. In other typical cases, you are asking IDL to repeat the calculation of an identical index loop that you can simply pre-cache for a large savings.

I replaced yours instead with the rather similar:

```
for i=1,s[d]-1 do a[i*off]=a[(i-1)*off:i*off-1]>a[i*off:(i+1)*off-1]
```

and got about 2.5x speedup for the final dimension. For non-final dimensions, the speedup is much less, since TRANSPOSE imposes a reasonably large overhead. Since this challenged my first "rule of thumb", I decided to check the next one: that TRANSPOSE and then in-order operation saves time over indexing out of memory order. That one holds for non-final dimensions, by at least a factor of 2.

BTW, it's now *15x* faster than MAX(DIMENSION=3), for the reasons Lajos mentions.
Thanks for your thoughts.

JD

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by Heinz Stege on Fri, 09 Mar 2012 17:58:13 GMT
View Forum Message <> Reply to Message

Hi JD.

On Fri, 9 Mar 2012 08:58:44 -0800 (PST), JDS wrote:

> I replaced yours instead with the rather similar:
>
> for i=1,s[d]-1 do a[i*off]=a[(i-1)*off:i*off-1]>a[i*off:(i+1)*off-1]
>
Good point. I didn't take into account to use the loop variable within the loop. This gives me a new idea:

```
  for i=off,ns-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
```

I didn't test it. However it is shorter and may be easier to read. I don't really think, that it is faster, but it may be. (5 instead of 9 scalar operations within the array subscripts is not the world.) :-)

Heinz

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by Heinz Stege on Sat, 10 Mar 2012 17:21:48 GMT
View Forum Message <> Reply to Message

First: On Fri, 09 Mar 2012 18:58:13 +0100, I wrote:

```
>    for i=off,ns-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
```

There is a mistake. It has to read:

```
  for i=off,n_elements(a)-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
```

Second: My contribution => my work. ;-) I measured the times for an array a=byte(randomu(seed,60,400,3000),/long). I tested the following versions:

```
[1] i1=0 &i2=off-1 & $
    for i=1,s[d]-1 do a[i*off]=a[i1:i2]>a[(i1+=off):(i2+=off)]
[2] for i=1,s[d]-1 do $
    a[i*off]=a[(i-1)*off:i*off-1]>a[i*off:(i+1)*off-1]
[3] for i=off,n_elements(a)-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
```

I ran each version 2000 times and found no significant differences in

the run-time. There only seems to be a slight trend for [2] beeing the slowest. However it does not have any practical relevance. Here are the details:

[1] 134.3 (+/-0.2) ms (calculation of i1 and i2 included)
[2] 134.9 (+/-0.2) ms
[3] 134.2 (+/-0.2) ms

The given errors are statistical standard-errors ("1 sigma").

Heinz

Subject: Re: Cumulative max() in *arbitrary* dimension?
Posted by JDS on Fri, 16 Mar 2012 21:26:44 GMT
View Forum Message <> Reply to Message

On Saturday, March 10, 2012 12:21:48 PM UTC-5, Heinz Stege wrote:
> First: On Fri, 09 Mar 2012 18:58:13 +0100, I wrote:
>
>> for i=off,ns-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
>
> There is a mistake. It has to read:
>
> for i=off,n_elements(a)-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
>
>
> Second: My contribution => my work. ;-) I measured the times for an
> array a=byte(randomu(seed,60,400,3000),/long). I tested the following
> versions:
>
> [1] i1=0 &i2=off-1 & $
> for i=1,s[d]-1 do a[i*off]=a[i1:i2]>a[(i1+=off):(i2+=off)]
> [2] for i=1,s[d]-1 do $
> a[i*off]=a[(i-1)*off:i*off-1]>a[i*off:(i+1)*off-1]
> [3] for i=off,n_elements(a)-off,off do a[i]=a[i-off:i-1]>a[i:i+off-1]
>
> I ran each version 2000 times and found no significant differences in
> the run-time. There only seems to be a slight trend for [2] beeing the
> slowest. However it does not have any practical relevance. Here are
> the details:
>
> [1] 134.3 (+/-0.2) ms (calculation of i1 and i2 included)
> [2] 134.9 (+/-0.2) ms
> [3] 134.2 (+/-0.2) ms
>
> The given errors are statistical standard-errors ("1 sigma").

Thanks, Heinz. When compared to processing 10^5 or 10^6 numbers within each loop iteration (which is what makes this a fast loop), the indexing math isn't really significant. Yours is a bit easier on the eye though!

JD
