Subject: Re: efficient comparing 1D and 3D arrays
Posted by Chris[5] on Wed, 11 Jun 2008 21:24:04 GMT
View Forum Message <> Reply to Message

> I am hoping there is a fell
> swoop with which I can do this in one step; a way to create a [ns, nl]
> bytearr with 0/1 to match the condition that each band for that bixel
> falls within the desired range...

Let me try to understand what you are trying to do:
-data is a datacube of dimensions (nb,ns,nl).
-for each pixel along the first dimension (the one with nb elements), you want to test whether it is greater than minval and less than maxval. These are functions of where you are along the first dimension, so minval and maxval are vectors of length nb.
-you want to create an array, of size ns x nl, such that result[x,y]=1 if data[i,x,y] falls between minval[i] and maxval[i] for all i.

Is this a correct summary? If so, I would recommend:

```
pro test

nb=3
ns=2
nl=2

data=randomu(seed,nb,ns,nl);just make up data
minval=fltarr(nb)+.1
maxval=fltarr(nb)+.9

;make cubes out of these
minval=rebin(minval,nb,ns,nl)
maxval=rebin(maxval,nb,ns,nl)

print,cube

hit=(data gt minval) and (data lt maxval)
result=total(hit,1) eq nb

print,result

end
```

Explanation:

You turn minval and maxval into cubes such that every pixel in the

data cube needs to be between the corresponding pixels in minval and maxval. You then do a pixel by pixel test of this, returning the 'hit' cube (ones and zeros). You only care about cuts through the first dimension which satisfy your bounds for every pixel along that dimension. Thus, you sum up the cube along the first dimension (returning an ns by nl array), and test whether or not it equals nb (meaning every pixel was a hit).

Aside:
I was doing something similar this weekend, and agree with the earlier poster about weighing the pros and cons of looping vs large array creation. If you have lots of pixels, allocating the memory for the minval and maxval cubes will take some time. Looping through every pixel in the cube is, of course, a dumb thing to do in idl - it would rather work with arrays than individual elements. However, if you loop through each PLANE (say, step along the fist dimension), and then at each step in the loop analyze a 2D array, you will balance IDL's efficiency at working with arrays with the time penalty associated with allocating big chunks of your system memory. I have found (to my surprise) that this kind of looping can be much, much faster than avoiding the loop altogether. I am trying to quantitatively understand this behavior more (there are lots of qualitative descriptions of this here and on David Fanning's website), but the moral seems to be the following: IDL's 'sweet spot' is to do operations on arrays as large as possible, AS LONG AS any memory allocation you need to do to allow such a procedure is small (say a few percent of the total memory you have available).

-chris