Subject: Re: Using where() on slices of data cubes
Posted by JDS on Tue, 20 Oct 2009 20:03:30 GMT
View Forum Message <> Reply to Message

On Oct 20, 8:22 am, Conor <cmanc...@gmail.com> wrote:
> I feel like this should be an easy one, but I've never quite figured
> it out. Let's say I got a data cube and I want to do something on
> just a slice of it, say I want to turn certain values in a column into
> something else:
>
> w = where(cube[1,*,*] lt 0)
>
> It seems like you should be able to do something like this:
>
> cube[1,w] = 1e24
>
> But that doesn't work... Somehow I can't quite figure out the right
> way to do this.

It's impossible for IDL to know the dimensionality of the original array which from which you extracted a given set of elements. WHERE simply gives you the zero-based running index into a given pile of data, independent of its form and (much less) the form of the parent array from which it was derived. So it's not at all surprising that the returned set of indices doesn't "plug right in" in some convenient way. The (likely) quickest solution in this case is the REFORM,/OVERWRITE method given by Greg, since it doesn't actually copy any data. But unfortunately, it's not at all general. For example, suppose you had been interested in longitudinal slices instead, ala:

```
w=where(cube[*,1,*] lt 0)
```

A solution similar to Greg's would require first TRANSPOSE'ing the cube such that the elements of the slice of interest could be put in order along some dimension, e.g.:

```
w=where(cube[*,1,*] lt 0)
sz=size(cube,/dim)
cube=reform(transpose(cube,[0,2,1]),[sz[0]*sz[2],sz[1]])
cube[w,1]=1e24
 cube=transpose(reform(cube,sz[0],sz[2],sz[1],/overwrite),[0, 2,1])
```

This obviously creates two transposed copies of the cube, which is memory and CPU inefficient, not to mention difficult to remember. For operating on small cubes, or a large fraction of the cube elements, it might be reasonable. If, however, you have only a few elements to access in a very large array (say just a sprinkling of negatives, in your example), this would be very wasteful indeed.

In my opinion, a far better general solution to these sorts of problems is to master the ability to recreate *yourself* any of the index computations IDL does for you (and, by extension, any that it doesn't). For example, when you write

```
slice = cube[1,*,*]
```

IDL doesn't just conjure the appropriate elements out of thin air. It examines the dimensions of 'cube', and implicitly constructs a one-dimensional index vector for all of the indices being referenced. This happens in the background, but you can easily verify it by seeing how much memory IDL has used for this temporary index vector. As a side note, this can occasionally be memory inefficient, which is why it's sometimes *preferable* to construct your own indices, even when IDL could have done it for you (see http://www.dfanning.com/misc_tips/submemory.html).

As for the posed problem, let's skip to the end. In your example, the answer looks like:

```
sz=size(cube,/DIMEN)
cube[1+sz[0]*(w mod sz[1] + w/sz[1]*sz[1])] = 1e24
```

Where does this come from? Your original cube "slice" has dimensions [sz[1],sz[2]], and is at an x position of 1. Recall that

```
slice_col = w mod sz[1]
```

gives the column inside this slice, and

```
slice_row = w/sz[1]
```

gives the row. Nothing fancy there. If you forget these, you can easily use the IDL provided convenience function ARRAY_INDICES instead:

```
slice_col_row = array_indices(sz[1:2],w,/DIMENSIONS)
```

However, this is just doing the same pair of computations, which are not that hard to remember.

So far so good. We need to create a one dimensional index vector appropriate for the cube's dimensionality, ala (very generally):

```
ind = x + sx * (y + sy * z)
```

Now comes the (slightly) tricky bit. We need to do this *for the selected slice elements*. But wait! The slice's column (x direction)

is the full cube's row (aka y direction), and the slice's row is the full cube's plane (aka z direction). So what we need looks like:

```
ind = x + sx * (slice_col + sy * slice_row)
```

aka

```
ind = 1 + sz[0] * (w mod sz[1] + w/sz[1]*sz[1])
```

One other wrinkle. Notice I didn't write sz[1] * w/sz[1]. This is because operator precedence would compute this as (sz[1]*w)/sz[1], which would give you.... w -- not what you want. You can either write sz[1]*(w/sz[1]), or just rearrange terms as I have done.

OK, you may be asking yourself, what has this really gained me? A lot, actually. Once you become fluent in converting back and forth between one-dimensional indices and [x,y,z,...] index sets in arbitrary dimensions, you are able to manipulate with ease the full range of indexing problems, and are, most importantly, no longer reliant on IDL's convenient but limited higher-order indexing operators. For example, suppose I had originally issued the call:

```
w=where(cube[1,5:*,10:1024] lt 0)
```

you should easily be able to generalize the above arguments to access these elements.
