Subject: VALUE_LOCATE tutorial
Posted by Jeremy Bailin on Wed, 24 Jun 2009 05:21:06 GMT
View Forum Message <> Reply to Message

I've taken David's encouragement to write up this tutorial on all
the fun uses of the insufficiently-appreciated VALUE_LOCATE
function. ;-)

- Introduction -

There are probably two reasons why VALUE_LOCATE is underused. The
first
is that it was only introduced in IDL 5.3, well after many people
developed their core techniques. The second is that the help page is
somewhat opaque on what it actually does. The basic idea is pretty
simple:
given two arrays Values and Array,

Result = VALUE_LOCATE(Values, Array)

tells you where within Values the elements of Array are located. A
concrete example will help:

```
IDL> Values = [-10, -5, 23, 109]
IDL> Array = [-5, -5, 23, 23, -10, 109]
IDL> print, VALUE_LOCATE(Values, Array)
           1           1           2           2           0
3
IDL> print, Values[VALUE_LOCATE(Values, Array)]
      -5      -5      23      23     -10     109
```

In other words, -5 is element number 1 of Values, 23 is element number
2,
-10 is element number 0, and 109 is element number 3. They are the
subscripts
of the located elements within Values - and indeed, if we subscript
Values
by those indices, we end up with the original Array.

One important caveat is that Values must be strictly increasing. You
will
get nonsense answers otherwise:

```
IDL> Values = [-10, 23, 109, -5]
IDL> Array = [-5, -5, 23, 23, -10, 109]
IDL> print, VALUE_LOCATE(Values, Array)
           0           0           1           1           0
3
```

```
IDL> print, Values[VALUE_LOCATE(Values, Array)]
    -10     -10      23      23     -10      -5
```

(technically, Values can also be monotonically decreasing, but the return
value doesn't have exactly the same meaning - for a beginner, I would recommend sticking with monotonically increasing case)

- Mapping Between Sets -

I use VALUE_LOCATE in this vein all the time as a way of creating a mapping between the set of integers and any other finite set of numbers.
Let's say I have an array whose elements can only take on the following 5
values:
-23.5, 19.4, 2.0, -9999, 14.1.
For a great many purposes (such as the HISTOGRAM examples that I'll get into
below) integers between 0 and 4 are a much nicer set of numbers to deal
with than this smorgasbord of floating point numbers. We can map back and forth between these two representations quite easily using VALUE_LOCATE:

```
IDL> Array = [2.0, 2.0, 19.4, -9999, 14.1, -9999, 19.4, -9999, 2.0]
IDL> Values = [-23.5, 19.4, 2.0, -9999, 14.1]
IDL> Values = Values[SORT(Values)]
IDL> MappedArray = VALUE_LOCATE(Values, Array)
IDL> print, MappedArray
         2         2         4         0         3
0
         4         0         2
```

Here we have taken some floating point data and converted it into a much
simpler set of integers (note that we had to sort Values first!). If we
have an array of integers, the reverse operations is equally simple:

```
IDL> MappedResult = [1, 1, 3, 4, 0, 0]
IDL> print, Values[MappedResult]
    -23.5000     -23.5000      14.1000      19.4000     -9999.00
-9999.00
```

This is in some ways similar to the enumeration type that is available in C and some other languages.

We can also map between two different non-integer enumerations by sticking
a forward mapping from one onto the reverse mapping of the other:

```
IDL> Values1 = [-9999, -23.5, 2.0, 14.1, 19.4]
IDL> Values2 = [100,   100.5, 101, 101.5, 102]
IDL> Array = [14.1, -23.5, -9999, 2.0]
IDL> print, Values2[VALUE_LOCATE(Values1, Array)]
      101.500      100.500      100.000      101.000
```

Here -9999 gets mapped to 100, -23.5 gets mapped to 100.5, etc.

- Ranges -

In every example so far, each element of Array occurs exactly within Values. But what if some of the elements don't appear there? Let's try it out!

```
IDL> Values = [0,10,20,30]
IDL> Array = [-5,5,15,25,35]
IDL> print, VALUE_LOCATE(Values, Array)
         -1           0           1           2           3

Values:     0      10      20      30
Index:      0       1       2       3
Array:  -5     5       15      25       35
Return: -1     0        1       2        3
```

We see that if an element of Array lies between two elements of Values,
VALUE_LOCATE rounds down to the lower index. For example, 15 lies between 10 (index 1) and 20 (index 2), so VALUE_LOCATE returns 1. This rounding down even occurs when values of Array lie outside of the range of Values: in the case above, -5 is less than Values[0], so VALUE_LOCATE rounds down to -1; similarly, 35 rounds down to the highest value, Values[3], and so VALUE_LOCATE returns 3.

- Using Ranges For Partitioning -

There are many applications for using VALUE_LOCATE in this manner. One example is partitioning floating point data into unevenly-spaced bins for display purposes. To repeat the example from http://www.dfanning.com/code_tips/partition.html , say you have a 2D array of values ranging from 0 to 1, and want to display it as an image with a small number of colours depending on the value:

< 0.2: white
0.2 - 0.3: green
0.3 - 0.5: yellow
0.5 - 0.8: blue
> 0.8: red

The first thing to do is to set up a colour table that loads white into
colour index 1, green into 2, etc. But how do we then turn our floating
point values into colour indices? With VALUE_LOCATE, it's simple:

```
IDL> Cutoffs = [0.2, 0.3, 0.5, 0.8]
IDL> Image = BYTE(VALUE_LOCATE(Cutoffs, Array) + 2)
```

What have we done here? We have asked where each floating point number in Array would fall in Cutoffs. All of the ones that lie below 0.2 will return -1, all of the ones that lie between 0.2 and 0.3 will return 0, all of the ones that lie between 0.3 and 0.5 will return
1, etc. We just need to add 2 to get to the colour index for each range, and convert to byte for displaying.

- A Serving of VALUE_LOCATE With A Side Of HISTOGRAM -

HISTOGRAM has a well-deserved reputation as the foundation of most IDL optimization strategies because of its combination of speed and the wonderful REVERSE_INDICES facility. However, some problems appear
difficult to solve using HISTOGRAM because it can only use fixed bin sizes.
As I'll demonstrate below, VALUE_LOCATE can be coupled with HISTOGRAM to
make it even more powerful (yikes!).

A common question to answer with HISTOGRAM is "which elements lie in each bin?" This is straightforward if we have equally spaced bins, but what if we want our bin edges to be spaced non-uniformly?

The trick is to get VALUE_LOCATE to partition the data into integers, and then run HISTOGRAM on the uniformly-spaced integers that result. For example:

```
IDL> Cutoffs = [0.2, 0.3, 0.5, 0.8]
IDL> Data = RANDOMU(43L, 10)
IDL> print, Data
```

```
    0.331022     0.151196     0.114072     0.203458    0.0409741
0.614608
    0.951897     0.191795    0.0152987     0.709563
IDL> MappedData = VALUE_LOCATE(Cutoffs, Data)
IDL> print, MappedData
         1        -1        -1         0        -1
2
         3        -1        -1         2
IDL> h = HISTOGRAM(MappedData, MIN=-1, REVERSE_INDICES=ri)
IDL> PRINT, h
         5         1         1         2         1
IDL> PRINT, Data[ri[ri[0]:ri[1]-1]]
    0.151196     0.114072    0.0409741     0.191795    0.0152987
(values less than 0.2)
IDL> PRINT, Data[ri[ri[1]:ri[2]-1]]
    0.203458
(values between 0.2 and 0.3)
IDL> PRINT, Data[ri[ri[2]:ri[3]-1]]
    0.331022
(values between 0.3 and 0.5)
IDL> PRINT, Data[ri[ri[3]:ri[4]-1]]
    0.614608     0.709563
(values between 0.5 and 0.8)
IDL> PRINT, Data[ri[ri[4]:ri[5]-1]]
    0.951897
(values greater than 0.8)
```

My favourite example of coupling VALUE_LOCATE and HISTOGRAM is in the case of sparse data. For example, let's say we want to know which values
are duplicated in the following data:

```
Data = [5, 1000000000000ULL, 1000000000000ULL, 6]
```

The obvious answer is HISTOGRAM:

```
h = histogram(data, omin=mindata)
print, where(h gt 1)+mindata
```

...but this will fail miserably because the required histogram has almost one trillion elements and would require almost 4TB of memory! That's ridiculous overkill given that there are only 3 distinct data values.

The solution is to use VALUE_LOCATE to map those values onto the set of integers
from 0 to 2, and the run histogram on those mapped values. First we

need
to get a list of all the possible values that Data can take on:

```
IDL> sorteddata = data[sort(data)]
IDL> dataenum = sorteddata[uniq(sorteddata)]
IDL> print, dataenum
                5                   6        1000000000000
```

Now we use dataenum to map the original data to the set of integers:

```
IDL> mappeddata = value_locate(dataenum, data)
IDL> print, mappeddata
          0           2           2           1
```

Then we run histogram:

```
IDL> h = histogram(mappeddata, min=0)
IDL> print, h
          1           1           2
```

and figure out which elements have more than one drop in a histogram
bucket:

```
IDL> print, dataenum[where(h gt 1)]
        1000000000000
```

This technique can be used to compress any sparse data set into a
range that
histogram can run on. Any algorithmic tricks that are based on
REVERSE_INDICES (and there are a great many!) can now be extended to
work
on sparse data sets.

-Jeremy.
