Subject: Re: Is there a standard 'null' array?
Posted by JD Smith on Fri, 28 Mar 2003 16:32:09 GMT
View Forum Message <> Reply to Message

On Fri, 28 Mar 2003 06:47:01 -0700, Matt Feinstein wrote:

> This is an embarassingly elementary question, and I'm -sure- that it has
> a simple, elegant, and obvious answer, but...
>
> I often find myself wanting to do things kinda like
>
> Data_I_want_vec = Init
> ...
> for ix = 0 , (size(data_array))[1]-1 do $
> if condition(data_array[ix]) then $
> Data_I_want_vec = [Data_I_want_vec, data_array[ix]]
>
> I realize that there is analogous code for a vectorized version of my
> question, but I think the for-loop version is clearer.
>
> The questions that arise are things like: What to use for 'Init' ? What
> do I do to test for a null result? What if there's no data?
>
> Some notes:
> a) Matlab has an empty vector = [] that serves these purposes. It's
> possible that my desire to program in this fashion is simply a
> Matlab-ism, and I need instruction in the IDL way. b) I do not, really
> and truly, need instruction in the use of the WHERE function or in the
> use of vectorization. Believe me, I know all about that. The analog of
> my question in the vectorized case is that it looks like I have to
> surround the vectorized expression with boring and error-prone tests for
> null inputs and outputs.
>
> Is there a 'standard' way of doing this sort of thing?

Yes. By vectorizing it ;). Despite your awareness of vector techniques, you might be unaware of the following: in the above code you've combined two of the absolute least efficient operations in all of IDL: long loops (assuming your data is substantial), and array concatenation. Loops are slow because each trip around a for loop takes a spin through the full IDL interpreter loop, which is necessarily pokey, thanks to all the wonderful conveniences it provides. With concatenation, every time you append an element, you're allocating space for a new larger array, copying the entire original array into it, and then copying the appended element(s). With time and experience, you'll find that vectorized versions are more terse and quickly understandable. It's not the cleanest syntax, but it does become second-nature in time.

Of course, there are plenty of times where "just vectorize it" is not an option, and you'd really like to build dynamic-sized arrays efficiently. You have (at least) two choices:

1. Just eat the large overheads and test/concatenate on each round.
   This is the standard recipe:

```
  for i=0,cowscomehome do begin
    if some_test then do $
 if n_elements(vec) eq 0 then vec=[complicated_function(i)] else $
 vec=[vec,complicated_function(i)]
  endfor
```

2. Be smart about array concatenation. Guess how big your array will
   be, pre-allocate, and fill it in. If you run out of room, grow the
   array with concatenation:

```
  nacc=1000                  ; or whatever
  vec=fltarr(nacc,/NOZERO)
  pos=0
  for i=0,cowscomehome do begin
    if NOT some_test(i) then continue
    new_vec=complicated_function(i)
    nnv=n_elements(new_vec)
    if pos+nnv ge nacc then begin  ; grow as necessary
      vec=[vec,fltarr(nacc,/NOZERO)]
      nacc=2*nacc
    endif
    vec[pos]=new_vec
    pos=pos+nnv
  endfor
  vec=vec[0:pos-1]           ; trim the empty bits
```

Here I double the array size whenever new space is needed, and trim to size afterwards. Typically this will result in far fewer concatenations, but the details depend on your initial guess and array growing prescription compared to the problem at hand. Sometimes it's not so easy to guess.

I for one would love to have true efficient array concatenation in IDL, ala:

```
  push,vec,append_vec
  unshift,vec,prepend_vec
```

These functions from Perl are efficient because Perl does all the ugly work typified in #2 above for you. Each PUSH does not necessarily copy the entire array. Instead, arrays are pre-allocated to be larger than

you need, and grown in intelligently-sized chunks, shifted, unshifted, spliced, etc., but only re-allocated and copied if it's absolutely necessary. This would come with a tradeoff though: arrays would need to be smarter, and hence, in more pedestrian usage (e.g. a=b+c) would be slower. Perhaps two kinds of interoperable arrays would be in order: the fancy kind that grow/shrink/prepend/append/pre-allocate automagically, and those that are built purely for speed, and never take up more space than the data they contain.

By the way, if you don't like to type all the repetitive "if n_elements()" bit, you can use a routine to do it for you (albeit even slightly more inefficiently thanks to the routine-call overhead):

```
pro push, array, append
  if n_elements(array) eq 0 then array=[append] else array=[array,append]
end
```

Good luck,

JD

P.S. There was talk of adding a zero-length-array type a few years ago, but the consensus was that it's probably too late to fit it easily into a 25 year old program.
