Subject: Re: algorithm question. Can I get rid of the for loop?
Posted by Jeremy Bailin on Wed, 27 Mar 2013 18:55:43 GMT
View Forum Message <> Reply to Message

Two more notes:

 - I just realized that what I wrote about the speed is ambiguous. My
3-histogram version is about 3x faster than your loop version.

 - In this line:
longarr = lindgen(nwindow_runningtot[ny-1])
  I had defined ny=n_elements(y) earlier in my test code, so you can
just plug that in instead.

-Jeremy.

On 3/27/13 12:53 PM, Jeremy Bailin wrote:
> On 3/27/13 9:31 AM, Sï¿½ren Frimann wrote:
>> Den fredag den 22. marts 2013 14.25.44 UTC+1 skrev Heinz Stege:
>>>
>>> Before entering the loop get the indices of the lower and upper limits
>>>
>>> for all x values by use of the VALUE_LOCATE function. Then you can use
>>>
>>> this pre-calculated indices instead of the index array from the WHERE
>>>
>>> function.
>>
>> That was a very nice hint indeed! Below an implementation where this
>> has been done (as well as a few general updates). It's much faster
>> than my older solution although, it retains the for loop
>>
>> ##
>>
>> FUNCTION hampel, x, y, dx, THRESHOLD=threshold
>>
>> Compile_Opt idl2
>>
>> IF N_Elements(threshold) EQ 0 THEN threshold = 3
>>
>> s0 = FltArr(N_Elements(y))
>> y0 = FltArr(N_Elements(y))
>> yy = y
>>
>> lower_Boundary = Value_Locate(x,x-dx)+1 ; indices of lower boundaries
>> upper_Boundary = Value_Locate(x,x+dx)   ; indices of upper boundaries
>>

```
>> FOR i=0,N_Elements(y)-1 DO BEGIN
>>   IF lower_Boundary[i] EQ upper_Boundary[i] THEN BEGIN
>>     ; only one point in gap
>>     y0[i] = y[i]
>>     s0[i] = !Value.F_NAN
>>   ENDIF ELSE BEGIN
>>     ; Two or more points in gap
>>     y_temp = y[lower_Boundary[i]:upper_Boundary[i]]
>>     y0[i] = Median(y_temp) ; median filtering
>>     s0[i] = 1.4826*Median(Abs(y_temp - y0[i])) ; estimating uncertainty
>>   ENDELSE
>> ENDFOR
>>
>> gp = Where(Abs(y - y0) LT threshold*s0)  ;index of good points
>> ol = Where(Abs(y - y0) GE threshold*s0,n)  ;index of outliers
>>
>> yy[ol] = y0[ol] ; replace outliers
>>
>> result = Create_Struct('y'    ,yy, $
>>                        'sigma',s0, $
>>                        'gp'   ,gp, $
>>                        'ol'   ,ol, $
>>                        'n'    ,n) ; number of outliers
>>
>> RETURN, result
>>
>> END
>>
>> ##############################################################
>>
```

>> Den fredag den 22. marts 2013 18.29.31 UTC+1 skrev bobgst...@gmail.com:
>>> Using histogram and reverse indices would be much faster than looping
>>> and whereing. (i.e. get all of your "index" arrays in one call,
>>> rather than n_elements(y) calls of where).
>>>
>>
>> I've looked into it, and I really don't see a way of using histogram,
>> since it involves binning of the data, and my data aren't binned -
>> rather they are subject to a moving window running smoothly over the
>> data set.
>>
>> Cheers,
>> Sï¿½ren
>>
>
> Anything can be turned into a solution using histogram. ;-) You just
> need to create an array that *can* be binned. Which, of course, probably
> also involves histogram! It may be much less readable, and involve 3

> histograms including a histogram-of-a-histogram, but here's my
> replacement for your loop, which is about 3x faster in my tests:
>
>
>
> nwindow_per_element = upper_boundary - lower_boundary + 1
> nwindow_runningtot = total(nwindow_per_element, /cumulative, /int)
>
> ; we're going to treat all possible points in the window around
> ; each element as one gigantic long array:
> longarr = lindgen(nwindow_runningtot[ny-1])
>
> ; So we need easy ways
> ; of figuring out, for each point i in that long array, which
> ; element it is in the window of.
> ; The number of points in the window of each element is given
> ; by nwindow_per_element, so we can use that as the number
> ; of times to repeat each integer.
> ; Use the histogram trick to do the repeats - see the histogram tutorial
> reph = histogram(nwindow_runningtot-1, bin=1, min=0,$
>   reverse_indices=repri)
> elementnum = repri[0:n_elements(reph)-1] - repri[0]
>
> ; We also need to know, for each point in the long array, what
> ; element in y it refers to. First figure out which point within
> ; the window that is:
> windownum = longarr - longarr[([0,nwindow_runningtot])[elementnum]]
> ; Then combine them with lower_boundary to figure out where in y that is
> ynum = lower_boundary[elementnum] + windownum
>
> ; now histogram the elementnums so each window falls into
> ; a separate bin
> winh = histogram(elementnum, min=0, reverse_indices=winri)
>
> ; and use the double-histogram trick so we only need to loop through
> ; repeat counts instead of through elements - see the drizzling/chunking
> ; page
> h2 = histogram(winh, reverse_indices=ri2, min=1)
> if h2[0] gt 1 then begin
>   ; single-element windows
>   vec_inds = ri2[ri2[0]:ri2[1]-1]
>   y0_jb[vec_inds] = y[ynum[winri[winri[vec_inds]]]]
>   s0_jb[vec_inds] = !value.f_nan
> endif
> for j=1,n_elements(h2)-1 do if h2[j] gt 0 then begin
>   ; windows of width j+1
>   element_inds = ri2[ri2[j]:ri2[j+1]-1]
>   vec_inds = rebin(winri[element_inds], h2[j], j+1, /sample) + $

```
>       rebin(transpose(lindgen(j+1)), h2[j], j+1, /sample)
>     y_temp = y[ynum[winri[vec_inds]]]
>     ; first dimension is which element, second is where in the window
>     ; so to do operations over the window, work on the second dimension
>     y0_jb[element_inds] = median(y_temp, dimension=2)
>     y0_temp = rebin(y0_jb[element_inds], h2[j], j+1, /sample)
>     s0_jb[element_inds] = 1.4826*median(abs(y_temp - y0_temp), dimension=2)
>  endif
>
>
>  -Jeremy.
>
```