Subject: Another "IDL way to do this" question
Posted by Fabzou on Tue, 08 Nov 2011 14:50:23 GMT
View Forum Message <> Reply to Message

Dear IDLers,

I have to interpolate modelled fields from eta to pressure coordinates, which is a relatively easy thing to do when using a simple linear approach. Since my 3D grid is irregular "only" on the third dimension (the fourth dimension being time), well I did not find any other solution than looping on the three regular dimensions and use INTERPOL for each vertical column.(interpol_3d in the code posted below)

This works fine, but is not very very fast.

Then I tried to loop on the time dimension outside the "main loop" (interpol_3d_alt in the code posted below), but results are only slightly better, as shown if I run the test program compare_interp:

% Compiled module: INTERPOL_3D.
% Compiled module: INTERPOL_3D_ALT.
% Compiled module: COMPARE_INTERP.
IDL> compare_interp
Method1: 19.713714
Method2: 18.455919

Do you have an idea how to make this faster? I thought about other types of 3D interpolation, but they all seem so "overkill" for such a simple problem...

Thanks a lot

Fab

```
;+
; :Description:
;    Interpolates model data vertically (height or pressure levels).
;
; :Params:
;    varin: in, float, required
;          Data on model levels that will be
;          interpolated[nx,ny,nz,(nt)]
;    z_in: in, float, required
;         Array of vertical coordinate to interpolate into.
;         This must either be pressure/height.
;         Dimensions must be the same as those of `varin`.
;    loc_param: in, float, required
```

```
;              the locatations to interpolate to
;               (typically pressure or height)
;
; :Returns:
;     Data interpolated to horizontal plane(s), with the third dimension
;     being equal to the number of elements in `loc_param`
;
;-
function interpol_3d, varin, z_in, loc_param

   ; Set Up environnement
   COMPILE_OPT idl2

   dims = SIZE(varin, /DIMENSIONS)
   nd = N_ELEMENTS(dims)
   nlocs = N_ELEMENTS(loc_param)

   if nd eq 3 then begin ; no time dimension
     out_var = FLTARR(dims[0], dims[1], nlocs)
     for i=0, dims[0]-1 do begin
       for j=0, dims[1]-1 do begin
         _z = z_in[i,j,*]
         out_var[i,j,*] = INTERPOL(varin[i,j,*],_z,loc_param)
         p = where(loc_param gt max(_z) or loc_param lt min(_z), cnt)
         if cnt ne 0 then out_var[i,j,p] = !VALUES.F_NAN
       endfor
     endfor
   endif else if nd eq 4 then begin
     out_var = FLTARR(dims[0], dims[1], nlocs, dims[3])
     for i=0, dims[0]-1 do begin
       for j=0, dims[1]-1 do begin
         for t=0, dims[3]-1 do begin
           _z = z_in[i,j,*,t]
           out_var[i,j,*,t] = INTERPOL(varin[i,j,*,t],_z,loc_param)
           p = where(loc_param gt max(_z) or loc_param lt min(_z), cnt)
           if cnt ne 0 then  out_var[i,j,p,t] = !VALUES.F_NAN
         endfor
       endfor
     endfor
   endif else Message, 'VARIN dimensions should be 3 or 4'

   return, out_var

end

function interpol_3d_alt, varin, z_in, loc_param

   ; Set Up environnement
```

```
  COMPILE_OPT idl2

  dims = SIZE(varin, /DIMENSIONS)
  nd = N_ELEMENTS(dims)
  nlocs = N_ELEMENTS(loc_param)

  if nd eq 3 then begin ; no time dimension
    out_var = FLTARR(dims[0], dims[1], nlocs)
    for i=0, dims[0]-1 do begin
      for j=0, dims[1]-1 do begin
        _z = z_in[i,j,*]
        out_var[i,j,*] = INTERPOL(varin[i,j,*],_z,loc_param)
        p = where(loc_param gt max(_z) or loc_param lt min(_z), cnt)
        if cnt ne 0 then out_var[i,j,p] = !VALUES.F_NAN
      endfor
    endfor
  endif else if nd eq 4 then begin
    out_var = FLTARR(dims[0], dims[1], nlocs, dims[3])
    for t=0, dims[3]-1 do $
        out_var[*,*,*,t]= interpol_3d_alt(reform(varin[*,*,*,t]), $
          reform(z_in[*,*,*,t]), loc_param)
  endif else Message, 'VARIN dimensions should be 3 or 4'

  return, out_var

end

pro compare_interp

  varin = FLTARR(200,200,27,24)
  p = LINDGEN(200,200,27,24) * 1.

  pressure_levels = [850., 700., 500., 300.]
  t1 = SYSTIME(/SECONDS)
  one = interpol_3d(varin, p, pressure_levels)
  print, 'Method1: ' + str_equiv(SYSTIME(/SECONDS) - t1)
  t1 = SYSTIME(/SECONDS)
  two = interpol_3d_alt(varin, p, pressure_levels)
  print, 'Method2: ' + str_equiv(SYSTIME(/SECONDS) - t1)

end
```

Subject: Re: Another "IDL way to do this" question
Posted by Jeremy Bailin on Tue, 08 Nov 2011 20:08:46 GMT
View Forum Message <> Reply to Message

On 11/8/11 9:50 AM, Fabzou wrote:

> Dear IDLers,
>
> I have to interpolate modelled fields from eta to pressure coordinates,
> which is a relatively easy thing to do when using a simple linear
> approach. Since my 3D grid is irregular "only" on the third dimension
> (the fourth dimension being time), well I did not find any other
> solution than looping on the three regular dimensions and use INTERPOL
> for each vertical column.(interpol_3d in the code posted below)
>
> This works fine, but is not very very fast.
>
> Then I tried to loop on the time dimension outside the "main loop"
> (interpol_3d_alt in the code posted below), but results are only
> slightly better, as shown if I run the test program compare_interp:
>
> % Compiled module: INTERPOL_3D.
> % Compiled module: INTERPOL_3D_ALT.
> % Compiled module: COMPARE_INTERP.
> IDL> compare_interp
> Method1: 19.713714
> Method2: 18.455919
>
> Do you have an idea how to make this faster? I thought about other types
> of 3D interpolation, but they all seem so "overkill" for such a simple
> problem...
>
> Thanks a lot
>
> Fab

I would do the interpolation by hand over the relevant dimension. Here's
a somewhat general solution - I haven't tested it, and I *know* it fails
on the edge cases, so check it against results you know first, but it
should be much much faster than what you're doing:

```
;+
; NAME:
;   INTERPOL_D
;
; PURPOSE:
;   Perform linear interpolation on an irregular grid, a la INTERPOL(V,
X, U), but only performing
;   the interpolation over one particular dimension.
;
; CATEGORY:
;   Math
;
; CALLING SEQUENCE:
```

```
;   Result = INTERPOL_D(V, X, U, D)
;
; INPUTS:
;   V:    Input array.
;
;   X:    Abscissae values for dimension D of array V. Must have the
same number
;         of elements as the appropriate dimension of V.
;
;   U:    Abscissae values for the result. The result will have the same
number of
;         elements as U in dimension D.
;
;   D:    Dimension over which to interpolate, starting at 1.
;
; OUTPUTS:
;   INTERPOL_D returns an array with the same dimensions as V except
that dimension D
;   has N_ELEMENTS(U) elements in dimension D.
;
; MODIFICATION HISTORY:
;   Written by:   Jeremy Bailin, 8 November 2011
;
;-
function interpol_d, v, x, u, d

vsize = size(v,/dimen)
nvdimen = n_elements(vsize)
nx = n_elements(x)
nu = n_elements(u)
if n_elements(d) ne 1 then message, 'D must have only one element.' else
d=d[0]  ; scalarize it

; check that vsize contains a dimension d
if d gt nvdimen then message, 'V must contain dimension D.'
; check that X has the right number of elements
if nx ne vsize[d] then message, 'X must have the same number of elements
as dimension D of array V.'

; where do U (output) values lie w/r/t X (input) values?
u_in_x_low = value_locate(x, u)
; fractional distance between u_in_x_low and u_in_x_low+1
xfrac = (u-x[u_in_x_low])/(x[u_in_x_low+1]-x[u_in_x_low])
; reform V so that dimension D is in the first dimension and all other
dimensions are in a single
; dimension afterward
all_but_d = where(indgen(nvdimen) ne d)
n_allbutd = product(/int, vsize[all_but_d])
```

```
v = reform(transpose(temporary(v), [d,all_but_d]), [nx, n_allbutd])
; do the linear interpolation
output = v[u_in_x_low,*]*(1.-xfrac) + v[u_in_x_low+1,*]*xfrac
; and reform back so that the interpolated dimension is where it should be
resorted_dimenlist = sort([d,all_but_d])
output = reform(transpose(temporary(output), resorted_dimenlist), vsize)

return, output

end
```

Subject: Re: Another "IDL way to do this" question
Posted by Jeremy Bailin on Tue, 08 Nov 2011 21:19:30 GMT
View Forum Message <> Reply to Message

After some debugging (still will give funny results for extrapolation,
but should work perfectly for interpolation):

```
;+
; NAME:
;   INTERPOL_D
;
; PURPOSE:
;   Perform linear interpolation on an irregular grid, a la INTERPOL(V,
X, U), but only performing
;   the interpolation over one particular dimension.
;
; CATEGORY:
;   Math
;
; CALLING SEQUENCE:
;   Result = INTERPOL_D(V, X, U, D)
;
; INPUTS:
;   V:    Input array.
;
;   X:    Abscissae values for dimension D of array V. Must have the
same number
;         of elements as the appropriate dimension of V.
;
;   U:    Abscissae values for the result. The result will have the same
number of
;         elements as U in dimension D.
;
;   D:    Dimension over which to interpolate, starting at 1.
;
; OUTPUTS:
```

```
;   INTERPOL_D returns an array with the same dimensions as V except
that dimension D
;   has N_ELEMENTS(U) elements in dimension D.
;
; WARNINGS:
;   Untested. Use at your own risk.
;
; MODIFICATION HISTORY:
;   Written by:   Jeremy Bailin, 8 November 2011
;
;-
function interpol_d, v, x, u, d

vsize = size(v,/dimen)
nvdimen = n_elements(vsize)
nx = n_elements(x)
nu = n_elements(u)
if n_elements(d) ne 1 then message, 'D must have only one element.' else
d0=d[0]-1  ; scalarize it and zero-index

; check that vsize contains a dimension d
if d0 ge nvdimen then message, 'V must contain dimension D.'
; check that X has the right number of elements
if nx ne vsize[d0] then message, 'X must have the same number of
elements as dimension D of array V.'

; where do U (output) values lie w/r/t X (input) values?
u_in_x_low = value_locate(x, u)
; fractional distance between u_in_x_low and u_in_x_low+1
xfrac = (u-x[u_in_x_low])/(x[u_in_x_low+1]-x[u_in_x_low])
; reform V so that dimension D is in the first dimension and all other
dimensions are in a single
; dimension afterward
all_but_d = where(indgen(nvdimen) ne d0)
n_allbutd = product(/int, vsize[all_but_d])
v = reform(transpose(temporary(v), [d0,all_but_d]), [nx, n_allbutd])
; do the linear interpolation
xfrac = rebin(xfrac, [nu, n_allbutd], /sample)
output = v[u_in_x_low,*]*(1.-xfrac) + v[u_in_x_low+1,*]*xfrac
; and reform back so that the interpolated dimension is where it should be
resorted_dimenlist = sort([d0,all_but_d])
output = transpose(reform(temporary(output), [nu, vsize[all_but_d]]),
resorted_dimenlist)

; reform v back to original dimensions so that if it's used outside we
haven't clobbered it
v = reform(v, vsize)
```

return, output

end

Subject: Re: Another "IDL way to do this" question
Posted by Fabzou on Wed, 09 Nov 2011 08:58:16 GMT
View Forum Message <> Reply to Message

Wow Jeremy, you produced a nice peace of code here, I will try to decipher it!

The problem is that in my case, the abscissae values X are of the same dimension as V. In the third dimension case, for example, each column [Xn,Yn] has different pressure values, which does not work with your assumption that X must have the same number of elements as the dimension 3 in V.

Your algorithm is indeed much faster (couldn't check the results, though):

varin = FLTARR(200,200,27,24), interpolation on DIM3:

Mine: 19.391076 sec
JB : 0.43421888 sec

Do you think I could adapt it?

Thanks a lot,

Fab

On 11/08/2011 10:19 PM, Jeremy Bailin wrote:
> ;+
> ; NAME:
> ; INTERPOL_D
> ;
> ; PURPOSE:
> ; Perform linear interpolation on an irregular grid, a la INTERPOL(V,
> X, U), but only performing
> ; the interpolation over one particular dimension.
> ;
> ; CATEGORY:
> ; Math
> ;
> ; CALLING SEQUENCE:
> ; Result = INTERPOL_D(V, X, U, D)
> ;
> ; INPUTS:

```
> ;   V:    Input array.
> ;
> ;   X:    Abscissae values for dimension D of array V. Must have the
> same number
> ;         of elements as the appropriate dimension of V.
> ;
> ;   U:    Abscissae values for the result. The result will have the same
> number of
> ;         elements as U in dimension D.
> ;
> ;   D:    Dimension over which to interpolate, starting at 1.
> ;
> ; OUTPUTS:
> ;   INTERPOL_D returns an array with the same dimensions as V except
> that dimension D
> ;   has N_ELEMENTS(U) elements in dimension D.
> ;
> ; WARNINGS:
> ;   Untested. Use at your own risk.
> ;
> ; MODIFICATION HISTORY:
> ;   Written by:   Jeremy Bailin, 8 November 2011
> ;
> ;-
> function interpol_d, v, x, u, d
>
> vsize = size(v,/dimen)
> nvdimen = n_elements(vsize)
> nx = n_elements(x)
> nu = n_elements(u)
> if n_elements(d) ne 1 then message, 'D must have only one element.' else
> d0=d[0]-1  ; scalarize it and zero-index
>
> ; check that vsize contains a dimension d
> if d0 ge nvdimen then message, 'V must contain dimension D.'
> ; check that X has the right number of elements
> if nx ne vsize[d0] then message, 'X must have the same number of
> elements as dimension D of array V.'
>
> ; where do U (output) values lie w/r/t X (input) values?
> u_in_x_low = value_locate(x, u)
> ; fractional distance between u_in_x_low and u_in_x_low+1
> xfrac = (u-x[u_in_x_low])/(x[u_in_x_low+1]-x[u_in_x_low])
> ; reform V so that dimension D is in the first dimension and all other
> dimensions are in a single
> ; dimension afterward
> all_but_d = where(indgen(nvdimen) ne d0)
> n_allbutd = product(/int, vsize[all_but_d])
```

> v = reform(transpose(temporary(v), [d0,all_but_d]), [nx, n_allbutd])
> ; do the linear interpolation
> xfrac = rebin(xfrac, [nu, n_allbutd], /sample)
> output = v[u_in_x_low,*]*(1.-xfrac) + v[u_in_x_low+1,*]*xfrac
> ; and reform back so that the interpolated dimension is where it should be
> resorted_dimenlist = sort([d0,all_but_d])
> output = transpose(reform(temporary(output), [nu, vsize[all_but_d]]),
> resorted_dimenlist)
>
> ; reform v back to original dimensions so that if it's used outside we
> haven't clobbered it
> v = reform(v, vsize)
>
> return, output
>
> end

Subject: Re: Another "IDL way to do this" question
Posted by Brian Wolven on Wed, 09 Nov 2011 16:08:53 GMT
View Forum Message <> Reply to Message

This routine doesn't require that they be the same, but if they were you'd get an additional boost in speed as you could reuse the indexing from one point to the next. It should still be much faster than interpol - author claims a speed-up of about 60 times. YMMV

Subject: Re: Another "IDL way to do this" question
Posted by Fabzou on Wed, 09 Nov 2011 16:43:33 GMT
View Forum Message <> Reply to Message

On 11/09/2011 05:08 PM, Brian Wolven wrote:
> This routine doesn't require that they be the same, but if they were you'd get an additional boost in speed as you could reuse the indexing from one point to the next. It should still be much faster than interpol - author claims a speed-up of about 60 times. YMMV

Ah sorry I didn't look at the routine carefully. Yes, this may be
usefull in the regular case where z values are the same for each x-y-t
slice. But now it appears to be twice slower than interpol (on my
tests). David made a comment about this routine, which is now obsolete:

http://www.idlcoyote.com/tips/fast_interpolate.html

Thanks a lot!

Fabien

Subject: Re: Another "IDL way to do this" question
Posted by Brian Wolven on Wed, 09 Nov 2011 16:48:40 GMT
View Forum Message <> Reply to Message

Aha! That was a routine I'd used in code prior to the 1998-1999 time frame when INTERPOL was modified, so that is good to know.

Thank *you*! (and David...)

Subject: Re: Another "IDL way to do this" question
Posted by Jeremy Bailin on Wed, 09 Nov 2011 18:26:03 GMT
View Forum Message <> Reply to Message

On 11/9/11 11:48 AM, Brian Wolven wrote:
> Aha! That was a routine I'd used in code prior to the 1998-1999 time frame when INTERPOL was modified, so that is good to know.
>
> Thank *you*! (and David...)

Since the creation of VALUE_LOCATE, it should be much faster than a hand-coded binary search... in my code (which I'm working on modifying for the general case) I use the following two lines to efficiently get the fractional index:

```
u_in_x_low = value_locate(x, u)
xfrac = (u - u[u_in_x_low]) / (x[u_in_x_low+1] - x[u_in_x_low])
```

(the fractional index is then u_in_x_low+xfrac)

-Jeremy.

Subject: Re: Another "IDL way to do this" question
Posted by Jeremy Bailin on Wed, 09 Nov 2011 20:08:55 GMT
View Forum Message <> Reply to Message

On 11/9/11 3:58 AM, Fabzou wrote:
> Wow Jeremy, you produced a nice peace of code here, I will try to
> decipher it!
>
> The problem is that in my case, the abscissae values X are of the same
> dimension as V. In the third dimension case, for example, each column
> [Xn,Yn] has different pressure values, which does not work with your
> assumption that X must have the same number of elements as the dimension
> 3 in V.
>
> Your algorithm is indeed much faster (couldn't check the results, though):

>
> varin = FLTARR(200,200,27,24), interpolation on DIM3:
>
> Mine: 19.391076 sec
> JB : 0.43421888 sec
>
> Do you think I could adapt it?
>
> Thanks a lot,

Ah, I see! Sorry about that. Okay, well I'm sure I can adapt the
algorithm. The trick is to effectively create a multi-dimensional
version of VALUE_LOCATE. How about this (just done for your case where
the dimension is 3, so it doesn't have some of the fancy dimensional
footwork):

```
function value_locate_3d, source, data
  compile_opt idl2

  ssize = size(source,/dimen)
  ssize_3d1 = ssize
  ssize_3d1[2] = 1
  ndata = n_elements(data)
  dsize = ssize
  dsize[2] = ndata

  ; change source minimum to be zero (otherwise this will fail if it's
negative)
  ; and find the maximum
  minsource = min(source, dimen=3)
  source += rebin(reform(minsource, ssize_3d1), ssize, /sample)
  maxsource = max(source)*1.01
  source += rebin(findgen(ssize_3d1)*maxsource, ssize, /sample)

  ; now do the same to data and then use value_locate
  data_uniq = rebin(reform(data, 1,1,ndata,1), dsize, /sample) + $
    rebin(reform(minsource, ssize_3d1) + findgen(ssize_3d1)*maxsource,
dsize, /sample)
  data_in_source_uniq = value_locate(source, temporary(data_uniq))

  ; restore source
  source -= rebin(reform(minsource, ssize_3d1), ssize, /sample) + $
    rebin(findgen(ssize_3d1)*maxsource, ssize, /sample)

  ; and turn into a 1D index into the third dimension
  return, reform((array_indices(source,
```

```
temporary(data_in_source_uniq)))[2,*], dsize)
end



function interpol_3d_jb, varin, z_in, loc_param

   compile_opt idl2

   vsize = size(varin, /dimen)
   outsize = vsize
   outsize[2] = n_elements(loc_param)

   ; where do output values lie w/r/t input values
   loc_in_z  = value_locate_3d(z_in, loc_param)
   outsize_3d1 = outsize
   outsize_3d1[[0,1,3]] = 1
   noutput = n_elements(loc_in_z)

   ; prepare indices
   index1 = rebin(indgen(vsize[0],1,1,1),outsize,/samp)
   index2 = rebin(indgen(1,vsize[1],1,1),outsize,/samp)
   index4 = rebin(indgen(1,1,1,vsize[3]),outsize,/samp)

   ; fractional difference between locations and z_in
   xfrac = (rebin(reform(loc_param,outsize_3d1),outsize,/sample) -
z_in[index1,index2,loc_in_z,index4]) / $
     (z_in[index1,index2,loc_in_z+1,index4] -
z_in[index1,index2,loc_in_z,index4])

   ; do the linear interpolation
   output = varin[index1,index2,loc_in_z,index4]*(1.-xfrac) + $
     varin[index1,index2,loc_in_z+1,index4]*xfrac

return, output

end


-Jeremy.
```