Subject: Image warping in IDL
Posted by Wox on Wed, 08 Nov 2006 12:53:58 GMT
View Forum Message <> Reply to Message

Hello,

I have a question concerning image warping: "Is there a fast way of doing forward mapping in IDL?"

To make myself clear, some remarks:

1. IDL's WARP_TRI uses inverse mapping. It uses triangulation and surface interpolation to get a "non-integer" position in the original image for each "integer" position, ie. pixel, in the output (destination) image. The reverse mapping now involves surface interpolation of the original image at the non-integer positions.

2. The triangulation and surface interpolation step is already done, so that leaves only the mapping. However the non-integer positions in the "output" image were calculated in the triang-interpol step for each integer position, i.e. pixel, in the "original" image. Additionally, this "triang-interpol" step uses other techniques, using external information (spline coefficients) that can't be changed.

3. Because of step 2, forward mapping has to be performed.

Possible answers to the question above (+ why there's a problem):

1. Just do the forward mapping the hard way:

```
; img: original 2D array
;loop over rows
for i=0,nrow-1 do $
resamplearr, xmap[*,i], img, interimg, ncol, 1, i*ncol

; loop over columns
for i=0,ncol-1 do $
resamplearr, ymap[i,*], interimg, img, nrow, ncol, i
; img: destination array
```

So this loops over all rows, resampeling them separate, creating an intermediate image. Then it does the same for all columns of the intermediate. For 2000x2000 arrays, you can imagine how slow it is.

2. Another possibility would be, when we have the xmap and ymap of the forward mapping, i.e. pixel [0,0] mapped to [xmap[0,0],ymap[0,0]]

...etc., to convert them for inverse mapping, i.e.
[xmap[0,0],ymap[0,0]] mapped to [0,0] ...etc.

The problem is how? You could do this:
img = WARP_TRI(xmap, ymap, indgen(ncol)#replicate(1b,nrow),
replicate(1b,ncol)#indgen(nrow) , img)

But here the number of controlpoints equals the number of pixels in
img. Except for the speed, the general idea seems strange. You use the
"triang-interpol" step on the "triang-interpol", if you know what I
mean.

3. Maybe there is a way of making solution 2 faster. In remark 2, I
stated that the "triang-interpol" is already done using external
parameters. I can't change the parameters (I'm not calculating them)
but maybe I could convert these parameters so "inverse" mapping xmap
and ymap are calculated. The problem is, I wouldn't know how. To be
specific, these parameters are coefficients of 2D splines, one for x
and one for y.

This text is getting longer and longer. My apologies for this. At this
point I would have to thank you for reading this in the first place.
Thanks!

So the initial question becomes now:
1. Is there a way of converting forward mapping splines to inverse
splines?
2. If not, is there a fast way of doing forward mapping (cfr. solution
1)?

I hope this is making any sense. And thanks again.

Subject: Re: Image warping in IDL
Posted by JD Smith on Thu, 09 Nov 2006 19:50:15 GMT
View Forum Message <> Reply to Message

On Thu, 09 Nov 2006 13:37:21 +0100, Wox wrote:

> On 8 Nov 2006 21:49:19 -0800, "Robbie" <retsil@iinet.net.au> wrote:
>
>> I don't really understand the problem fully, but I'm sure that
>> INTERPOLATE does all the hard work for you.
>

> That would be for reverse mapping. In forward mapping, it's the
> resampling (the for-loops in the original post) that takes processing
> time. There isn't really a memory problem here.
>
> So the problem in short:
> 1. How to get ride of the looping in the resampling step
> OR
> 2. How to prevent having to do the resampling in the first place
> (going to reverse mapping by having a "magical operation" converting
> the 2D spline coefficients)

I'm afraid if you want to get any further with this, you are going to have to illustrate the issue with a small amount of actual IDL code. What do you actually have in hand? The output of TRIANGULATE without the anchor points themselves? Just the map of fractional pixel positions in the input image for each pixel in the output image (as obtained by, e.g. TRIGRID)? If it's the latter, there must have been some higher-level method for defining the warp. Rather than start with the pixel-by-pixel mapping, I'd suggest going upstream. If you can't go upstream, you'll have to triangulate your entire forward map, and then sample it at the fixed x,y grid positions of the input array. This may not give a unique solution, depending on how perverse the mapping is (it could fold onto itself, for instance).

JD

Subject: Re: Image warping in IDL
Posted by Wox on Fri, 10 Nov 2006 09:05:25 GMT
View Forum Message <> Reply to Message

On Thu, 09 Nov 2006 12:50:15 -0700, JD Smith <jdsmith@as.arizona.edu> wrote:

> I'm afraid if you want to get any further with this, you are going to
> have to illustrate the issue with a small amount of actual IDL code.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; img points to the input image
imgs=size(*img)
seval=indgen(imgs[1])
teval=indgen(imgs[2])
xmap=seval#replicate(1b,imgs[2])
ymap=replicate(1b,imgs[1])#teval

; X-distortion
 xmap+=bsplineint2Dcp(xuvec,xvvec,xp,xq,xn,xm,xh,xk,seval,tev al,xCP)
; -> these parameters come from an external source
```

```
; Y-distortion
 ymap+=bsplineint2Dcp(yuvec,yvvec,yp,yq,yn,ym,yh,yk,seval,tev al,yCP)
; -> these parameters come from an external source

; At this point we have the corresponding output pixel (non-integer)
for each input pixel

; Forward mapping: resample_array performs Fant's resampling
algorithm:
interimg=ptr_new(*img)

; loop over rows
for i=0,imgs[2]-1 do $
 resample_array, xmap[*,i], img, interimg, imgs[1], 1,
i*imgs[1]
; loop over columns
for i=0,imgs[1]-1 do $
 resample_array, ymap[i,*], interimg, img, imgs[2], imgs[1], i

ptr_free,interimg

; img now points to the output image
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The two loops are the time consumers.
So to repeat the question:
1. Can I do something faster then the looping to resample the image?
OR
2. Is there a "magic operation" for converting CP's and uvec/vvec so
that xmap and ymap describe distortion of output pixels?

I'd like to refer to this URL for the spline surface evaluated in
bsplineint2Dcp:
 http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/surface/bs pline-construct.html
So I've got the control points(CP), the knot vectors(uvec and vvec)
and I have the degrees(p,q).

Subject: Re: Image warping in IDL
Posted by Wox on Fri, 10 Nov 2006 09:12:34 GMT
View Forum Message <> Reply to Message

On Fri, 10 Nov 2006 10:05:25 +0100, Wox <nomail@hotmail.com> wrote:

> 2. Is there a "magic operation" for converting CP's and uvec/vvec so
> that xmap and ymap describe distortion of output pixels?

Maybe some simple analog example to make this clear:

Take a 1D polynomial for which you know the coeff. and degree:
y(t)=a+b.t+c.t^2+d.t^3

Is there a "magic operation" that can estimate/calculate a',b',c' and
d' in the following :
t(y)=a'+b'.y+c'.y^2+d'.y^3

The 2D splines are a little bit more complicated, but they are just
piecewise polynomial surfaces.

Subject: Re: Image warping in IDL
Posted by James Kuyper on Fri, 10 Nov 2006 16:38:59 GMT
View Forum Message <> Reply to Message

Wox wrote:
> On Fri, 10 Nov 2006 10:05:25 +0100, Wox <nomail@hotmail.com> wrote:
>
>> 2. Is there a "magic operation" for converting CP's and uvec/vvec so
>> that xmap and ymap describe distortion of output pixels?
>
> Maybe some simple analog example to make this clear:
>
> Take a 1D polynomial for which you know the coeff. and degree:
> y(t)=a+b.t+c.t^2+d.t^3
>
> Is there a "magic operation" that can estimate/calculate a',b',c' and
> d' in the following :
> t(y)=a'+b'.y+c'.y^2+d'.y^3
>
> The 2D splines are a little bit more complicated, but they are just
> piecewise polynomial surfaces.

The solution is straightforward, in one dimension. Choose a set of t
values that is sufficiently large, and use your splines for the forward
transformation to calculate corresponding y values. Then fit a spline
in y to your t values. This will provide a spline approximation to the
inverse of your first spline. It will be exactly correct at the points
you chose, and less accurate as you move away from those points.

However, this approach doesn't generalize to 2-D data very well. I hope
someone else can help you, but the only efficient algorithms I know of
for fitting 2-D splines require that the function being fitted, z(x,y),
is tabulated on the outer product of a set of x values and a set of y
values. When the only way to determine the x and y values is by
evaluating a spline interpolant, I don't see any easy way to arrange
that they form an outer-product set.

Subject: Re: Image warping in IDL
Posted by JD Smith on Fri, 10 Nov 2006 22:51:34 GMT
View Forum Message <> Reply to Message

On Fri, 10 Nov 2006 10:05:25 +0100, Wox wrote:

> ;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
> ; img points to the input image
> imgs=size(*img)
> seval=indgen(imgs[1])
> teval=indgen(imgs[2])
> xmap=seval#replicate(1b,imgs[2])
> ymap=replicate(1b,imgs[1])#teval
>
> ; X-distortion
> xmap+=bsplineint2Dcp(xuvec,xvvec,xp,xq,xn,xm,xh,xk,seval,tev al,xCP) ; ->
> these parameters come from an external source ; Y-distortion
> ymap+=bsplineint2Dcp(yuvec,yvvec,yp,yq,yn,ym,yh,yk,seval,tev al,yCP) ; ->
> these parameters come from an external source
>
> ; At this point we have the corresponding output pixel (non-integer) for
> each input pixel

Here's a possibility:

Take the fractional xmap, ymap pair of vectors, and use HIST_ND to bin into a unit cell grid the size of the output array, saving the reverse indices. Most bins (== output pixels) will have 3 input pixel or fewer mapped into them (depending on how severe the warping is).

Using the "histogram of histogram" method to loop over the small number of values in the original histogram frequency distribution (e.g. 1,2,3), and build up the output array as so:

1. Calculate the fraction of the input pixels xmap[inds],ymap[inds] which fall on each of the 9 neighbors surrounding that output bin. 5 will have zero fraction, and can be disregarded. This will depend on your pixel coordinate system (I use 0.5, 0.5 as center of first pixel, other people use other systems). See below.

2. Accumulate in the output array "f_pix * input[pix]" for each of the 4 pixels that it overlaps, careful of falling off the edge.

3. Accumulate a separate array of f_pix for each of the 4.

4. Divide the output by the accumulated f_pix array, to get the fractional area-weighted average.

All of these steps can be done without a loop. The only loop will be very short (maybe 2-3 iterations at most), over the unique repeat counts in the original 2D histogram.

This is a flux-conserving algorithm similar to drizzle, but vastly simplified by the lack of rotation/skew/etc. between input and output pixels. If a given output pixel has no input pixel "touching it", the f_pix array will be zero there after everything, and you'll have to interpolate from neighbors.

Here's a suggestion for calculating f_pix from two (potentially long) 1xn vectors, xm & ym. This is the part drizzle has to do the hard way by clipping polygons:

```
outpix=floor(xm) + im_width*floor(ym)
dx=xm-floor(xm)-.5 & dy=ym-floor(ym)-.5
sx=1-2*(dx lt 0.) & sy=1-2*(dy lt 0.)
dx=fabs(dx) & dy=fabs(dy)

f_pix=[dx*dy, (1.-dx)*dy, dx*(1.-dy), (1.-dx)*(1.-dy)]
off_x=[sx,    0,          sx,         0            ]
off_y=[sy,    sy,         0,          0            ]

output_pix=rebin(outpix,4,npix) + off_x + im_width * off_y
add_vals=rebin(input[inpix],4,npix)*f_pix
```

You'll also want to zero out the f_pix and add_vals for pixels off the array.

Note that you can't now just say:

```
output[output_pix]+=add_vals
fpix_sum[output_pix]+=f_pix,
```

since there are likely many duplicates among output_pix. See the histogram tutorial for a method using (you guessed it) HISTOGRAM. In fact, another dual histogram would do nicely there.

Yes, that's a lot of HISTOGRAMs:

H1. 2D histogram of fractional mapping coords in output pixel grid.
H2. Histogram of mapped pixel density in H1.
H3. Histogram of output pixels, for each "repeat count" density from H2.
H4. Histogram of repeat counts in the output pixels from H3.

H1 and H2 are called once. H3 and H4 are called once for each number of repeats in the 2d map histogram (i.e. 2 or 3 times, likely). If

you code this up, let us know whether it was faster.

JD

Subject: Re: Image warping in IDL
Posted by David Fanning on Fri, 10 Nov 2006 23:11:00 GMT
View Forum Message <> Reply to Message

JD Smith writes:

> If you code this up, let us know whether it was faster.

Yes, and be sure to enter the code in the IEPA code
contest. There is a slim possibility you might be offered
membership just on the basis of this program. It already
has the requisite "What the hell!?" factor. (All that
really remains for membership qualification, assuming
the program works as expected, is to know if you can
hold your liquor.)

Cheers,

David
--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Sepore ma de ni thui. ("Perhaps thou speakest truth.")

Subject: Re: Image warping in IDL
Posted by Wox on Tue, 14 Nov 2006 10:20:38 GMT
View Forum Message <> Reply to Message

On Fri, 10 Nov 2006 15:51:34 -0700, JD Smith <jdsmith@as.arizona.edu>
wrote:

<snip>

> If you code this up, let us know whether it was faster.

Wow, thanks! A nice opportunity to brush up my histogram skills :-).

I'm almost there and I will post some benchmarks when finished.

One question though: "Don't you need a second loop (a loop over the

repeat counts in H4)?"

Subject: Re: Image warping in IDL
Posted by Wox on Tue, 14 Nov 2006 16:20:52 GMT
View Forum Message <> Reply to Message

> On Fri, 10 Nov 2006 15:51:34 -0700, JD Smith <jdsmith@as.arizona.edu>
> wrote:
>
> <snip>
> If you code this up, let us know whether it was faster.
>

Version: IDL 6.2
OS: WinXP
CPU: Intel Pentium 4A, 2018 MHz (20 x 101)
Mem: 512MB

Benchmark forward mapping of array(1200x1200). Repeated 10 times,
average seconds:

Obtaining the fractional coord, using 2D spline: 0.435900

Different forward mappings:
Dummy method: 0.212500
Fant's resampling(double loop): 36.6000
Drizzle like algo. (empty pixels): 8.18290
Drizzle like algo. (trigrid empty pixels): 9.92030

The mapping seems smooth enough to omit interpolating the "empty
pixels". However in general, it should probably be used. So your
suggestion speeds it up 3/4 times! So the message seems to be the same
as always:"Use histogram." :-)

Maybe some code for the ones who are interested:

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;Obtaining the fractional coord
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
imgs=size(*img)
seval=indgen(imgs[1])
teval=indgen(imgs[2])
xmap=seval#replicate(1b,imgs[2])
```

```
ymap=replicate(1b,imgs[1])#teval

; Correct: xmap,ymap in CCD => calculate where they are in corrected
frame
; X-distortion
 xmap+=bsplineint2Dcp(tck1.uvec,tck1.vvec,tck1.p,tck1.q,tck1.
n,tck1.m,tck1.h,tck1.k,seval,teval,tck1.CP)
; Y-distortion
 ymap+=bsplineint2Dcp(tck2.uvec,tck2.vvec,tck2.p,tck2.q,tck2.
n,tck2.m,tck2.h,tck2.k,seval,teval,tck2.CP)




;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;Dummy method
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
temp=*img
(*img)[*]=0
(*img)[xmap,ymap]=temporary(temp) ; => black pixels + duplicates
overwritten!!!




;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;Fant's resampling
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
interimg=ptr_new(*img)

; loop over rows
for i=0,imgs[2]-1 do $
resample_array, xmap[*,i], img, interimg, imgs[1], 1, i*imgs[1]
; loop over columns
for i=0,imgs[1]-1 do $
resample_array, ymap[i,*], interimg, img, imgs[2], imgs[1], i

ptr_free,interimg




;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;Drizzle like algo
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>   ; Intermediate image: add boarder of two pixels
>   interimg=MAKE_ARRAY(imgs[1]+4,imgs[2]+4,type=size(*img,/type ))
>  fsum=interimg
```

```
>  fsum[*]=0
>
>  imgsinter=imgs+4
>  npixmax=imgs[1]*imgs[2]
>
>  ; H1: Group fractional pixels per pixel
>  v=[reform(xmap,1,npixmax),reform(ymap,1,npixmax)]
>   h1=hist_nd(v,1,MIN=[0,0],MAX=[imgs[1]-1,imgs[2]-1],REVERSE_I NDICES=r1)
>  ; h1 has dimensions of output image, and each pixel contains the number of pixels mapped into
it
>  ; Fractional pixels that map in pixel i: v[*,r1[r1[i] : r1[i+1]-1]]
>
>  ; H2: Histogram of number of fractional pixels per pixel
>  h2=histogram(h1,omax=omax2,omin=omin2,REVERSE_INDICES=r2)
>
>  ; Loop over the different fractional pixel counts:
>  ; handle repeat counts >0
>  i0 = omin2>1
>  for i=i0,omax2-omin2 do begin
>   if r2[i] ne r2[i+1] then begin
>    ;-----------Get output pixels + values to assign:-------------
>
>    ; indices in h1 with pixel count= i+omin2
>    indh1=r2[r2[i] : r2[i+1]-1]
>
>    ; indices in v with pixel count= i+omin2
>    ; since we know the pixel count:
>    ; r1[indh1] : r1[indh1+1]-1
>    ; <>
>    ; r1[indh1] : r1[indh1]+pixelcount-1
>    nc=omin2+i
>    indh1=rebin(r1[indh1],h2[i],nc,/SAMPLE)+ $
>     rebin(lindgen(1,nc),h2[i],nc,/SAMPLE)
>    npix=h2[i]*nc
>    indh1=r1[indh1]
>
>    ; Devide pixels in 9 neighbors (5 will get 0% of pixel)
>    ; first pixel of the 4 receiving output pixels
>    outpix=floor(v[*,indh1])
>    ; difference between fractional pixel and first pixel
>    dxy=v[*,indh1]-outpix
>    ; area of 4 rectangles (total area=1)
>    f_pix=[dxy[0,*]*dxy[1,*], (1.-dxy[0,*])*dxy[1,*], $
>     dxy[0,*]*(1.-dxy[1,*]), (1.-dxy[0,*])*(1.-dxy[1,*])]
>    off_x=rebin([3,2,3,2],4,npix) ; [1,0,1,0]+2 for boarder
>    off_y=rebin([3,3,2,2],4,npix) ; [1,1,0,0]+2 for boarder
>
>    ; For each input pixel: 4 output pixels + values
```

```
>    ; Pixels that fall off: clip (double boarder will catch them)
>    off_x=0>(rebin(outpix[0,*],4,npix)+off_x)<(imgsinter[1]-1)
>    off_y=0>(rebin(outpix[1,*],4,npix)+off_y)<(imgsinter[2]-1)
>    outpix=off_x + imgsinter[1] * off_y
>    add=rebin(reform((*img)[indh1],1,npix),4,npix)*f_pix
>
>    ;-----------Handle multiple indices in:----------
>    ; > interimg[outpix]+=add
>    ; > fsum[outpix]+=f_pix
>
>    ; H3: find duplicate outpix's
>     h3=histogram(outpix,omax=omax3,omin=omin3,REVERSE_INDICES=r3 )
>    ; H4
>    ; skip repeat count 0 (by setting min=1)
>     h4=histogram(h3,omax=omax4,omin=omin4,min=1,REVERSE_INDICES= r4)
>    ; handle repeat count 1
>    if r4[0] ne r4[1] then begin
>     ; indices in h3 with repeat count= 1
>     indh3=r4[r4[0] : r4[1]-1]
>     ; indices in outpix with with repeat count= 1
>     indh3=r3[r3[indh3]]
>     ; Add
>     interimg[outpix[indh3[*,0]]]+=add[indh3]
>     fsum[outpix[indh3[*,0]]]+=f_pix[indh3]
>    endif
>    ; handle repeat count >1
>    for j=1,omax4-omin4 do begin
>     if r4[j] ne r4[j+1] then begin
>      ; indices in h3 with repeat count= j+omin4
>      indh3=r4[r4[j] : r4[j+1]-1]
>
>      ; indices in outpix with with repeat count= j+omin4
>      nc=omin4+j
>      indh3=rebin(r3[indh3],h4[j],nc,/SAMPLE)+ $
>       rebin(lindgen(1,nc),h4[j],nc,/SAMPLE)
>      npix=h4[j]*nc
>      indh3=r3[indh3]
>
>      ; Total values for duplicate indices and add
>      interimg[outpix[indh3[*,0]]]+=total(add[indh3],2)
>      fsum[outpix[indh3[*,0]]]+=total(f_pix[indh3],2)
>     endif
>    endfor
>    ;---------------------------
>   endif
>  endfor
>  ; fractional area-weighted average
>  outpix=fsum eq 0
```

```
>  fsum+=outpix
>  interimg/=fsum
>
>  ; cut boarders before interpolation
>  interimg=interimg[2:2+imgs[1],2:2+imgs[2]]
>  outpix=outpix[2:2+imgs[1],2:2+imgs[2]]
>
>  ; interpolation of the pixels that didn't receive anything
>  h4=histogram(outpix,min=1,REVERSE_INDICES=r4)
>  if r4[0] ne r4[1] then begin
>   ind=r4[r4[0] : r4[1]-1]
>   xmap2=ind/imgs[1]
>   ymap2=ind mod imgs[1]
>
>   TRIANGULATE, xmap2, ymap2, tr, bounds
>   gs = [1,1] ;Grid spacing: for pixels -> 1 in each direction
>   b = [0,0, imgs[1]-1, imgs[2]-1] ;Bounds: 0,0,maxx,maxy
>
>   interimg=TRIGRID(xmap2, ymap2,interimg[xmap2, ymap2],tr, gs, b, /QUINT,EXTRA = bounds)
>  endif
>
>
>  ; Replace original
>  (*img)=temporary(interimg)
```

Subject: Re: Image warping in IDL
Posted by Wox on Wed, 15 Nov 2006 12:18:20 GMT
View Forum Message <> Reply to Message

> Drizzle like algo. (trigrid empty pixels): 9.92030

Sorry, I used "convol with 3x3 averaging kernel" to fill the empty
pixels. The trigrid solution made the code hopelessly slow (150sec).

Subject: Re: Image warping in IDL
Posted by Wox on Mon, 20 Nov 2006 09:18:07 GMT
View Forum Message <> Reply to Message

On Sat, 18 Nov 2006 16:22:56 -0700, Jeff Hester <jhester@asu.edu>
wrote:

<snip>
> (1) Set up a grid of points x_i, y_i spanning the image that you want to
> warp, then transform them into eta_i, xce_i in space you are warping
> into. (This is the transformation that you know how to do.)

>
> (2) Do a least squares fit for some function, (x_i, y_i) = F(eta_i,
> xce_i) using these sample points.
>
> (3) Do the "reverse" transformation in the standard way, marching
> through the output (eta, xce) space using F() to map the regularly
> gridded coordinates back into the original image.
<snip>

Thanks for your reply. The problem has been solved thanks to JD Smith's comments. However I'm not sure whether I understood the method you described:

[1] You are talking about the input and output tie points? If there was a transformation function for this, is there a point in having step [2]? (Sorry if this sounds stupid, I'm a little confused)

[2] This F is a function from R^2 -> R^2? I'm always looking at this step as two functions from R^2 -> R

[3] This was a subquestion I had before. This would be something like having y=f(x) with f a polynomial from which you know the coeff. and the evaluate x for a series of y (without fitting a second polynomial to y's calculated from a choosen series of x's, as stated in kuyper's reply).

Subject: Re: Image warping in IDL
Posted by Jeff Hester on Mon, 20 Nov 2006 17:00:39 GMT
View Forum Message <> Reply to Message

Wox wrote:
> On Sat, 18 Nov 2006 16:22:56 -0700, Jeff Hester <jhester@asu.edu>
> wrote:
>
> <snip>
>
>> (1) Set up a grid of points x_i, y_i spanning the image that you want to
>> warp, then transform them into eta_i, xce_i in space you are warping
>> into. (This is the transformation that you know how to do.)
>>
>> (2) Do a least squares fit for some function, (x_i, y_i) = F(eta_i,
>> xce_i) using these sample points.
>>
>> (3) Do the "reverse" transformation in the standard way, marching
>> through the output (eta, xce) space using F() to map the regularly
>> gridded coordinates back into the original image.

>
> <snip>
>
>
> Thanks for your reply. The problem has been solved thanks to JD
> Smith's comments. However I'm not sure whether I understood the method
> you described:
>
> [1] You are talking about the input and output tie points? If there
> was a transformation function for this, is there a point in having
> step [2]? (Sorry if this sounds stupid, I'm a little confused)
>
> [2] This F is a function from R^2 -> R^2? I'm always looking at this
> step as two functions from R^2 -> R
>
> [3] This was a subquestion I had before. This would be something like
> having y=f(x) with f a polynomial from which you know the coeff. and
> the evaluate x for a series of y (without fitting a second polynomial
> to y's calculated from a choosen series of x's, as stated in kuyper's
> reply).
>
The forward transformation is known, but you need the back transformation to do the resampling efficiently. (Presumably the forward transition is nontrivial to invert analytically.) So I run a sparse set of tie points through the forward transition, then do a fit to the tie points to get the back transformation. The key is to choose a functional form that when fit does an adequate job of representing the back transformation. Once you have the fit to the reverse transformation you can back-transform the regularly gridded points in the output image.

Sorry if I was unclear earlier.

Subject: Re: Image warping in IDL
Posted by JD Smith on Mon, 20 Nov 2006 20:17:03 GMT
View Forum Message <> Reply to Message

On Tue, 14 Nov 2006 11:20:38 +0100, Wox wrote:

> On Fri, 10 Nov 2006 15:51:34 -0700, JD Smith <jdsmith@as.arizona.edu>
> wrote:
>
> <snip>
>
>> If you code this up, let us know whether it was faster.
>
> Wow, thanks! A nice opportunity to brush up my histogram skills :-).

>
> I'm almost there and I will post some benchmarks when finished.
>
> One question though: "Don't you need a second loop (a loop over the
> repeat counts in H4)?"

Yes, as I see you figured out. Nice implementation. As you found, explicitly loop from 1 to omax in your histogram of repeat counts is fine, and solves the problem without any monkeying of indices. In fact, the snippet j=1,omax4-omin4 works only when omin4 is zero (which it seems to be always for you). j=1,omax should work. If you want to handle the j=1 case seperately for efficiency (as you've done), just do so and start the loop at 2. Also, I couldn't quite understand the rebin([3,2,3,2],4,npix) for selecting which 4 of the 9 output pixels actually receive any data. It seems like those are fixed offsets, which wouldn't work when the offset direction rotates around. Maybe something about your mapping lets you get away with that.

JD

Subject: Re: Image warping in IDL
Posted by Wox on Tue, 21 Nov 2006 09:18:02 GMT
View Forum Message <> Reply to Message

On Mon, 20 Nov 2006 13:17:03 -0700, JD Smith <jdsmith@as.arizona.edu>
wrote:

> Yes, as I see you figured out. Nice implementation. As you found,
> explicitly loop from 1 to omax in your histogram of repeat counts is
> fine, and solves the problem without any monkeying of indices. In
> fact, the snippet j=1,omax4-omin4 works only when omin4 is zero (which
> it seems to be always for you).

For H4, min=1, so omin4 is always 1. (I should have used j=1,omax4-1)

So
j=0 => repeat count 1 (handle separate)
j=1 => repeat count 2 (init loop)
...

This way we skip the 0, which is what we want. These are the "empty pixels" that need some interpolation from it's neighbours afterwards.

> j=1,omax should work. If you want to
> handle the j=1 case seperately for efficiency (as you've done), just do so
> and start the loop at 2. Also, I couldn't quite understand the
> rebin([3,2,3,2],4,npix) for selecting which 4 of the 9 output pixels

> actually receive any data. It seems like those are fixed offsets, which
> wouldn't work when the offset direction rotates around. Maybe something
> about your mapping lets you get away with that.

This is because I added a "boarder" of two pixels to the output image.

 interimg=MAKE_ARRAY(imgs[1]+4,imgs[2]+4,type=size(*img,/type))

I did this for the pixels that "fall-off". I just have to use < and >
as in:

off_x=0>(rebin(outpix[0,*],4,npix)+off_x)<(imgsinter[1]-1)
off_y=0>(rebin(outpix[1,*],4,npix)+off_y)<(imgsinter[2]-1)

After that, I cut off the 2 pixel boarder that accumulated all
fall-off pixels. I thought this was the most efficient way. Otherwise
I had to use if statements or something.

Subject: Re: Image warping in IDL
Posted by JD Smith on Tue, 21 Nov 2006 17:10:57 GMT
View Forum Message <> Reply to Message

On Tue, 21 Nov 2006 10:18:02 +0100, Wox wrote:

> This is because I added a "boarder" of two pixels to the output image.
>
> interimg=MAKE_ARRAY(imgs[1]+4,imgs[2]+4,type=size(*img,/type))
>
> I did this for the pixels that "fall-off". I just have to use < and >
> as in:
>
> off_x=0>(rebin(outpix[0,*],4,npix)+off_x)<(imgsinter[1]-1)
> off_y=0>(rebin(outpix[1,*],4,npix)+off_y)<(imgsinter[2]-1)
>
> After that, I cut off the 2 pixel boarder that accumulated all
> fall-off pixels. I thought this was the most efficient way. Otherwise
> I had to use if statements or something.

Interesting method. What I was specificially referring to is that you
have no "sign" term for dx or dy, so I'm not sure how you know which
quadrant relative to the target pixel your 4 output pixels occupy (UL,
UR, LL, LR). It seems you're always hitting a single quadrant. For the
final fsum eq 0. test for empty pix, a simple where(fsum eq 0.) should
suffice.

JD

Subject: Re: Image warping in IDL
Posted by Wox on Wed, 22 Nov 2006 08:21:27 GMT
View Forum Message <> Reply to Message

On Tue, 21 Nov 2006 10:10:57 -0700, JD Smith <jdsmith@as.arizona.edu>
wrote:

> Interesting method. What I was specificially referring to is that you
> have no "sign" term for dx or dy, so I'm not sure how you know which
> quadrant relative to the target pixel your 4 output pixels occupy (UL,
> UR, LL, LR). It seems you're always hitting a single quadrant.

It's because I use [0,0] instead of [0.5, 0.5] as center of the first
pixel. floor(xy) gives then the lower-left pixel. So UR, UL, LR and LL
are given by:
floor(x)+offx[1,0,1,0]
floor(y)+offy[1,1,0,0]

As a consequence:
dxy=xy-floor(xy)
i.e. without the 0.5, so never negative.

The result will be shifted [0.5,0.5] with your "first-pixel"
definition. I guess if one keeps this definition in further
pixel-coordinate related operations, this isn't a problem.

> For the
> final fsum eq 0. test for empty pix, a simple where(fsum eq 0.) should
> suffice.

I got carried away by the histograms there ;-).

Subject: Re: Image warping in IDL
Posted by JD Smith on Wed, 22 Nov 2006 16:22:46 GMT
View Forum Message <> Reply to Message

On Wed, 22 Nov 2006 09:21:27 +0100, Wox wrote:

> It's because I use [0,0] instead of [0.5, 0.5] as center of the first
> pixel. floor(xy) gives then the lower-left pixel. So UR, UL, LR and LL
> are given by:
> floor(x)+offx[1,0,1,0]
> floor(y)+offy[1,1,0,0]

That's pretty clever... a case where [0,0] pixels centers as a coordinate system works better (rare in my opinion ;).

JD
