
Subject: Re: Spatial normalisation

Posted by [Ben Marriage](#) on Thu, 16 Dec 1999 08:00:00 GMT

[View Forum Message](#) <> [Reply to Message](#)

Dave Brennan wrote:

>
> Hi,
>
> We have some single slice MR scans from different patients which we want
> to spatially normalise to one another. Does anyone have an algorithm
> which can do this? If not does anyone know how I would go about doing it
> myself?
>

I have written an IDL function which calculates a normalised image (invariant to scaling, skew, translation and rotation) based on moments of the image (among other things). It's based on a paper by Pei, S. and Lin, C. (Image normalisation for pattern recognition).

Ben

```
function normalise, img, display=display, itransform=inverse, ftransform=afftran, $  
                Cx=Cx, Cy=Cy, origx=origx,origy=origy,ox=ox,oy=oy
```

```
;+  
; NAME:  
;  
;   normalised.pro  
;  
; PURPOSE:  
;  
;   to produce an image which is invariant to translation, scaling,  
;   rotation and skew (for comparisons with templates, etc...)  
;  
; CATEGORY:  
;  
;   image processing  
;  
; CALLING SEQUENCE:  
;  
;   Res = NORMALISE(Image)  
;  
; INPUTS:  
;  
;   IMAGE  
;  
;   input image  
;  
; OPTIONAL INPUTS:  
;  
;
```



```

; central moments (20,11,11,02)

U20 = total(((x-Cx)^2)*((y-Cy)^0)*f)
U11 = total(((x-Cx)^1)*((y-Cy)^1)*f)
U02 = total(((x-Cx)^0)*((y-Cy)^2)*f)
U30 = total(((x-Cx)^3)*((y-Cy)^0)*f)
U21 = total(((x-Cx)^2)*((y-Cy)^1)*f)
U12 = total(((x-Cx)^1)*((y-Cy)^2)*f)
U03 = total(((x-Cx)^0)*((y-Cy)^3)*f)

; covariance matrix is
;
; [U20 U11]
; [U11 U02]
;
; and subsequently eigenvalues

eigenval1 = (U20 + U02 + sqrt((U20-U02)^2 + 4*U11^2))/2
eigenval2 = (U20 + U02 - sqrt((U20-U02)^2 + 4*U11^2))/2

; and eigenvectors

eigenvec1x = U11/sqrt((eigenval1-U20)^2 + U11^2 )
eigenvec1y = (eigenval1-U20)/sqrt((eigenval1-U20)^2+U11^2)

; rotational matrix is:

E = [[eigenvec1x,eigenvec1y],[-eigenvec1y,eigenvec1x]]

; calculate c:

c = sqrt(sqrt(eigenval1)*sqrt(eigenval2))

; W is the scaling matrix which preserves the overall size of the
; normalised shape.

W = [[c/sqrt(eigenval1),0],[0,c/sqrt(eigenval2)]]

; the transform to apply to the input image is determined by the
; matrix multiplication of W with E:

compact_trans = W##E

; the individual elements of this (to compute the third order central
; moments of the compact image using the third order central moments
; of the input image) are:

a11 = compact_trans[0,0]

```

```
a12 = compact_trans[1,0]
a21 = compact_trans[0,1]
a22 = compact_trans[1,1]
```

```
; We calculate the third order central moments of the compact image by
; combining some of the previous steps into one and actually using
; the 3rd order central moments of the input image
```

```
Up30 = (a11^3*U30) + (3*a11^2*a12*U21) + (3*a11*a12^2*U12) + (a12^3*U03)
Up21 = (a11^2*a21*U30) + (a11^2*a22 + 2*a11*a12*a21)*U21 + (2*a11*a12*a22 +
a12^2*a21)*U12 + (a12^2*a22*U03)
Up12 = (a11*a21^2*U30) + (a21^2*a12 + 2*a11*a21*a22)*U21 + (2*a12*a21*a22 +
a22^2*a11)*U12 + (a12*a22^2*U03)
Up03 = (a21^3*U30) + (3*a21^2*a22*U21) + (3*a21*a22^2*U12) + (a22^3*U03)
```

```
; calculate the tensors
```

```
t1 = Up12 + Up30
t2 = Up03 + Up21
```

```
; calculate alpha which has two solutions (see next step)
```

```
alpha = atan(-t1/t2)
```

```
; do a tensor test to determine which solution for alpha we want:
```

```
test = -t1*sin(alpha) + t2*cos(alpha)
```

```
if test lt 0. then alpha = alpha + !pi
```

```
; rotation transform is:
```

```
R=[[cos(alpha),sin(alpha)],[-sin(alpha),cos(alpha)]]
```

```
; final affine transformation is:
```

```
;
; [x'] = [cos(alpha) sin(alpha)][c/sqrt(eigenval1) 0][eigenvec1x eigenvec1y][x-Cx]
; [y'] = [-sin(alpha) cos(alpha)][0 c/sqrt(eigenval2)][-eigenvec1y eigenvec1x][y-Cy]
```

```
;
; which is : R.W.E.[x-Cx]
;             [y-Cy]
```

```
; affine transform matrix is
```

```
afftran = R##W##E
```

```
; calculate the normalised coordinates of the image. this is just done
```

```
; by extracting the two components of the matrix multiplication which
; affect the x and y coordinates separately.
```

```
normx = afftran[0,0]*(x-Cx) + afftran[1,0]*(y-Cy)
normy = afftran[0,1]*(x-Cx) + afftran[1,1]*(y-Cy)
```

```
; set up the new coordinate system for the output image.
```

```
; only transform those points where there is a 1 in the original image
; or where the original image is greater than 0
```

```
goodplot = where(img gt 0)
```

```
if not keyword_set(display) then window,/free,xs=ox,ys=oy,/pix,title='NORMALISED IMAGE' else
$
```

```
  window,/free,xs=ox,ys=oy,title='NORMALISED IMAGE'
win2 = !d.window
```

```
  plot,/nodata,[min(normx[goodplot]),max(normx[goodplot])],[min(
normy[goodplot]),max(normy[goodplot])],$
  xstyle=1,ystyle=1,position=[0.0,0.0,1.0,1.0],/iso
```

```
; calculate the inverse transform to determine each pixel value in the
; output
```

```
inverse = float(LU_COMPLEX(complex(afftran),-1,/inverse))
```

```
; normalise output image array
```

```
normi=fltarr(ox,oy)
nx = fltarr(ox,oy)
ny = fltarr(ox,oy)
```

```
for i=0,oy-1 do nx[* ,i] = indgen(ox)
for j=0,ox-1 do ny[j, *] = indgen(oy)
```

```
nxy = convert_coord(nx,ny,/dev,/to_data)
```

```
origx = (inverse[0,0]*nxy[0,*] + inverse[1,0]*nxy[1,*]) + Cx
origy = (inverse[0,1]*nxy[0,*] + inverse[1,1]*nxy[1,*]) + Cy
```

```
origx = reform(origx,ox,oy)
origy = reform(origy,ox,oy)
```

```
good = where(origx ge 0. and origy ge 0. and origx lt imgsize[0]-1 and origy lt imgsize[1]-1)
```

```
normi[good] = img[origx[good],origy[good]]
```

```
if winflag1 eq 1 then wdelete, win1  
wdelete, win2  
wset,oldwin
```

```
return, normi
```

```
end
```

File Attachments

1) [normalise.pro](#), downloaded 89 times
