

Wellenfrontanalyse mit Zernike-Polynomen und Shack-Hartmann Gradientenmessungen

Stefan Hippler, August 2006

Mit dem Shack-Hartmann Wellenfrontsensor werden die Verschiebungen $\Delta x, \Delta y$ der Spots gemessen. Diese Verschiebungen, oft auch Shack-Hartmann Gradienten genannt, stehen mit den partiellen Ableitungen der einfallenden, aberrierten Wellenfront $W(x, y)$ wie folgt im Zusammenhang:

$$\frac{\partial W(x, y)}{\partial x} = \frac{\Delta x(x, y)}{f} \quad (1)$$

$$\frac{\partial W(x, y)}{\partial y} = \frac{\Delta y(x, y)}{f} \quad (2)$$

Die Wellenfront $W(x, y)$ kann als Summe von Zernike Polynomen P_n dargestellt werden:

$$W(x, y) = \sum_n C_n P_n(x, y) \quad (3)$$

dabei entsprechen die Koeffizienten C_n der Zernike Polynome der Standardabweichung (rms) der Wellenfront für den Mode P_n . Für die partiellen Ableitungen der Wellenfront gilt:

$$\frac{\partial W(x, y)}{\partial x} = \sum_n C_n \frac{\partial P_n(x, y)}{\partial x} \quad (4)$$

$$\frac{\partial W(x, y)}{\partial y} = \sum_n C_n \frac{\partial P_n(x, y)}{\partial y} \quad (5)$$

Einsetzen von Gl. 1 und Gl. 2 ergibt:

$$\frac{\Delta x(x, y)}{f} = \sum_n C_n \frac{\partial P_n(x, y)}{\partial x} \quad (6)$$

$$\frac{\Delta y(x, y)}{f} = \sum_n C_n \frac{\partial P_n(x, y)}{\partial y} \quad (7)$$

Mit $\frac{\Delta x(x, y)}{f} = a(x, y)$ und $\frac{\Delta y(x, y)}{f} = b(x, y)$ sowie $\frac{\partial P_n(x, y)}{\partial x} = g(x, y)$ und $\frac{\partial P_n(x, y)}{\partial y} = h(x, y)$ erhält man ein lineares Gleichungssystem, das sich in Matrixform wie folgt aufschreiben läßt:

$$\begin{bmatrix} a(x_1, y_1) \\ a(x_1, y_2) \\ \vdots \\ a(x_s, y_s) \\ b(x_1, y_1) \\ b(x_1, y_2) \\ \vdots \\ b(x_s, y_s) \end{bmatrix} = \begin{bmatrix} g_1(x_1, y_1) & g_2(x_1, y_1) & \cdots & g_n(x_1, y_1) \\ g_1(x_1, y_2) & g_2(x_1, y_2) & \cdots & g_n(x_1, y_2) \\ \vdots & \vdots & \cdots & \vdots \\ g_1(x_s, y_s) & g_2(x_s, y_s) & \cdots & g_n(x_s, y_s) \\ h_1(x_1, y_1) & h_2(x_1, y_1) & \cdots & h_n(x_1, y_1) \\ h_1(x_1, y_2) & h_2(x_1, y_2) & \cdots & h_n(x_1, y_2) \\ \vdots & \vdots & \cdots & \vdots \\ h_1(x_s, y_s) & h_2(x_s, y_s) & \cdots & h_n(x_s, y_s) \end{bmatrix} \cdot \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{bmatrix} \quad (8)$$

Besteht die Shack-Hartmann Mikrolinsenmaske aus s^2 Mikrolinsen (angeordnet auf einem quadratischen Gitter $s \times s$, können maximal $2s^2$ Gradientensätze a (x Gradienten) und b (y Gradienten) gemessen werden. Achtung, die im Versuch verwendete Mikrolinsenmaske hat keine quadratische

sondern eine ringförmige Geometrie. Für die maximale Modenzahl gilt $n < 2s^2$. Schreibt man Gl. 8 als:

$$\vec{G} = \vec{\alpha}\vec{C} \quad (9)$$

dann erhält man \vec{C} über die Pseudoinverse von $\vec{\alpha}$, $\vec{\alpha}^+$:

$$\vec{C} = \vec{\alpha}^+\vec{G} \quad (10)$$

Die Pseudoinverse einer Matrix ist die Verallgemeinerung der Inversen einer quadratischen Matrix für beliebige (m,n) Matrizen. Die Pseudoinverse einer Matrix A kann für $m > n$ wie folgt definiert werden:

$$A^+ = (A^T A)^{-1} A^T \quad (11)$$

Der beste Weg, die Pseudoinverse zu berechnen, geschieht mit Hilfe der Singulärwert Zerlegung. Mit $A = USV^T$, so wie den orthogonalen Matrizen $U(n,n)$ und $V(n,n)$ und der Singulärwert Diagonal-Matrix $S(m,n)$ erhält man:

$$A^+ = V(S^T S)^{-1} S^T U^T \quad (12)$$

Die Diagonale von S besteht dabei aus realen, nicht negativen Zahlen. Das im Praktikum verwendete Programm IDL kann die Singulärwert Zerlegung mit den Prozeduren *LA_SVD* oder *SVDC* berechnen. Im Anhang ist die IDL-Helpseite für das *LA_SVD Programm* abgedruckt.

Beispiel für eine Wellenfrontrekonstruktion mit Hilfe von Zernike-Moden und gemessenen Shack-Hartmann Gradienten: die IDL Prozedur f36.pro

```

PRO F36
;PRO F36_ZERNIKE_FIT
;
; NAME:
;   F36_Zernike_fit
;
; Stefan Hippler, MPIA, Heidelberg, 27 July 2006
; PURPOSE:
;   Fit Zernike polynomials to Wavefront and Shack-Hartmann data
;
; CALLING SEQUENCE:
;   F36_ZERNIKE_FIT or just F36 (check PRO statement above)
;
; INPUTS:
;   None.
;
; RESTRICTIONS:
;   Makes use of the functions NOLLSEQUENCE and ZERNIKE.
;   and DIST_CIRCLE (astrolib)
;
; REFERENCE:
;   Noll, R. J., 1976: Zernike Polynomials and Atmospheric Turbulence, Journal
;   of the Optical Society of America, Volume 66, 207.
;
; ORIGINAL:
;   pasp-zernike_fit.pro by:
;   2005-11-08 Carsten Denker, New Jersey Institute of Technology, Center
;   for Solar-Terrestrial Research, 323 Martin Luther King Blvd, Newark,
;   NJ 07102, cdenker@adm.njit.edu
;
; error handling
ONERROR, 2

```

```

; true color device
; check with help,/device
DEVICE, decomposed=1

;
; COLOR DEFS, hex format BGR
; device has to be a true color device
; check with help,/device ----> Graphics Pixels: decomposed
; set with device,decomposed=1
;
TRED='0000FF'x
TGREEN='00FF00'x
TBLUE='FF0000'x
TWHITE='FFFFFF'x
TBLACK='000000'x
TYELLOW='00FFFF'x
MODENAME=[ 'Piston', 'Tip', 'Tilt', 'Defocus', 'Astigmatism', 'Astigmatism', $
           'Coma', 'Coma', 'Trifoil', 'Trifoil', 'Spherical_Aberration' ]

; definitions for the F36 KS28 array
;;;KS28 nx = 8      ; pixel in x-direction
;;;KS28 ny = 8      ; pixel in y-direction
; remember that the Zernike array is surrounded by zero rows/columns

;
; definitions for the Zernike derivatives test with 512 x 512 pixels
;
nx = 512      ; pixel in x-direction
ny = 512      ; pixel in y-direction

;
; remember that the Zernike array is surrounded by zero rows/columns
;
nz = 30 ; number of Zernike polynomials including piston
dx = 1  ; first partial derivative of Z, dZ/dx
dy = 1  ; first partial derivative of Z, dZ/dy

;
; azimuthal frequency and radial order according to the Noll sequence
;
mn = NOLL_SEQUENCE( nz )

;
; random variable to create the coefficients for the wavefront representation
; random Zernike coefficients
;
randomzc = RANDOMN( s, nz )

;
; use the piston term to define the aperture
;
wf = DOUBLE( ZERNIKE( mn[ 0, 0 ], mn[ 1, 0 ], nx ) )
mask = wf
index = WHERE(mask, npix)

;
; create mask for first derivatives (gradients) = diameter of mask - 4
; ignore 2 outermost circles
;
dist_circle, dmask,512, 255.5, 255.5
dmask[where(dmask LT 254.)] = 1
dmask[where(dmask GE 254.)] = 0
gindex = WHERE (dmask, gpix)

;
; Zernike polynomials as column vectors

```

```

;
zk = FLTARR( nz, npix )
zk[ 0, * ] = randomzc[ 0 ] * wf[ index ]

;
; first x derivative as column vector
;
zkdx = FLTARR( nz, gpix )
zkdx[ 0, * ] = randomzc[ 0 ] * wf[ gindex ]

;
; first y derivative as column vector
;
zkdy = FLTARR( nz, gpix )
zkdy[ 0, * ] = randomzc[ 0 ] * wf[ gindex ]

WINDOW, XSIZE = 2 * nx, YSIZE = ny
LOADCT, 13

FOR i = 1, nz-1 DO BEGIN
;
; create a wavefront composed of Zernike polynomials
;
tmp = DOUBLE( ZERNIKE( mn[ 0, i ], mn[ 1, i ], nx, dx=dx, dy=dy ) )
;
; save one Zernike mode for tests
;
IF i EQ 6 THEN zt = tmp
IF i EQ 6 THEN iiit = i
wf = TEMPORARY( wf ) + randomzc[ i ] * tmp
zk[ i, * ] = tmp[ index ]
zkdx[ i, * ] = dx[ gindex ]
zkdy[ i, * ] = dy[ gindex ]
;
m = stddev( tmp * mask )
m = m * 2.
;
; display the Zernike polynomials
;
TV, BYTSCL( tmp * mask, -m, m ), 0
TV, BYTARR( nx, ny ), 1
XYOUTS, 20, ny - 20, 'Zernike_Polynomial_Z', $
COLOR = TBLACK, /DEVICE, FONT = 0
XYOUTS, nx + 20, ny - 20, $
'F36_Wavefront_Modal_Reconstruction_with_Zernike_Polynomials,[ July_27,_2006] ', $
COLOR = TYELLOW, /DEVICE, FONT = 0
;
XYOUTS, nx + 20, ny - 40, STRING( FORMAT = $
'("Zernike_mode_number_[Noll_ordering]_j:_",_I2)', i+1 ), $
COLOR = TRED, /DEVICE, FONT = 0
;
; negative m indices mean Noll sin() Zernike functions
;
ZTERM = 'cos()'
IF mn[0, i] LT 0 THEN ZTERM = 'sin()'

XYOUTS, nx + 20, ny - 60, STRING( FORMAT = $
'("Azimuthal_frequency_m_[Noll_",_A,_term]:_",_I2)', $
ZTERM, mn[ 0, i ] ), $
COLOR = TRED, /DEVICE, FONT = 0
XYOUTS, nx + 20, ny - 80, STRING( FORMAT = $
'("Radial_order_n:_",_I2)', mn[ 1, i ] ), $
COLOR = TRED, /DEVICE, FONT = 0

IF (i+1 LE 11) THEN BEGIN
XYOUTS, nx + 20, ny - 100, STRING( FORMAT = $

```

```

        '("Aberration_name: ",_A)', MODENAME[i] ), $
        COLOR = TYELLOW, /DEVICE, FONT = 0
    ENDIF
;
; display the first partial derivated dZ/dx
;
    tmpdx = dx * mask
    m = stddev(dx * mask)
    m = m * 2.
    tmpdx = rebin(tmpdx, nx/2, ny/2)
    TV, BYTSCL( tmpdx, -m, m ), nx, 0
    XYOUTS, nx + 20, ny/2 + 20, 'First_partial_derivative_dZ/dx', $
        COLOR = TYELLOW, /DEVICE, FONT = 0

;
; display the first partial derivated dZ/dy
;
    tmpdy = dy * mask
    m = stddev(dy * mask)
    m = m * 2.
    tmpdy = rebin(tmpdy, nx/2, ny/2)
    TV, BYTSCL( tmpdy, -m, m ), nx+nx/2, 0
    XYOUTS, nx+nx/2 + 20, ny/2 + 20, 'First_partial_derivative_dZ/dy', $
        COLOR = TYELLOW, /DEVICE, FONT = 0
    WAIT, 0.5

ENDFOR

; erase display
ERASE

;
; check DERIV on pure Zernike no #iiit
;
wft = zt
; compute dynamic range for image display
m = MAX( ABS( wft[ index ] ) ) / 2.
wftm = rebin(wft*mask, nx/2, ny/2)
TV, BYTSCL( wftm, -m, m ), 0, 0
XYOUTS, 20, ny/2 + 20, 'Zernike_#7', $
    COLOR = TRED, /DEVICE, FONT = 0

;
; Zernikes are defined on the unit circle
; DINDGEN generates indices from 0:nx-1.
;
; xc = [-1,1]
; yc = [-1,1]
;
xc = DINDGEN( nx ) / ( nx - 1 ) * 2.0 - 1.0
yc = REPLICATE( 1.0D, nx ) # xc
xc = xc # REPLICATE( 1.0D, nx )

; compute dwft / dx
dwftdx = DBLARR( nx, ny)
FOR iii = 0, nx - 1 DO dwftdx[ *, iii ] = DERIV( xc[*,iii], wft[*,iii] )
;
m = MAX( ABS( dwftdx[ gindex ] ) ) / 2.
dwftdxmask = rebin(dwftdx * mask, nx/2, ny/2)
TV, BYTSCL( dwftdxmask, -m, +m), nx, 0
XYOUTS, nx + 20, ny/2 + 20, 'd(Zernike_#7)/dx', $
    COLOR = TYELLOW, /DEVICE, FONT = 0

; now the same with some noise on Z6
wft = zt
wft = wft + 0.1 * RANDOMN( s, nx, ny )
;

```

```

m = MAX( ABS( wft[ index ] ) ) / 2.
wftm = rebin(wft*mask, nx/2, ny/2)
TV, BYTSCL( wftm, -m, +m ), nx/2, 0
XYOUTS, nx/2 + 20, ny/2 + 20, 'Noisy_Zernike_#7', $
    COLOR = TRED, /DEVICE, FONT = 0

; compute dwft / dx
dwftdx = DBLARR( nx, ny)
FOR iii = 0, nx - 1 DO dwftdx[ *, iii ] = DERIV( xc[*,iii], wft[*,iii] )
;
m = MAX( ABS( dwftdx[ gindex ] ) ) / 2.
dwftdxmask = rebin(dwftdx * mask, nx/2, ny/2)
TV, BYTSCL( dwftdxmask, -m, +m), nx + nx/2, 0
XYOUTS, nx + nx/2 + 20, ny/2 + 20, 'd(Noisy_Zernike_#7)/dx', $
    COLOR = TYELLOW, /DEVICE, FONT = 0

WAIT, 5.0

; erase display
ERASE

;
; the real fun
;

; compute dynamic range for image display
m1 = MAX( ABS( wf[ index ] ) ) / 2.
wfmask = rebin(wf*mask, nx/2, ny/2)
TV, BYTSCL( wfmask, -m1, m1 ), 0, 0
XYOUTS, 20, ny/2 + 20, 'Random_wavefront_(without_noise)_W', $
    COLOR = TWHITE, /DEVICE, FONT = 0

; add some noise to the wavefront to make the fit more interesting
; wfn = wf + RANDOMN( s, nx, ny )
; wfnmask = rebin(wfn * mask, nx/2, ny/2)
; TV, BYTSCL( wfnmask, -m, m ), nx/2, 0
; XYOUTS, nx/2 + 20, ny/2 + 20, 'Random wavefront + noise', $
; COLOR = TRED, /DEVICE, FONT = 0

; compute dwf / dx
dwfdx = DBLARR( nx, ny)
FOR iii = 0, nx - 1 DO dwfdx[ *, iii ] = DERIV( xc[*,iii], wf[*,iii] )
;

;
; xxx=dwfdx[gindex]
; print,min(xxx),max(xxx),mean(xxx),stddev(xxx)
; dwfdx[gindex] = xxx - mean(xxx)
;

m2 = MAX( ABS( dwfdx[ gindex ] ) ) / 3.
dwfdxmask = rebin(dwfdx * mask, nx/2, ny/2)
TV, BYTSCL( dwfdxmask, -m2, +m2), nx, 0
XYOUTS, nx + 20, ny/2 + 20, 'dW/dx', $
    COLOR = TYELLOW, /DEVICE, FONT = 0

; compute dwfmask / dy
dwfdy = DBLARR( nx, ny)
FOR iii = 0, ny - 1 DO dwfdy[ iii, * ] = DERIV( yc[ iii, * ], wf[ iii, * ])
;

;
; xxx=dwfdy[gindex]
; print,min(xxx),max(xxx),mean(xxx),stddev(xxx)
; dwfdy[gindex] = xxx - mean(xxx)
;

```

```

m3 = MAX( ABS( dwfdy[ gindex ] ) ) / 3.
dwfdymask = rebin(dwfdy * mask, nx/2, ny/2)
TV, BYTSCL( dwfdymask, -m3, +m3), nx + nx/2, 0
XYOUTS, nx + nx/2 + 20, ny/2 + 20, 'dW/dy', $
    COLOR = TYELLOW, /DEVICE, FONT = 0
;
; WAIT, 5.0
; not necessary???, SVDC takes some time
;

XYOUTS, nx + 20, ny - 20, $
    'Computing_Singular_Value_Decomposition_of_Zernike_modes_matrix...', $
    COLOR = TYELLOW, /DEVICE, FONT = 0
;
; Singular Value Decomposition of zk to get the coefficients from the wavefront
; zk = U S V^T
; SVDC returns the singular values w first, then the u and v matrices
;
; zk contains the Zernike coefficients in a 2-dimensional matrix [nz,npix]
;
;SVDC, zk, w, u, v
LA_SVD, zk, w, u, v
FOR i = 0, nz - 1 DO IF ABS( w[ i ] ) LE 1.0e-5 THEN w[ i ] = 0
;
; now solve the linear equations system
; Zernike_Coefficients = Wavefront * PseudoInverse(Zernike Polynomials)
; using SVDC and SVSOL
;
; SVSOL = Singular Value Solve
; solve the linear equations system WF = ZC * ZK
; with the output of the singular value decomposition of ZK
; ZC = [WF] * [ZK]+
;
zc = SVSOL( u, w, v, wf[ index ] )

; 2D Zernike fit
wff = FLTARR( nx, ny )
wff[ index ] = REFORM( zk ## zc )
m = stddev(wff * mask)
m=m*2.
TV, BYTSCL( wff * mask, -m, m ), 0
XYOUTS, 20, ny - 20, 'Reconstructed_wavefront', $
    COLOR = TBLACK, /DEVICE, FONT = 0

; residuals
TV, BYTSCL( ( wf - wff ) * mask, -m / 100. , m / 100. ), 1
XYOUTS, nx + 20, ny - 20, $
    '(Wavefront-reconstructed_wavefront)_x_100', $
    COLOR = TBLACK, /DEVICE, FONT = 0
WAIT, 5.0

;
; reset color table
;
LOADCT, 0

; take care of the piston term, NOLL numbering starts with 1 (piston)
x = INDGEN( nz - 1 ) + 2
y = randomzc
y = y[ 1 : * ]

;
; show the random Zernike coefficients
; P.MULTI [No. of plots, COL, ROW, STACK, ORDER]
;
!P.MULTI = [ 0, 2, 1 ]

```

```

PLOT, x, y, TITLE = 'Zernike_coefficients_ZC_of_random_wavefront', $
    XTITLE = 'Mode_number_(Noll)', $
    YTITLE = 'ZC', YRANGE = [ -3, 3 ], YSTYLE = 1
;
; show random - reconstructed modes
;
x = INDGEN( nz - 1 ) + 2
y = randomzc - zc
y = y[ 1 : * ]
PLOT, x, y, TITLE = 'Fitting_error_of_Zernike_coefficients', $
    XTITLE = 'Mode_number_(Noll)', YTITLE = 'Fitting_error', $
    YRANGE = [ -0.01, 0.01 ]

WAIT, 5.0

;
; reset plot window
;
!P.MULTI = 0

;
; erase display
; and replot W, dW/dx, and dW/dy
ERASE
;
TV, BYTSCL( wfmask, -m1, m1 ), 0, 0
XYOUTS, 20, ny/2 + 20, 'Random_wavefront_(without_noise)_W', $
    COLOR = TWHITE, /DEVICE, FONT = 0
;
TV, BYTSCL( dwfdxmask, -m2, +m2), nx, 0
XYOUTS, nx + 20, ny/2 + 20, 'dW/dx', $
    COLOR = TYELLOW, /DEVICE, FONT = 0
;
TV, BYTSCL( dwfdymask, -m3, +m3), nx + nx/2, 0
XYOUTS, nx + nx/2 + 20, ny/2 + 20, 'dW/dy', $
    COLOR = TYELLOW, /DEVICE, FONT = 0
;
XYOUTS, nx + 20, ny - 20, $
    'Computing_Singular_Value_Decomposition_of_Zernike_GRADIENTS_matrix...', $
    COLOR = TYELLOW, /DEVICE, FONT = 0
;
;
; Find Zernike coefficients using first derivatives of wavefront;
;
; Shack-Hartmann measures dW/dx and dW/dy
; ZCG = [dWF/dx dWF/dy] * [dZP/dx dZP/dy]+
;
; create Zernike derivatives matrix with dimension [nmodes,2*size(gindex)]
;
zkdx dy = [transpose(zkdx),transpose(zkdy)]
zkdx dy = transpose(zkdx dy)
wf dx dy = [dwfdx[gindex],dwfdy[gindex]]

; help, zkdx dy, wf dx dy

;SVDC, zkdx dy, w, u, v
LA.SVD, zkdx dy, w, u, v
FOR i = 0, nz - 1 DO IF ABS( w[ i ] ) LE 1.0e-5 THEN w[ i ] = 0
zcg = SVSOL(u, w, v, wf dx dy)

;
; 2D Zernike fit
;
wff = FLTARR( nx, ny )
wff[ index ] = REFORM( zk ## zcg )
m = stddev(wff * mask)
m=m*2.

```



```

TV, BYTSCL( wff * mask, -m, m ), 0
XYOUTS, 20, ny - 20, 'Wavefront_Reconstruction', $
    COLOR = TBLACK, /DEVICE, FONT = 0
XYOUTS, 20, ny - 40, 'from_Gradients', $
    COLOR = TBLACK, /DEVICE, FONT = 0

;
; residuals
;
TV, BYTSCL( ( wf - wff ) * mask, -m / 100. , m / 100. ), 1
XYOUTS, nx + 20, ny - 20, $
    '(Wavefront_-reconstructed_wavefront)_x_100', $
    COLOR = TBLACK, /DEVICE, FONT = 0
WAIT, 5.0

;
; reset color table
;
LOADCT, 0

;
; show coefficients
;
!P.MULTI = [ 0, 2, 1 ]
; take care of the piston term and NOLL numbering starts with 1 (IDL starts with 0)
x = INDGEN( nz - 1 ) + 2
y = randomzc
y = y[ 1 : * ]
PLOT, x, y, TITLE = 'Zernike_coefficients_ZC_of_random_wavefront', $
    XTITLE = 'Mode_number_(Noll)', $
    YTITLE = 'ZC', YRANGE = [ -3, 3 ], YSTYLE = 1

;
; take care of the piston term and NOLL numbering starts with 1 (IDL starts with 0)
;
x = INDGEN( nz - 1 ) + 2
y = randomzc - zcg
y = y[ 1 : * ]
PLOT, x, y, TITLE = 'Fitting_error_of_Zernike_coefficients_(from_gradients)', $
    XTITLE = 'Mode_number_(Noll)', YTITLE = 'Fitting_error', $
    YRANGE = [ -0.01, 0.01 ]

;
; reset plot window
;
!P.MULTI = 0

print, 'Random_Zernike_coefficients_of_input_wavefront'
print, '_____',
print, randomzc
print, 'Reconstructed_Zernike_coefficients_from_wavefront_data'
print, '_____',
print, zc
print, 'Reconstructed_Zernike_coefficients_from_wavefront_gradients'
print, '_____',
print, zcg

```

END

Anhang: IDL Help Ausdruck für LA_SVD



LA_SVD

[Syntax](#) | [Arguments](#) | [Keywords](#) | [Examples](#) | [Version History](#) | [See Also](#)

The LA_SVD procedure computes the singular value decomposition (SVD) of an n -columns by m -row array as the product of orthogonal and diagonal arrays:

A is real: $A = U S V^T$

A is complex: $A = U S V^H$

where U is an orthogonal array containing the left singular vectors, S is a diagonal array containing the singular values, and V is an orthogonal array containing the right singular vectors. The superscript T represents the transpose while the superscript H represents the Hermitian, or transpose complex conjugate.

If $n < m$ then U has dimensions $(n \times m)$, S has dimensions $(n \times n)$, and V^H has dimensions $(n \times n)$. If $n \geq m$ then U has dimensions $(m \times m)$, S has dimensions $(m \times m)$, and V^H has dimensions $(n \times m)$. The following diagram shows the array dimensions:

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \cdot \begin{bmatrix} S \end{bmatrix} \cdot \begin{bmatrix} V^T \end{bmatrix} \quad n < m$$

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \cdot \begin{bmatrix} S \end{bmatrix} \cdot \begin{bmatrix} V^T \end{bmatrix} \quad n \geq m$$

LA_SVD is based on the following LAPACK routines:

Table 3-66: LAPACK Routine Basis for LA_SVD

Output Type	LAPACK Routine	
	QR Iteration	Divide-and-conquer
Float	sgestd	sgestd
Double	dgestd	dgestd
Complex	cgestd	cgestd
Double complex	zgestd	zgestd

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_SVD, Array, W, U, V [, /DOUBLE] [, /DIVIDE_CONQUER] [, STATUS=variable]

Arguments

Array

The real or complex array to decompose.

W

On output, W is a vector with $\text{MIN}(m, n)$ elements containing the singular values.

U

On output, U is an orthogonal array with $\text{MIN}(m, n)$ columns and m rows used in the decomposition of $Array$. If $Array$ is complex then U will be complex, otherwise U will be real.

V

On output, V is an orthogonal array with $\text{MIN}(m, n)$ columns and n rows used in the decomposition of $Array$. If $Array$ is complex then V will be complex, otherwise V will be real.

Note

To reconstruct $Array$, you will need to take the transpose or Hermitian of V .

Keywords

DIVIDE_CONQUER

If this keyword is set, then the divide-and-conquer method is used to compute the singular vectors, otherwise, QR iteration is used. The divide-and-conquer method is faster at computing singular vectors of large matrices, but uses more memory and may produce less accurate singular values.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set $\text{DOUBLE} = 0$ to use single-precision for computations and to return a single-precision (real or complex) result. The default is DOUBLE if $Array$ is double precision, otherwise the default is $\text{DOUBLE} = 0$.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- $\text{STATUS} = 0$: The computation was successful.
- $\text{STATUS} > 0$: The computation did not converge. The STATUS value specifies how many superdiagonals did not converge to zero.

Note

If STATUS is not specified, any error messages will output to the screen.

Examples

Construct a sample input array A , consisting of smoothed random values:

```

PRO ExLA_SVD
; Create a smoothed random array:
n = 100
m = 200
seed = 12321
a = SMOOTH(RANDOMN(seed, n, m, /DOUBLE), 5)

; Compute the SVD and check reconstruction error:
LA_SVD, a, w, u, v
arecon = u ## DIAG_MATRIX(w) ## TRANSPOSE(v)
PRINT, 'LA_SVD error:', MAX(ABS(arecon - a))

; Keep only the 15 largest singular values
wfiltered = w
wfiltered[15:*] = 0.0
; Reconstruct the array:
afiltered = u ## DIAG_MATRIX(wfiltered) ## TRANSPOSE(v)
percentVar = 100*(w^2)/TOTAL(w^2)
PRINT, 'LA_SVD Variance:', TOTAL(percentVar[0:14])
END

```

When this program is compiled and run, IDL prints:

```

LA_SVD error:  1.0103030e-014
LA_SVD variance:      82.802816

```

Note

More than 80% of the variance is contained in the 15 largest singular values.

Version History

5.6 Introduced

See Also

LA_CHOLDC, LA_LUDC, SVDC

IDL Online Help (March 01, 2006)