

Linear Algebra and Image Processing: Additional Theory regarding Computer Graphics and Image Processing not covered by David C. Lay

Dr. D.P. Huijsmans
LIACS, Leiden University

February 2012

1 Differences in conventions between Mathematics, Computer Graphics textbooks and Programming Languages

1.1 Order of multiplication

In Lay the European convention with respect to matrix vector multiplication is used, i.e., *Column-After convention*: Matrix in front of a column vector; for matrices to be concatenated later transformations are put in front of earlier transformations.

In many (especially the early) US computer graphics textbooks a *Row-Vector-in-Front convention* is followed: a row-vector is followed by a series of matrices; early ones closest to the row, later ones are added to the right.

When converting between these systems one should take into account that: Both vector and matrix data have to be replaced by their transposed versions and the order of matrices to be concatenated should be reversed.

1.2 Choice of coordinate system

In mathematical texts an (x, y) coordinate system is usually accompanied by an origin at the lower-left position and $+x$ towards the right, $+y$ pointing upwards.

In Computer Graphics and Image Processing environment the video refresh (row by row from top-left to bottom-right) is usually followed with the origin at the top left of the screen and $+x$ to the right and $+y$ pointing downwards! Matlab has an even a stranger convention for image arrays, by storing it in the (y, x) instead of (x, y) order. So, for entry (i, j) in Matlab, i represents the y -line and j the x -line.

1.3 Choice of serialization between programming languages

Images are multidimensional arrays, 2D for grayvalue images and 3D for color images (R,G,B triplet intensities per pixel). Any multidimensional arrays computer representation is serialized into an internal 1 dimensional range of byte addresses. If we take an image array to have rows, columns and planes one should keep in mind for a particular programming language which index is the slowest and which the fastest changing one. Differences exist between programming languages:

- The Fortran (column by column) serialization for instance for a 2D array is the opposite of a C(++) (row by row) serialization
- Most programming languages with N size rows have addresses running from $[1, N]$, but C(++) has its row indices running from $[0, N - 1]$.
- Matlab image and coordinate convention: Matlab stores pixels in (y, x) order (line by line top to bottom) but for many coordinate transformation functions expects the pixel coordinates in (x, y) order! Matlab uses the $[1, N]$ convention so the origin is at the topleft position just outside the actual image!

2 Image Processing Convolution Filters

An image processing convolution filter uses a set of global position-dependent weights to sum intensities in each local $n \times m$ pixel area into an inner product value that is placed at the corresponding center of an output image.

For a 3×3 local filter, with pixel-intensities $I(i, j)$:

Local environment of pixel(i, j)	Weight coefficients used
$\begin{bmatrix} I(i-1, j-1) & I(i-1, j) & I(i-1, j+1) \\ I(i, j-1) & I(i, j) & I(i, j+1) \\ I(i+1, j-1) & I(i+1, j) & I(i+1, j+1) \end{bmatrix}$	$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix}$

The 2D environments of pixel-values and coefficients are serialized row-wise into:

- a row-vector $(I(i-1, j-1), I(i-1, j), \dots, I(i, j), \dots, I(i+1, j), I(i+1, j+1))$; and
- a column vector: $(C_{11}, C_{12}, \dots, C_{22}, \dots, C_{32}, C_{33})^T$.

Their inner product I_{out} is written to the corresponding middle position (i, j) of the output image. Of course different serializations are possible here but the only important thing is that it is done the same way for the local pixel values and position-dependent weight factors.

2.1 Gradient Estimators

The spatial derivatives of intensity patterns are often used in Image Processing. (Because the image is 2D, the derivative, or gradient as it is often called, is direction dependent.)

2.1.1 Roberts gradient

From the definition of a derivative of $I(x, y)$,

$$\frac{\partial I}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{I(x + \Delta x, y) - I(x, y)}{\Delta x}, \quad \frac{\partial I}{\partial y} = \lim_{\Delta y \rightarrow 0} \frac{I(x, y + \Delta y) - I(x, y)}{\Delta y}.$$

In a discrete setting with integer grid values for x and y , take $\Delta x = \Delta y = 1$:

$$\frac{\Delta I}{\Delta x}(i, j) = I(i+1, j) - I(i, j), \quad \frac{\Delta I}{\Delta y}(i, j) = I(i, j+1) - I(i, j).$$

In other words, the differences between two neighboring intensities in a row or column. These gradient are known as the Roberts x and y gradients. Rewritten as local 1×2 or 2×1 environments of intensities and weight coefficients, we have

$$R_x(i, j) = \begin{bmatrix} I(i+1, j) & I(i, j) \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \text{and} \\ R_y(i, j) = \begin{bmatrix} I(i, j+1) & I(i, j) \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

The combination of two Roberts operators create a second derivative estimator with coefficients horizontally or vertically (or diagonally) with weights $\begin{bmatrix} -1 & 2 & -1 \end{bmatrix}$ that are combined Laplace 3×3 operators like

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}, \quad \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}.$$

Theory has it that zero crossings of this second derivative is an optimal segmentation choice.

2.1.2 Sobel gradient

Other famous gradient operators are those from Sobel:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}.$$

Effectively this gives an average of 4 Roberts gradient estimators, halving noise.

All gradient like operators can be recognized by the fact that their sum of coefficients equals zero (in a homogeneous local environment no gradient would be present).

Smoothing filters can be recognized by the fact that their sum of coefficients equals one (a homogeneous local environment remains equal).

3 Including Translations in coordinate transformations via linear algebra

Elementary coordinate transformations in 2D and 3D using 2×2 or 3×3 transformation matrices cannot implement translations of the origin, whereas such a translation is an often used elementary transformation in Computer Graphics and Image Processing, for instance when rotating an image around its center instead of its origin in the top left corner.

The following trick is applied to include translations as an elementary transformation: an extra dimension is added, the so-called homogeneous coordinate which is chosen to be 1 and remain 1.

The 2×2 and 3×3 transformation matrices for 2D and 3D coordinate transformations are also enlarged to 3×3 and 4×4 homogeneous transformation matrices for 2D and 3D transformations: Elementary homogeneous transformations now look like this in 2D:

$$\begin{array}{ccc} \text{Translation 2D} & \text{Scaling 2D} & \text{Rotation 2D} \\ \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Homogeneous coordinate transformations in 2D are applied as follows:

$$\begin{array}{ccccc} \text{2D Input} & \text{Homogeneous} & \text{Transformed} & \text{Normalised} & \text{2D Result} \\ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} & \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} & \begin{bmatrix} x' \\ y' \\ h \end{bmatrix} & \begin{bmatrix} x'/h \\ y'/h \\ 1 \end{bmatrix} & \begin{bmatrix} x'/h \\ y'/h \end{bmatrix} \end{array}$$

For instance: rotate an image 30° (degrees) around its center (30, 70). To do this we have to concatenate three elementary operations: translate origin to center, rotate around this new center, translate back to old origin position.

The concatenated transformation matrix C becomes:

$$\begin{aligned} C &= \begin{bmatrix} 1 & 0 & 30 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ & 0 \\ \sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -30 \\ 0 & 1 & -70 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ & (-30 \cos 30^\circ + 70 \sin 30^\circ + 30) \\ \sin 30^\circ & \cos 30^\circ & (-30 \sin 30^\circ - 70 \cos 30^\circ + 70) \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

Applied to a pixel position (x, y) :

$$\begin{aligned} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &\rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow C \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} x \cos 30^\circ - y \sin 30^\circ - 30 \cos 30^\circ + 70 \sin 30^\circ + 30 \\ x \sin 30^\circ + y \cos 30^\circ - 30 \sin 30^\circ - 70 \cos 30^\circ + 70 \\ 1 \end{bmatrix}. \end{aligned}$$

Since this result is already in normalized form, the top 2 positions are the resulting transformed point (x', y') .

Note that under this transformation the rotation center (30, 70) remains invariant: if $(x, y) = (30, 70)$,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 30 \cos 30^\circ - 70 \sin 30^\circ - 30 \cos 30^\circ + 70 \sin 30^\circ + 30 \\ 30 \sin 30^\circ + 70 \cos 30^\circ - 30 \sin 30^\circ - 70 \cos 30^\circ + 70 \end{bmatrix} = \begin{bmatrix} 30 \\ 70 \end{bmatrix}.$$

4 Why Inverse Coordinate Transformations are used with Images

If one would use a forward coordinate transformation to copy a pixels intensity value(s) to the new position, it may well happen that, due to rotation and/or scaling effects, not all resulting pixels will be completely filled in the output image area: when input positions are mapped onto non-neighboring output positions, intermediate pixels in the output image might not receive a copy of any input pixel position. Thus, the output image may have small holes.

To prevent this from happening, in Image Processing, image coordinate transformations are carried out inversely: each pixel position in the output image is visited; using the inverse coordinate transformation, its input location (not necessarily integer-valued) is located; since the input image consists of a set of integer grid positions (row and column numbers) this real position may end up either:

- outside the input grid, in which case a background value is used; or
- within an input grid cell with 4 intensity values at its corners, in which case a so-called interpolation recipe is used to determine what value will be used as the output value.

The easiest interpolation recipe is to discard the fractional value of the calculated input floating point position and use the resulting integer corner position intensity value. One can also round the floating point position and take the nearest corner intensity value. One can also calculate a bi-linear interpolated value from the 4 nearest corner intensity values. The inverse concatenated transformation matrix can be obtained from the forward one by determining its inverse. But the inverse is often most easily constructed from re-ordered inverse elementary transformations, which are simple versions of the forward ones. In 2D:

Inverse translation	Inverse rotation	Inverse scaling
$\begin{bmatrix} 1 & 0 & -dx \\ 0 & 1 & -dy \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos(-\alpha) & -\sin(-\alpha) & 0 \\ \sin(-\alpha) & \cos(-\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

5 3D perspective transformations

A lucky coincidence of the use of homogeneous coordinates is that not only the extra column at the back (for translations) but also the extra row below can serve a useful purpose.

When a 4×4 coordinate transformation matrix has non-zero entries in the $(4, 1)$, $(4, 2)$, $(4, 3)$ positions, the matrix acts as a perspective transformation with convergence points in the accompanying direction. The use of a single non-zero value at the $(4, 3)$ position introduces a central perspective transformation along the z -axis (or x_3 direction) with a central vanishing point. But also in the horizontal direction and vertical direction, (pairs of) vanishing points can be introduced for x going towards $+\infty$ or $-\infty$ and/or y going towards $+\infty$ or $-\infty$.

An example of a central perspective projection is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & V & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 + Vz \end{bmatrix} \rightarrow \begin{bmatrix} x/(1 + Vz) \\ y/(1 + Vz) \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x/(1 + Vz) \\ y/(1 + Vz) \\ 0 \end{bmatrix}$$

The 4×4 matrix is a concatenated perspective and projection matrix that models projecting 3D space onto a 2D screen, as seen by an observer located at $(0, 0, -1/V)$. Note that the first $\rightarrow$ is *not* a linear transformation, but on the other hand it only needs to be done once during the calculation. Note also that input points with $z = -1/V$ or $z < -1/V$ must be treated with care: these correspond to locations that are “beside” or “behind” the observer, relative to the screen.

In Lay Ch. 2.7, perspective projections are treated in a way that differs from the usual set-up in Computer Graphics: in fig. 6 page 163 they place the model points to be perspectively projected in a location between eye/camera and projection screen, whereas Computer Graphics prefers to have the screen between the eye/camera and the model points. Also the choice of which point is the origin and which direction is positive may differ between users.

Users therefore have to make very clear how eye/camera, screen and model scene are positioned with respect to each other and which point will be taken as origin and in which direction $+x$, $+y$ and $+z$ are pointing.

6 Transforming Computer Graphics Models

In computer graphics points play the main role; points can be interpolated pairwise into edges; triplets of points are interpolated into triangular patches; patches build up surfaces; textures are mapped to surfaces. Since anything but the defining points themselves can be interpolated, they are the only quantities (point coordinates) that have to be transformed; everything else

(ordering of vertices; edges and surface triangles) remain the same. CG therefore only has to calculate a coordinate transformation for list of points. The data matrix D (see Lay Ch. 2.7) consists of these vertices (one column per vertex).

7 Continuous vs. discrete lines

Whereas continuous lines when not parallel or coinciding will always have an intersection point, this need not be true for discretized lines! For instance, the following discrete lines have no discrete intersection:

```

+           *
  +   *
    *   +
      *       +
        *           +

```

8 In-betweening with affine and convex combinations of point vectors

In Lay 4th Edition Ch. 8, affine and convex point combinations in space or over time (often used in CG animations) of line segments and filled triangles are treated.