Applications of Mathematics in Information and Communication Technology

School of Information and Communication Technology,
Aalborg University

July 2013

Contents

Chapter 1

Geometric Transformations of Images

Keywords: Image processing, computer graphics, linear transformation, affine transformation, transformation matrix.

Images can be transformed in many ways. One class of transformations is called "linear transformations," which include scaling, rotation, and shearing. For the purpose of illustration, we can think of an image as a grid of small "picture elements" (i.e. pixels) where each pixel has coordinates (x, y), see Figure 1.1a. The coordinates (x, y) of each pixel are then transformed to new coordinates (x', y') in order to transform the whole image.

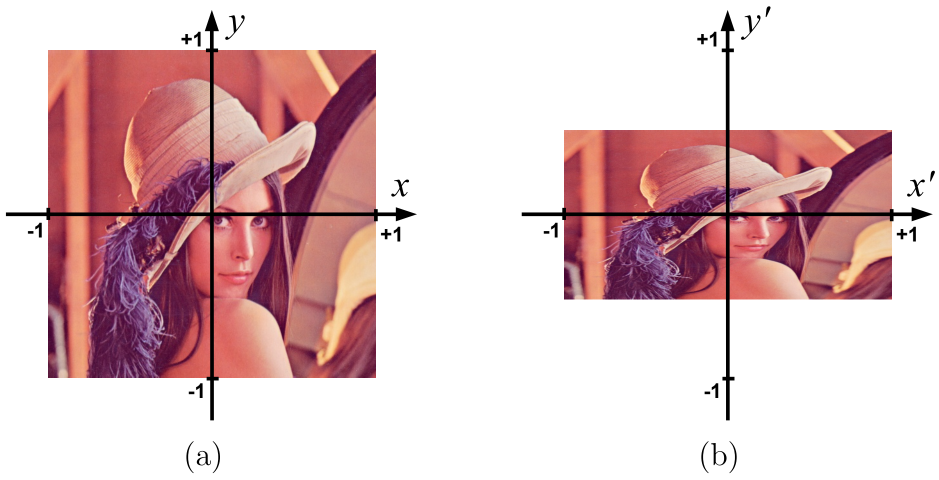


Figure 1.1: (a) Original image with pixel coordinates x and y between -1 and $+1$ and (b) scaled image with pixel coordinates x' and y'.

1.1 Scaling

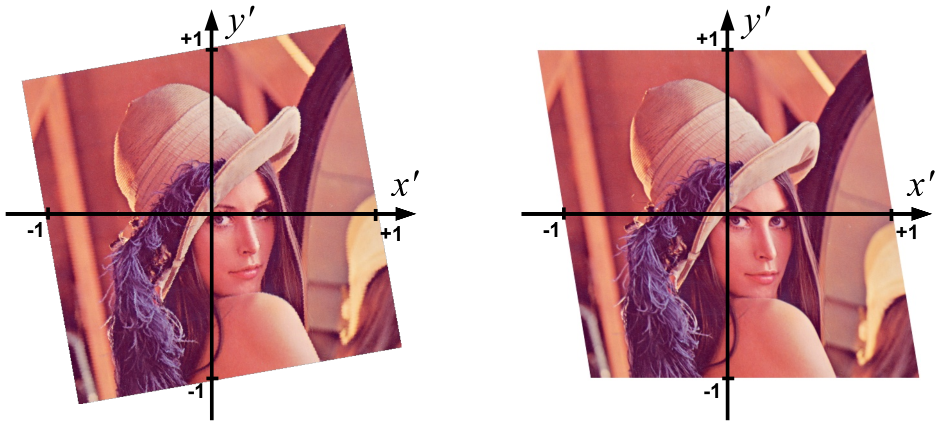
For a (linear) scaling transformation with scaling factor s_x in x direction and scaling factor s_y in y direction, the equations are:

$$x' = xs_x \tag{1.1}$$
$$y' = ys_y \tag{1.2}$$

If the coordinates of each pixel are transformed this way and the pixels are displayed at their new positions (x', y'), the resulting image is a scaled version of the original; see Figure 1.1b for an example. We can also write the equations of the transformation in vector notation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.3}$$

The matrix is called a "transformation matrix." Often only the matrix is used to specify the transformation, which allows for a very compact specification. Note that negative scaling factors result in scaled reflections of the image at the coordinate axes.



(a) (b)

Figure 1.2: (a) Rotated image and (b) sheared image.

1.2 Rotation

A counter-clockwise rotation by an angle θ is described by this transformation:

$$x' = x\cos\theta - y\sin\theta \tag{1.4}$$
$$y' = x\sin\theta + y\cos\theta \tag{1.5}$$

An example with a horizontal shift is depicted in Figure 1.2a. In vector notation, the equation becomes:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.6}$$

1.3 Shearing

Shearing is a kind of shift of the image — horizontally by b_x or vertically by b_y; an example is depicted in the Figure 1.2b. The equations are:

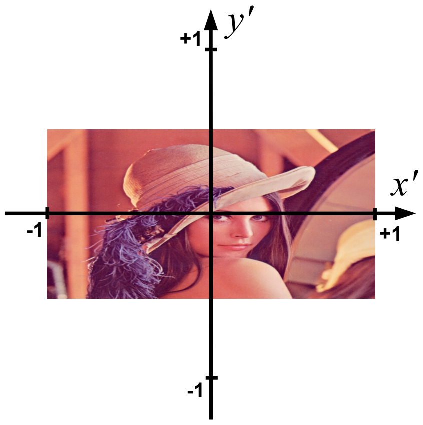
$$x' = x + yb_x \tag{1.7}$$
$$y' = xb_y + y \tag{1.8}$$

And in vector notation:

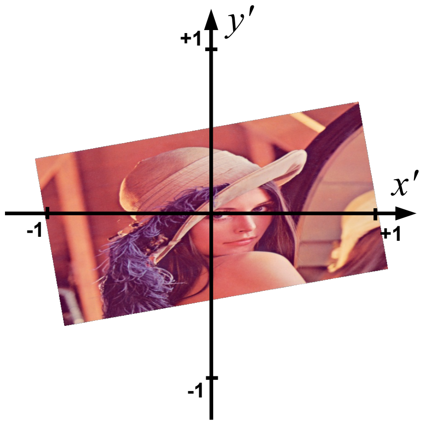
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & b_x \\ b_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.9}$$

1.4 Combination of Transformations

One powerful insight is that the basic linear transformations (i.e. scaling, rotation, and shearing) can be combined by applying a transformation to an already transformed image. Thus, any number of linear transformations can be combined (or concatenated) by applying each transformation to the image that has been transformed with the previous transformations. It turns out that the combination of any two linear transformations is again a linear transformation. Furthermore, the transformation matrix of a combined linear transformation is just the matrix product of the transformation matrices corresponding to the linear transformations that are combined. Note, however, that these matrix products have to be read from right to left: the right-most matrix is applied first (since vectors are multiplied from the right) and the left-most matrix is applied last. This order is important



(a)



(b)

Figure 1.3: (a) Scaled image and (b) scaled and then rotated image.

because matrix products are not commutative (i.e., you are not allowed to change the order of factors in a matrix product).

For an example, consider Figure 1.3. We first scale the image and then rotate it. The vector equation would be:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x\cos\theta & -s_y\sin\theta \\ s_x\sin\theta & s_y\cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.10}$$

Note that the matrix product has to be read from right to left: the order here is first the scaling and then the rotation.

1.5 Geometric Meaning of the Column Vectors of a Transformation Matrix

Before we move on to more general transformations, consider the transformation of the "canonical basis vectors" $(1,0)$ and $(0,1)$ by an arbitrary linear transformation, which is specified by a 2×2 transformation matrix:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{1,1} \\ a_{2,1} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{2,1} \\ a_{2,2} \end{bmatrix} \tag{1.11}$$

These equations mean that the vector $(1,0)$ is transformed to the first column vector $(a_{1,1}, a_{2,1})$ of the transformation matrix; i.e., the first column vector is the "image" (in the mathematical sense) of the first canoncial basis vector $(1, 0)$. Similarly, the vector $(0, 1)$ is transformed to the second column vector $(a_{1,2}, a_{2,2})$. Since the transformation matrix specifies the linear transformation completely, this observation has important consequences:

1. To understand what any linear transformation does, you only need to look at the column vectors of the transformation matrix, which specify how the canoncial basis vectors are transformed. This describes the behavior of the transformation completely.

2. To create a transformation matrix for a specific purpose, all you need to know is how the canonical basis vectors should be transformed. Once you know where the canoncial basis vectors are transformed to (i.e., what their "images" are), you can simply put these transformed vectors into the columns of a matrix. The resulting matrix is the correct transformation matrix.

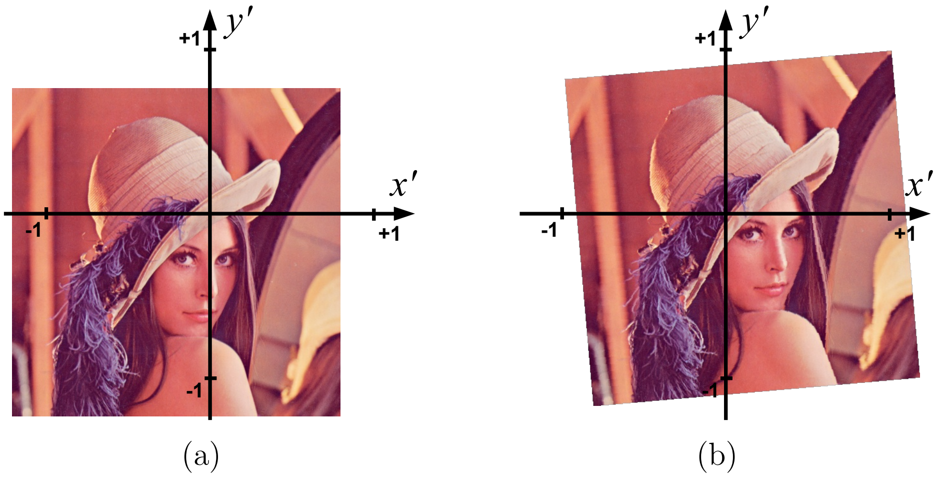


Figure 1.4: (a) Translated image. (b) Image rotated about the upper right corner.

1.6 Translation and Affine Transformations

The last basic transformation that is discussed here is a translation by a horizontal offset Δx and a vertical offset Δy. An example is depicted in Figure 1.4a. The equations are:

$$x' = x + \Delta x \tag{1.12}$$

$$y' = y + \Delta y \tag{1.13}$$

And in vector notation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{1.14}$$

If a linear transformation is combined with a translation, the result is an "affine transformation." If the linear transformation matrix consists of column vectors $(a_{1,1}, a_{2,1})$ and $(a_{1,2}, a_{2,2})$ and the translation is specified by the vector $(\Delta x,\ \Delta y)$, the resulting equations are:

$$x' = a_{1,1}x + a_{1,2}y + \Delta x \tag{1.15}$$
$$y' = a_{2,1}x + a_{2,2}y + \Delta y \tag{1.16}$$

In order to write this as a matrix-vector product, we can use "homogeneous coordinates." These are also used for projective transformations, which will not be discussed here. The main difference is that all vectors are given a third coordinate, which is always 1 for the transformations discussed here:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \Delta x \\ a_{2,1} & a_{2,2} & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.17}$$

As shown in the equation, the top-left 2×2 part of the transformation matrix is the same as the linear transformation matrix, the third column $(\Delta x,\ \Delta y, 1)$ is just the translation vector with 1 as third coordinate, and the third row is always $(0, 0, 1)$ for 2D affine transformations.

One special case is a translation without a linear transformation, which can be represented as a translation combined with an identity linear transformation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.18}$$

1.7 Rotation about an Arbitrary Point

Similarly to linear transformations, affine transformations can be combined to form new affine transformations. Moreover, the 3×3 transformation matrix for the combined affine transformation is just the matrix product of the 3×3 transformation matrices of the affine transformations that are combined. Again, the matrix product has to be read from right to left and the order is important.

One example is the rotation about an arbitrary point, for example, the point $(1,1)$ as in Figure 1.4b. With a linear transformation, we can only describe rotations about the origin $(0,0)$. Affine transformations allow us to specify rotations about any point. Let's assume that we want to rotate the image counter-clockwise by an angle θ about the point (c_x, c_y). To compute the required transformation matrix, we first translate the image such that the point (c_x, c_y) is transformed to the origin $(0,0)$; i.e., we set $\Delta x = -c_x$ and $\Delta y = -c_y$. Then we can rotate with the linear transformation matrix for a rotation about the origin (without translation; thus, the third column is $(0,0,1)$). Finally, we have to translate the origin back to the point (c_x, c_y) by using the inverse translation with $\Delta x = c_x$ and $\Delta y = c_y$. Remember that the matrix product has to be read from right to left; therefore, we get:

$$\begin{aligned}
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta & -\sin\theta & c_x \\ \sin\theta & \cos\theta & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta & -\sin\theta & -c_x\cos\theta + c_y\sin\theta + c_x \\ \sin\theta & \cos\theta & -c_x\sin\theta - c_y\cos\theta + c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (1.19)
\end{aligned}$$

Computing such a matrix product once and transforming all pixels with it is usually much more efficient than applying all individual transformations to all pixels.

1.8 Forward and Backward Mapping

So far, we considered the transformation of each pixel from a position (x, y) to a new position at (x', y') which can be computed with a transformation matrix, say M:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (1.20)$$

This is called forward mapping, and conceptually it is perfectly fine. However, in more than 99% of all cases, this is not how computers actually transform images. Instead, a computer would usually loop over all new positions (x', y') and compute for each position (x', y') the position (x, y) in the original image from where a pixel is mapped to (x', y'). This is called backward mapping. (The reason why computers use backward mapping is that looking up a pixel color at any position (x, y) in the original image is extremely fast, and looping over all output pixels results in processing each of them only once; i.e., it is not less and not more work than absolutely necessary.) Thus, the computer has to compute (x, y) from (x', y'), which is possible with the inverse matrix M^{-1}:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = M^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{1.21}$$

Thus, image transformations on computers usually require the computation of inverse matrices.

Further Reading

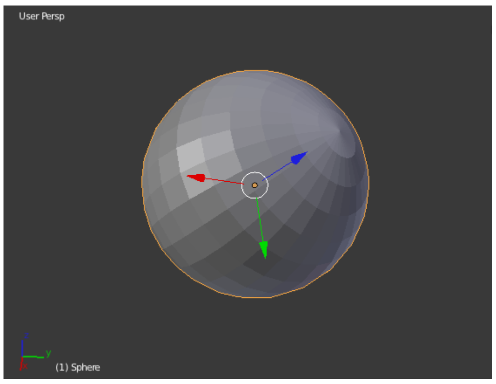
More details and more general transformations are discussed in Chapter 10 of the book T.B. Moeslund, *Introduction to Video and Image Processing*, Springer Verlag London Limited, 2012.

Chapter 2

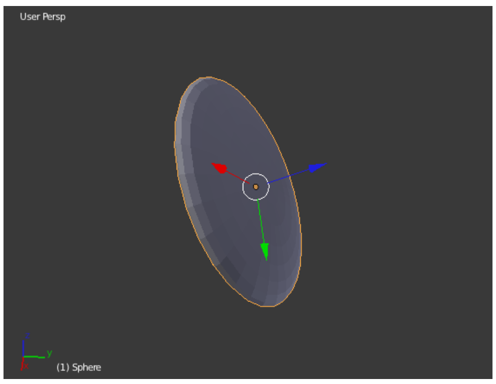
Geometric Transformations in 3D Computer Graphics

Keywords: Computer graphics, affine transformation, modeling transformation.

In 3D computer graphics — in particular in real-time 3D computer graphics — most surfaces are represented by meshes. The vertices of each mesh (i.e. the corners of its polygons) are usually defined in a coordinate system that is specific to each mesh; i.e., each mesh has its own local coordinate system. These coordinates are also called "object coordinates" or "model coordinates." Here, we will denote the coordinates of a 3D point in object coordinates by (x_o, y_o, z_o). Figure 2.1a shows a screen shot of a spherical mesh in the free modeling tool Blender. The large colored axes represent the local coordinate system of the spherical object. (By convention, the x axis is red, the y axis is green, and the z axis is blue.)



(a)



(b)

Figure 2.1: (a) A spherical mesh in Blender with colored axes representing the local object coordinate system (large arrows) and the world coordinate system (small arrows). (b) A spherical mesh that is scaled and rotated.

2.1 General Form of the Modeling Transformation

The vertices of each mesh are transformed from the local coordinate system to a "world coordinate system" by an affine transformation, which can include scaling, rotation, shearing, translation, etc. This transformation is often called the "modeling transformation." While there are many local object coordinate systems (one for each object), there is usually only one common world coordinate system in a scene. Each object usually has its own specific modeling transformation, which allows for an individual transformation of each object from its local object coordinate system to the world coordinate system. Here, we denote the coordinates of a 3D point in world coordinates by (x_w, y_w, z_w). In Figure 2.1a, the directions of the world coordinate axes are represented by the small colored arrows in the lower left corner.

The modeling transformation (from object coordinates (x_o, y_o, z_o) to world coordinates (x_w, y_w, z_w)) usually consists of a linear transformation (for scalings, rotations, shearings, etc.), which could be represented by a 3×3 matrix with column vectors $(a_{1,1}, a_{2,1}, a_{3,1})$, $(a_{1,2}, a_{2,2}, a_{3,2})$, and $(a_{1,3}, a_{2,3}, a_{3,3})$, and a translation by some vector, say (t_x, t_y, t_z); i.e., it is an affine transformation. The vector equation is then written as:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{2.1}$$

In order to write this equation as a single matrix-vector product, homogeneous coordinates are usually used. (These allow also for perspective transformations, which are not discussed here.) This means mainly that all points get a fourth coordinate, which is always 1 as long as we only use affine transformations. The general form of an affine modeling transformation is then:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_x \\ a_{2,1} & a_{2,2} & a_{2,3} & t_y \\ a_{3,1} & a_{3,2} & a_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.2}$$

This general transformation matrix has 12 independent parameters, i.e., there are 12 degrees of freedom. However, it is much more common to work with more basic modeling transformations that have only 1 to 4 parameters as discussed below.

While points get 1 as a fourth coordinate, directions get 0 as their fourth coordinate. Thus, the vector $(\hat{x}_o, \hat{y}_o, \hat{z}_0, 0)$ describes a direction in object coordinates. The transformation to world coordinates uses the same transformation matrix as the transformation of points:

$$\begin{bmatrix} \hat{x}_w \\ \hat{y}_w \\ \hat{z}_w \\ 0 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_x \\ a_{2,1} & a_{2,2} & a_{2,3} & t_y \\ a_{3,1} & a_{3,2} & a_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x}_o \\ \hat{y}_o \\ \hat{z}_o \\ 0 \end{bmatrix} \tag{2.3}$$

This is equivalent to:

$$\begin{bmatrix} \hat{x}_w \\ \hat{y}_w \\ \hat{z}_w \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} \hat{x}_o \\ \hat{y}_o \\ \hat{z}_o \end{bmatrix} \tag{2.4}$$

This is the same as the transformation of points, except that the translation is missing. This is in fact correct because directions are not affected by translations. For example, the vector from one point to another point (as computed by the difference of the two points) will always be the same regardless of any simultaneous translation of both points. In this sense, this difference vector is not affected by translations.

Thus, we can correctly transform points and directions with the same 4×4 transformation matrix provided that the fourth coordinate of points is 1 and the fourth coordinate of directions is 0.

2.2 Basic Modeling Transformations

This section discusses some important special cases of modeling transformations and the corresponding modeling matrices. For example, a translation by the vector (t_x, t_y, t_z) is achieved with a model matrix of the following form:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.5}$$

A scaling with a factor s_x in x direction, s_y in y direction, and s_z in z direction, is achieved this way:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.6}$$

An example that includes a scaling along the z axis is presented in Figure 2.1b.

A rotation by an angle θ about the x axis, is described by this equation:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.7}$$

Similarly, a rotation by an angle ψ about the y axis is described by this equation:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.8}$$

And a rotation by an angle φ about the z axis is specified by this equation:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.9}$$

A more general rotation by an angle α about a normalize axis (x, y, z) would compute the vector $(x_w, y_w, z_w, 1)$ by:

$$\begin{bmatrix} (1-\cos\alpha)x\,x+\cos\alpha & (1-\cos\alpha)x\,y-z\sin\alpha & (1-\cos\alpha)z\,x+y\sin\alpha & 0 \\ (1-\cos\alpha)x\,y+z\sin\alpha & (1-\cos\alpha)y\,y+\cos\alpha & (1-\cos\alpha)y\,z-x\sin\alpha & 0 \\ (1-\cos\alpha)z\,x-y\sin\alpha & (1-\cos\alpha)y\,z+x\sin\alpha & (1-\cos\alpha)z\,z+\cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

2.3 Combination of Modeling Transformations

Affine transformations can be combined to form new affine transformations. Correspondingly, the basic modeling transformations can be combined to form more complex modeling transformations. In practice, this is performed by multiplying the corresponding modeling matrices. For example, a scaling operation followed by a translation is described by this equation:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.10}$$

$$= \begin{bmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.11}$$

Note that the scaling matrix appears to the right of the translation matrix because it is applied first and these matrix products have to be read from right (where the vector is multiplied) to the left.

Another very common combination is the multiplication of three rotation matrices about fixed axes to describe any rotation. The three angles are then often called "Euler angles." There are various ways of defining this kind of rotation. In the game engine Unity, the order is: a rotation by φ about the z axis, followed by a rotation by θ about the x axis, followed by a rotation by ψ about the y axis. Again, the matrix product has to be read from right to left; thus, it is:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.12}$$

In computer graphics, the matrix product of the three matrices would be evaluated to compute a new modeling matrix because it is more efficient to transform all vertices with this new modeling matrix than to apply all three rotation matrices to all vertices.

It should be mentioned that combinations of rotation matrices and translation matrices are sometimes referred to as "rigid transformations" because these are the transformations that don't deform the shape of objects; i.e., they respect the rigidity of rigid bodies. Since all translations are described by 3 coordinates (t_x, t_y, t_z) and all rotations are described by 3 Euler angles φ, θ, and ψ, these rigid-body transformations have $3 + 3 = 6$ degrees of freedom.

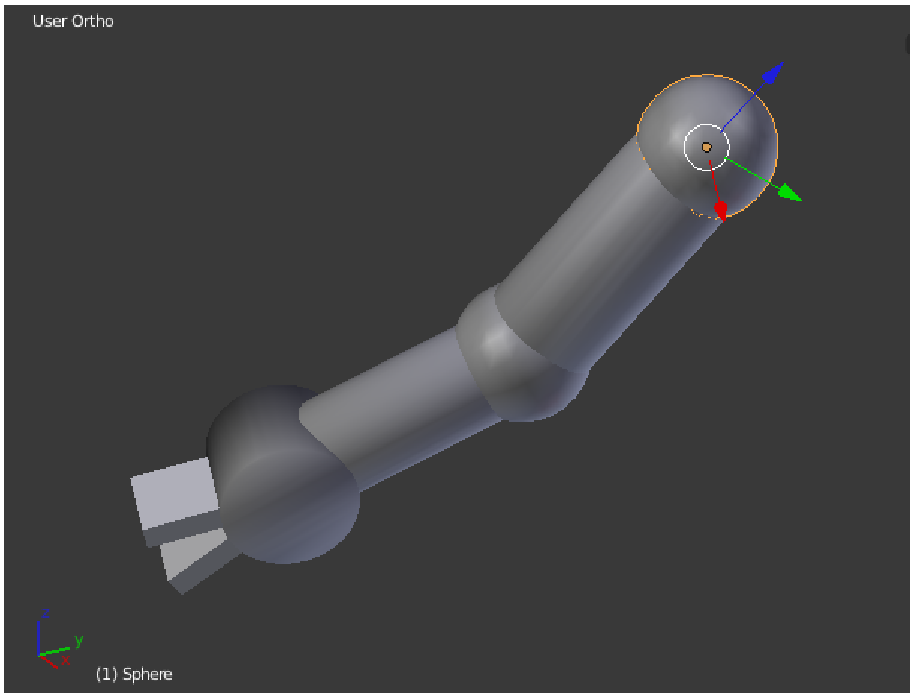


Figure 2.2: A simple model of a robot arm in Blender.

2.4 Object Hierarchies

Many compound objects are organized in an object hierarchy. Consider the robot arm in Figure 2.2. The right-most sphere represents a shoulder joint. Let's denote its transformation matrix by M_{shoulder}. Thus, the vertices of that shoulder sphere are transformed this way:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = M_{\text{shoulder}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.13}$$

The idea of an object hierarchy is that some objects are "children" to other objects. For example, the sphere representing the elbow joint could be a child of the shoulder sphere. From the point of view of transformations, this means that the elbow sphere should follow the transformation of its parent; i.e., if the shoulder is moved, the elbow should follow; if the shoulder is rotated, the shoulder should move accordingly. Technically, this is achieved by specifying a local transformation matrix M_{elbow} for the elbow sphere which is multiplied with the transformation matrix of the shoulder

M_{shoulder} to compute the actual modeling matrix for the vertices of the elbow sphere:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = M_{\text{shoulder}} M_{\text{elbow}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.14}$$

In this way, the vertices of the elbow sphere are affected by both, the transformation matrix of the shoulder joint and the local transformation of the elbow joint.

As we go along the arm, we go deeper into the hierarchy: the cylinder representing the wrist joint should be affected by a local transformation M_{wrist} relative to the elbow joint but also by the transformation matrices M_{elbow} and M_{shoulder}:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = M_{\text{shoulder}} M_{\text{elbow}} M_{\text{wrist}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{2.15}$$

In general, the local transformation matrix of each object is multiplied with all the transformation matrices of its parent, grandparent, great-grandparent, etc. Note that the order of the multiplication is important: when reading from right (where vertices are multiplied) to left, we have to start with the "deepest" transformation matrix, which is also "closest" to the vertices.

In summary, matrix multiplication is not only a mathematical tool for combining basic modeling transformations such as rotations and translations but it is also the technical foundation of arbitrarily complex hierarchies of meshes.

Further Reading

A detailed introduction into affine transformations in 3D computer graphics is provided in Chapter 4 of James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications — A Programmer's Guide*, 2nd ed., Morgan Kaufmann Publishers, 2008.

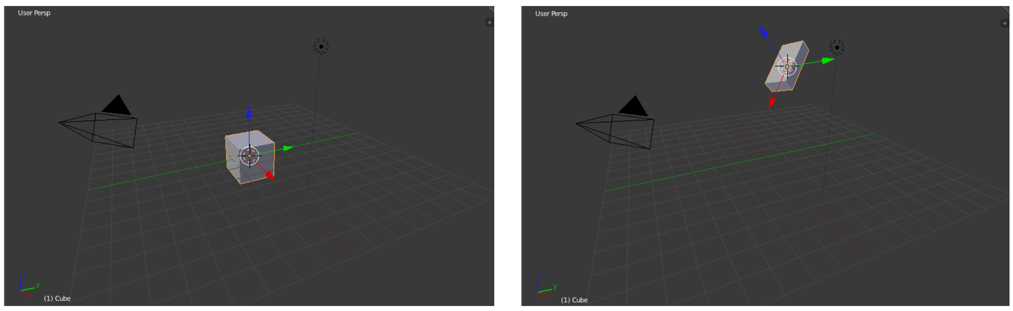
The screen shots were produced with the "Blender 3d content creation suite" by the Blender Foundation, see `http://www.blender.org`.

Chapter 3

Active and Passive Transformations in Computer Graphics

Keywords: Computer graphics, active transformation, passive transformation, change of basis.

Transforming points and directions is an important part of 3D computer graphics. In fact, the whole process of rendering a polygonal mesh onto a screen is often described as a sequence of coordinate transformations; for example, from object coordinates to world coordinates etc. The transformations are described by 4×4 matrices, points are described by 4D vectors with 1 in the fourth coordinate, and directions are described by 4D vectors with 0 in the fourth coordinate. So far, so good.



(a) (b)

Figure 3.1: (a) The startup scene of Blender: a cube centered at the origin in world coordinates. (b) The same cube but scaled, rotated, and translated.

Unfortunately, there are two basic interpretations of transformations: the “active interpretation” and the “passive interpretation.” In the active interpretation, points are transformed from one position to another position in the same coordinate system. In the passive interpretation, points stay in the same position but their coordinate representation is transformed from one coordinate system to another coordinate system (i.e. a “change of basis” is performed). How could two such different operations be called two interpretations of the same transformation? Simply because the equation of the two transformations (and therefore the implementation in a computer program) is exactly the same for both interpretations. In practice, however, it is often easier to think in terms of one of the two interpretations. In particular, when reading a text (or source code) that uses a certain interpretation, it is a lot easier to think in terms of the same interpretation. Thus, since both interpretations are commonly used, it is very useful to understand both of them.

To this end, we will look at an example where both interpretations make sense in order to see what exactly the difference is.

3.1 Active Interpretation

Consider the startup scene of the modeling tool Blender in Figure 3.1a: a cubic mesh is centered at the origin of the world coordinate system. In Figure 3.1b, however, the same mesh is scaled, rotated, and translated by a 4×4 transformation matrix, say M.

In the *active interpretation*, we can think of placing the vertices of the cubic mesh into world coordinates in Figure 3.1a. Then the matrix M actively transforms each vertex position $(x_w, y_w, z_w, 1)$ to a new position $(x'_w, y'_w, z'_w, 1)$ as shown in Figure 3.1b. Both positions are specified in world coordinates. The large arrows in Figure 3.1b just show how the points $(1, 0, 0, 1)$, $(0, 1, 0, 1)$, and $(0, 0, 1, 1)$ would be transformed (if there was no scaling). We can write the transformation as:

$$\begin{bmatrix} x'_w \\ y'_w \\ z'_w \\ 1 \end{bmatrix} = M \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.1}$$

I.e. the transformation is just a matrix-vector multiplication with M.

3.2 Passive Interpretation

The *passive interpretation* is quite different: We only look at Figure 3.1b where the large arrows represent a local object coordinate system (apart from the scaling). The vertices of the mesh are specified in this coordinate system; for example, a vertex $(0, 0, 0, 1)$ in this coordinate system would appear at the intersection of the three large arrows in Figure 3.1b, i.e. at the origin of the local coordinate system. The transformation matrix M doesn't change those positions at all, it just allows us to compute the coordinate representation of the passive vertices with respect to another coordinate system, specifically the world coordinate system (which is represented by the three large arrows in Figure 3.1a). Thus, given a vertex in object coordinates $(x_o, y_o, z_o, 1)$, we can transform this coordinate representation to world coordinates $(x_w, y_w, z_w, 1)$ with the transformation matrix M:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = M \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \tag{3.2}$$

Again, this is just a matrix-vector multiplication with M. The only differences to the previous equation are the names of the variables, which were chosen to clarify the interpretation. If numbers are supplied, there is no difference at all: the exact same computations are performed in both cases.

3.3 Real-World Usage

Which of the two interpretations is preferable? Unfortunately, there is no clear answer to this question. In the case of "modeling transformations" (from object coordinates to world coordinates) both interpretations are commonly used. In fact, even the source code of one and the same application often uses an active interpretation in one part and a passive interpretation of the very same modeling transformation in another part. Thus, the choice of the interpretation doesn't only depend on the specific transformation and the personal preferences of a programmer or author but also on the specific context in which the transformation is used.

For many other transformations (in particular the viewing transformation, the projection transformation, and the viewport transformation), the passive interpretation is usually preferred when applying the transformation matrix; however, the active interpretation is often preferred when computing the transformation matrix. Thus, readers (and authors) of source code and/or publications in computer graphics not only have to understand both interpretations but also have to be able to switch from one interpretation to the other.

Further Reading

There is a discussion of "thinking about transformations" in Chapter 3 of Dave Shreiner, *OpenGL Programming Guide: The Official Guide to Learn-*

ing OpenGL®, Versions 3.0 and 3.1, 7th ed., Addison-Wesley, 2010. The section “Grand, Fixed Coordinate System” describes the active interpretation while section “Moving a Local Coordinate System” describes the passive interpretation of combined modeling transformations.

The Wikipedia page “Active and Passive Transformation” (as of October 25, 2012) discusses a 2D rotation as an active and as a passive transformation. In the case of the passive transformation, the Wikipedia article transforms the coordinate representation of a point from a fixed coordinate system to the transformed coordinate system while the example in Figure 3.1 transformed the coordinate representation of points from the transformed coordinate system to a fixed coordinate system. That is why the Wikipedia article requires the inverse transformation in the case of the passive transformation.

The screen shots were produced with the “Blender 3d content creation suite” by the Blender Foundation, see `http://www.blender.org`.

Chapter 4

Transformations of Special Vectors in Computer Graphics

Keywords: Computer graphics, transformation, normal.

Transforming points and directions is an important part of 3D computer graphics. In fact, the whole process of rendering a polygonal mesh onto a screen is often described as a sequence of coordinate transformations. Most transformations are affine transformations; i.e., they consist of a linear transformation, which can be represented by a 3×3 matrix, say A, and a translation by a vector, say $\mathbf{v}$. A 3D point $\mathbf{p}$ is then transformed this way:

$$\mathbf{p}' = A\mathbf{p} + \mathbf{v} \tag{4.1}$$

This transformation can be used, for example, to describe the positions and orientations of rigid bodies such as depicted in Figure 4.1. Apart from points, there are, however, additional kinds of vectors that have to be transformed, e.g. the surface normal vectors, which are required for the correct shading of surfaces.

4.1 Directions

Directions (or “vectors” that aren’t points) are usually not affected by translations. Consider the vector $\mathbf{d}$ from point $\mathbf{p}$ to point $\mathbf{q}$ with $\mathbf{d} = \mathbf{q} - \mathbf{p}$. If $\mathbf{p}$ is transformed to $\mathbf{p}' = A\mathbf{p} + \mathbf{v}$ and $\mathbf{q}$ is transformed to $\mathbf{q}' = A\mathbf{q} + \mathbf{v}$ then the difference $\mathbf{d}'$ of the transformed vectors is:

$$\mathbf{d}' = \mathbf{q}' - \mathbf{p}' = (A\mathbf{q} + \mathbf{v}) - (A\mathbf{p} + \mathbf{v}) = A(\mathbf{q} - \mathbf{p}) = A\mathbf{d} \tag{4.2}$$

Thus, the difference $\mathbf{d}$ between two points is only affected by the linear transformation but not by the translational part of an affine transformation. Similarly, all vectors that represent directions (e.g. light directions, camera directions, velocities, accelerations, forces, etc.) are not affected by translations.

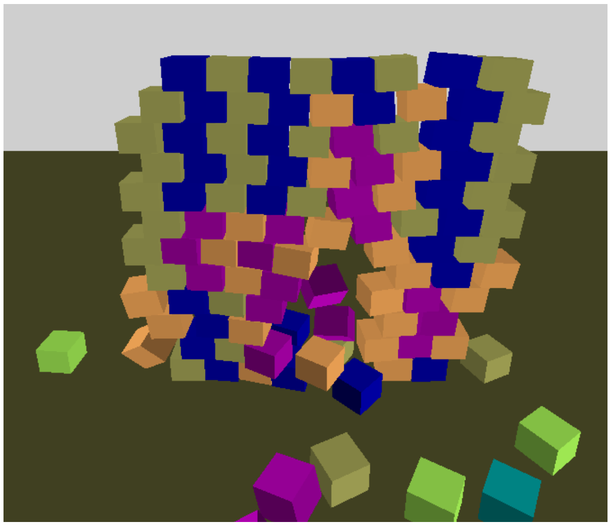


Figure 4.1: A rather simple rigid-body simulation. Image source: http://commons.wikimedia.org/wiki/ File:Bullet_Wall.png .

4.2 Cross Products

There are further vectors that behave differently. For example, a vector **c** that is always defined as the cross product of two directions **a** and **b** will usually not transform like a regular direction since in general we have:

$$\mathbf{c}' = \mathbf{a}' \times \mathbf{b}' = A\mathbf{a} \times A\mathbf{b} \neq A(\mathbf{a} \times \mathbf{b}) = A\mathbf{c} \tag{4.3}$$

In order to compute the transformed vector $\mathbf{c}'$, it is therefore usually best to transform the vectors **a** and **b** and compute the cross product of the transformed vectors $\mathbf{a}'$ and $\mathbf{b}'$.

4.3 Surface Normal Vectors

Surface normal vectors are another class of vectors that transform differently. A surface normal vector **n** at a point on a surface is defined to be always orthogonal (or perpendicular) to the surface at that point, which also means that it is orthogonal to any tangential direction **t** at that point; see Figure 4.2a. However, a linear transformation M (in particular a shear transformation) doesn't in general preserve the angles between any two vectors (e.g. **a** and **b** in Figure 4.2b); thus, two vectors that are orthogonal might become non-orthogonal vectors if they are linearly transformed like directions. Thus, surface normal vectors cannot be transformed like directions since they have to be orthogonal to tangential vectors, which are transformed like directions.

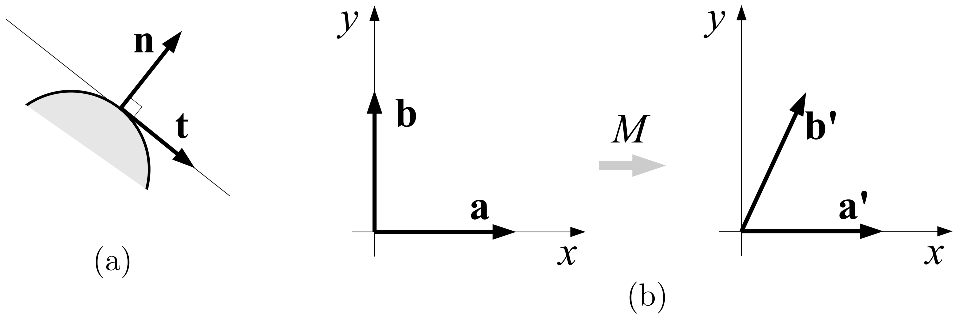


Figure 4.2: (a) Surface normal vector $\mathbf{n}$ at a point of a surface (here: a curve) orthogonal to tangent vector $\mathbf{t}$. (b) After a linear shear transformation M, the vectors $\mathbf{a}$ and $\mathbf{b}$ are no longer orthogonal.

Thus, the question is: which transformation of surface normal vectors would guarantee that the transformed normal vector is orthogonal to a transformed tangent vector?

In order to answer the question, we first make the assumption that for every "reasonable" transformation matrix M that is used to transform a tangent vector $\mathbf{t}$, there exists a linear transformation matrix Q such that the transformed normal vector $\mathbf{n}' = Q\mathbf{n}$ is orthogonal to the transformed tangent vector $\mathbf{t}' = M\mathbf{t}$ iff (i.e. if and only if) $\mathbf{n}$ is orthogonal to $\mathbf{t}$. Two vectors (unequal to $\mathbf{0}$) are orthgonal iff their dot product is equal to 0. Thus, we can write a stronger requirement this way:

$$\mathbf{n}' \cdot \mathbf{t}' \overset{!}{=} \mathbf{n} \cdot \mathbf{t} \tag{4.4}$$

If $\mathbf{n}'$ and $\mathbf{t}'$ are orthogonal then the left-hand-side is 0; thus, the right-hand-side has to be 0, and, therefore, $\mathbf{n}$ and $\mathbf{t}$ have to be orthogonal. Analogously for the case that they are not orthogonal.

From this equation it is straightforward (but slightly cumbersome) to compute Q:

$$\begin{align} \mathbf{n}' \cdot \mathbf{t}' &= \mathbf{n}'^T \mathbf{t}' \tag{4.5}\\ &= (Q\mathbf{n})^T (M\mathbf{t}) \tag{4.6}\\ &= \mathbf{n}^T Q^T M \mathbf{t} \tag{4.7}\\ &\overset{!}{=} \mathbf{n}^T \mathbf{t} = \mathbf{n} \cdot \mathbf{t} \tag{4.8} \end{align}$$

We can easily fulfill the requirement $\mathbf{n}^T Q^T M \mathbf{t} \overset{!}{=} \mathbf{n}^T \mathbf{t}$ with this requirement:

$$Q^T M \overset{!}{=} \text{Id} \qquad \text{(Id is the identity matrix)} \tag{4.9}$$

$$Q^T \overset{!}{=} M^{-1} \tag{4.10}$$

$$Q \overset{!}{=} (M^{-1})^T \tag{4.11}$$

Thus, if $(M^{-1})^T$ exists, i.e. if M is invertible (which is the "reasonable" requirement mentioned above) then surface normal vectors are transformed with $Q = (M^{-1})^T$; i.e., the transformed surface normal vector $\mathbf{n}'$ is:

$$\mathbf{n}' = (M^{-1})^T \mathbf{n} \tag{4.12}$$

In other words, surface normal vectors are transformed with the transpose of the inverse of the regular transformation matrix.

There is one special case: if M is orthogonal, i.e. $M^{-1} = M^T$, then $Q = (M^{-1})^T = (M^T)^T = M$. Thus, in the special case of an orthgonal transformation matrix M, surface normal vectors are transformed with M like regular direction vectors.

Further Reading

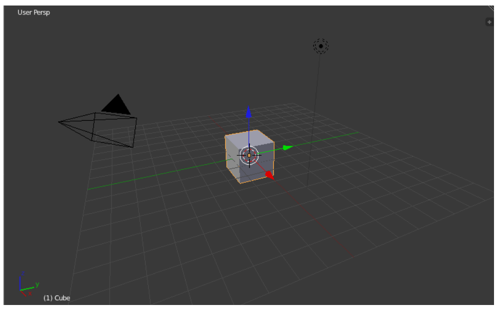
The transformation of surface normal vectors is also discussed in Section 4.3.7 of James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications — A Programmer's Guide*, 2nd ed., Morgan Kaufmann Publishers, 2008.

Chapter 5

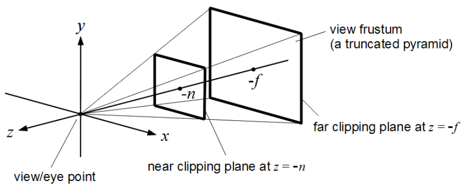
Viewing Transformation in Computer Graphics

Keywords: Computer graphics, viewing transformation, view transformation, camera.

In computer graphics, a virtual camera is usually employed to produce an image of a virtual 3D scene; see Figure 5.1a. Like a real camera, the virtual camera has a position and an orientation, as well as other properties that are not taken into account here, e.g. a field of view, an aspect ratio, a focal length, etc. Different rendering techniques use the position and orientation of a virtual camera in different ways; here, we will look at the standard procedure for perspective cameras in the real-time rendering library OpenGL. (Other real-time rendering libraries, in particular Direct3D, work similarly.)



(a)


y
z
x
view/eye point
-n
-f
view frustum
(a truncated pyramid)
far clipping plane at $z = -f$
near clipping plane at $z = -n$

(b)

Figure 5.1: (a) The startup scene of Blender with a representation of the camera's view frustum in black on the left-hand side. The view frustum is the volume that is visible to the camera. The frustum's top is at the camera position; the frustum points in the view direction; and the "up" direction is indicated by the solid black triangle. (b) The view frustum of a camera in view/eye coordinates.

5.1 View Coordinates

In OpenGL, vectors are transformed from the common world coordinate system to "view coordinates" (also known as "eye coordinates"). While world coordinates don't depend on the virtual camera, the view coordinate system is defined by the position and orientation of the virtual camera. Specifically, the origin of the view coordinate system is placed at the position of the camera (called $\mathbf{t}$ in the following); the negative z axis of the view coordinate system points to the view direction (called $\mathbf{d}$ below); and the positive y axis points "up" (as close to a world-up direction $\mathbf{k}$ as possible). See Figure 5.1b for an illustration of the view frustum in view coordinates.

When rendering a scene, all points are (passively) transformed from world coordinates to view coordinates. This transformation depends on the virtual camera. Once in view coordinates, however, there is no more dependency on the position and orientation of the camera, and the rendering can proceed in the same way for all positions and orientations of the camera.

5.2 Computing the Viewing Transformation

In order to apply a viewing transformation, we have to compute the corresponding matrix. Here, we will denote the viewing transformation matrix from world coordinates to view coordinates as $M_{\text{world}\to\text{view}}$ and the inverse matrix from view coordinates to world coordinates as $M_{\text{view}\to\text{world}}$.

Conceptually, it is easier to first compute the transformation matrix from view coordinates to world coordinates $M_{\text{view}\to\text{world}}$ in an active interpretation. The origin in view coordinates $[0\ 0\ 0\ 1]^T$ (with a 4th coordinates of 1 because it is a point) has to be translated to the position of the camera in world coordinates, say $\mathbf{t} = [t_1\ t_2\ t_3\ 0]^T$. This will become the translation part of $M_{\text{view}\to\text{world}}$.

For the linear part of $M_{\text{view}\to\text{world}}$, we first consider the positive(!) z axis of the view coordinate system (more precisely, the vector $[0\ 0\ 1\ 0]^T$), which has to be mapped to the negative(!) view direction. Let's assume the view direction is given in world coordinates by an unnormalized vector $\mathbf{d}$, then the "image" $\mathbf{z}$ (in the mathematical sense) of the positive z axis (i.e. of the vector $[0\ 0\ 1\ 0]^T$) is:

$$\mathbf{z} = -\frac{\mathbf{d}}{\|\mathbf{d}\|} \tag{5.1}$$

Second, the image $\mathbf{x}$ (in world coordinates) of the positive x axis of the view coordinate system (i.e. of the vector $[1\ 0\ 0\ 0]^T$) has to be orthogonal to $\mathbf{z}$

and to a world-up direction $\mathbf{k}$ (also in world coordinates) because $\mathbf{x}$ should point to the right (not up). These requirements can be easily fulfilled with a normalized cross product:

$$\mathbf{x} = \frac{\mathbf{d} \times \mathbf{k}}{\|\mathbf{d} \times \mathbf{k}\|} \tag{5.2}$$

Thirdly, the image $\mathbf{y}$ of the y axis of the view coordinate system (i.e. of the vector $[0\ 1\ 0\ 0]^T$) has to be orthogonal to $\mathbf{z}$ and $\mathbf{x}$. Since the latter two vectors are already normalized and orthogonal, a simple cross product is sufficient to compute $\mathbf{y}$:

$$\mathbf{y} = \mathbf{z} \times \mathbf{x} \tag{5.3}$$

From the three images $\mathbf{x} = [x_1\ x_2\ x_3\ 0]^T$, $\mathbf{y} = [y_1\ y_2\ y_3\ 0]^T$, and $\mathbf{z} = [z_1\ z_2\ z_3\ 0]^T$ (in world coordinates) of the vectors $[1\ 0\ 0\ 0]^T$, $[0\ 1\ 0\ 0]^T$, and $[0\ 0\ 1\ 0]^T$ (in view coordinates), and the translation vector $\mathbf{t} = [t_1\ t_2\ t_3\ 0]^T$ (also in world coordinates), the matrix $M_{\text{view}\to\text{world}}$ for the transformation from view coordinates to world coordinates can be easily constructed by using the vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{t}$ as column vectors:

$$M_{\text{view}\to\text{world}} = \begin{bmatrix} x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ x_3 & y_3 & z_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

Since the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are normalized and orthogonal to each other, the upper-left 3×3 part, say R, of this matrix is orthogonal, i.e. $R^{-1} = R^T$. The matrix $M_{\text{view}\to\text{world}}$ can then be written this way:

$$M_{\text{view}\to\text{world}} = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \quad \text{with} \quad R = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} \tag{5.5}$$

It turns out that the inverse of a matrix of this form is given by:

$$M^{-1}_{\text{view}\to\text{world}} = \begin{bmatrix} R^{-1} & -R^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{5.6}$$

Since R is orthogonal; i.e., $R^{-1} = R^T$, this is just:

$$M^{-1}_{\text{view}\to\text{world}} = \begin{bmatrix} R^T & -R^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{5.7}$$

Thus, $M^{-1}_{\text{view}\to\text{world}}$ is easy to compute using $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{t}$. Furthermore, the viewing transformation matrix $M_{\text{world}\to\text{view}}$ is equal to $M^{-1}_{\text{view}\to\text{world}}$:

$$M_{\text{world}\to\text{view}} = M^{-1}_{\text{view}\to\text{world}} = \begin{bmatrix} R^T & -R^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{5.8}$$

Thus, this matrix is the correct viewing transformation matrix to transform points from world coordinates to view coordinates.

Further Reading

The computation is also described in Section 6.2 of James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications — A Programmer's Guide*, 2nd ed., Morgan Kaufmann Publishers, 2008.

The presented algorithm assumes that the view direction always points to the center of the view plane. In the more general case of an arbitrary position of the view point and the view plane, the computation is somewhat more difficult. In particular, the "view direction" **d** is then just the direction that is orthogonal to the view plane (regardless of the direction into which the camera is pointing); see Robert Kooima's publication "Generalized Perspective Projection" (available at: `http://csc.lsu.edu/~kooima/pdfs/gen-perspective.pdf`) for details.

The screen shot was produced with the "Blender 3d content creation suite" by the Blender Foundation, see `http://www.blender.org`.

Chapter 6

Encoding High Dynamic Range Images

Keywords: Dynamic range, image processing, color transformation, logarithm, exponential.

The human visual system can adjust itself to environments of a very large range of light intensities. Traditionally, cameras and displays were able to capture and reproduce only a fraction of this intensity range at a time; thus, image formats were designed for these images of limited intensity range. High Dynamic Range (HDR) images, however, require a larger range of intensities; thus, traditional image formats are not sufficient and new HDR image formats had to be defined. One of them is the "HDR" format (a.k.a. "Radiance" picture format). One of the encodings of the HDR format is the RGBE encoding, which is discussed here as an example for an application of exponential and logarithmic functions.

6.1 RGBE Encoding

This encoding uses 4 bytes (or 32 bits) for each pixel: one byte for the red component (R), one for green (G), one for blue (B), and one byte for a common exponential (E).

Given an original RGB color with components R_{org}, G_{org}, and B_{org} between approximately 10^{-38} and 10^{+38}, the encoded bytes R, G, B, and E between 0 and 255 are computed this way:

$$E = \text{ceiling}\left(\log_2\left(\max\left(R_{\text{org}}, G_{\text{org}}, B_{\text{org}}\right)\right) + 128\right) \quad (6.1)$$

$$R = \text{floor}\left(\frac{256 R_{\text{org}}}{2^E - 128}\right) \quad (6.2)$$

$$G = \text{floor}\left(\frac{256 G_{\text{org}}}{2^E - 128}\right) \quad (6.3)$$

$$B = \text{floor}\left(\frac{256 B_{\text{org}}}{2^E - 128}\right) \quad (6.4)$$

where floor(x) is the largest integer that is smaller than or equal to x, and ceiling(x) is the smallest integer greater than or equal to x.

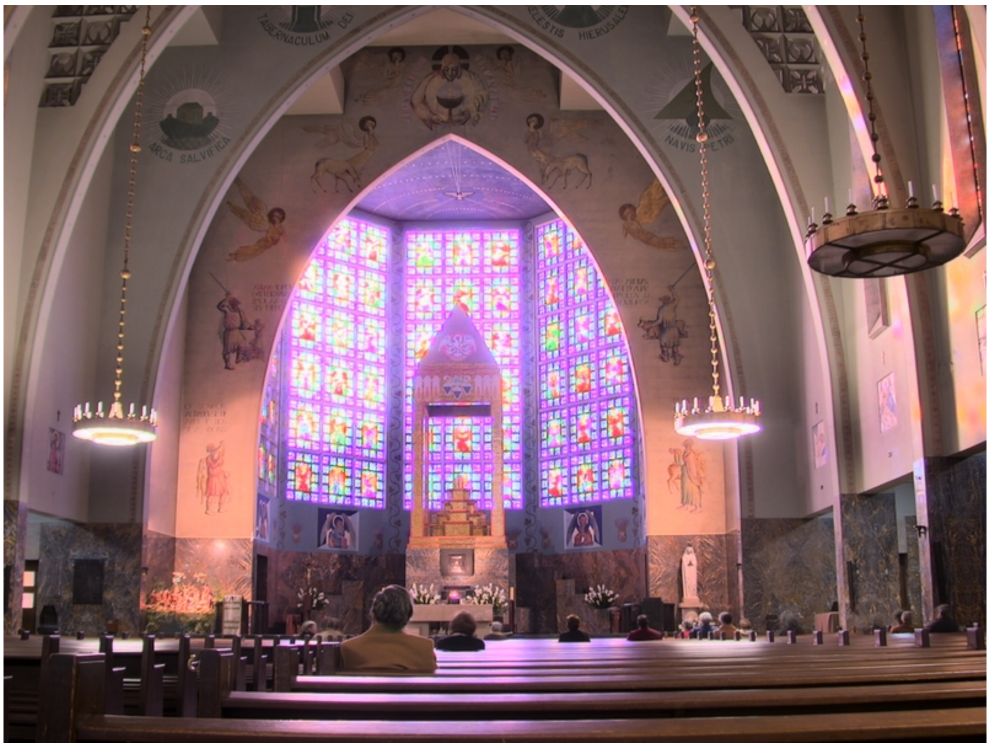


Figure 6.1: Example of an HDR image, which was rendered with a particular algorithm (i.e. some form of tone mapping) to show a large range of color intensities in the same image. Image source: http://commons.wikimedia.org/wiki/File:Igreja_Nossa_Senhora_de_Fatima_(Tone_Mapped).jpg

The computation of E uses the $\log_2$ function, which changes very slowly for large values x. This makes it possible to encode a large range of values in just one byte.

6.2 Decoding RGBE

Given the 4 bytes R, G, B, and E between 0 and 255, an approximation to the original color components R_{org}, G_{org}, and B_{org} between approximately

10^{-38} and 10^{+38} can be decoded this way:

$$R_{\text{org}} \approx \frac{R + 0.5}{256} 2^{E-128} \tag{6.5}$$

$$G_{\text{org}} \approx \frac{G + 0.5}{256} 2^{E-128} \tag{6.6}$$

$$B_{\text{org}} \approx \frac{B + 0.5}{256} 2^{E-128} \tag{6.7}$$

Here, the exponential function changes rapidly for positive arguments, which makes it possible to generate a large range of values even though E has only 256 potential values.

6.3 Example

In order to illustrate the RGBE encoding with an actual example, Figure 6.2 shows how the (non-HDR) image in Figure 6.2a would be encoded in R, G, and B components, which are depicted in Figure 6.2b, and E, which is shown as grayscale image in Figure 6.2c. E is always rather close to 128

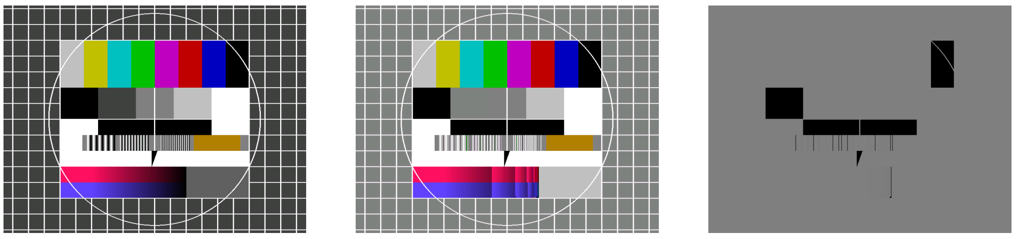

(a) (b) (c)

Figure 6.2: (a) Original image (http://commons.wikimedia.org/wiki/File:FuBK-Testbild.png). (b) Encoded R, G, and B components. (c) Encoded E component as grayscale image.

(i.e. gray) except for pure black. Note in particular the color gradients in the lower, left part of the image: almost all of the colors are encoded with only 7 (out of 256) values of E.

Further Reading

The book "High Dynamic Range Imaging" (2nd ed., Morgan Kaufmann, 2010) by Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski discusses many aspects of HDR imaging, including the HDR format in Section 3.3.1.

Chapter 7

Motion of a Particle under Forces

Keywords: Simulation, physics, mechanics, calculus, velocity, acceleration, force.

The computer simulation of the motion of objects under forces has numerous applications: from highly accurate simulations in ballistics to the inaccurate motion of balls, projectiles, or angry birds in computer games. Even some elements of graphical user interfaces move based on a simulation of spring forces. Here, we consider the basic case of a point-like particle with no volume but some mass that moves under a known force, for example, gravity. Other forces, e.g. collision forces, could also be integrated provided that they can be computed. Point-like particles are often a useful approximation for projectiles, e.g. in computer games.

Figure 7.1: Example of a "projectile." If the rotation of the ball is ignored, its motion can be approximately computed with the model of a point-like particle. Collision detection, however, should still take the volume of the ball into account. Source: http://commons.wikimedia.org/wiki/File:2011-06-07_Basketball_in_hoop_still_shot.jpg

7.1 Equation of Motion and Force

The basic equation for the computation of some motion is usually called the "equation of motion." In the case of a particle of mass m (usually in kilogram: kg) under force $\mathbf{F}$ (usually in Newton: $1\text{N} = 1\text{kg}\,\text{m}\,\text{s}^{-2}$), the equation of motion is:

$$\mathbf{a} = m^{-1}\mathbf{F} \tag{7.1}$$

This equation specifies the acceleration $\mathbf{a}$ (in meter per second squared: $\text{m}\,\text{s}^{-2}$) as a function of mass and force. The equation is also known as Newton's second law, which is usually written in this equivalent form: $\mathbf{F} = m\mathbf{a}$. In three dimensions, it is vector equation, i.e. it is actually a system of three equations:

$$a_x = m^{-1}F_x \tag{7.2}$$
$$a_y = m^{-1}F_y \tag{7.3}$$
$$a_z = m^{-1}F_z \tag{7.4}$$

with $[a_x\ a_y\ a_z]^T = \mathbf{a}$ and $[F_x\ F_y\ F_z]^T = \mathbf{F}$.

The gravity force on the surface of the Earth is a good example. If the y axis is pointing upwards, the gravity force is approximately: $\mathbf{F}_{\text{grav}} = m\left[0\ \ -9.81\frac{\text{m}}{\text{s}^2}\ 0\right]^T$. Note that m (in italics) denotes the mass while m (not in italics) is the symbol for meter. Also note that not all forces are that simple, and in many cases the computation of forces is more costly than the rest of the simulation. Here, however, we won't discuss any other specifc forces.

If multiple forces act on the same particle (e.g. gravity force, aerodynamic drag force, repulsive collision force, electromagnetic Lorentz force, etc.), all the force vectors have to be added and the total sum of the forces is used in the equation of motion.

7.2 Definition of Acceleration and Computation of Velocity

With the equation of motion, we can compute the acceleration $\mathbf{a}$ of a particle of mass m under a (total) force $\mathbf{F}$. The acceleration specifies the change of velocity $\Delta\mathbf{v}$ per time Δt for infinitely small time intervals:

$$\mathbf{a} = \lim_{\Delta t \to 0} \frac{\Delta\mathbf{v}}{\Delta t} \tag{7.5}$$

Again, this is actually a system of three equations since $\mathbf{a}$ and $\Delta\mathbf{v}$ are vectors.

If the velocity $\mathbf{v}(t')$ of the particle is described as a function of time t' then the acceleration $\mathbf{a}(t)$ at t can be written as the time derivative of $\mathbf{v}(t')$ with respect to t' at $t' = t$:

$$\mathbf{a}(t) = \left.\frac{\mathrm{d}\mathbf{v}(t')}{\mathrm{d}t'}\right|_{t'=t} \tag{7.6}$$

As before, this is actually a system of three equations with three time derivatives. The equation is often written in a more compact form:

$$\mathbf{a}(t) = \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} \qquad \text{or even} \qquad \mathbf{a} = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} \tag{7.7}$$

This definition of the acceleration can be used to compute the particle velocity from the acceleration by "integrating" the equation:

$$\mathbf{a}(t) = \left.\frac{\mathrm{d}\mathbf{v}(t')}{\mathrm{d}t'}\right|_{t'=t} \qquad \Rightarrow \qquad \mathbf{v}(t) = \mathbf{v}(t_0) + \int_{t_0}^{t} \mathbf{a}(t')\mathrm{d}t' \tag{7.8}$$

The integral equation is actually three integral equations: one for each coordinate. We will not worry whether these integrals exist since we know that the physical motion exists. Note, however, that an integration constant $\mathbf{v}(t_0)$ had to be introduced, which cannot be determined by the equation of motion. Here, $\mathbf{v}(t_0)$ is just the velocity at some time t_0, which has to be known. In the language of differential equations, this is the initial condition.

7.3 Definition of Velocity and Computation of Position

So far, we can compute the velocity of a particle from its acceleration, which we can compute from its mass and the force on the particle. A computer simulation will, however, usually require the computation of the position of the particle. To this end, we "integrate" the velocity once more. From a formal, mathematical point of view, this is completely analogous to the previous integration of the acceleration. However, the physical meaning is different.

Velocity is defined as the change of the position vector $\Delta\mathbf{r}$ per time Δt for infinitely small time intervals:

$$\mathbf{v} = \lim_{\Delta t \to 0} \frac{\Delta\mathbf{r}}{\Delta t} \tag{7.9}$$

Again, we can write this as a derivative of the time-dependent position $\mathbf{r}(t)$ (or three derivatives of the three coordinates of $\mathbf{r}(t)$):

$$\mathbf{v}(t) = \left.\frac{\mathrm{d}\mathbf{r}(t')}{\mathrm{d}t'}\right|_{t'=t} \tag{7.10}$$

This equation is integrated with the initial position $\mathbf{r}(t_0)$ at t_0 to yield:

$$\mathbf{r}(t) = \mathbf{r}(t_0) + \int_{t_0}^{t} \mathbf{v}(t')\mathrm{d}t' \tag{7.11}$$

In this way, we can compute the position $\mathbf{r}(t)$ of the particle for any time t based on the initial position $\mathbf{r}(t_0)$, the initial velocity $\mathbf{v}(t_0)$, the particle's mass m, and the (possibly time-dependent) force $\mathbf{F}(t)$.

There are, however, some serious potential problems. While the mentioned approximation to the gravity force could be easily integrated analytically (this is left as an exercise for the reader), most forces on particles actually depend on the velocity or the position of the particle. For example, the force of a spring between two particles usually depends on the distance between the two particles and therefore on their positions. However, it is the position that we want to compute from the force; thus, we cannot use the (unknown) position to compute the force. Fortunately, the situation is not quite as bad in the case of a numerical integration.

7.4 Outlook: Discrete Numerical Integration

In a numerical simulation, time t is usually discretized into discrete times t_i with $i = 0, 1, 2, 3, \ldots$ and all time-dependent variables are only computed at those times: $\mathbf{r}_i = \mathbf{r}(t_i)$, $\mathbf{v}_i = \mathbf{v}(t_i)$, $\mathbf{a}_i = \mathbf{a}(t_i)$, and $\mathbf{F}_i = \mathbf{F}(t_i)$. The idea of the discrete integration is to start with the known initial values for t_0 and then to compute the values for t_1 based on these initial values. In the next step, the values for t_2 are computed based on the previous values and so on. In general, we want to compute the values for t_{i+1} from the values for t_i (and possibly $t_{i-1}, t_{i-2}, \ldots, t_0$).

One of the most basic discrete integration methods is called explicit Euler integration. For this method, the computation for time t_{i+1} based on the values for t_i would be:

$$\mathbf{a}_i = m^{-1}\mathbf{F}_i \tag{7.12}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i\Delta t \tag{7.13}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_i\Delta t \tag{7.14}$$

where $\Delta t = t_{i+1} - t_i$ is the length of the time step. Here, $\mathbf{F}_i$ can depend on $\mathbf{r}_i$ and $\mathbf{v}_i$ without causing any trouble. However, explicit Euler integration is not very precise. Actually, it is a rather poor choice for an integration method; thus, many better techniques have been suggested, e.g., Runge-Kutta methods.

Further Reading

A more detailed introduction is provided in Chapter 13 of James M. Van Verth and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications — A Programmer's Guide*, 2nd ed., Morgan Kaufmann Publishers, 2008.

More details are given in Chapter 22 of Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann, *Physics-Based Animation*, Charles River Media, 2005.

Chapter 8

Face Recognition using Principal Component Analysis

Keywords: Eigenvalues, eigenvectors, face recognition.

Principal Component Analysis (PCA) is a well-known technique in linear algebra for mapping high-dimensional observations of possibly correlated variables to a lower dimensional space in which the variables are linearly uncorrelated. Since the second space is of lower dimension compared to the original space, there will be some loss of information during this process. PCA finds the basis of the new space such that this loss is minimal. The basis vectors of the new space are known as principal components and they are usually found by eigenvalue decomposition. In the following subsection eigenvalue decomposition is explained for a small matrix, and then it will be used for the PCA in a real-world example, namely face recognition.

8.1 Eigenvalue Decomposition

For a square matrix A, an eigenvector $\vec{x}$ is a vector unequal to $\vec{0}$ with:

$$A\vec{x} = \lambda\vec{x} \tag{8.1}$$

where λ is a real or complex number and is called the eigenvalue of A associated with its eigenvector $\vec{x}$. The eigenvalues of A can be calculated by:

$$\det(A - \lambda \mathrm{Id}) = 0 \tag{8.2}$$

where det is the determinant function and Id is the identity matrix. To do the eigenvalue decomposition, the eigenvalues of a given matrix are first found using Eq. 8.2. Then each of the computed eigenvalues is substituted into Eq. 8.1 to find its associated eigenvector.

Let's assume that, for example, $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ which means that the eigenvectors will be of the form $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$. Using Eq. 8.2, we have:

$$\det\left(\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}\right) = 0 \tag{8.3}$$

This can be simplified to $(1-\lambda)^2 - 4 = 0$ which results in $\lambda_1 = 3$ and $\lambda_2 = -1$. Substituting λ_1 into Eq. 8.1 results in the following system of equations:

$$\begin{cases} -2x_1 + 2x_2 &= 0 \\ 2x_1 - 2x_2 &= 0 \end{cases}$$

Setting $x_1 = t$, it is obvious that $x_2 = t$. This means that the corresponding eigenvector of the largest eigenvalue, $\lambda_1 = 3$, is: $[1\ 1]^T$. (Note that the scaling of eigenvectors is arbitrary.) Repeating the process for the second eigenvalue, $\lambda_2 = -1$, the following eigenvector will be obtained: $[1\ -1]^T$. These perpendicular eigenvectors are the basis vectors of a new space, usually called eigenspace. Now let's see how projections to eigenspaces can be used in practical problems.

8.2 Face Recognition

In this section, we show how an eigenvalue decomposition can be used for a PCA in the context of face recognition. This has been first reported in [1] which provides further details about the algorithm.

To use PCA for face recognition, we obviously need a database of facial images. Here, we use the ORL Face Database for this purpose [2]. Some of the facial images of this database are shown in Figure 8.1. There should be more than one sample image per person in the database. Some of the samples of each person will be used for training and the rest for testing the trained system.

The first step in using PCA for face recognition is reshaping the two-dimensional matrix of facial images into one-dimensional vectors and combining them into a large matrix. The number of rows of this large matrix is equal to the number of the pixels of the images (they are assumed to be of the same size) and the number of its columns is equal to the number of the images. Let's say the employed database contains M training images as Γ_1, Γ_2, ..., Γ_M, each of size $N \times N$. Having reshaped these images into vectors

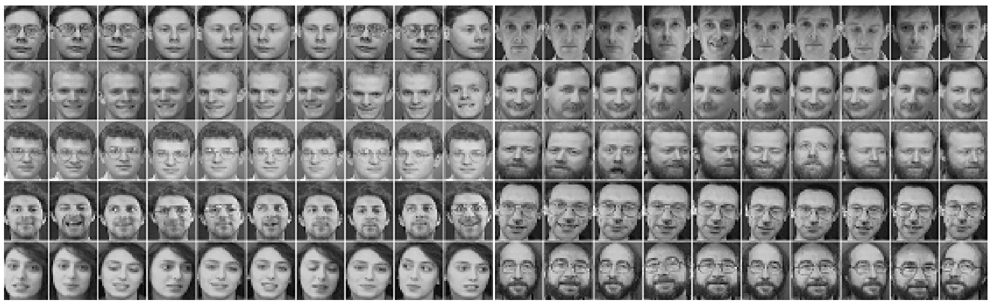


Figure 8.1: Some of the face images of the ORL Face Database [2].

of size $N^2 \times 1$, then a so-called mean face is calculated as: $\Psi = \frac{1}{M}\sum_{n=1}^{M}\Gamma_n$. This mean face which is shown in Figure 8.2 will then be subtracted from each face sample for the normalization purposes as: $\Phi_i = \Gamma_i - \Psi$, where Φ_i is the normalized face image.

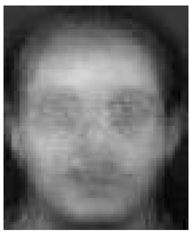


Figure 8.2: The mean face image of the employed database

The set of all of these normalized vectors is then combined in a matrix as $A = [\Phi_1\ \Phi_2\ ...\ \Phi_M]$ of size $N^2 \times M$ to which the PCA technique will be applied. To this end, the eigenvalues and eigenvectors of the following covariance matrix need to be calculated:

$$C = \sum_{n=1}^{M} \Phi_n \Phi_n^T = AA^T \tag{8.4}$$

Since this matrix is of a very large size, $N^2 \times N^2$, obtaining its eigenvalues is not easy. However, it can be shown (see [1] for details) that the eigenvectors of AA^T are the same as those for $L = A^T A$ which is of the significantly smaller size $M \times M$. The M eigenvectors of this matrix, which can be obtained using the method described in the previous section, are the basis vectors of the eigenspace. This means that every face image can be expressed as a linear combination of these basis vectors. The 18 eigenfaces corresponding to the 18 largest eigenvalues of matrix L for the employed database are shown in Figure 8.3.

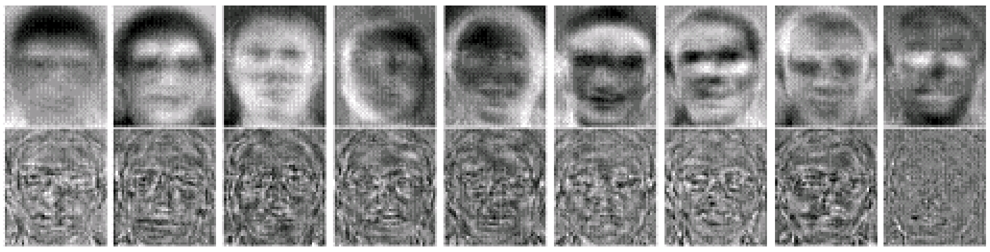


Figure 8.3: Some of the eigenfaces of the employed database.

These 18 eigenfaces form the basis of a new, 18-dimensional space. Every face image can be approximated as a linear combination of these basis vectors. For the recognition purpose, every image of the database is mapped in this way to the new space using these basis vectors. Doing so results in a weighting vector for each face image where the elements of the vector reveal the influence of the corresponding eigenface for mapping that face.

Having an input test image, it will first be projected to the eigenspace and the weights of each of the eigenfaces for doing this projection will be calculated. Then the result of the recognition will be a face image which has the closest weight vector to the weight vector of the test image. To measure the closeness, distance metrics like simple Euclidean distance can be used.

8.3 References

[1] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. *P*roceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 586-591.

[2] ORL Face Database, available at: http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html, accessed: 27, Nov., 2012.

Chapter 9

Recognizing words: Finite automata

Keywords: Automata.

In this chapter we describe an application of ideas from set theory and graph theory to algorithms for recognizing which words are valid. The notion of *finite automaton* originally arose in the study of neural networks; its fundamental theory is due to Kleene [2], Rabin and Scott [3].

Finite automata can be seen as very simple algorithms for testing if a given word belongs to some set of words.

Two very important applications of finite automata are

- Their use in text editors; one can define a search pattern by means of a regular expression (these are defined later in this text) and by means of an algorithm extract a finite automaton that will accept precisely the set of words that match the search pattern, and
- Their use in compilers for programming languages; the compiler will as one of its first steps perform a so-called lexical analysis that will check if the program identifiers, reserved words and numerical constants follow the syntax of the programming language. This check is performed by a finite automaton.

9.1 Letters, words and languages

First we need to fix the terminology. An *alphabet* is a finite set of *letters*. A *word* over Σ is a finite sequence of letters from the alphabet Σ. The *empty word* is the sequence of letters that has length 0; it is denoted by ε. Note that we do not ascribe any meaning to words; a word can be any sequence of characters from any given alphabet.

The *concatenation* of two words w_1 and w_2 is the word formed by the letters of w_1 followed by the letters of w_2 and is denoted by w_1w_2.

A *language* is a set of words; this set may be finite or infinite. Again, note that we do not ascribe any structural demands on languages; a language is just a set whose elements are words (in the sense just defined).

9.2 Finite automata

A *finite automaton* is a 5-tuple $A = (Q, \Sigma, q_0, \rightarrow, F)$ with the following properties:

- $(Q, \rightarrow)$ is a directed graph.
- The edges in the set of edges $\rightarrow$ are labelled with elements from Σ. We write $u \xrightarrow{a} v$ if there is an edge from u to v labelled a. We call an edge a *transition*. We allow loops, i.e. transitions $u \xrightarrow{a} u$.
- The vertices in Q are called *states* and satisfy the following condition:

  For every $u \in Q$ and $a \in \Sigma$ there is exactly one $v \in Q$ such that $u \xrightarrow{a} v$
- $q_0 \in Q$ is a designated state called the *start state*.
- We have $F \subseteq Q$, and F is called the set of *final states*.

We can think of a finite automaton as a flow diagram for reading a word. We start in the finite state and follow the transitions of the graph labelled with the letters of the word.

Let $w = a_1 \dots a_k$ be a word over the alphabet Σ. A *computation* for a finite automaton is a sequence of edges

$$q_0 \xrightarrow{a_1} q_1 \cdots q_{k-1} \xrightarrow{a_k} q_k$$

We sometimes write this as simply $q_0 \xrightarrow{w} q_k$.

If $q_k \in F$ we say that the computation is *accepting* and we say that M accepts w.

A language is called a *regular language* if there exists a finite automaton such that $L = \{w \mid M \text{ accepts } w\}$. We say that M *recognizes* L.

9.3 Closure under set operations

The class of regular languages is closed under the set operations of union, intersection and set difference. By this we mean that

Theorem 1. *If L_1 and L_2 are regular languages over the same alphabet Σ, then*

1. *$L_1 \cup L_2$ is a regular language.*
2. *$L_1 \cap L_2$ is a regular language.*

3. *$L_1 \setminus L_2$ is a regular language.*

The proof of this theorem proceeds as follows.

Suppose L_1 is a regular language. Then there exists a finite automaton $M_1 = (Q_1, \Sigma, q_{01}, \rightarrow_1, F_1)$ that recognizes L_1. Similarly, there exists a finite automaton $M_2 = (Q_2, \Sigma, q_{02}, \rightarrow_2, F_2)$ that recognizes L_1.

In all three cases, we define a new finite automaton $M_{12} = (Q_{12}, \Sigma, q_0, \rightarrow, F)$ where the states are given by

$$\begin{aligned} Q &\overset{\text{def}}{=} Q_1 \times Q_2 = \{(q', q'') \mid q' \in Q_1, q'' \in Q_2\} \\ q_0 &\overset{\text{def}}{=} (q_{01}, q_{02}) \end{aligned}$$

The idea behind this construction is the new automaton describes what would happen if M_1 and M_2 were run at the same time. The idea is that a state (q', q'') represents the joint state of the two original automata. This automaton is called the *product automaton* of M_1 and M_2.

Therefore the transitions are given by

$$(q', q'') \xrightarrow{a} (q'_1, q''_2) \text{ if } q' \xrightarrow{a}_1 q'_1 \text{ and } q'' \xrightarrow{a}_2 q''_2$$

If we want the product automaton to recognize $L_1 \cup L_2$, we let

$$F = \{(q', q'') \mid q' \in F_1 \text{ or } q'' \in F_2\}$$

If we want the product automaton to recognize $L_1 \cap L_2$, we let

$$F = \{(q', q'') \mid q' \in F_1 \text{ and } q'' \in F_2\}$$

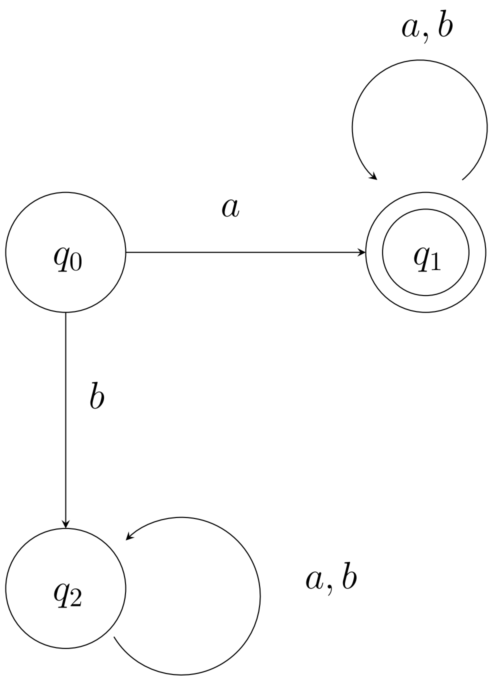
And finally, if we want the product automaton to recognize $L_1 \setminus L_2$, we let

$$F = \{(q', q'') \mid q' \in F_1 \text{ and } q'' \notin F_2\}$$

The proof that the construction that we have just described is a proof of the fact that

$$(q', q'') \xrightarrow{w} (q'_1, q''_2) \text{ iff we have } q' \xrightarrow{w}_1 q'_1 \text{ and } q'' \xrightarrow{w}_2 q''_2$$

for every word w. This proof proceeds by induction in the length of w.



Example 1.

Figure 9.1: A finite automaton recognizing the language L_1

9.4 Concatenation, Kleene closure and regular expressions

We can define two further operations on finite automata, namely concatenation and Kleene closure. They are

Definition 1 (Concatenation of languages)**.** Let L_1 and L_2 be languages. Then the concatenation of L_1 and L_2 is the language

$$L_1 \circ L_2 = \{uv \mid u \in L_1, v \in L_2\}$$

Definition 2 (Kleene closure)**.** Let L be a language. Then the Kleene closure of L is the language

$$L^* = \{w_1 \dots w_n \mid n \geq 0, w_i \in L \text{ for all } i \leq n\}$$

Since we in the definition of L^* allow for words formed by 0 strings from L, the empty word ε will always be a member of L^*.

An automaton which recognizes L_1 is depicted in Figure 9.1. We follow the convention of denoting accept states with a double circle and by depicting transitions with different labels by a single edge.

Note that there may be several different automata that recognize the same language. An important result due to Myhill and Nerode [1] is that there always exists a *minimal automaton* recognizing a given regular language and that this automaton can be found by means of an algorithm.

The proof that the class of regular languages is also closed under these operators mentioned here goes beyond the scope of this short note, but taken together, the *regular operations* of concatenation, Kleene closure and union are enough to specify precisely the class of regular languages [2, 3].

A set expression that only uses the regular operations is called a *regular expression.*

In fact the proof that regular expressions capture precisely the class of regular languages is constructive: there is an algorithm that, given any finite automaton can build the regular expression of the language that it recognizes. Conversely, there is an algorithm that, given any regular expression defining a language can build the finite automaton recognizing it.

Example 2. Suppose our alphabet $\Sigma = \{a, b\}$. The language L of all words that either begin with a or ends with two b's is given by the regular expression

$$L = \{a\} \circ \{a, b\}^* \cup \{a, b\}^* \circ \{bb\}$$

9.5 References

[1] A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, Vol. 9, No. 4 (Aug., 1958) (pp. 541-544).

[2] S. C. Kleene, Representation of events in nerve nets and finite automata. *Automata Studies*, Princeton University Press, 1956, pp. 3-41.

[3] M.O. Rabin and D.S. Scott. Finite Automata and Their Decision Problems, *IBM Journal of Research and Development*, Volume 3, Issue 3, April 1959.

Chapter 10

Error-collecting cødes $\mapsto$ *Error-correcting codes*

Keywords: Error-correcting codes, noisy channel, digital data transmission.

Digital data transmission is vulnerable to errors and erasures. This may happen when a picture is send from the surface of Mars to the Earth, when broadcasting TV or radio, when reading information on a CD-ROM etc. The solution to the problem is to add some redundancy to the message. This should be understood in the information theoretic way: Rather than sending a message vector $\vec{m}$ of lenght k a corresponding codeword $\vec{c}$ of length n, with $k < n$, is being send. The digits of $\vec{m}$ need NOT appear in $\vec{c}$.

The theory of error-correcting codes was born with the pioneering work of Shannon during the second world war. Today it is an important part of most digital communication. As an example QR-codes use error-correcting codes.

10.1 The repetition code

A file is a sequence of 0's and 1's. The most naive approach to protect a data transmission would be to send every symbol three times: Instead of sending 0 we send 000 and instead of 1 we send 111. The corresponding code is called the repetition code of obvious reasons. Until the publication of Shannon's work in 1948 it was the common belief that this was the only way to protect digital data from noise. Note that the repetition code is expensive in use, the ratio of information symbols pr. send symbol (called the information rate) being only 1/3. As a preparation to describe better codes we start by describing the repetition code in the language of error-correcting codes.

In most Linear Algebra courses the scalars are either real numbers or complex numbers. In mathematical language this amounts to saying that the field is $\mathbb{R}$ or $\mathbb{C}$. In digital communication the field is often of finite size, the most simple case being the binary field $\mathbb{F}_2 = \{0, 1\}$. The rules for addition and multiplication in $\mathbb{F}_2$ are as follows:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Most results from the Linear Algebra course (but not all) hold when replacing $\mathbb{R}$ (or $\mathbb{C}$) with $\mathbb{F}_2$ or any other finite field. In a similar way as $\mathbb{R}^n$ means the vectorspace

$$\{[\ c_1 \quad c_2 \quad \cdots \quad c_n\ ]^T \mid c_1, c_2, \ldots, c_n \in \mathbb{R}\}$$

by $\mathbb{F}_2^n$ we shall mean the vectorspace

$$\{[\ c_1 \quad c_2 \quad \cdots \quad c_n\ ]^T \mid c_1, c_2, \ldots, c_n \in \mathbb{F}_2\}.$$

For the case of the repetition code the messages are either 0 or 1. We identify these with vectors of length 1, namely [0] and [1]. The corresponding code words are $[\ 0 \quad 0 \quad 0\ ]^T$ and $[\ 1 \quad 1 \quad 1\ ]^T$. We see that the encoding of a message can be thought of as using the linear transformation $T : \mathbb{F}_2^1 \to \mathbb{F}_2^3$, $T(\vec{m}) = G\vec{m}$, where

$$G = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

is called a generator matrix.

10.2 The $[7,4,3]$ binary Hamming code

We next describe the $[7,4,3]$ binary Hamming code. When using this code we transform messages of length 4 to code words of length 7. Hence, the information rate is 4/7. As we shall demonstrate we will be able to correct one error in each code word.

Consider the binary matrices

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

A message $\vec{m} \in \mathbb{F}_2^4$ is encoded to a code word $\vec{c}$ by $\vec{c} = T(\vec{m}) = G\vec{m}$. Again, of course T is a linear transformation and G is called a generator matrix. The above encoding rule means that the collection of code words (called the code) is the column space of G. One can check that the columns of G are linearly independent (over $\mathbb{F}_2$). Hence, a code word can uniquely be identified with a message (we have a bijective map). Actually, for the $[7,4,3]$ binary Hamming code $\vec{m}$ is found as the first four entries of the corresponding code word $\vec{c}$.

Using standard techniques from linear algebra one can check that $\operatorname{Col} G$ is the same as $\operatorname{Nul} H$. This amounts to observing that both G and H are of full rank and that $HG = O$ holds (here O is the 3×4 zero matrix). If the codeword is not changed when passing the information channel we can check this at the receiving end by calculating $H\vec{c}$ which should then be $[\ 0\ \ 0\ \ 0\ \ 0\ ]^T$ (H is called the parity check matrix). We next consider the case that exactly one error occurs. This means that exactly one of the elements in $\vec{c}$ is changed from either 0 to 1 or vice versa. We can formalize this by saying that $\vec{r} = \vec{c} + \vec{e}$ is received where $\vec{e}$ is 0 in all positions except in one where it is 1. For instance we may receive:

$$\vec{r} = [\ 1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ ]^T + [\ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ]^T = [\ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ ]^T.$$

The decoding algorithm now relies on the very nice fact that

$$H\vec{r} = H(\vec{c} + \vec{e}) = H\vec{c} + H\vec{e} = \vec{0} + H\vec{e} = H\vec{e}.$$

Now if there is an error in position i meaning that the only non-zero element of $\vec{e}$ is in position i. Then $H\vec{r} = H\vec{e}$ will equal the ith column of H. But all the columns of H are different which makes it possible to read of the position of error from the matrix vector product $H\vec{r}$. Let us assume $\vec{r} = [\ 1 \ \ 1 \ \ 1 \ \ 0 \ \ 0 \ \ 0 \ \ 1\ ]^T$ as above. At the receiving end we calculate

$$H\vec{r} = [\ 1 \ \ 0 \ \ 1\ ]^T,$$

which tells us that indeed there is an error in the second position. We then recover the codeword simply by flipping the symbol in position two. Finally, as the upper part of G is an identity matrix the first four elements of the recovered word $\vec{c}$ is the message $\vec{m}$.
In general the $[7, 4, 3]$ code is capable of correcting one error but no more than that. We will say that its error-correcting capability is 1.

10.3 The theory of error-correcting codes

Above we presented two of the simplest error-correcting codes. In general a good error-correcting code must have a reasonable transmission rate and be able to correct many errors. This involves that the code should be structured in such a way that fast decoding algorithms can be designed. Many deep results have been established since Shannon's seminal work, but many others are still awaiting their discovery.

10.4 References

[1] J. Justesen and T. Høholdt, “A Course In Error-Correcting Codes”, European Mathematical Society, *Textbooks in Mathematics*, 2004

Chapter 11

Modular arithmetic and public-key cryptography

Keywords: Cryptography.

11.1 Cryptography

Cryptography is the practice and study of being able to communicate confidential information securely, that is, in the presence of a potential adversaries. A *cryptosystem* consists of an encryption function E, a decryption function D, a set of plaintexts $\mathcal{P}$, a set of ciphertexts $\mathcal{C}$, a set of encryption keys $\mathcal{K}_e$ and a set of decryption keys $\mathcal{K}_d$.

An *instance* of the cryptosystem chooses a specific encryption key and a specific decryption key.

The encryption function takes as its arguments an encryption key $k_e \in \mathcal{K}_e$ and a text t and produces a ciphertext $E(k_e, t)$. The decryption function takes as its arguments a decryption key $k_d \in \mathcal{K}_d$ and a ciphertext c and produces a plaintext $E(k_d, c)$.

Historically, the first cryptosystems were *symmetric cryptosystem.* In such cryptosystems we always choose the decryption key and the encryption key to be the same in an instance of it. Let us call this key instance k. Decryption and encryption are therefore inverses of each other, that is, we have that

$$D(k, (E(k, t))) = t$$

The problem with this approach is that all users of an instance of a symmetric cryptosystem need to share the key and to keep it secret.

The idea of a *asymmetric cryptosystem* (also known as a public-key cryptosystem) is much more recent. The idea dates back to Ellis (secret paper, 1970) but the first cryptosystem is due to Diffie and Hellman and from 1976. The first important system is due to Rivest, Shamir and Adleman and from 1977; an equivalent system was invented by Cocks in 1973.

An asymmetric cryptosystem uses *different* keys for encryption and decryption. Each user of the cryptosystem has their own pair of keys, K and K^{-1}. The encryption key K is made public, whereas K^{-1} is kept secret.

We must have that

$$D(K^{-1}, E(K,t)) = t \text{ for all } t$$

The advantage of this approach is that any two users can exchange secret messages without sharing a key pair – to send an encrypted message, each party only needs to know the public encryption key of the other party.

Most asymmetric cryptosystems make use of modular aritmetic, which we shall now introduce.

11.2 Modular arithmetic

We can define an equivalence relation on the natural numbers by

$$a \equiv b \mod n$$

if a and b, where $a, b, x \in \mathbb{N}$, have the same remainder under division by x. That is, for some remainder $r < n$ we have that $a = d_1 n + r$ and $b = d_2 n + r$. We say that a and b are equivalent modulo x.

It is not hard to prove that $\equiv$ is an equivalence relation, i.e. that for any $n \in \mathbb{N}, n > 0$ it satisfies the axioms

$$\begin{aligned} a \equiv a \mod n \quad &\text{for all } a \in \mathbb{N} \\ a \equiv b \mod n \Rightarrow b \equiv a \mod n \ &\text{for all } a, b \in \mathbb{N} \\ a \equiv b \mod n \wedge b \equiv c \mod n \Rightarrow a \equiv c \mod n \ &\text{for all } a, b, c \in \mathbb{N} \end{aligned}$$

Moreover, the relation respects multiplication and addition, i.e. that for any $x \in \mathbb{N}, x > 0$ it satisfies the axioms

$$\begin{aligned} a \equiv b \mod n \Rightarrow a + m \equiv b + m \mod n \text{ for all } a, b, m \in \mathbb{N} \\ a \equiv b \mod n \Rightarrow a \cdot m \equiv b \cdot m \mod n \text{ for all } a, b, m \in \mathbb{N} \end{aligned}$$

We can therefore think of $\equiv \mod n$ as defining a new kind of equality over the natural numbers.

In the following we let $\mathbb{Z}_n \stackrel{\text{def}}{=} \{a \in \mathbb{N} \mid a < n\}$.

We can define the additive inverse $-a$ in $\mathbb{Z}_n$ as the number $b \in \mathbb{Z}_n$ such that $a + b \equiv 0 \mod n$. This number is easily seen to be $n - a$. Likewise, we can define the multiplicative inverse (if this inverse exists; it does not always) $a^{-1} \mod n$ as the number $b \in \mathbb{Z}_n$ such that $ab \equiv 1 \mod n$.

One can show that if p is a prime number, then every $n \in \mathbb{Z}_p$ will have a multiplicative inverse $\mod p$.

Example 3. Let $n = 7$. Then 4 is the additive inverse of 3 in $\mathbb{Z}_7$, since $3 + 4 \equiv 0 \mod 7$. And $5 = 3^{-1}$, since $3 \cdot 5 \equiv 1 \mod 7$.

11.3 Greatest common divisors and the Euclidean algorithm

For any two natural numbers m and n their greatest common divisor $\gcd(m, n)$ is the largest $d \in \mathbb{N}$ such that $m = dx_1$ for some x_1 and $n = dx_2$ for some x_2. We say that m and n are *relatively prime* if $\gcd(m, n) = 1$.

We now present a simple algorithm that can be used to find $\gcd(m, n)$ and also to find the multiplicative inverse $\mod p$ for any number in $\mathbb{Z}_p$. This algorithm was first presented in Euclid's *Elements* and is therefore known as the Euclidean algorithm.

Note that whenever $m \geq n$, we have

$$\gcd(m, n) = \gcd(n, m \mod n)$$

since $m = kn+r$ where $r < n$ and $r = m \mod n$, and the greatest common divisor has to divide both summands on the right. This can be used to give an algorithm for finding the greatest common divisor.

```
GCD$(a, b)$

$r_0 \leftarrow a$
$r_1 \leftarrow b$
while  $r_m \neq 0$  do
      $q_m \leftarrow \lfloor \frac{r_{m-1}}{r_m} \rfloor$
      $r_{m+1} \leftarrow r_{m-2} - q_m r_m$
      $m \leftarrow m + 1$
return $r_{m-1}$
```

We can now show that $\gcd(m, n)$ can be written as a linear combination of m and n. The algorithm produces a sequence of equations:

$$r_0 = q_1 r_1 + r_2 \tag{11.1}$$

$$r_1 = q_2 r_2 + r_3 \tag{11.2}$$

$$\dots \tag{11.3}$$

$$r_{m-2} = q_{m-1} r_{m-1} + r_m \tag{11.4}$$

$$r_{m-1} = q_m r_m \tag{11.5}$$

We have $r_2 = r_0 - q_1 r_1$. Apply this to (11.2) to get $r_3 = r_1 - q_2(r_0 - q_1 r_1)$. Continuing in this fashion, we see that each r_i can be written as a linear combination of r_0 and r_1. We can find the coefficients as follows. Define

t_j's and s_j's:

$$t_j = \begin{cases} 1 & \text{for } j = 0 \\ 0 & \text{for } j = 1 \\ t_{j-2} + q_{j-1}t_{j-1} & \text{otherwise} \end{cases}$$

$$s_j = \begin{cases} 0 & \text{for } j = 0 \\ 1 & \text{for } j = 1 \\ s_{j-2} + q_{j-1}s_{j-1} & \text{otherwise} \end{cases}$$

At each stage we have that $r_j = s_j r_0 + t_j r_1$.

We can find a multiplicative inverse of a modulo m, if $\gcd(m, a) = 1$. But then we have $1 = s_m m + t_m a$. Reducing this modulo m, we get $1 = t_m a$, so t_m is the multiplicative inverse. This gives us an algorithm for finding a^{-1}.

11.4 One-way functions

For an asymmetric cryptosystem, the encryption function should be a 'one-way function', i.e. it should be " difficult" to compute t from $E(K, t)$. A more precise definition of what we mean by this requires knowledge of the theory of computational complexity; this is beyond the scope of the present text.

At present, no one-way functions are known to exist, but there are several good candidates. One such candidate is modular exponentiation:

$$f(x) = x^b \mod n$$

The RSA cryptosystem (and many others) rely on this and situations (choices of n) for which f^{-1} is also a modular exponentiation.

11.5 RSA

The RSA cryptosystem is named after Rivest, Shamir and Adleman.

Let p, q be primes and let $n = pq$. Let a, b which satisfy that $ab \equiv 1 \mod (p-1)(q-1)$. Then we can define a pair of keys (K, K^{-1}) as follows.

The public key is $K \overset{\text{def}}{=} (n, b)$, the private key is $K^{-1} \overset{\text{def}}{=} (a, p, q)$.

Encryption and decryption are defined as follows.

$$E(K, x) = x^b \mod n$$
$$D(K^{-1}, y) = y^a \mod n$$

One can now prove that for all $x \in \mathbb{Z}_n$ we have $D(K^{-1}, (E(K, x))) \equiv x \mod n$.

Here is how we create an instance of RSA:

1. Generate large primes p, q
2. Let $n = pq$. We let $\phi(n) = (p - 1)(q - 1)$.
3. Find a b such that $\gcd(b, \phi(n)) = 1$.
4. Find $a = b^{-1} \mod \phi(n)$ using the extended Euclidean algorithm.

Chapter 12

McEliece Cryptosystem

Keywords: cryptography, error-correcting codes, McEliece cryptosystem, quantum computer.

The emergence of quantum computers will imply that cryptosystems such as RSA are no longer secure. Fortunately, the research community provides other candidates. One of the most promising ones is the McEliece cryptosystem. The McEliece cryptosystem is not yet used in practical application due to the fact that its keys are much longer than the keys used in classical systems such as RSA. The development of quantum computers most likely would change that situation over night. The McEliece cryptosystem is a

OH ALICE... YOU'RE THE ONE FOR ME
BUT BOB... IN A QUANTUM WORLD HOW CAN WE BE SURE?
ψ^+ or ψ^- ?

public key system based on error-correcting codes (if you are not familiar with this object you should read first the chapter "Error-collecting cødes $\mapsto$ Error-correcting codes"). Its security relies on the observation that decoding *general* linear code is NP-complete (meaning it is expected to be a very difficult problem to solve). The idea in the cryptosystem is, given a secret code C for which an efficient decoding algorithm is known to modify it slightly to another code C' which appears impossible to decode unless one knows the modifications made.

The McEliece cryptosystem is asymmetric. This means that there are two sets of keys. The one set is public and the other set is private. Using the public key anybody can encrypt a message and send it to the owner of the private key. Only he can then decrypt it. Observe that a third party cannot decrypt the encrypted message even though the public key is also known by him (the public key could be announced at a homepage or similar). There are a vast number of error-correcting codes but so far only very specific families of codes resist cryptoanalysis. The most secure codes still seems to be the Goppa codes suggested by McEliece in the first place. It is beyond the scope of this manuscript to treat Goppa codes. We shall therefore explain the principle of the McEliece cryptosystem using a much simpler code namely a binary Hamming code. This code along with its decoding algorithm is described in the chapter "Error-collecting cødes $\mapsto$ Error-correcting codes". If you are familiar with that chapter you can skip the following section.

12.1 The $[7, 4, 3]$ Hamming code

Consider the binary matrices

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

As we work in the binary field $\mathbb{F}_2 = \{0, 1\}$ we have the rule $2 = 0$. This gives us

$$0+0=0, \quad 0+1=1, \quad 1+0=1, \quad 1+1=0,$$
$$0\cdot 0=0, \quad 0\cdot 1=0, \quad 1\cdot 0=0, \quad 1\cdot 1=1.$$

The $[7, 4, 3]$ binary Hamming code by definition equals Col G – the column space of G. Hence, the code consists of the $2^4 = 16$ words that can be made as binary linear combinations of the four columns of G. For instance taking 1 times the first column, 0 times the second column, 1 times the third column, and finally 0 times the fourth column produces the code word

$$[\ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1\ ]^T + [\ 0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ ]^T = [\ 1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ ]^T,$$

where we used the above rules for addition and multiplication. Using standard techniques from linear algebra one can check that $\operatorname{Col} G$ is the same as $\operatorname{Nul} H$. This amounts to observing that both G and H are of full rank and that $HG = O$ holds (here O is the 3×4 zero matrix).

Messages $\vec{m} = [\ m_1 \quad m_2 \quad m_3 \quad m_4\ ]^T \in \mathbb{F}_2^4$ are encoded into code words by the linear transformation $T : \mathbb{F}_2^4 \to \mathbb{F}_2^7$, $T(\vec{m}) = G\vec{m}$. We assume a codeword $\vec{c} = T(\vec{m})$ is send over some noisy channel. If the codeword is not disturbed we can check this at the receiving end by calculating $H\vec{c}$ which should then be $[\ 0 \quad 0 \quad 0 \quad 0\ ]^T$. We next consider the case that exactly one error occurs. This means that exactly one of the elements in $\vec{c}$ is changed from either 0 to 1 or vice versa. We can formalize this by saying that $\vec{r} = \vec{c} + \vec{e}$ is received where $\vec{e}$ is 0 in all positions except in one where it is 1. For instance we may receive:

$$\vec{r} = [\ 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1\ ]^T + [\ 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0\ ]^T = [\ 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1\ ]^T.$$

The decoding algorithm now relies on the very nice fact that

$$H\vec{r} = H(\vec{c} + \vec{e}) = H\vec{c} + H\vec{e} = \vec{0} + H\vec{e} = H\vec{e}.$$

Now if there is an error in position i meaning that the only non-zero element of $\vec{e}$ is in position i. Then $H\vec{r} = H\vec{e}$ will equal the ith column of H. But all the columns of H are different which makes it possible to read of the position of error from the matrix vector product $H\vec{r}$. Let us assume $\vec{r} = [\ 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1\ ]^T$ as above. At the receiving end we calculate

$$H\vec{r} = [\ 1 \quad 0 \quad 1\ ],$$

which tells us that indeed there is an error in the second position. We then recover the codeword simply by flipping the symbol in position two. Finally, as the upper part of G is an identity matrix the first four elements of the recovered word $\vec{c}$ is the message $\vec{m}$.
In general the $[7, 4, 3]$ code is capable of correcting one error but no more than that. We will say that its error-correcting capability is 1.

12.2 The decoding problem

Let $n > k$. Any $n \times k$ matrix G of $\operatorname{rank} G = k$ defines a code in the same way as did the 7×4 matrix G above. It is interesting to observe that if the entries of such a matrix G are chosen by random then with very high probability the requirement $\operatorname{rank} G = k$ is met. This is another way of saying that there are quite a lot of possible choices of G with rank G given fixed n and k.

The *decoding problem* is as follows. Let G and a corresponding codeword $\vec{c}$ be given. Assume we receive the word $\vec{r} = \vec{c} + \vec{e}$ where the number of non-zero elements in $\vec{e}$ is smaller than or equal to the error-correcting capability of the code. How can we recover $\vec{c}$? The problem size is specified by the parameter (n, k).

In 1978 Berlekamp, McEliece and van Tilborg showed that the decoding problem is NP-complete. Hence, there is (presumable) no simple general algorithm or principle that decodes general codes. Therefore, if we just receive a matrix G without any information on how it was constructed then we will have huge difficulties in decoding.

12.3 The cryptosystem

We now describe how Bob can set up a protocol and construct a corresponding public key and private key. The idea is that the public key is made public (surprise) and that Bob keeps the private key to himself (again surprise). When Alice wants to send a message (plain text) to Bob which should be kept secret to Eve, Alice uses the protocol and the public key to encrypt it. Only Bob will be able to decrypt it as he is the only one having the private key. The details are as follows:

- Bob chooses an $n \times k$ matrix G with rank $G = k$. He does this in such a way that the code has good error-correcting capability, say it can correct up to t errors. The code is chosen from a family of codes for which efficient decoding algorithms are known.
- Bob chooses an invertible $k \times k$ matrix S. (S stands for substitution).
- Bob chooses an $n \times n$ permutation matrix P. This is a binary matrix with exactly one 1 in each row and in each column.
- Bob calculates $G' = PGS$ (which is possible for him as he followed the linear algebra course).

Public key: G' (appears to be random, but is not), and t.

Private key: S^{-1}, $P^{-1} = P^T$, and a decoding algorithm for the code corresponding to G.
Encryption: Alice wants to send the message (plain text) $\vec{m}$ to Bob. She first calculates the matrix-vector product $\vec{c}' = G'\vec{m}$. She then adds up to t random errors as follows: $\vec{r}' = \vec{c}' + \vec{e}'$. This is the cipher text to be send over a public channel of some kind.

Decryption: Bob receives $\vec{r}'$ which he decrypts as follows: He first calculate $\vec{r} = \vec{r}'P^{-1}$. Then he use the decoding algorithm for G. This produces a codeword $\vec{c}$ which differs from $\vec{r}$ in at most t positions. He then determine $\vec{m}'$ such that $G\vec{m}' = \vec{c}$. Finally, the plain text is found as $\vec{m} = S^{-1}\vec{m}'$.

It remains to be shown that actually the above decryption method finds the plain text. We have

$$\begin{aligned} \vec{r} &= \vec{r}'P^{-1} \\ &= (\vec{c}' + \vec{e}')P^{-1} \\ &= \vec{c}'P^{-1} + \vec{e}'P^{-1} \\ &= \vec{c} + \vec{e} \\ &= G(S\vec{m}) + \vec{e}. \end{aligned}$$

The following two crucial observations is what makes the decryption work:

1. $\vec{e} = P^{-1}\vec{e}'$ has at most t non-zero elements.

2. $\vec{m}' = S\vec{m}$ is not the “real” message, but it can be thought of as a related message with corresponding codeword $\vec{c}$.

Observe, that during the decryption we find at no point the code word corresponding to the plain text.

12.4 A worked through example

We shall use the $[7, 4, 3]$ binary Hamming code and the corresponding matrices G and H described previously. Bob chooses

$$S = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and calculates

$$S^{-1} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad G' = PGS = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Alice wants to send the message $\vec{m} = [\ 1 \ \ 1 \ \ 0 \ \ 1\ ]^T$ to Bob. Using G' he encodes it to $\vec{c}' = [\ 0 \ \ 1 \ \ 1 \ \ 0 \ \ 1 \ \ 1 \ \ 0\ ]^T$. She then adds an error in position 6. that is she sends

$$\vec{r}' = \vec{c}' + \vec{e}' = [\ 0 \ \ 1 \ \ 1 \ \ 0 \ \ 1 \ \ 0 \ \ 0\ ]^T$$

At the receiving end Bob first calculate

$$\vec{r} = P^T\vec{r}' = (\vec{r}'P)^T = [\ 1 \ \ 0 \ \ 0 \ \ 0 \ \ 1 \ \ 0 \ \ 1\ ]^T.$$

Using the decoding method for $[7, 4, 3]$ binary Hamming codes as described previously Bob find the following codeword which is very "close" to $\vec{r}$, namely $\vec{c} = [\ 1 \ \ 0 \ \ 0 \ \ 0 \ \ 1 \ \ 1 \ \ 1\ ]^T$. This corresponds to the "message" $\vec{m}' = [\ 1 \ \ 0 \ \ 0 \ \ 0\ ]^T$. The plain text is recovered as

$$\vec{m} = S^{-1}\vec{m}' = [\ 1 \ \ 1 \ \ 0 \ \ 1\ ]^T$$

Further Reading

Wikipedia: `http://en.wikipedia.org/wiki/McEliece_cryptosystem`

12.5 References

[1] E. Berlekamp, R. McEliece, and H. van Tilborg. "On the inherent intractability of certain coding problems", *IEEE Transactions on Information Theory 24(3)*, pp. 384–386, 1978

Chapter 13

Just relax: Computing shortest paths

Keywords: Algorithms, graph theory.

13.1 Finding the shortest paths

Given a map with finitely many points, one of which is taken as the *source*, and distances between pairs of points, find the shortest paths to all other points on the map. This problem arises in many different settings. Some examples are [3]:

- Maps
- Robot navigation.
- Texture mapping.
- Game programming.
- Typesetting in $\TeX$.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Here we shall describe two different algorithms for solving the shortest-paths problem. Both have their strengths and weaknesses.

13.2 Directed graphs

The mathematical apparatus that can be used to describe the shortest-paths problem and solutions to it is that of directed graphs.

A directed graph G is a pair (V, E) where V is a set of vertices and E is a set of edges. An edge from u to v is a pair (u, v).

A weighted directed graph G is a directed graph (V, E) together with a weight function $w : E \to \mathbb{R}$. We often think of $w(u, v)$ as the cost of travelling from u to v. If $w(u, v) > 0$ for all $u, v \in V$ we say that w is a positive weight function.

13.3 Dijkstra's algorithm

As the name suggests, this algorithm is due to Edsger W. Dijkstra [2]. It only works for weighted directed graphs whose weight function is positive.

The underlying idea is to find the shortest path leading to a vertex that has not been examined yet. We now include this vertex in the set S of examined vertices. When all vertices have been examined, the algorithm terminates.

The algorithm fills out two tables. Firstly, it updates a table f of predecessors. When the algorithm has finished and the table says that $f[v] = u$, this means that the vertex visited just before v on the shortest path from s to v is u. Secondly, the algorithm updates a table d of shortest distances. When the algorithm has finished and the table says that $d[u] = k$, this means that the shortest path from s to u has a total length of u.

The important step in the algorithm is that of *relaxation* with respect to an edge. Whenever a new vertex u has been selected, we check for every other vertex v if we can get a shorter path by going via u:

```
RELAX(u,v) =
if $d[u] + w(u,v) < d[v]$ then
        $f[v] \leftarrow u$
        $d[v] \leftarrow d[u] + w(u,v)$
```

The algorithm now looks as follows.

```
DIJKSTRA($(G, w, s)$)
```

```
$d[s] \leftarrow 0$
for all $v \neq u, v \in V$ do
    $d[v] = \infty$
$S \leftarrow \emptyset$
while $S \neq V$ do
    find a $v \notin U$ such that $d[v]$ is minimal
    $S \leftarrow S \cup \{v\}$
        for all $v \in V \setminus S$ do
            RELAX(u,v)
```

How many relaxation steps will Dijkstra's algorithm need? If our graph $G = (V, E)$ has $n = |V|$, then every unvisited vertex in $V \setminus S$ *may* be relaxed in every step, so the total number of relaxation steps can be at most n^2.

13.4 Bellman and Ford's algorithm

The idea of the Bellman-Ford algorithm [1] is to relax with respect to *every* edge in every iteration. The algorithm will perform this $n-1$ times, where n is the number of vertices in the graph under consideration.

The advantage of this approach is that it can be used to detect cycles of negative weight. For if there exists a cycle of negative weight, then this cycle can be extended to have an arbitrarily small total weight by further relaxation. Therefore we can finish the algorithm by if further relaxation is possible.

```
BELLMAN-FORD($(G, w, s)$)

$d[s] \leftarrow 0$
for all $u \neq s$
    $d[u] \leftarrow \infty$
for $i \leftarrow 1$ to $n-1$
    for each $(u, v) \in E$
        RELAX$(u, v)$ for each $(u, v) \in E$
    if $d[v] > d[u] + w(u, v)$
    then return *The graph contains a negative cycle*
```

How many relaxation steps will the Bellman-Ford algorithm need? If our graph $G = (V, E)$ has $n = |V|$ and $m = |E|$, then the algorithm will *always* perform exactly nm relaxations.

A graph can have up to n^2 edges, and in this case the running time is n^3 relaxation steps.

13.5 References

[1] Bellman, Richard. On a routing problem. *Quarterly of Applied Mathematics* 16: 8790, 1958.

[2] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik* 1: 269271, 1959.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

Chapter 14

Data compression and its limits

Keywords: Data compression, Kolmogorov complexity.

14.1 Alphabets and words

Here we shall concern ourselves with compressing text.

Definition 3. An *alphabet* A is a finite set of characters. A *word* over A is a finite sequence of characters from A. The *length* of a word w is the number of characters in w; this number is denoted by $|w|$.

Example 4. A well-known alphabet is the Latin alphabet

$$D = \{\text{a,b,c,d,}\ldots\text{,z}\}$$

A string over D could be `house`. Anoter word is `kpst`.

Example 5. Another alphabet that we shall use in the following is $\mathcal{B} = \{\mathtt{0}, \mathtt{1}\}$. We refer to words over $\mathcal{B}$ as *binary words.*

For every natural number k there are exactly 2^k binary words of length k. To see this, note that each of the k characters in a word of length k can be either `0` or `1`, so in total there are $\underbrace{2 \cdot 2 \cdots 2}_{k\text{times}}$ words of this length.

Definition 4. Let x and y be strings. The *concatenation* of x and y is the string that we obtain by plaing the symbols of x before those of y; the resulting string is denoted xy.

Example 6. Let $x = \text{snow}$ and $y = \text{mand}$. Then $xy = \text{snowman}$.

We sometimes need to compare binary strings; to this end we define the so-called *lexicographic ordering.* This is the usual ordering known from dictionaries, except that a short string is always “ less than” a longer string. Our alphabet is $\{\mathtt{0}, \mathtt{1}\}$ and we assume that `0` is the first letter of the alphabet.

Definition 5. Let x and y be binary strings. We say that x is *lexicographically less than* y if x appears earlier in the dictionary of binary strings than y. We then write $x < y$.

Example 7. We have that `0` $<$ `01`, that `01` $<$ `11` and that `11` $<$ `110`.

Definition 6. next(s) is the next string that is lexicographically greater than s.

next(x) thus denotes the "next word in the dictionary after x". x'.

Example 8. Let $x =$ `01`; then we have next(x) $=$ `10`. Let $y =$ `111`; then we have next(y) $=$ `0000`.

14.2 The binary numerals

Any integer can be represented as a binary string by using base 2 numerals. In this binary representation we do not represent numbers using powers of 10 but as powers of 2, so the only digits we need are `0` and `1`.

For instance

$$23 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

which means that the binary representation of 23 is `10111`$\dot{\text{W}}$e denote the binary representation of n by binary(n).

Exercise 1. Find binary(x) where x is

- 32
- 17
- 119
- 204

How many binary digits will we need to write down binary(n)? We need as many as the least power of 2 that is greater than n. This is the *base-2 logarithm of* n.

Definition 7. Let $m > 0$. The base 2 logarithm of m is defined by

$$\log_2 m = x$$

where $2^x = m$.

We have that

$$\ln m = \ln(2^{\log_2 m})$$

and therefore

$$\log_2 m = \frac{\ln m}{\ln 2} \tag{14.1}$$

Exercise 2. Look at the answers you gave in Exercise 1 and check, using (14.1) that the number of digits was correct.

14.3 Programs

The theory of data compression considers algorithms. I will here only consider algorithms written in a very simple programming language which nevertheless (this will not be proved here) is sufficient to describe any computable function over the naturam numbers.

A program in our language always receives a string as input. We shall assume that strings are binary strings and that numbers are represented as binary numerals. Within our actual program text, we represent numerals in decimal notation (i.e. in base 10).

Programs can use integer values, modify integer-valued variables and string-valued variables. We assume a small collection of functions on numbers and strings: *Integers* can be manipulated using addition, subtraction and multiplication. *Strings* can be concatenated, and for any string s we can find the string $\mathsf{next}(s)$.

The statements that we allow are

- Reading a text string from input. If the input consists of the streng `1101` the statement `read(u)` will ask for a text string from input and assign the value `1101` to the string-valued variable `u`.
- Reading an integer from input written in binary. In other words: to input the integer n we must supply the program with the string $\mathsf{binary}(n)$. In the input consists of the string `101` the statement `read(x)` will result in the integer-valued variable `x` being assigned the value 5, since $\mathsf{binary}(5) = \texttt{101}$.
- Writing a text string to output. For example `write(011)` will write the string `011` to output.
- Modifying the values of integer variables. For example

```
x := x+2
```

will increment the value of the variable `x` by 2.

- **repeat**-loops. For example the program

  ```
  x := 4;
  y := 2;
  r := 1;
  repeat
     r := r * y;
     x := x-1
  until x = 0
  ```

  will, upon termination, satisfy that the variable `r` holds the value 2^4, that the variable `x` holds the value 0 and the variable `y` holds the value 2.

- `if-then-else`-statements. For example we could write

  ```
  if x = 3 sthen
         write(0)
  else write(1)
  ```

- A program can call procedures. A procedure has a sequence of *arguments* and will return a value. For example we can write

  ```
  procedure square(x)
    y := x *x;
    return  y;
  end of procedure;

  z := 5;
  z := square(z);
  ```

  In this tiny program the variable `z` gets the value 25.

Exercise 3. Write a program that does the following

1. If the input is `0` the program computes the sum of the values of the integernvariables `x` and `y`. If the input is `1` the program computes the product of the values of the integernvariables `x` and `y`.

2. Print out the result in binary (such that if e.g. the result is 7, the program prints out `111`).

Exercise 4. Write a procedure `length` that computes the length of a binary streng. *Hint:* Use the `next`()-function and remember how many strings there are of length i.

14.4 Programs are binary strings

For the remainder of this text I shall make an important assumption that will life a lot easier: *A program is a binary string.* It is easy to encode each character in our usual set of characters as a sequence of `0`'s and `1`'s.

Exercise 5. Find out how to translate the program

```
x := 7;
x := x+2;
udskriv(1)
```

to a binary string. *Hint:* First find out how to translate each character.

Why is this such a good idea? Remember that we assume that the input of a program is a binary string. We can now speak of *programs that take programs as input.*.

The following program examples are written so as to be readable but we could easily translate them to binary strings, if need be.

14.5 Data compression

A data compression algorithm is a *function* over strings. It must satisfy some natural requirements.

Compression must not increase the size of data

First and foremost a compression function must not yield a string longer than the original one.

In other words we are only interested in functions f that satisfy the condition

$$|f(s)| \leq |s|$$

for every string s.

Compression must not lead to information loss

We distinguish between *lossy* and *lossless* data compression. In the case of lossy compression, the result of applying the compression function and subsequently uncompressing the result will in general not give us back the original data – information is lost. Lossy compression is well-suited for compressing complicated data that human senses are only able to distinguish up to a certain levelas pictures and sound. The MPEG and MP3 data formats make use of lossy compression.

On the other hand, it is of course not acceptable to lose information when compressing a written text. In the following we shall only consider lossless compression .

Decompression must always be well-defined

A compressed datum must of course be decompressable to one and only one original piece of text. Therefore, the compression function must be 1-1.

Definition 8. Let f be a function with domain A and range B. f is 1-1 (injective) if we have for all $x, y \in A$ that if $f(x) = f(y)$ then also $x = y$.

Definition 9. Let f be a 1-1 function with domain A and range B. If for some $z \in B$ we have that there exists an $x \in A$ such that $f(x) = z$, we denote this x by $f^{-1}(z)$ and refer to x as the *inverse image* of z.

Compression must be computable

Finally, we require that a compression function f must be computable in the sense that there exists a program that for any input string w will write the string $f(w)$ to output and terminate.

All of these lead to the following.

Definition 10. A compression function is a computable, 1-1 function f over strings such that $|f(s)| \leq |s|$.

14.6 Kolmogorov complexity

The underlying intuition of the Kolmogorov theory of information is that we can measure the information of a text string s as the *length of the shortest description of* s. In this setting a description is a program with an associated input.

Definition 11. Let x be a binary string. A *description* of x is a pair $\langle P, w\rangle$ where P is a program and w is string such that P, when run with input w, will write x and nothing else to output and then terminate. The length of $\langle P, w\rangle$ is $|P| + |w|$.

Note the Definition 11 tells us that *distinct strings must have distinct descriptions.*

Example 9. Let s be the string `010101010101`. The program

```
t := 1;
read(x);
repeat
   x := x01
   t := t+1
until t = 6;
write(x)
```

and the input string `01` are a description of s.

Definition 12. Let x be a binary string. The *minimal description* of x, written $d(x)$ is the shortest description of x[1]

Example 10. The program

```
write(1)
```

together with the empty input string form a minimal description of the string `1`.

One can in general have more than one minimal description of a string; for this reason we do not speak of *the* minimal description.

Definition 13. Let x be a string. *The Kolmogorov complexity* of x, denoted $\mathrm{K}(x)$, is the length of a minimal description of x. That is,

$$\mathrm{K}(x) = |d(x)|$$

The Kolmogorov complexity of a string is at most the length of a string plus a constant whose value does not depend on the string.

Theorem 2. *There exists a constant c such that for every x we have that $K(x) \leq |x| + c$.*

[1]If there are several possible descriptions, we choose the pair that is lexicographically smallest.

Proof. The program P_{ident}:

```
read(x);
write(x)
```

reads a string from input and outputs it. The pair $\langle P_{\text{ident}}, x\rangle$ is therefore a description of x, and we can choose c to be the length of P. □

The following result tells us that there exist strings of arbitrarily high Kolmogorov complexity.

Theorem 3. *For every natural number c there exists a string s such that $K(s) \geq c$.*

Proof. Let c be an arbitrary natural number. We know that there are at must $B = \sum_{i=0}^{c} 2^i$ strings of length less than our equal to c.

Since this means that there can be at most B different descriptions of length at most c, and since there are infinitely many strings, some string must have a Kolmogorov complexity at least c. □

14.7 Some strings cannot be compressed

How is data compression related to Kolmogorov complexity? The connection is this: Data compression has as its goal to represent a string s by a shorter string that contains the same information as s – in other words, a shorter description of s.

Intuitively a string s can only be compressed if we can find a description of s that is shorter than s, i.e. if $\mathrm{K}(s) < |s|$.

Definition 14. Let $c > 0$ be a natural number. A string s is *c-compressible*if we have that

$$\mathrm{K}(s) \leq |s| - c$$

If s cannot be compressed by at least c characters, we say that s is *c-uncompressible.* If s is even 1-uncompressible, we say that s is *uncompressible.*

Theorem 4. *There exist uncompressible binary strings of every length.*

Proof. We show that for any given n there exists an uncompressible string of length n. First we recall that there are 2^n binary strings of length n.

Let S be the number of distinct descriptions whose length is less than n. We have that

$$S = \sum_{i=0}^{n-1} 2^i = 1 + 2 + 4 + 8 + \ldots + 2^{n-1} \tag{14.2}$$

If we multiply by 2 on both sides of (14.2) we get

$$2S = 2\sum_{i=0}^{n-1} 2^i = 2 + 4 + 8 + 16 \ldots + 2 \cdot 2^{n-1}$$

But from this we see that

$$\begin{aligned} 2S &= 2\sum_{i=0}^{n-1} 2^i = \underbrace{2 + 4 + 8 + 16 \ldots + 2 \cdot 2^{n-2}} + 2 \cdot 2^{n-1} && (14.3) \\ &= \left(\sum_{i=0}^{n-1} 2^i\right) + 2^n - 1 && (14.4) \\ &= S + 2^n - 1 && (14.5) \end{aligned}$$

If we subtract S we see that

$$S = 2^n - 1 \tag{14.6}$$

So there can be at most $2^n - 1$ descriptions of length $< n$, but there are 2^n strings of length n. This means that there has to exist a string of length 2^n whose description has length n or more. □

The consequence of Theorem 4 is that no algorithm for data compress will be able to compress every string.

14.8 K(s) is not computable

It would be nice if we could devise a program that could compute K(s) for a given string s. K(s) tells us the length of the shortest description of s and therefore tells us how much we can ever hope to compress s. If we could compute $K(s)$, this would tell us if there any point at all to using a compression algorithm.

But we cannt even find out if $K(s)$ is less than a given number k. This result is a direct consequence of a version of *Berry's paradox*.

Berry's paradox is due to the Welsh mathematician and philosopher Bertrand Russell who named it after a librarian at Oxford University.

The paradox goes as follows. Consider this description of a natural number:

> The least integer that cannot be described using fewer than twenty-four syllables.

If you count the syllables in the above description you will notice that it uses twenty-three syllables! Still, it describes a number that cannot be described using fewer than twenty-four syllables.

Theorem 5. *There is no program that can take a string s and a number k as input and correctly tell us if $K(s) \geq k$.*

Proof. Assume that we had such a program P that could provide us with the correct answer for every string s and number k. But then we could use P as a sub-program of the following program P_1:

```
subprogram p(s,k)
...
end of subprogram;
read(n)
w := "";
repeat
  if p(w,n) is true then
     return(w)
  else
     w := next(w)
until p(w,n) is true;
return(w)
```

P_1 takes the string binary(n) for a number n and examines all strings in lexicographic order until it discovers a string w such that $\mathrm{K}(w) \geq n$. Such a string must exist; we know this from Theorem3.

If we give P_1 the number n we see that $\langle P_1, \text{binary(n)}\rangle$ has length $|P_1| + \log_2 n$.

We know that $n > \log_2 n$ for all $n > 1$, so we can find a number N such that

$$N > |P_1| + \log_2 N$$

Let N be the least such N. But $\langle P_1, \text{binary(N)}\rangle$ is a program of length $|P_1| + \log_2 N$ that returns a string s where $\mathrm{K}(s) \geq N$. In other words $\langle P_1, \text{binary(N)}\rangle$ is a description of this s. But this contradicts the claim that $\mathrm{K}(s) \geq N$, which says that the shortest description of s has length at least N.

The only possible conclusion is that the program P cannot exist. □

Why does this tell us that we cannot compute $K(s)$? The answer is simple: If we had a program for computing $K(s)$, we would immediately know if $\mathrm{K}(s) \geq n$.

14.9 References

[1] G.J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13(4):547-569, October 1966.

[2] A. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Inf. Trans.*, 1 (s. 1–7), 1965.

[3] `http://www.kolmogorov.com`

[4] MPEG. `http://www.mpeg.org` . Set 1. november 2011.

[5] Audio & Multimedia MPEG Audio Layer-3. `http://www.iis.fraunhofer.de/amm/techinf/layer3/` Set 1. november 2011.

[6] M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing 1997.

[7] R.J. Solomonoff. A Preliminary Report on a General Theory of Inductive Inference Report V-131, Zator Co., Cambridge, Mass, February 1960.

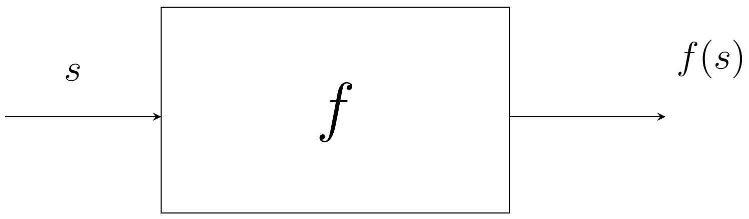


Figure 14.1: A compression function f creates the string $f(s)$ from the input string s