Affine transformations

A brief encounter with linear algebra

DSHEAL
Norman Juchler

Life Sciences and Facility Management
Institute of Computational Life Sciences

Introduction

In image processing, it is common to geometrically transform images through operations such as scaling, rotating, flipping, shifting and others. These transformations are instrumental in manipulating the geometric properties of an image.

The geometric transformation of an image involves the mapping of pixel coordinates $x = (x_0, x_1)^T$ from the input image to the corresponding pixel $y = (y_0, y_1)^T$ in the output image. Formally, this transformation can be expressed as a function f: $y = f(x)$. Specifically, we are dealing here with an important subset of transformations known as *affine transformations*. We will see that these transformations can be mathematically described using linear algebra, taking the concise form: $y = A \cdot x$.

Detail: Note that throughout this primer, we disregard the fact that the pixel coordinates take only integral values $x_0, x_1, y_0, y_1 \in \mathbb{N}_0$. Instead, we assume that $x_0, x_1, y_0, y_1 \in \mathbb{R}$. In the context of image processing, we are always able to "reach" the next pixel position by rounding a value to the next whole number.

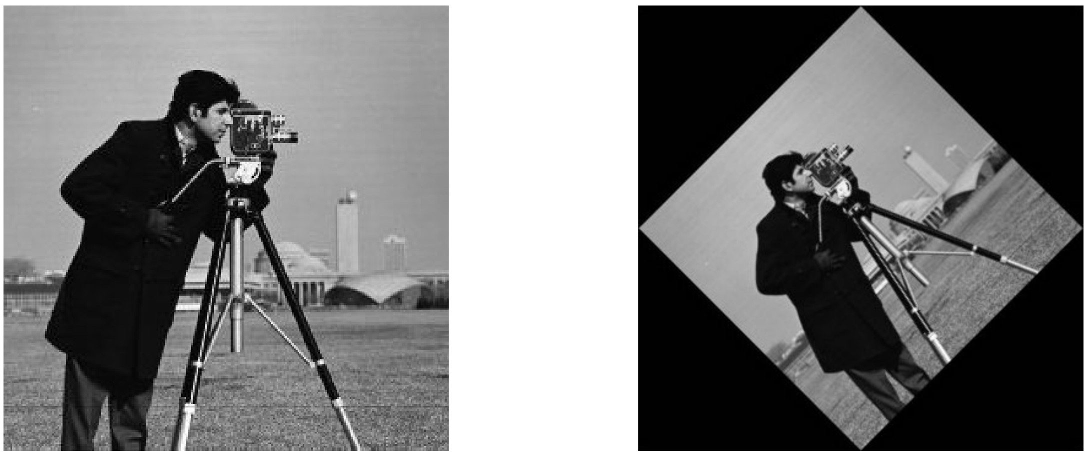


Figure 1: Example of a geometric image transformation: The image on the right is created by first scaling the input image (left) by a factor of $\frac{1}{\sqrt{2}}$ and then rotating it counterclockwise by 45°. The center of the image is the reference point for both transformations.

Linear systems of equations and matrices

In simple terms, a matrix is a rectangular array of numbers, arranged in rows and columns. Each number in a matrix is called an element. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

has two rows and three columns. The element in the first row, second column is 2 (denoted as a_{01}), and the element in the second row, third column is 6 (denoted as a_{12}):

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}$$

We can use matrices to express linear systems of equations in condensed form. Consider the following example:

$$\begin{vmatrix} y_0 = a_{00}x_0 + a_{01}x_1 + a_{02}x_2 \\ y_1 = a_{21}x_0 + a_{22}x_1 + a_{23}x_2 \end{vmatrix} \tag{1}$$

This exemplary system can be represented in matrix form as $y = A \cdot x$:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \tag{2}$$

In this notation, it holds that

- $A \in \mathbb{R}^{3\times 3}$ is the coefficient matrix, containing the coefficients a_{ij} of the variables
- $x \in \mathbb{R}^3$ is the column vector of variables $(x_0, x_1, x_2)^T$
- $y \in \mathbb{R}^2$ is the column vector of constants $(y_0, y_1)^T$

Note: To refer to row i of A, we write $a_{i,:}$ or $A(i,:)$ – and $a_{:,j}$ or $A(:,j)$ for column j. The use of ":" reminds us of the slice operator that we would use in Python for arrays.

Besides collecting the numbers in matrices and vectors, the notation also implies a calculation scheme (dt.: *Rechenschema*): The multiplication of a matrix A with a column vector x is defined as:

- (Step 1): Row-wise multiplication of the elements of A with the elements of x
- (Step 2): Subsequent summation of the resulting products

We can express this also very compactly using the sum notation:

$$y_i = A(i,:) \cdot x = \sum_{j=0}^{N} a_{ij} \cdot x_j \ , \quad i \in \{0, 1\}$$

If we apply this scheme, we indeed recover the original system of equations (1):

$$y_0 = A_{0,:} \cdot x = a_{00}x_0 + a_{01}x_1 + a_{02}x_2$$
$$y_1 = A_{1,:} \cdot x = a_{10}x_0 + a_{11}x_1 + a_{12}x_2$$

Notes:

- In the above formulas, we employ zero-based indices, meaning that we start counting the indices at zero: $x = (x_0, x_1, x_3)^T$. This choice corresponds to the convention used in Python and other programming languages, where vectors and matrices also use zero-based indices. Conversely, in mathematics and certain other coding languages such as R, it is common to use indices that start with the value 1: $x = (x_1, x_2, x_3)^T$. This is always a source of confusion! Here we use the computer scientist notation.
- With the operator ^{T}, we can *transpose* (flip) of a vector or matrix. For example:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^T = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}^T = \begin{pmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \\ a_{02} & a_{12} \end{pmatrix}$$

- We used matrix notation in equation (2) to multiply a matrix with a column vector. We can generalize this scheme to *multiply two matrices* A and B.

$$A \cdot B = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = C$$

$$C(i,j) = A(i,:) \cdot B(:,j)$$

  As an example (see colored elements), by using the above scheme of multiplying and summing row $A(0,:)$ with column $B(:,1)$, we compute matrix element $C(0,1) = c_{01}$. Note that the matrices must have matching sizes for this to work!

There is a lot more to say about matrices. The mathematical discipline that studies the properties of matrices and related topics is called *linear algebra*.

Transformation matrices

The linear system of equations (1) can be interpreted in the following manner: The values $x = (x_0, x_1, x_2)^T$ undergo a *mapping* or *transformation* to yield new values $y = (y_0, y_1, y_2)^T$. Matrix A comprises all parameters that define this *linear* transformation:

$$y = f(x) = A \cdot x$$

In the context of image analysis, we work with 2D coordinates x and y: $x = (x_0, x_1)^T$ and $y = (y_0, y_1)^T$.

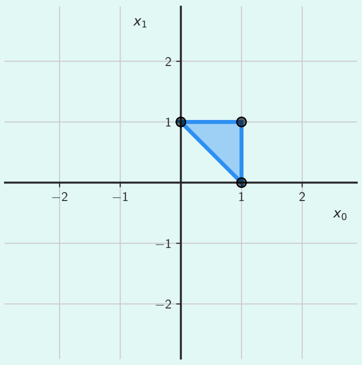
$$\begin{vmatrix} y_0 = a_{00}x_0 + a_{01}x_1 \\ y_1 = a_{21}x_0 + a_{22}x_1 \end{vmatrix}$$

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \tag{3}$$

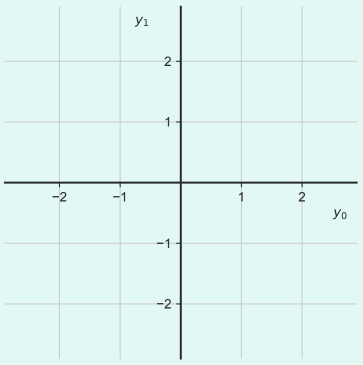
With this notation, we can specify different linear transformations.

Identity: (points are mapped onto each other)	$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Reflection: (vertical, about x-axis)	$A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Reflection: (horizontal, about y-axis)	$A = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
Scaling: (isotropic, about origin)	$A = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix}$
Scaling: (anisotropic, about origin)	$A = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$
Rotation: (around origin, about angle ϕ)	$A = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$

Exercise 1: For three points $x_A = (1, 0)$, $x_B = (1, 1)$ and $x_C = (0, 1)$, compute the transformed points y_A, y_B and y_C by evaluating equation (3) for any of the above transformation matrices. Visualize the results in a x-y diagram.



$$\Rightarrow$$
$$A \cdot x = y$$



Exercise 2: Observe that the translation is missing in the above list of transformations. Indeed, it is not possible to find a 2×2 matrix that represents a translational shift $t = (t_0, t_1)^T$ of vector x. Try to understand this limitation by looking at equation (1).

Affine transformations

Although we can represent a veritable zoo of different transformations using 2×2 matrices, the translation operation is still missing. To support also translations, we can extend the above linear transformation (3):

$$\begin{vmatrix} y_0 = a_{00}x_0 + a_{01}x_1 + t_0 \\ y_1 = a_{10}x_0 + a_{11}x_1 + t_1 \end{vmatrix} \tag{4}$$

$$y = A \cdot x + t$$

Technically, because of the additional term $+t$, this is no longer a linear transformation (but almost!). Mathematicians call this an *affine transformation*.

Take note that the parameters defining the transformation $y = f(x)$ are here embedded in two variables A and t. To revert the equation back to linear form $y = A \cdot x$, we can apply a small notational trick. If we extend the vectors x and y by a constant value 1,

$$x_H = (x_0, x_1, 1)^T$$
$$y_H = (y_0, y_1, 1)^T,$$

then the corresponding matrix notation will look as follows:

$$\begin{pmatrix} y_0 \\ y_1 \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} a_{00} & a_{01} & t_0 \\ a_{10} & a_{11} & t_1 \\ 0 & 0 & 1 \end{pmatrix}}_{A_H} \cdot \begin{pmatrix} x_0 \\ x_1 \\ 1 \end{pmatrix}$$

This is the *homogeneous* form of the affine transformation. Note how the translation parameters t_0, t_1 are now contained in the homogenous matrix A_H.

With this form, we can represent all affine transformations, including translation, with a single function: $y_H = A_H \cdot x_H$. Figure 2 summarizes the different elementary types of transformation that can be described using this function.

In image analysis, it is very common to represent affine transformations in this homogeneous form. In the remainder of this text, we will continue to use this form and, for the sake of simplicity, neglect the subscript $_H$.

Exercise 3: Take a close look at the transformations summarized in Figure 2. For every type of transformation, identify the parameters that define it. Also, make sure you understand that some transformations (like scaling and rotation) use the *origin* as a reference point.

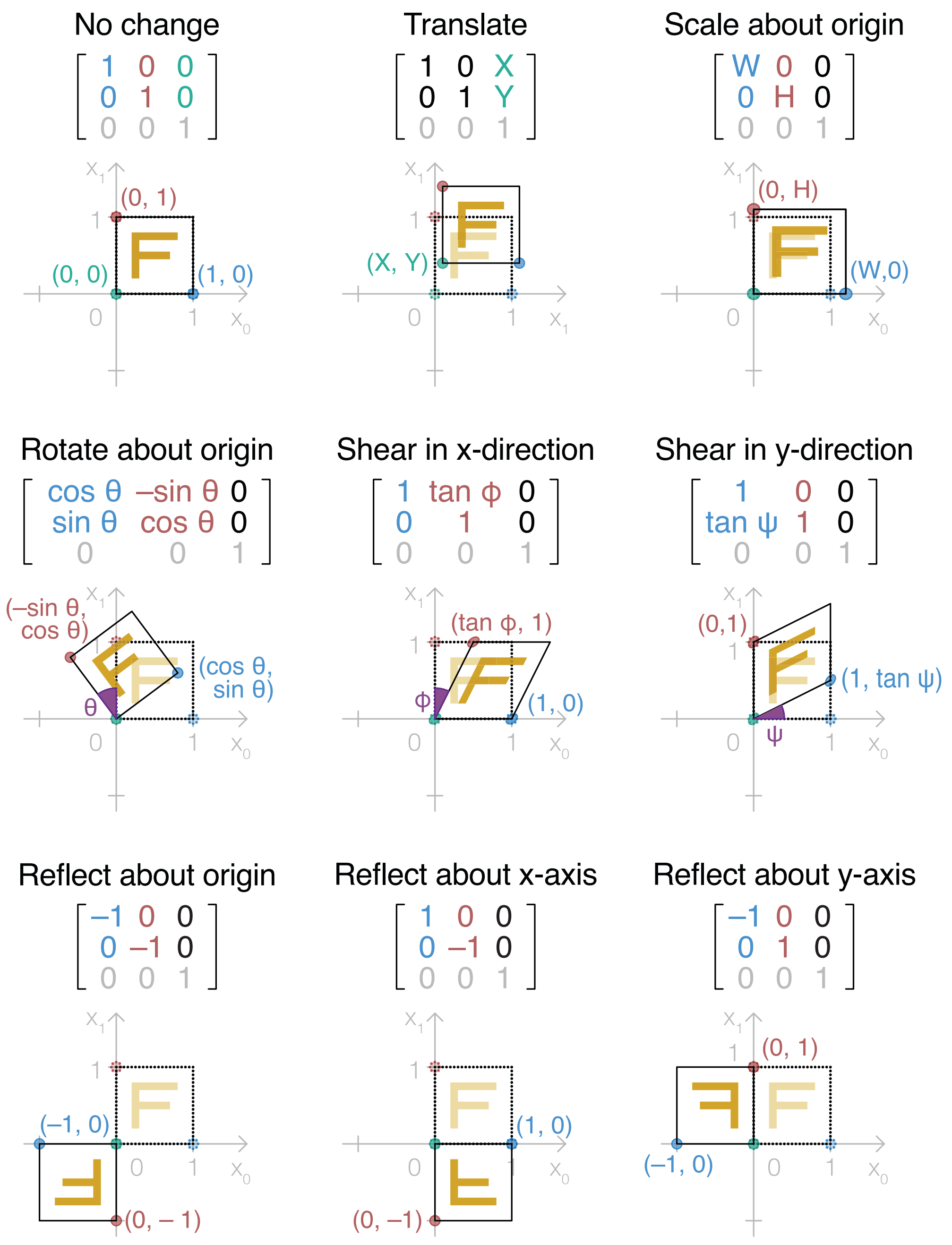

No change
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(0, 1)
(0, 0)
(1, 0)
Translate
$\begin{bmatrix} 1 & 0 & X \\ 0 & 1 & Y \\ 0 & 0 & 1 \end{bmatrix}$
(X, Y)
Scale about origin
$\begin{bmatrix} W & 0 & 0 \\ 0 & H & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(0, H)
(W,0)
Rotate about origin
$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(–sin θ, cos θ)
(cos θ, sin θ)
θ
Shear in x-direction
$\begin{bmatrix} 1 & \tan\phi & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(tan ϕ, 1)
(1, 0)
ϕ
Shear in y-direction
$\begin{bmatrix} 1 & 0 & 0 \\ \tan\psi & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(0,1)
(1, tan ψ)
ψ
Reflect about origin
$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(–1, 0)
(0, – 1)
Reflect about x-axis
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(1, 0)
(0, –1)
Reflect about y-axis
$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
(0, 1)
(–1, 0)
x_0
x_1

Figure 2: Overview of elementary affine transformations. Source, CC BY-SA 3.0

Compound transformations

A very useful property of homogeneous transformation matrices is that we can compose arbitrary affine transformations by sequentially applying multiple elementary transformations.

For example, if R represents a rotation, S an isotropic scaling, and T a translation matrix (see Figure 2), we can compute a compound transformation as follows:

$$y = T \cdot (S \cdot (R \cdot x)) \overset{*}{=} (T \cdot S \cdot R) \cdot x \tag{5}$$

Recall that this equation involves matrices. It requires a couple of extra steps to actually show that the equality marked with $\overset{*}{=}$ holds true. Here, we just assume it to be correct.

Using our basic understanding for matrix multiplications, we can compute the resulting matrix $A = T \cdot S \cdot R$ for this example:

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 & t_0 \\ 0 & 1 & t_1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \\ &= \begin{pmatrix} 1 & 0 & t_0 \\ 0 & 1 & t_1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s\cos\theta & -s\sin\theta & 0 \\ s\sin\theta & s\cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} s \cdot \cos\theta & -s \cdot \sin\theta & t_0 \\ s \cdot \sin\theta & s \cdot \cos\theta & t_1 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Note that it matters, in which order the transformations are applied! In the above example, we first rotate, then scale and finally translate (see equation (5)): $A = T \cdot S \cdot R$. If we apply a different order, for example, if we first translate, then scale and finally rotate, the compound transformation matrix B looks differently:

$$B = R \cdot S \cdot T = \begin{pmatrix} s \cdot \cos\theta & -s \cdot \sin\theta & s \cdot t_0 \cdot \cos\theta - s \cdot t_0 \cdot \sin\theta \\ s \cdot \sin\theta & s \cdot \cos\theta & s \cdot t_0 \cdot \sin\theta + s \cdot t_1 \cdot \cos\theta \\ 0 & 0 & 1 \end{pmatrix}$$

Exercise 4: Take a look at Figure 1, illustrating a concrete affine transformation. Using the elementary transformations of Figure 2, compute the compound transformation matrix A for that affine transformation. Assume the size of the image to be $s \times s$, and use the information provided of the figure caption.

Hint: You can rotate the image about any point $p = (p_0, p_1)^T$ by first translating the image such that p is mapped to the origin ($t = -p$). The you can apply the rotation, and finally move it back again to its original position ($t = +p$).

Inverse transformations

In some situations, we start with a pixel y in the transformed image and wonder what the corresponding pixel x was in the original image. For this, we need the inverse function:

$$x = f^{-1}(y)$$

For affine transformations of the form $y = A \cdot x$, with $A \in \mathbb{R}^{3\times3}$, the inverse transformation will be $x = A^{-1} \cdot y$, with $A^{-1} \in \mathbb{R}^{3\times3}$ being the *inverse matrix*.

For an affine transformation matrices A, it is (almost always) possible to compute the inverse matrix. It goes beyond the scope of this course to introduce how to invert a matrix. However, the tools we use (NumPy, OpenCV) offer the functionality to compute the inverse of a matrix:

- NumPy: `np.linalg.inv()`
- OpenCV: `cv.invertAffineTransform()`
- scikit-image: `skimage.AffineTransform.inverse()`

The following code listing demonstrates how to compute the inverse for an affine matrix in NumPy or OpenCV.

```
# Compute an affine transformation matrix
width, height = img.shape
A_2x3 = cv.getRotationMatrix2D(center=(width//2, height//2),
                               angle=60,
                               scale=1)

# Note: OpenCV omits the last row of A (which is always [0, 0, 1]).
#       For NumPy operations, we require the complete matrix.
A_3x3 = np.vstack([A_2x3, [0,0,1]])

# Invert with OpenCV
A_inv_cv = cv.invertAffineTransform(A_2x3)

# Invert with NumPy
A_inv_np = np.linalg.inv(A_3x3)
```

If we want to transform an input image I using some affine matrix A (see Figure 1 for an example), we use the inverse matrix A^{-1} to actually calculate the transformation. Functions such as `cv.warpAffine2D()` use a procedure similar to this:

- Create the *output* image J with shape (width, height)
- Iterate over all pixels y of the *output* image:
  - Apply the inverse transformation $x = A^{-1} \cdot y$ to find the position of pixel y in the input image I.
  - Sample the color/intensity at position x in the input image I and assign the value to pixel y in the output image J
  - Note: For the previous step, one may has to apply an interpolation scheme, as the position x generally will not fall exactly on a pixel position of the input image.

Perspective transformations

With transformations of the form $y = A \cdot x$, we can cover even more type of transformations. In fact, if we admit values other than $(0, 0, 1)$ in the last row, we are able to describe so called *perspective transformations*.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}, \quad a_{ij} \in \mathbb{R}$$

The details get a little more intricate for perspective transformations. We end this introduction by providing an example created with `cv.getPerspectiveTransform()` and `cv.warpPerspective()`.

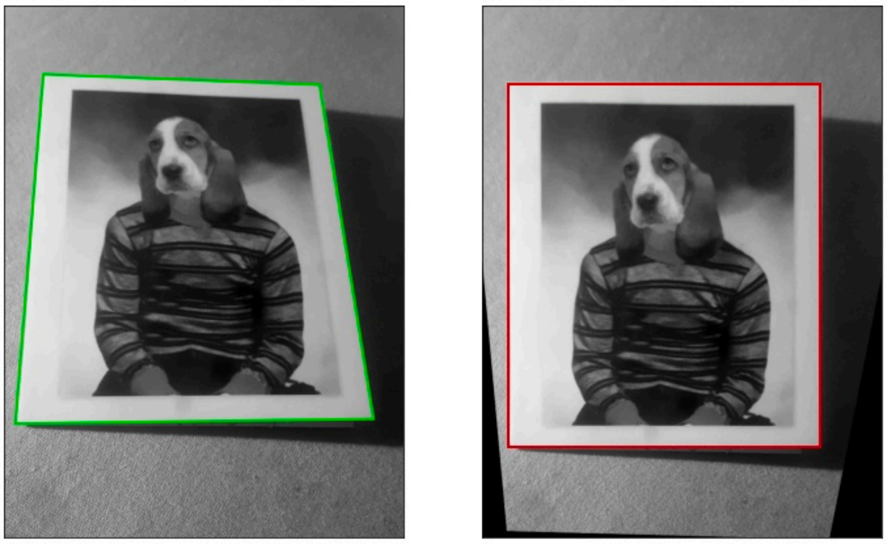


Figure 3: Example of a perspective image transformation: The image on the left is rectified using a perspective transformation such that the new image appears rectangular. Detail: The image shows a gift card that a 3-year-old kid chose for the author.

Further reading

- OpenCV tutorial on affine transformations. Link
- OpenCV tutorial on geometric transformations. Link