Lab: Interpolation and Warping

What you will learn

1. **Interpolation** as a way to calculate the image value 'inbetween pixels'.
2. **Geometrical transformations** (or warping) to change the 'shape' of an image.

Introduction

In many image processing applications you have access to a number of image transformation methods. These include rotating, stretching and even shearing an image.

These methods can be used to create some artistical effect, but they are also useful in correcting for some unwanted distortion, which were introduced, when the picture was taken (like camera rotation, perspective distortions etc.)

Image Warping

Optical Distorted Image

Corrected Image using Image Warping

The input for these methods (at least in this Exercise) is always a quadrilateral region of a discrete image F. That means that you only know the intensity values at discrete integral pixel positions. The output consists of a rectangular image G with axes aligned with the Cartesian coordinate axes x and y.

In an arbitrary image transformation, there is no guarantee, that an "input"-pixel will be positioned at a pixel in the "output" image as well. Rather, most of the time your output image pixels will "look at" image positions in the input image, which are "between" pixels.

So you need access to intensity values, which are not on the sampling grid of the original image, e.g. the intensitiy value at position $(6.4, 7.3)$

In this lab exercise you will examine and implement interpolation techniques, which solve this problem. Then you will implement geometrical image transformations, such as rotations, affine and perspecive transformations. These transformations require interpolation techniques.

Interpolation

The transformation algorithm needs a way to access the original image F in locations that are not on the sampling grid. We thus need a function `pV` (short for 'pixel value') that returns the value in the location (x, y) even for non-integer coordinate values.

Exercise Interpolation

Write a Python/Numpy function `pV` that returns the approximated value $f(x, y)$ for real valued x and y given the sampled image function F. The function could look like:

```
def pV(image, x, y, method):
  if inImage(image, x, y):
    # do interpolation
    return interpolatedValue
  else:
    # return a constant
    return constantValue
```

Implement nearest neighbor interpolation (`method = 'nearest'`) and bilinear interpolation (`method = 'linear'`) as are described in the lecture notes. Take the necessary steps to deal with the *border problem* (i.e. implement the function `inImage`). The simplest choice is to return a constant value in case (x, y) is not within the domain of the image.

Exercise Profile

Do a preliminary test on your function `pV` by using it in a profile function that samples the image in n equidistant points along a line from point (x_0, y_0) to point (x_1, y_1). The profile function is:

```
def profile(image, x0, y0, x1, y1, n, method):
  # profile of an image along line in n points
  return array( [ pV(image,x,y,method) for (x,y) in zip(linspace(x0,x1,n), linspace(y0,y1,n)) ])
```

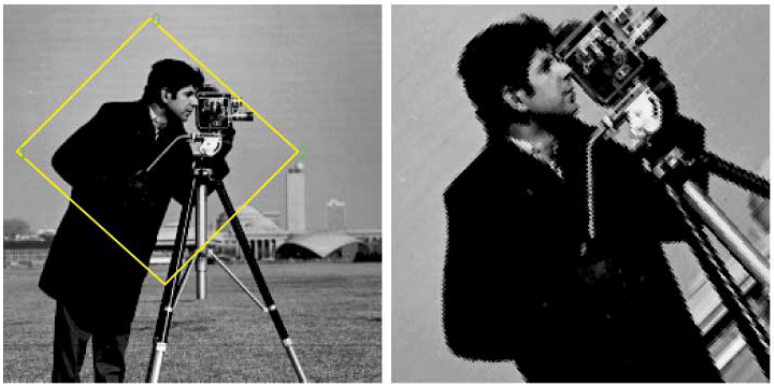
Test this function with:

```
a = imread('cameraman.tif')
plot( profile(a,100,100,120,120,'linear') )
```

Experiment a bit with a varying number of points on the profile line while keeping the begin and end point fixed. With many points on the line the difference between nearest neighbor interpolation and bilinear interpolation should become clearly visible.

Affine Geometrical Image Transform

We will use the affine transform to warp a parallelogram (a quadrilateral region in the input image) into another parallelogram (the entire resulting image).



An affine transform is described with:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\mathbf{x}' = A\mathbf{x}$$

where (x, y) are the coordinates in the input image and (x', y') are the coordinates of the point in the resulting image. Note that the affine transform is invertible

For the affine transform we need to know how three points map to their transformed points (in our case we take 3 corners from a parallelogram in the input image). Let (x_i, y_i) be a point in the input image with corresponding point (x_i', y_i') in the output image.

The relation between between these points and the affine transform matrix can be rewritten in the following form:

$$\begin{pmatrix} x_i' \\ y_i' \end{pmatrix} = \begin{pmatrix} x_i & y_i & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_i & y_i & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

The above is just for one point correspondence. For 3 point correspondences we can stack the x', y' values and also add two rows to the matrix on the right hand side. For three points we have:

$$\begin{pmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

or:

$$\mathbf{q} = M\mathbf{p}$$

The 6×6 matrix M is invertible (given three non collinear points) and thus:

$$\mathbf{p} = M^{-1}\mathbf{q}$$

In practice it is hard to define the three points with high accuracy, errors will be made. In that case the calculated transform matrix (represented now with the vector $\mathbf{p}$) can be inaccurate. If we are able to find more points correspondences we might be able to obtain a more accurate transform. Say we have n point correspondences. This leads to a vector $\mathbf{q}$ that has $2n$ elements and a M matrix that is $2n \times 6$ matrix. Then obviously we cannot simply invert the matrix M to obtain the transformation matrix characterized with parameter vector $\mathbf{p}$.

Instead we solve for a least squares solution for $\mathbf{p}$. For a detailed description we refer to Least Squares Estimators. Here we only give the solution:

$$\mathbf{p} = (M^T M)^{-1} M^T \mathbf{q}$$

Note that Numpy has a special function to solve a equation with one line of code:

```
p = lstsq(M,q)
```

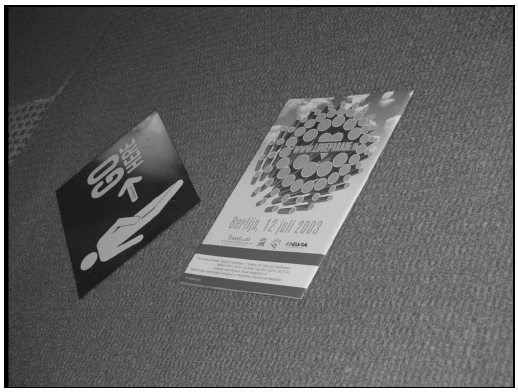
Exercise Affine Transformation

Write a function `affineTransform(image, x1, y1, x2, y2, x3, y3, M, N)` that warps the parallelogram defined by the three points (x1,y1), (x2,y2) and (x3,y3) onto a new image of size MxN.

Please implement the backward algorithm!

Perspective

A perspective transform maps a quadrilateral onto another quadrilateral. The flyer photographed while lying on the ground had no rectangular outline in the image (on the left). With a perspective transform we can warp the quadrilateral to the rectangle in the image on the right.



The perspective transform is the generic transform characterized with a 3x3 homogeneous matrix:

$$s\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\mathbf{x}' = P\mathbf{x}$$

Note the scaling factor s in the above expression which is due to the normalization that is inherent when using homogeneous coordinates (and a transform for which elements g and h are not zero).

The above can be rewritten as:

$$x' = \frac{ax + by + c}{gx + hy + i}$$
$$y' = \frac{dx + ey + f}{gx + hy + i}$$

Make sure you understand that this can be written as:

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y & -x' \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y & -y' \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Stacking 4 point correspondences we arrive at:

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1'x_1 & -y_1'y_1 & -y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2'x_2 & -x_2'y_2 & -x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_2'x_2 & -y_2'y_2 & -y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3'x_3 & -x_3'y_3 & -x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y_3'x_3 & -y_3'y_3 & -y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4'x_4 & -x_4'y_4 & -x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y_4'x_4 & -y_4'y_4 & -y_4' \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\mathbf{Mp} = \mathbf{0}$$

This is a homogeneous system of equations. It can be shown that in case the points are not colinear there is one non trivial solution for the vector $\mathbf{p}$.

In case we have more then 4 point correspondences (and we will when dealing with image mosaics) the matrix M is of size $2n \times 9$Then, due to noise in the measurements, there is little chance that there is any null vector. We then set out to find a vector $\mathbf{p}$ that minimizes $\|M\mathbf{p}\|$ subject to the constraint that $\|\mathbf{p}\| = 1$ (else the zero vector as trivial solution would suffice). So

Let UDV\T be the singular value decomposition of M, then:

\begin{split}\min \|UDV\T \v p\| \quad\text{s.t.}\quad \|\v p\| = 1\\ \min \|DV\T \v p\| \quad\text{s.t.}\quad \|\v p\| = 1\end{split}
due to the fact that U is orthogonal and therefore preserves the norm. Writing \v q = V\T \v p we have:

\begin{split}\min \|D \v q\| \quad\text{s.t.}\quad \|V \v q\| = 1\\ \min \|D \v q\| \quad\text{s.t.}\quad \|\v q\| = 1\\\end{split}
because V is also orthogonal.

Remember that D is a 'diagonal' matrix with the sorted singular values on the diagonal. Convince yourself that the optimal \v q then is q = (0 0 \cdots 0 1)\T. And thus \v p = V \v q which is the last column of V.

Exercise Perspective Transformation

Write a function `perspectiveTransform(image, x1, y1, x2, y2, x3, y3, x4, y4 M, N)` that warps the quadrilateral defined by the four points (x1,y1), (x2,y2), (x3,y3) and (x4,y4) onto a new image of size MxN.