Applied Computer Vision

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 6

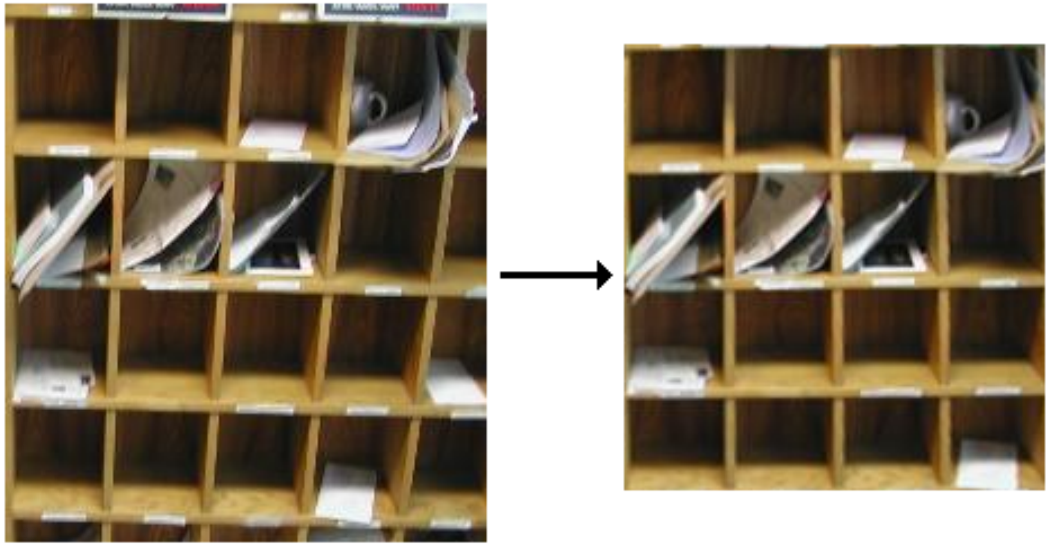
Image processing

Geometric operations

Geometric Operations

Change the spatial relationship between objects in an image

The relative distances between points a, b and c will typically be different after a geometric operation or warping



Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Geometric Operations

The applications of such warping include

- Geometric Decalibration

  the correction of geometric distortion introduced by the imaging system

- Image Registration

  the intentional distortion of one image with respect to another so that the objects in each image superimpose on one another

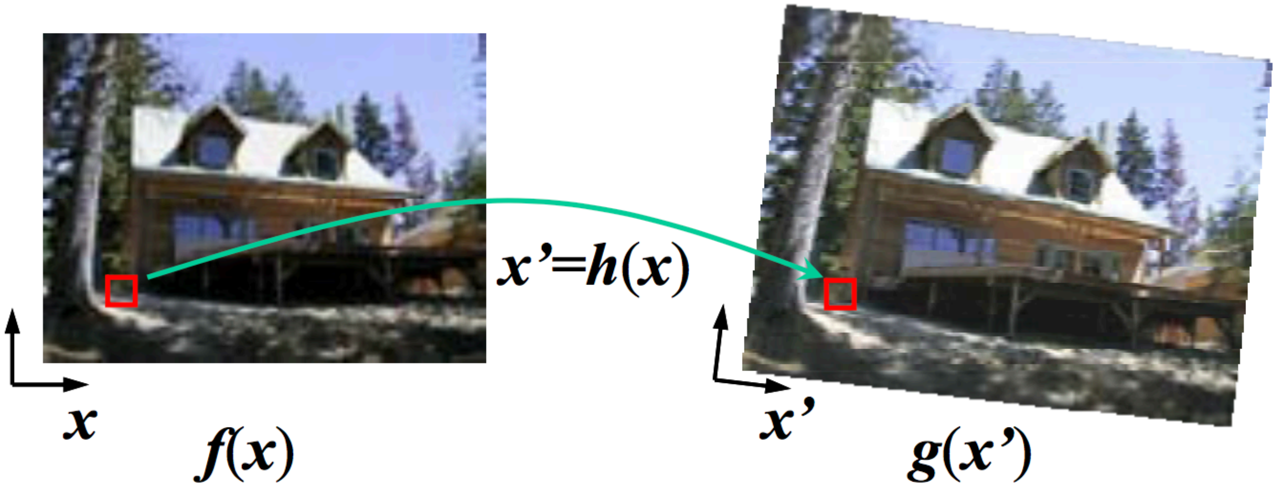
Geometric Operations

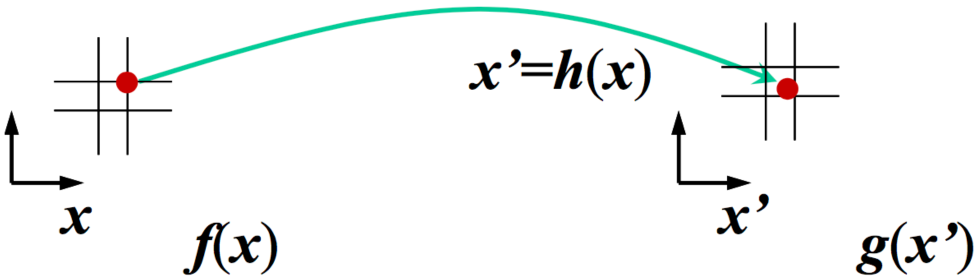
The approach to geometric image manipulation described here is called Spatial Warping and involves:

- the identification of a mathematical model of the required distortion
- its application to the image
- and the creation of a new corrected (decalibrated or registered) image

Geometric Operations

Pixel carry-over: forward mapping

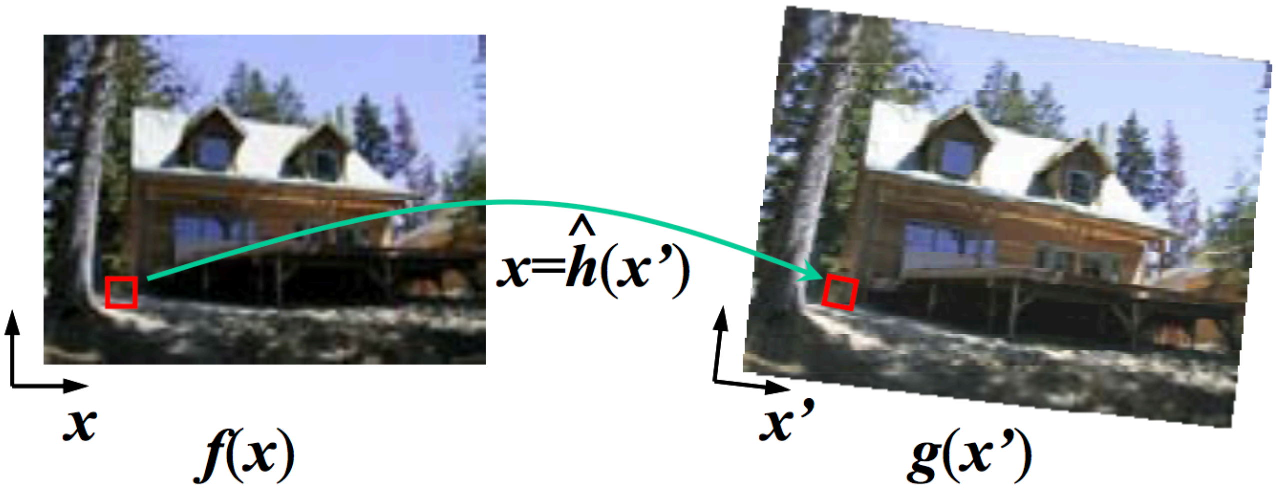


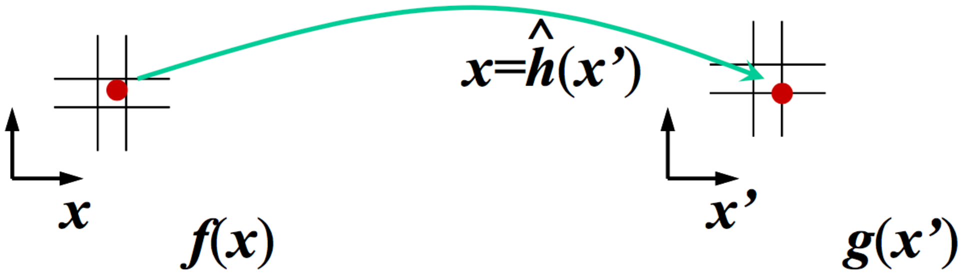


Credit: R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.

Geometric Operations

Pixel filling: reverse mapping





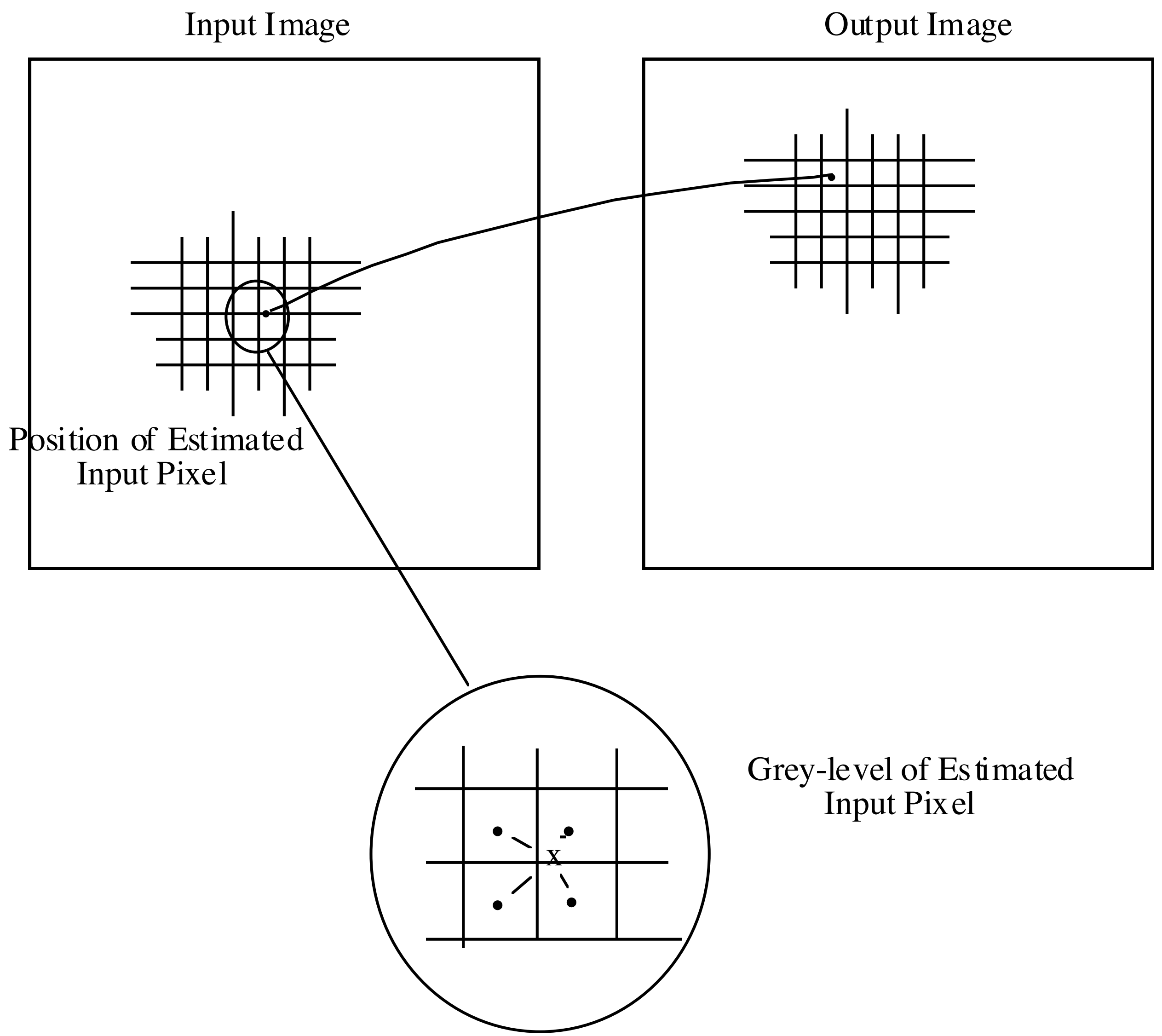
Credit: R. Szeliski, *Computer Vision: Algorithms and Applications,* Springer, 2010.

Geometric Operations

- The co-ordinates of input pixels yielded by the warping function will not correspond to exact (i.e. integer) pixel locations

- Need a method of estimating the grey-level of the output pixel when the "corresponding" pixel falls "between the integer co-ordinates"

Geometric Operations



Geometric Operations

The two requirements of geometric operations:

a) A spatial transformation which allows one to derive the position of a pixel in the input which corresponds to the pixel being “filled” or generated in the output

b) An interpolation scheme to estimate the grey level of this input pixel

Geometric Operations

The spatial transformation is expressed in general form as a mapping from a point (x, y) in the output image to its corresponding (warped) position (i, j) in the input image

$$(i, j) = \left(W_x(x, y), W_y(x, y)\right)$$

- the first co-ordinate, i, of the warped point is a function of the current position in the output
- likewise for the second co-ordinate, j.

Geometric Operations

Thus, given any point (x, y) in the output image,

the co-ordinates of the corresponding point in the input image may be generated using the warping functions

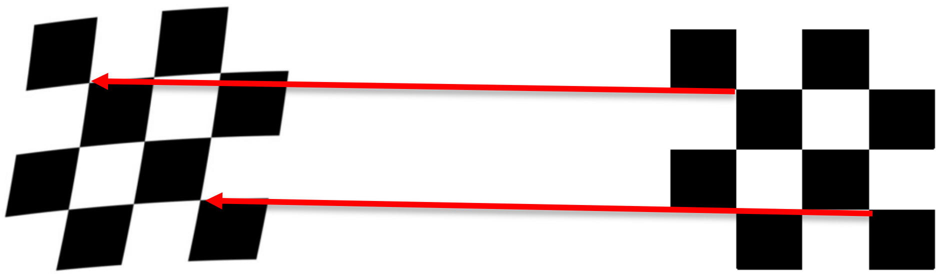
$$W_x \qquad\qquad W_y$$

Geometric Operations

The distortion may be specified by

- locating control points (also called fiducial points) in the input image (the image to be warped)

- identifying their corresponding control points in an ideal (undistorted or registered) image



Geometric Operations

The distortion model is then computed in terms of the transformation between these control points

- generating a spatial warping function
- which will allow one to build the output image pixel by pixel
- by identifying the corresponding point in the input image

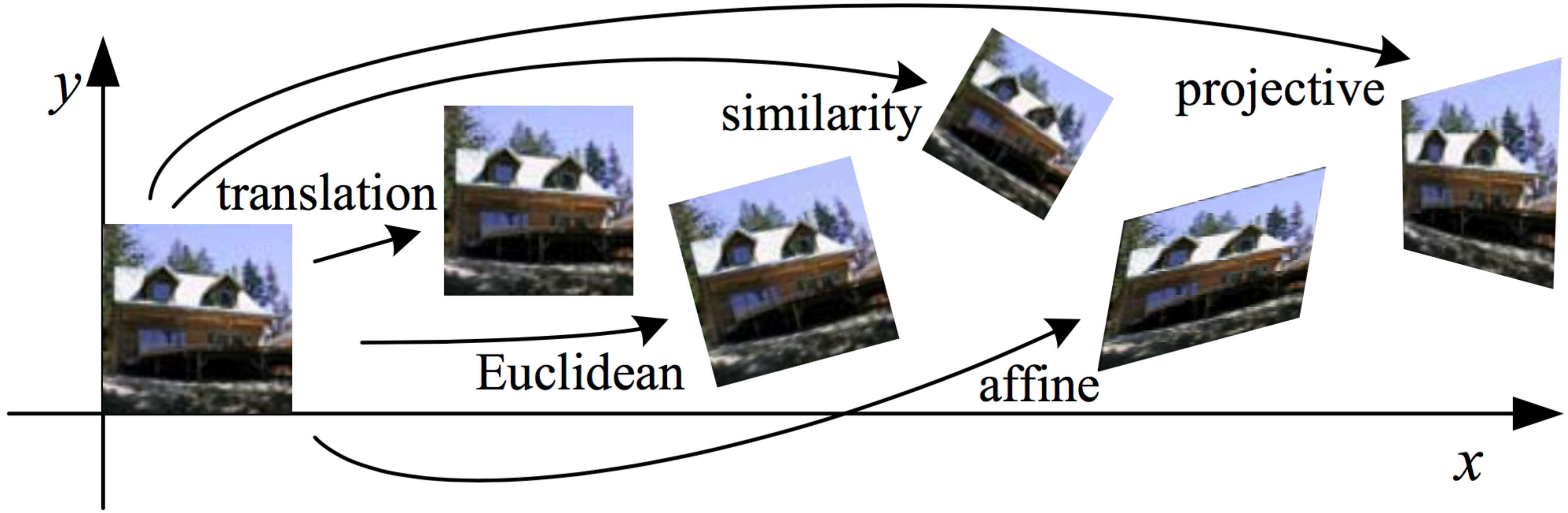
Geometric Operations

Application steps:

- Define the transformation
  - Known in advance, or
  - Determined through correspondences
    - Image to known pattern, or
    - Image to image
- Apply the transformation
  - For every point in the output image
    - Determine where it came from using W
    - Interpolate a value for the output point

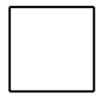
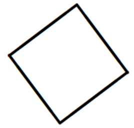
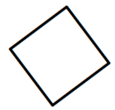
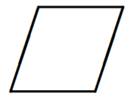
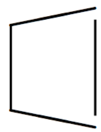
Geometric Operations

Different forms of distortion



R. Szeliski, *Computer Vision: Algorithms and Applications,* Springer, 2010.

Geometric Operations

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$	3	lengths	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$	4	angles	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times 3}$	8	straight lines	

Number of control points required to determine the transformation = [DoF / 2] since each point provides two constraints, one for x and one for y

Geometric Operations

If such transformations don't model the geometric distortion adequately, we can model each spatial warping function by a polynomial function

Geometric Operations

So, we assume that, for example, the warping functions are given by the following equations:

$$W_x(x, y) = \sum_p^n \sum_q^n a_{pq} x^p y^q$$

$$W_y(x, y) = \sum_p^n \sum_q^n b_{pq} x^p y^q$$

This gives us the (probably non-integer) coordinate i in the input image
(we are performing pixel-filling so we compute i from known integer coordinates x and y)

Note: it would have been better to use i and j subscripts instead of x and y in these equations to make it clear that the functions are used to compute the i and j coordinates, respectively

Geometric Operations

For example, if n=2
(which is adequate to correct for many distortions):

$$W_x(x,y) = a_{00}x^0y^0 + a_{10}x^1y^0 + a_{20}x^2y^0 + \\ a_{01}x^0y^1 + a_{11}x^1y^1 + a_{21}x^2y^1 + \\ a_{02}x^0y^2 + a_{12}x^1y^2 + a_{22}x^2y^2$$

$$W_y(x,y) = b_{00}x^0y^0 + b_{10}x^1y^0 + b_{20}x^2y^0 + \\ b_{01}x^0y^1 + b_{11}x^1y^1 + b_{21}x^2y^1 + \\ b_{02}x^0y^2 + b_{12}x^1y^2 + b_{22}x^2y^2$$

Geometric Operations

Now, the only thing that remains to complete the specification of the spatial warping function is to determine the values of these coefficients, *i.e.* to compute

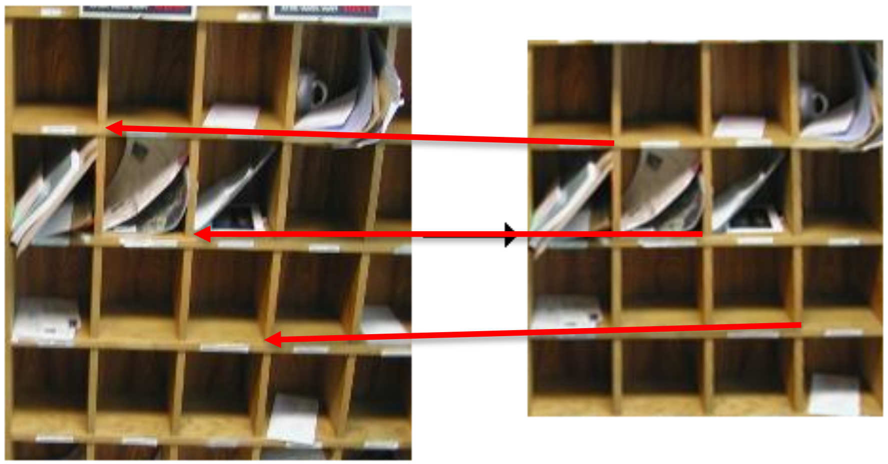
$$a_{00} - a_{22}$$

$$b_{00} - b_{22}$$

Geometric Operations

Assume we know the transformation exactly for a number of points (at least as many as the number of unknown coefficients; nine in this case):

i.e. assume we know the values of x and y and their corresponding i and j values



Geometric Operations

We then write the relationships explicitly in the form of the two equations above

We then solve these equations simultaneously to determine the value of the coefficients

There are two sets of simultaneous equations to set up

– one for the "a" coefficients

– one for the "b" coefficients

Geometric Operations

However, the same values relating x, y to i, j can be used in each case

This is now where the control points come in as we are going to use these to provide us with the (known) relationships between (x, y) and (i, j)

Geometric Operations

If we have nine unknown coefficients, then in order to obtain a solution we require at least nine such observations

$$\{(x_1, y_1), (i_1, j_1)\} \ldots \{(x_9, y_9), (i_9, j_9)\}$$

Such a system is said to be exactly determined

Geometric Operations

The solution of these exact systems are often ill-conditioned (numerically unstable)

It is usually good practice to over-determine the system by specifying more control points than you need (and hence generate more simultaneous equations)

Geometric Operations

The first point to note is that an over-determined system does not have an exact solution: there are likely to be some errors for some points

The idea, then, is to minimize these errors

Geometric Operations

The first point to note is that an over-determined system does not have an exact solution: there are likely to be some errors for some points

The idea, then, is to minimize these errors

We will use the common approach of minimizing the sum of the square of each error

i.e. to generate the least-square-error solution

Geometric Operations

Consider, again, a single control point and assume we are attempting to compute the a_{pq} coefficients:

$$\begin{aligned} i_1 = \; & a_{00}x_1^0 y_1^0 + a_{10}x_1^1 y_1^0 + a_{20}x_1^2 y_1^0 + \\ & a_{01}x_1^0 y_1^1 + a_{11}x_1^1 y_1^1 + a_{21}x_1^2 y_1^1 + \\ & a_{02}x_1^0 y_1^2 + a_{12}x_1^1 y_1^2 + a_{22}x_1^2 y_1^2 \end{aligned}$$

Geometric Operations

If we use *m* control points in total we will have *m* such equations which (noting that x^0 and y^0 are both equal to 1) we may write in matrix form as:

$$\begin{bmatrix} i_1 \\ i_2 \\ .. \\ i_m \end{bmatrix} = \begin{bmatrix} 1 & x_1^1 & x_1^2 & y_1^1 & x_1^1 y_1^1 & x_1^2 \; y_1^1 & y_1^2 & x_1^1 \; y_1^2 & x_1^2 \; y_1^2 \\ 1 & x_2^1 & x_2^2 & y_2^1 & x_2^1 y_2^1 & x_2^2 \; y_2^1 & y_2^2 & x_2^1 \; y_2^2 & x_2^2 \; y_2^2 \\ .. & & & & & & & & .. \\ 1 & x_m^1 & x_m^2 & y_m^1 & x_m^1 y_m^1 & x_m^2 \; y_m^1 & y_m^2 & x_m^1 \; y_m^2 & x_m^2 \; y_m^2 \end{bmatrix} * \begin{bmatrix} a_{00} \\ a_{10} \\ .. \\ a_{22} \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ .. \\ e_m \end{bmatrix}$$

Geometric Operations

We have to include the errors since there won't be a set of coefficients $a_{00} - a_{22}$

which will simultaneously provide us with exact $i_1 - i_m$ in the over-determined case

Geometric Operations

Let us abbreviate this matrix equation to :

$$i = Xa + e$$

Similarly

$$j = Xb + e$$

Geometric Operations

We require a so we might think of multiplying across by X^{-1} to obtain an appropriate expression

$$i = Xa \qquad \Rightarrow \qquad \begin{aligned} X^{-1} X a &= X^{-1} i \\ I a &= X^{-1} i \\ a &= X^{-1} i \end{aligned}$$

Unfortunately, X is non-square

Number of equations > number of coefficients

and one cannot invert a non-square matrix

Geometric Operations

Instead we use the pseudo-inverse of X

$$X^{\dagger} = \left(X^T X\right)^{-1} X^T$$

$$a = X^{\dagger} i$$

$$b = X^{\dagger} j$$

Geometric Operations

Aside: derivation of the pseudo-inverse … can be skipped

$$i = Xa + e$$
$$e = i - Xa$$

Forming the sum of the square of each error by computing

$$e^T e = (i - Xa)^T (i - Xa)$$

Geometric Operations

Differentiating $e^T e$ with respect to a to determine how the errors change as the coefficients change:

$$
\begin{aligned}
\frac{d\left(e^T e\right)}{d(a)} &= (0 - XI)^T (i - Xa) + (i - Xa)^T (0 - XI) \\
&= (- XI)^T (i - Xa) + \left(i^T - (Xa)^T\right)(- XI) \\
&= -IX^T (i - Xa) + \left(i^T - a^T X^T\right)(- XI) \\
&= -IX^T i + IX^T Xa - i^T XI + a^T X^T XI
\end{aligned}
$$

Geometric Operations

But noting that $i^T XI$ and $a^T X^T XI$ are 1 * 1 matrices and that the transpose of a 1 * 1 matrix is equal to itself, we transpose these two sub-expressions

$$\frac{d\left(e^T e\right)}{d(a)} = -IX^T i + IX^T Xa - IX^T i + IX^T Xa$$
$$= 2(I)\left(X^T Xa - X^T i\right)$$

Geometric Operations

The sum of the square of each error is minimized when $\dfrac{d\left(e^T e\right)}{d(a)}$ is equal to zero, thus:

$$0 = 2(I)\left(X^T X a - X^T i\right)$$

$$\left(X^T X\right)^{-1} X^T X a = \left(X^T X\right)^{-1} X^T i$$

$$a = \left(X^T X\right)^{-1} X^T i$$

Geometric Operations

Once the spatial mapping function has been found, the output image can be built, pixel by pixel and line by line

- The co-ordinates given by the warping function
  - denoting the corresponding points in the input image

  will not in general be integer values

- The grey-level must be interpolated from the grey-level of surrounding pixels

Geometric Operations

The simplest interpolation function

- “nearest neighbour” interpolation
  - the grey level of the output pixel (which is what we are trying to estimate)
  - is given by the grey-level of the input pixel which is nearest to the calculated point in the input image

Geometric Operations

101	100	103	105	107	105	103	110
110	140	120	122	130	130	121	120
134	134	135	131	137	138	120	121
132	132	132	133	133	150	160	155
134	140	140	135	140	156	160	174
130	138	139	150	169	175	170	165
126	133	138	149	163	169	180	185
130	140	150	169	178	185	190	200

Nearest Neighbour

Computed Point

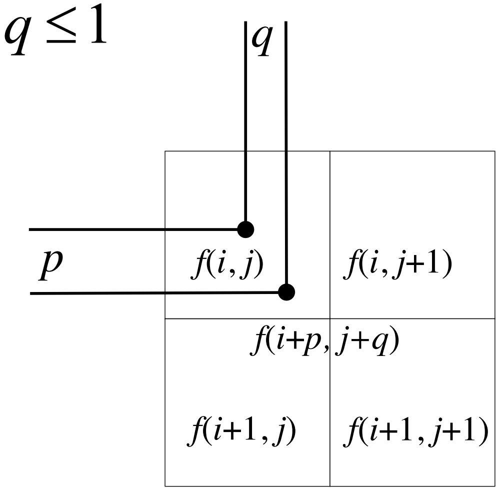
Geometric Operations

Better interpolation function

- bi-linear interpolation
  - estimate is made on the basis of four neighbouring input pixels
- bi-cubic interpolation
  - estimate is made on the basis of sixteen neighbouring pixels

Geometric Operations

Let the position of this point relative to pixel (i, j) be given by co-ordinates $(p,q);0 \leq p,q \leq 1$



The grey level at point (p, q) is constrained by the grey-level at the four neighbouring pixels and is a function of its position between these neighbours

Geometric Operations

To estimate the grey-level, $f(p, q)$, we fit a surface through the four neighbours' grey levels

- the equation of which will identify in general the grey-level at any point between the neighbours

The surface we fit is a hyperbolic paraboloid and is defined by the bilinear equation:

$$f(p, q) = ap + bq + cpq + d$$

There are four coefficients, a, b, c and d, which we must determine to identify this function for any given 2*2 neighbourhood in which we wish to interpolate

Geometric Operations

Thus we require four simultaneous equations in *a, b, c and d*;

These are supplied from our knowledge of the grey-levels at the four neighbours given by relative co-ordinates $(0, 0), (0, 1), (1, 0)$ and $(1, 1)$

Geometric Operations

Specifically, we know that :

$$a \text{ x } 0 + b \text{ x } 0 + c \text{ x } 0 \text{ x } 0 + d = f(i, j) \tag{4.1}$$

$$a \text{ x } 0 + b \text{ x } 1 + c \text{ x } 0 \text{ x } 1 + d = f(i, j+1) \tag{4.2}$$

$$a \text{ x } 1 + b \text{ x } 0 + c \text{ x } 1 \text{ x } 0 + d = f(i+1, j) \tag{4.3}$$

$$a \text{ x } 1 + b \text{ x } 1 + c \text{ x } 1 \text{ x } 1 + d = f(i+1, j+1) \tag{4.4}$$

Geometric Operations

Directly from (4.1), we have :

$$d = f(i, j) \tag{4.5}$$

Rearranging (4.2) and substituting for d, we have:

$$b = f(i, j+1) - f(i, j) \tag{4.6}$$

Rearranging (4.3) and substituting for d, we have:

$$a = f(i+1, j) - f(i, j) \tag{4.7}$$

Rearranging (4.4) and substituting for a, b and d, we have:

$$c = f(i+1, j+1) + f(i, j) - f(i+1, j) - f(i, j+1) \tag{4.8}$$

Geometric Operations

Equations (4.5) - (4.8) allow us to compute the coefficients a, b, c and d; which define the bilinear interpolation for a given 2*2 neighbourhood with known pixel grey-levels

For example, if the co-ordinates of the point at which we wish to estimate the grey-level are $(60.4, 128.1)$ and the grey-levels at pixels $(60, 128)$, $(60, 129)$, $(61, 128)$ and $(61, 129)$ are 10, 12, 14 and 15, respectively, then the grey level at this point, in relative co-ordinates, is given by:

$$
\begin{aligned}
f(0.4, 0.1) \quad &= \quad (14 - 10) * 0.4 + \\
&\quad\quad (12 - 10) * 0.1 + \\
&\quad\quad (15 + 10 - 14 - 12) * 0.4 * 0.1 + \\
&\quad\quad 10 \\
&= \quad 11.76
\end{aligned}
$$

Demos

The following code is taken from the geometricTransformation project in the lectures directory of the ACV repository

See:

```
geometricTransformation.h
geometricTransformationImplementation.cpp
geometricTransformationApplication.cpp
```

```
/*
  Example use of openCV to perform a geometric transformation

  We use a perspective transformation as an example
  The user must interactively select four control points by
  clicking on four consecutive pairs of source and target pixels
  ------------------------------------------------------------
  Implementation file

  David Vernon
  24 May 2017

*/
#include "geometricTransformation.h"

// Global variables to allow access by the display window callback functions
// specifically the mouse callback function to acquire the control point coordinates


Point2f source_points[4];
Point2f destination_points[4];
int     number_of_control_points;
char*   inputWindowName = "Input Image";
Mat     inputImage;  // strictly speaking we don't need this because we don't need to access the pixel values for this example
                     // however, we leave it in to show how this can be done
```

```
void geometricTransformation(char *filename) {

   char* outputWindowName = "Transformed Image";

   Mat inputImageCopy;
   Mat outputImage;
   Mat perspective_matrix( 3, 3, CV_32FC1 );

   bool debug = false;

   namedWindow(inputWindowName,   CV_WINDOW_AUTOSIZE);
   setMouseCallback(inputWindowName, getControlPoints);    // use this callback to get the coordinates of the four pairs of control points

   namedWindow(outputWindowName,  CV_WINDOW_AUTOSIZE);

   inputImage = imread(filename, CV_LOAD_IMAGE_UNCHANGED); // Read the file
   inputImageCopy = inputImage.clone();                    // make a copy of the input for warping

   if (!inputImage.data) {                                 // Check for invalid input
      printf("Error: failed to read image %s\n",filename);
      prompt_and_exit(-1);
   }

   printf("Clicking on four pairs of source and target pixels on the input image.\n");
   printf("When finished with this image, press any key to continue ...\n");

   /* display the input and a zero output */

   imshow(inputWindowName, inputImage );
   outputImage = Mat::zeros(inputImage.rows, inputImage.cols,inputImage.type());
   imshow(outputWindowName, outputImage);
   waitKey(30);
```

```
   /* now get four pairs of control points */

   number_of_control_points = 0;
   do{
      waitKey(30);
   } while (number_of_control_points < 8);

   if (debug) {
      for (int i=0; i<number_of_control_points/2; i++) {
         printf("%f %f - %f %f\n", source_points[i].x, source_points[i].y, destination_points[i].x, destination_points[i].y);
      }
   }

   perspective_matrix = getPerspectiveTransform( source_points, destination_points);
   warpPerspective(inputImageCopy, outputImage, perspective_matrix, outputImage.size() );

   imshow(outputWindowName, outputImage);

   do{
      waitKey(30);                                   // Must call this to allow openCV to display the images
   } while (!_kbhit());                              // We call it repeatedly to allow the user to move the windows
                                                     // (if we don't the window process hangs when you try to click and drag

   getchar(); // flush the buffer from the keyboard hit

   destroyWindow(inputWindowName);
   destroyWindow(outputWindowName);
}
```

```
void getControlPoints( int event, int x, int y, int, void* ) {

   extern char* inputWindowName;
   extern Mat   inputImage;
   extern Point2f source_points[4];
   extern Point2f destination_points[4];
   extern int number_of_control_points;

   int crossHairSize = 10;

   if (event != EVENT_LBUTTONDOWN) {
      return;
   }
   else {
      number_of_control_points++;

      if ((number_of_control_points %2) == 1) { // source  point

         line(inputImage,Point(x-crossHairSize/2,y),Point(x+crossHairSize/2,y),Scalar(0, 0, 255),1, CV_AA); // Red
         line(inputImage,Point(x,y-crossHairSize/2),Point(x,y+crossHairSize/2),Scalar(0, 0, 255),1, CV_AA);

      }
      else {                                    // target  point

         line(inputImage,Point(x-crossHairSize/2,y),Point(x+crossHairSize/2,y),Scalar(0, 255, 0),1, CV_AA); // Green
         line(inputImage,Point(x,y-crossHairSize/2),Point(x,y+crossHairSize/2),Scalar(0, 255, 0),1, CV_AA);
      }
```

```
      /* every alternate point goes in source and destination because we are specifying pairs */

      if ((number_of_control_points %2) == 1) {
         source_points[(number_of_control_points-1)/2].x = (float) x;
         source_points[(number_of_control_points-1)/2].y = (float) y;
      }
      else {
         destination_points[(number_of_control_points-1)/2].x = (float) x;
         destination_points[(number_of_control_points-1)/2].y = (float) y;
      }

      imshow(inputWindowName, inputImage); // show the image with the cross-hairs

      /* Example of the code required to access image values
      if (inputImage.type() == CV_8UC1) { // greyscale image
         printf("Input image value at coordinates (%d, %d): %d\n", x, y, inputImage.at<uchar>(y,x));              // note order of indices
      }
      else if  (inputImage.type() == CV_8UC3) { // colour image
         printf("Input image value at coordinates (%d, %d):RGB %d %d %d\n",
                x, y, inputImage.at<Vec3b>(y,x)[2], inputImage.at<Vec3b>(y,x)[1], inputImage.at<Vec3b>(y,x)[0]); //RGB & note order of indices
      }
      */
   }
}
```

Exercises

1. Read OpenCV documentation for all OpenCV functions in sample code

2. Study utility functions in sample code

Reading

R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.

Section 3.6 Geometric transformations