Geometric Transformations and Image Registration

We conclude this chapter with an introduction to geometric transformations for image restoration. Geometric transformations modify the spatial relationship between pixels in an image. They are often called *rubber-sheet transformations* because they may be viewed as printing an image on a sheet of rubber and then stretching this sheet according to a predefined set of rules.

Geometric transformations are used frequently to perform *image registration*, a process that takes two images of the same scene and aligns them so they can be merged for visualization, or for quantitative comparison. In the following sections, we discuss (1) spatial transformations and how to define and visualize them in MATLAB; (2) how to apply spatial transformations to images; and (3) how to determine spatial transformations for use in image registration.

Geometric Spatial Transformations

Suppose that an image, f, defined over a (w, z) coordinate system, undergoes geometric distortion to produce an image, g, defined over an (x, y) coordinate system. This transformation (of the coordinates) may be expressed as

$$(x, y) = T\{(w, z)\}$$

For example, if $(x, y) = T\{(w, v)\} = (w/2, z/2)$, the "distortion" is simply a shrinking of f by half in both spatial dimensions, as illustrated in Fig. 5.12.

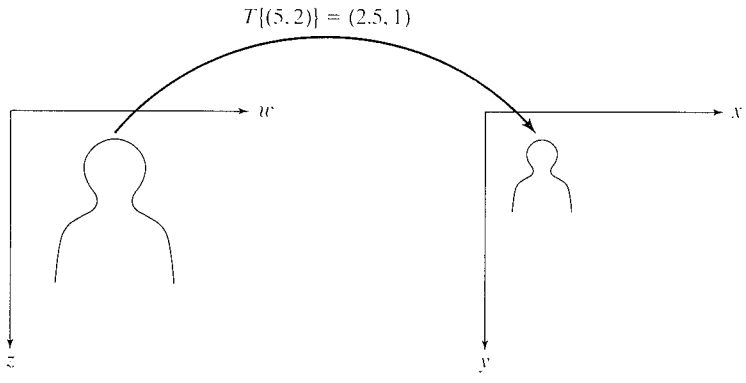


FIGURE 5.12 A simple spatial transformation. (Note that the xy-axes in this figure do not correspond to the image axis coordinate system defined in Section 2.1.1. As mentioned in that section, IPT on occasion uses the so-called spatial coordinate system in which y designates rows and x designates columns. This is the system used throughout this section in order to be consistent with IPT documentation on the topic of geometric transformations.)

One of the most commonly used forms of spatial transformations is the *affine transform* (Wolberg [1990]). The affine transform can be written in matrix form as

$$[x \quad y \quad 1] = [w \quad z \quad 1]\,\mathbf{T} = [w \quad z \quad 1]\begin{bmatrix} t_{11} & t_{12} & 0 \\ t_{21} & t_{22} & 0 \\ t_{31} & t_{32} & 1 \end{bmatrix}$$

This transformation can scale, rotate, translate, or shear a set of points, depending on the values chosen for the elements of **T**. Table 5.3 shows how to choose the values of the elements to achieve different transformations.

IPT represents spatial transformations using a so-called *tform structure*. One way to create such a structure is by using function `maketform`, whose calling syntax is

See Sections 2.10.6 and 11.1.1 for a discussion of structures.

```
tform = maketform(transform_type, transform_parameters)
```

TABLE 5.3
Types of affine transformations.

Type	Affine Matrix, T	Coordinate Equations	Diagram
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = z$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = s_x w$ $y = s_y z$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w\cos\theta - z\sin\theta$ $y = w\sin\theta + z\cos\theta$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w + \alpha z$ $y = z$	
Shear (vertical)	$\begin{bmatrix} 1 & \beta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = \beta w + z$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$	$x = w + \delta_x$ $y = z + \delta_y$	

The first input argument, `transform_type`, is one of these strings: `'affine'`, `'projective'`, `'box'`, `'composite'`, or `'custom'`. These transform types are described in Table 5.4, Section 5.11.3. Additional arguments depend on the transform type and are described in detail in the help page for `maketform`.

In this section our interest is on affine transforms. For example, one way to create an affine `tform` is to provide the **T** matrix directly, as in

```
>> T = [2 0 0; 0 3 0; 0 0 1];
>> tform = maketform('affine', T)
tform =
       ndims_in: 2
      ndims_out: 2
    forward_fcn: @fwd_affine
    inverse_fcn: @inv_affine
          tdata: [1 x 1 struct]
```

Although it is not necessary to use the fields of the `tform` structure directly to be able to apply it, information about **T**, as well as about $\mathbf{T}^{-1}$, is contained in the `tdata` field:

```
>> tform.tdata
ans =
       T: [3 x 3 double]
    Tinv: [3 x 3 double]
>> tform.tdata.T
ans =
    2   0   0
    0   3   0
    0   0   1
>> tform.tdata.Tinv
ans =
    0.5000         0         0
         0    0.3333         0
         0         0    1.0000
```

IPT provides two functions for applying a spatial transformation to points: `tformfwd` computes the forward transformation, $T\{(w, z)\}$, and `tforminv` computes the inverse transformation, $T^{-1}\{(x, y)\}$. The calling syntax for `tformfwd` is `XY = tformfwd(WZ, tform)`. Here, `WZ` is a $P \times 2$ matrix of points; each row of `WZ` contains the w and z coordinates of one point. Similarly, `XY` is a $P \times 2$ matrix of points; each row contains the x and y coordinates of a transformed point. For example, the following commands compute the forward transformation of a pair of points, followed by the inverse transform to verify that we get back the original data:

```
>> WZ = [1 1; 3 2];
>> XY = tformfwd(WZ, tform)
XY =
```

```
    2    3
    6    6

>> WZ2 = tforminv(XY, tform)
WZ2 =
    1    1
    3    2
```

To get a better feel for the effects of a particular spatial transformation, it is often useful to see how it transforms a set of points arranged on a grid. The following M-function, `vistformfwd`, constructs a grid of points, transforms the grid using `tformfwd`, and then plots the grid and the transformed grid side by side for comparison. Note the combined use of functions `meshgrid` (Section 2.10.4) and `linspace` (Section 2.8.1) for creating the grid. The following code also illustrates the use of some of the functions discussed thus far in this section.

vistformfwd

```
function vistformfwd(tform, wdata, zdata, N)
%VISTFORMFWD Visualize forward geometric transform.
%   VISTFORMFWD(TFORM, WRANGE, ZRANGE, N) shows two plots: an N-by-N
%   grid in the W-Z coordinate system, and the spatially transformed
%   grid in the X-Y coordinate system.  WRANGE and ZRANGE are
%   two-element vectors specifying the desired range for the grid. N
%   can be omitted, in which case the default value is 10.

if nargin < 4
   N = 10;
end

% Create the w-z grid and transform it.
[w, z] = meshgrid(linspace(wdata(1), zdata(2), N), ...
                  linspace(wdata(1), zdata(2), N));

wz = [w(:) z(:)];
xy = tformfwd([w(:) z(:)], tform);

% Calculate the minimum and maximum values of w and x,
% as well as z and y. These are used so the two plots can be
% displayed using the same scale.
x = reshape(xy(:, 1), size(w)); % reshape is discussed in Sec. 8.2.2.
y = reshape(xy(:, 2), size(z));
wx = [w(:); x(:)];
wxlimits = [min(wx) max(wx)];
zy = [z(:); y(:)];
zylimits = [min(zy) max(zy)];

% Create the w-z plot.
subplot(1,2,1) % See Section 7.2.1 for a discussion of this function.
plot(w, z, 'b'), axis equal, axis ij
hold on
plot(w', z', 'b')
hold off
xlim(wxlimits)
ylim(zylimits)
```

```
set(gca, 'XAxisLocation', 'top')
xlabel('w'), ylabel('z')

% Create the x-y plot.
subplot(1, 2, 2)
plot(x, y, 'b'), axis equal, axis ij
hold on
plot(x', y', 'b')
hold off
xlim(wxlimits)
ylim(zylimits)
set(gca, 'XAxisLocation', 'top')
xlabel('x'), ylabel('y')
```

EXAMPLE 5.12: Visualizing affine transforms using `vistformfwd`.

In this example we use `vistformfwd` to visualize the effect of several different affine transforms. We also explore an alternate way to create an affine `tform` using `maketform`. We start with an affine transform that scales horizontally by a factor of 3 and vertically by a factor of 2:

```
>> T1 = [3 0 0; 0 2 0; 0 0 1];
>> tform1 = maketform('affine', T1);
>> vistformfwd(tform1, [0 100], [0 100]);
```

Figures 5.13(a) and (b) show the result.

A shearing effect occurs when t_{21} or t_{12} is nonzero in the affine **T** matrix, such as

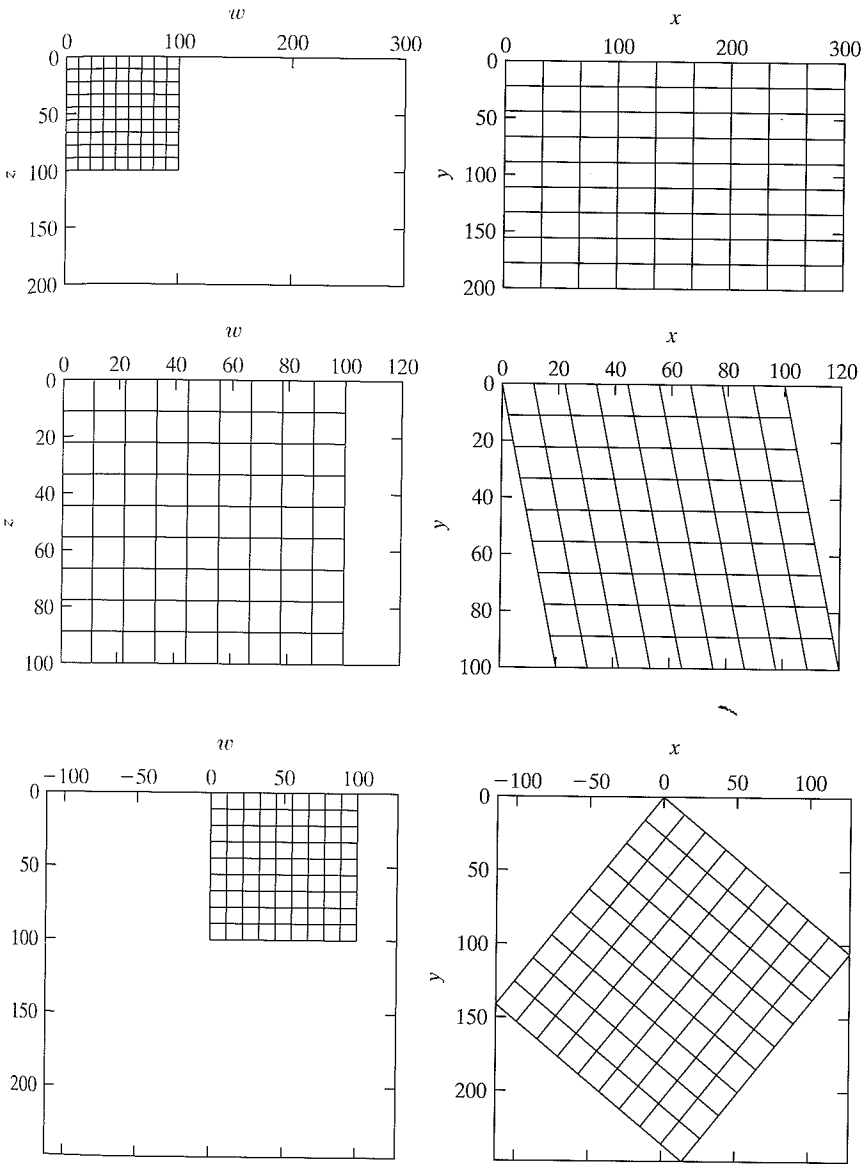
```
>> T2 = [1 0 0; .2 1 0; 0 0 1];
>> tform2 = maketform('affine', T2);
>> vistformfwd(tform2, [0 100], [0 100]);
```

Figures 5.13(c) and (d) show the effect of the shearing transform on a grid.

An interesting property of affine transforms is that the composition of several affine transforms is also an affine transform. Mathematically, affine transforms can be generated simply by using multiplication of the **T** matrices. The next block of code shows how to generate and visualize an affine transform that is a combination of scaling, rotation, and shear.

```
>> Tscale = [1.5 0 0; 0 2 0; 0 0 1];
>> Trotation = [cos(pi/4) sin(pi/4) 0
               -sin(pi/4) cos(pi/4) 0
                0 0 1];
>> Tshear = [1 0 0; .2 1 0; 0 0 1];
>> T3 = Tscale * Trotation * Tshear;
>> tform3 = maketform('affine', T3);
>> vistformfwd(tform3, [0 100], [0 100])
```

Figures 5.13(e) and (f) show the results.



a b
c d
e f

FIGURE 5.13 Visualizing affine transformations using grids. (a) Grid 1. (b) Grid 1 transformed using `tform1`. (c) Grid 2. (d) Grid 2 transformed using `tform2`. (e) Grid 3. (f) Grid 3 transformed using `tform3`.

5.11.2 Applying Spatial Transformations to Images

Most computational methods for spatially transforming an image fall into one of two categories: methods that use *forward mapping*, and methods that use *inverse mapping*. Methods based on forward mapping scan each input pixel in turn, copying its value into the output image at the location determined by $T\{(w, z)\}$. One problem with the forward mapping procedure is that two or more different pixels in the input image could be transformed into the same pixel in the output image, raising the question of how to

combine multiple input pixel values into a single output pixel value. Another potential problem is that some output pixels may not be assigned a value at all. In a more sophisticated form of forward mapping, the four corners of each input pixel are mapped onto quadrilaterals in the output image. Input pixels are distributed among output pixels according to how much each output pixel is covered, relative to the area of each output pixel. Although more accurate, this form of forward mapping is complex and computationally expensive to implement.

IPT function `imtransform` uses inverse mapping instead. An inverse mapping procedure scans each output pixel in turn, computes the corresponding location in the input image using $T^{-1}\{(x, y)\}$, and interpolates among the nearest input image pixels to determine the output pixel value. Inverse mapping is generally easier to implement than forward mapping.

The basic calling syntax for `imtransform` is


```
g = imtransform(f, tform, interp)
```

where `interp` is a string that specifies how input image pixels are interpolated to obtain output pixels: `interp` can be either `'nearest'`, `'bilinear'`, or `'bicubic'`. The `interp` input argument can be omitted, in which case it defaults to `'bilinear'`. As with the restoration examples given earlier, function `checkerboard` is useful for generating test images for experimenting with spatial transformations.

EXAMPLE 5.13: Spatially transforming images.

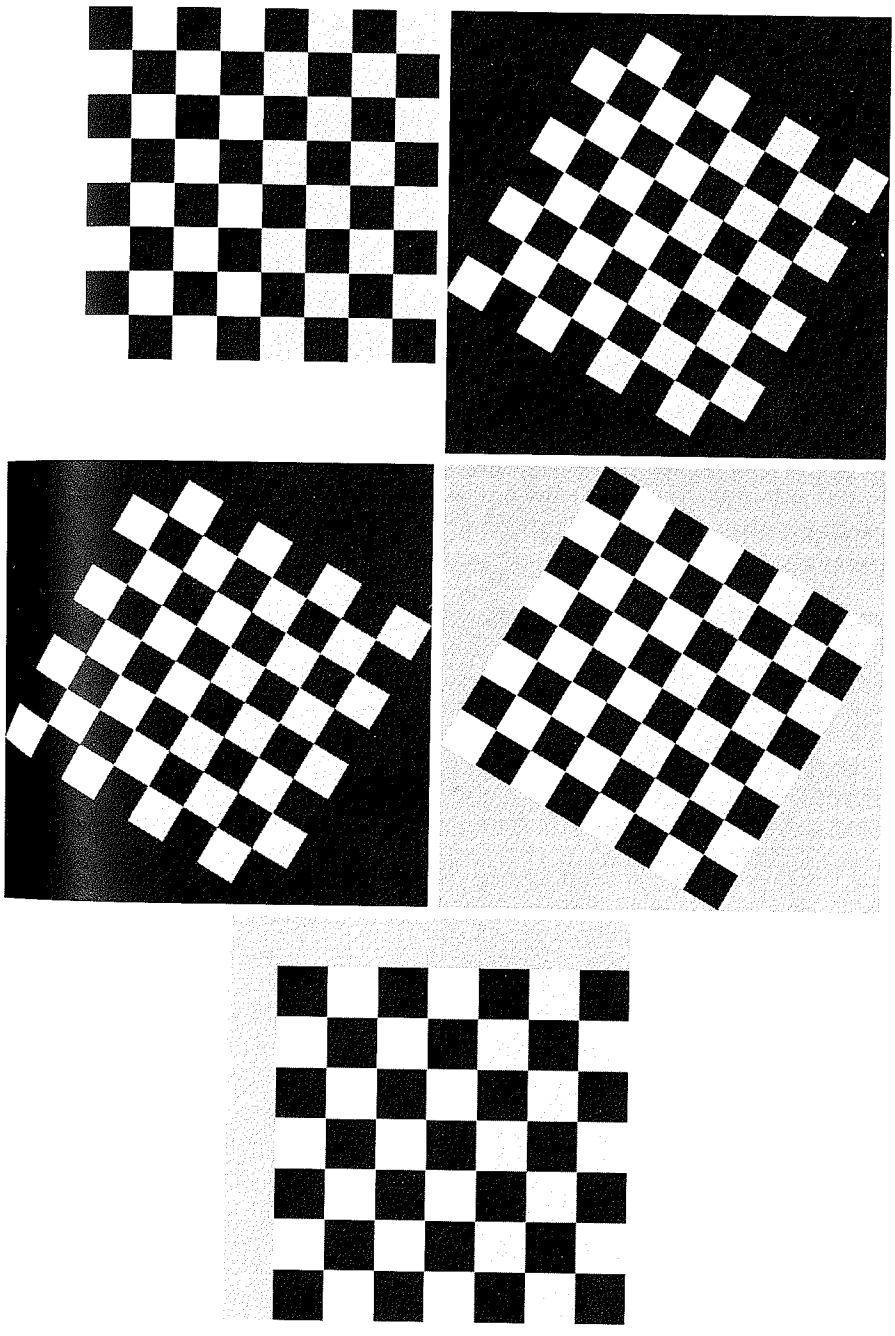
In this example we use functions `checkerboard` and `imtransform` to explore a number of different aspects of transforming images. A *linear conformal transformation* is a type of affine transformation that preserves shapes and angles. Linear conformal transformations consist of a scale factor, a rotation angle, and a translation. The affine transformation matrix in this case has the form

$$\mathbf{T} = \begin{bmatrix} s\cos\theta & s\sin\theta & 0 \\ -s\sin\theta & s\cos\theta & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$$

The following commands generate a linear conformal transformation and apply it to a test image.

```
>> f = checkerboard(50);
>> s = 0.8;
>> theta = pi/6;
>> T = [s*cos(theta) s*sin(theta) 0
       -s*sin(theta) s*cos(theta) 0
        0   0   1];
>> tform = maketform('affine', T);
>> g = imtransform(f, tform);
```

Figures 5.14(a) and (b) show the original and transformed checkerboard images. The preceding call to `imtransform` used the default interpolation method.



a b
c d
e

FIGURE 5.14 Affine transformations of the checkerboard image. (a) Original image. (b) Linear conformal transformation using the default interpolation (bilinear). (c) Using nearest neighbor interpolation. (d) Specifying an alternate fill value. (e) Controlling the output space location so that translation is visible.

`'bilinear'`. As mentioned earlier, we can select a different interpolation method, such as nearest neighbor, by specifying it explicitly in the call to `imtransform`:

```
>> g2 = imtransform(f, tform, 'nearest');
```

Figure 5.14(c) shows the result. Nearest neighbor interpolation is faster than bilinear interpolation, and it may be more appropriate in some situations, but it generally produces results inferior to those obtained with bilinear interpolation.

Function `imtransform` has several additional optional parameters that are useful at times. For example, passing it a `FillValue` parameter controls the color `imtransform` uses for pixels outside the domain of the input image:

```
>> g3 = imtransform(f, tform, 'FillValue', 0.5);
```

In Fig. 5.14(d) the pixels outside the original image are mid-gray instead of black.

Other extra parameters can help resolve a common source of confusion regarding translating images using `imtransform`. For example, the following commands perform a pure translation:

```
>> T2 = [1 0 0; 0 1 0; 50 50 1];
>> tform2 = maketform('affine', T2);
>> g4 = imtransform(f, tform2);
```

The result, however, would be identical to the original image in Fig. 5.14(a). This effect is caused by default behavior of `imtransform`. Specifically, `imtransform` determines the bounding box (see Section 11.4.1 for a definition of the term *bounding box*) of the output image in the output coordinate system, and by default it only performs inverse mapping over that bounding box. This effectively *undoes* the translation. By specifying the parameters `XData` and `YData`, we can tell `imtransform` exactly where in output space to compute the result. `XData` is a two-element vector that specifies the location of the left and right columns of the output image; `YData` is a two-element vector that specifies the location of the top and bottom rows of the output image. The following command computes the output image in the region between $(x, y) = (1, 1)$ and $(x, y) = (400, 400)$.

```
>> g5 = imtransform(f, tform2,'XData', [1 400], 'YData', [1 400], ...
                    'FillValue', 0.5);
```

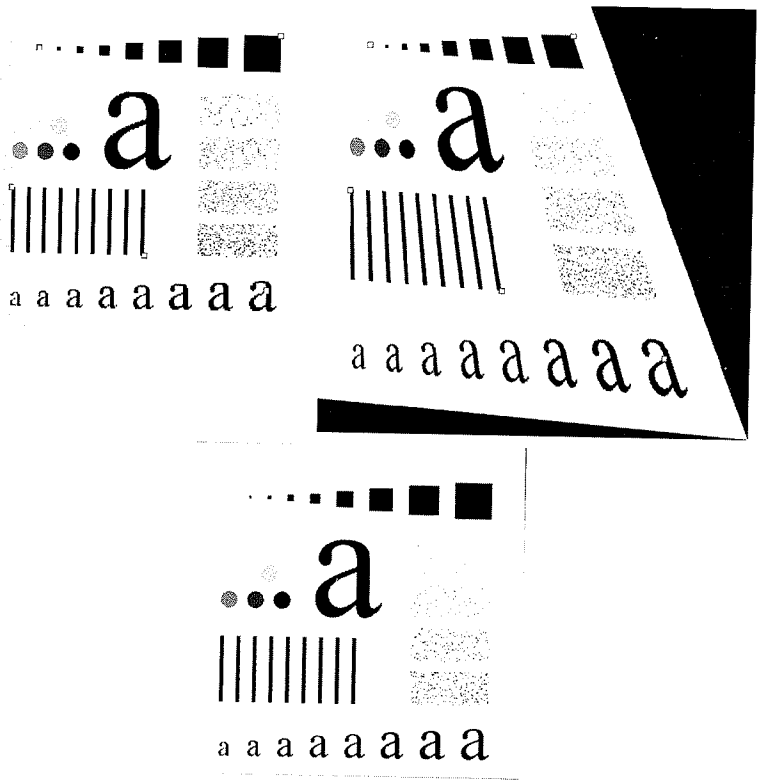
Figure 5.14(e) shows the result.

Other settings of `imtransform` and related IPT functions provide additional control over the result, particularly over how interpolation is performed. Most of the relevant toolbox documentation is in the help pages for functions `imtransform` and `makeresampler`.

5.11.3 Image Registration

Image registration methods seek to align two images of the same scene. For example, it may be of interest to align two or more images taken at roughly the same time, but using different instruments, such as an MRI (magnetic resonance imaging) scan and a PET (positron emission tomography) scan. Or, perhaps the images were taken at different times using the same instrument, such as satellite images of a given location taken several days, months, or even years apart. In either case, combining the images or performing quantitative analysis and comparisons requires compensating for geometric aberrations caused by differences in camera angle, distance, and orientation; sensor resolution; shift in subject position; and other factors.

The toolbox supports image registration based on the use of *control points*, also known as *tie points*, which are a subset of pixels whose locations in the two images are known or can be selected interactively. Figure 5.15 illustrates the idea of control points using a test pattern and a version of the test pattern that has undergone projective distortion. Once a sufficient number of control points have been chosen, IPT function `cp2tform` can be used to fit a specified type of spatial



a b
c

FIGURE 5.15 Image registration based on control points. (a) Original image with control points (the small circles superimposed on the image). (b) Geometrically distorted image with control points. (c) Corrected image using a projective transformation inferred from the control points.

TABLE 5.4 Transformation types supported by `cp2tform` and `maketform`.

Transformation Type	Description	Functions
Affine	Combination of scaling, rotation, shearing, and translation. Straight lines remain straight and parallel lines remain parallel.	`maketform` `cp2tform`
Box	Independent scaling and translation along each dimension; a subset of affine.	`maketform`
Composite	A collection of spatial transformations that are applied sequentially.	`maketform`
Custom	User-defined spatial transform; user provides functions that define T and T^{-1}.	`maketform`
Linear conformal	Scaling (same in all dimensions), rotation, and translation; a subset of affine.	`cp2tform`
LWM	Local weighted mean; a locally-varying spatial transformation.	`cp2tform`
Piecewise linear	Locally varying spatial transformation.	`cp2tform`
Polynomial	Input spatial coordinates are a polynomial function of output spatial coordinates.	`cp2tform`
Projective	As with the affine transformation, straight lines remain straight, but parallel lines converge toward vanishing points.	`maketform` `cp2tform`

transformation to the control points (using least squares techniques). The spatial transformation types supported by `cp2tform` are listed in Table 5.4.

For example, let `f` denote the image in Fig. 5.15(a) and `g` the image in Fig. 5.15(b). The control point coordinates in `f` are (83, 81), (450, 56), (43, 293), (249, 392), and (436, 442). The corresponding control point locations in `g` are (68, 66), (375, 47), (42, 286), (275, 434), and (523, 532). Then the commands needed to align image `g` to image `f` are as follows:

```
>> basepoints = [83 81; 450 56; 43 293; 249 392; 436 442];
>> inputpoints = [68 66; 375 47; 42 286; 275 434; 523 532];
>> tform = cp2tform(inputpoints, basepoints, 'projective');
>> gp = imtransform(g, tform, 'XData', [1 502], 'YData', [1 502]);
```

Figure 5.15(c) shows the transformed image.

FIGURE 5.16 Interactive tool for choosing control points.

The toolbox includes a graphical user interface designed for the interactive selection of control points on a pair of images. Figure 5.16 shows a screen capture of this tool, which is invoked by the command `cpselect`.

Summary

The material in this chapter is a good overview of how MATLAB and IPT functions can be used for image restoration, and how they can be used as the basis for generating models that help explain the degradation to which an image has been subjected. The capabilities of IPT for noise generation were enhanced significantly by the development in this chapter of functions `imnoise2` and `imnoise3`. Similarly, the spatial filters available in function `spfilt`, especially the nonlinear filters, are a significant extension of IPT's capabilities in this area. These functions are perfect examples of how relatively simple it is to incorporate MATLAB and IPT functions into new code to create applications that enhance the capabilities of an already large set of existing tools.