CHAPTER 4

Geometric Image Transformations

There are a number of important and useful geometric transformations of the plane that can be applied to images. Image transformations are implemented in the same way as subsampling: by scanning the pixels of the target and modifying them using interpolates of pixels from the source. This means it is important that transformations are invertible. Adopt the convention that a point $\mathbf{x} = (x, y)$ is mapped by a transformation to the point $\mathbf{u} = (u, v) = (u(x, y), v(x, y))$, and $\mathbf{u} = (u, v)$ is mapped to $\mathbf{x} = (x, y)$ by the inverse. In vector notation, $\mathbf{x}$ is mapped to $\mathbf{u}$, and so on. Write $\mathbf{A}$ for a 2×2 matrix, whose i, j'th component is a_{ij}.

4.1 GEOMETRIC TRANSFORMATIONS

You should either already know, or memorize, the following facts.

Definition: 4.1 *Translation*

Translation maps the point (x, y) to the point $(u, v) = (x + t_x, y + t_y)$ for two constants t_x and t_y. Here $(x, y) = (u - t_x, v - t_y)$. In vector notation,

$$\mathbf{u} = \mathbf{x} + \mathbf{t} \text{ and } \mathbf{x} = \mathbf{u} - \mathbf{t}.$$

Useful Fact: *Translation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a translation $(\mathbf{u}_1 - \mathbf{u}_2) = (\mathbf{x}_1 - \mathbf{x}_2)$. A similar argument shows that angles are preserved (***exercises***).*

Definition: 4.2 *Rotation*

Rotation takes the point (x, y) to the point

$$(u, v) = x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta.$$

Here θ is the angle of rotation, rotation is anti-clockwise, and

$$(x, y) = u\cos\theta + v\sin\theta, -u\sin\theta + v\cos\theta.$$

Write $\mathcal{R}$ for a 2×2 rotation matrix (a matrix where $\mathcal{R}^T\mathcal{R} = \mathcal{I}$ and $\det(\mathcal{R}) = 1$); then

$$\mathbf{u} = \mathcal{R}\mathbf{x} \text{ and } \mathbf{x} = \mathcal{R}^{-1}\mathbf{u} = \mathcal{R}^T\mathbf{u}.$$

Useful Fact: *Rotation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a rotation $(\mathbf{u}_1 - \mathbf{u}_2) = \mathcal{R}(\mathbf{x}_1 - \mathbf{x}_2)$ and $\mathcal{R}^T\mathcal{R} = \mathcal{I}$. A similar argument shows that angles are preserved (***exercises***).*

Definition: 4.3 *Euclidean transformations*

A Euclidean transformation is a rotation and translation, so $(u(x, y), v(x, y)) = (x\cos\theta - y\sin\theta + t_x, v(x, y) = x\sin\theta + y\cos\theta + t_y)$. Euclidean transformations preserve lengths and angles (and so areas) and are sometimes referred to as rigid body transformations. Here $(x, y) = ((u - t_x)\cos\theta + (v - t_y)\sin\theta, -(u - t_x)\sin\theta + (v - t_y)\cos\theta)$. In vector notation, for $\mathcal{R}$ a rotation,

$$\mathbf{u} = \mathcal{R}\mathbf{x} + \mathbf{t} \text{ and } \mathbf{x} = \mathcal{R}^T(\mathbf{u} - \mathbf{t}).$$

Useful Fact: *Euclidean transformations preserve lengths and angles (you can think of a Euclidean transformation as a rotation followed by a translation).*

Definition: 4.4 *Uniform scaling*

For **uniform scaling**, $(u, v) = (sx, sy)$ for $s > 0$. Here $(x, y) = (1/su, 1/sv)$. In vector notation,

$$\mathbf{u} = s\mathbf{x} \text{ and } \mathbf{x} = (1/s)\mathbf{u}.$$

Useful Fact: *Uniform scaling preserves angles, but not lengths (***exercises** *). Uniform scaling preserves ratios of lengths (***exercises** *)*

Definition: 4.5 *Non-uniform scaling*

For **non-uniform scaling**, $(u, v) = (sx, ty)$ for s and t both positive, and so $(x, y) = (1/su, 1/tv)$. Write $\mathsf{diag}\,((s, t))$ for the matrix with s and t on the diagonal. In vector notation,

$$\mathbf{u} = \mathsf{diag}\,((s, t))\mathbf{x} \text{ and } \mathbf{x} = \mathsf{diag}\,((1/s, 1/t))\mathbf{u}.$$

Useful Fact: *Non-uniform scaling will usually change both lengths and angles.*

Definition: 4.6 *Affine transformations*

Affine transformations are better written in vector notation. Write $\mathcal{A}$ for an invertible 2×2 matrix, and $\mathbf{t}$ for some constant vector. Then

$$\mathbf{u} = \mathcal{A}\mathbf{x} + \mathbf{t} \text{ and } \mathbf{x} = \mathcal{A}^{-1}(\mathbf{u} - \mathbf{t}).$$

Useful Fact: *Affine transformations will usually change both lengths and angles.*

Definition: 4.7 *Projective transformations*

Projective transformations involve quite inefficient notation if one does not know homogenous coordinates (Section **??**), and writing them in vector form is clumsy. Write p_{ij} for the i, j'th component of a 3×3 matrix $\mathcal{P}$ that is invertible. Then

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p_{11}x+p_{12}y+p_{13}}{p_{31}x+p_{32}y+p_{33}} \\ \frac{p_{21}x+p_{22}y+p_{23}}{p_{31}x+p_{32}y+p_{33}} \end{bmatrix}.$$

The inverse transformation is obtained by applying the inverse of $\mathcal{P}$ to $\mathbf{u}$ according to the recipe above. For a vector representation, write

$$\mathcal{P} = \begin{bmatrix} \mathbf{p}_1^T & p_{13} \\ \mathbf{p}_2^T & p_{23} \\ \mathbf{p}_3^T & p_{33} \end{bmatrix}$$

for a 3×3 array with inverse $\mathcal{Q}$. Then

$$\mathbf{u} = \begin{bmatrix} \frac{\mathbf{p}_1^T\mathbf{x}+p_{13}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \\ \frac{\mathbf{p}_2^T\mathbf{x}+p_{23}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} \frac{\mathbf{q}_1^T\mathbf{u}+q_{13}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \\ \frac{\mathbf{q}_2^T\mathbf{u}+q_{23}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \end{bmatrix}$$

This definition means that, if $\mathcal{P} = \lambda\mathcal{Q}$ for some $\lambda \neq 0$, then $\mathcal{P}$ and $\mathcal{Q}$ implement the same projective transformation.

Useful Fact: *All the classes of transformation described are special cases of a projective transformation (***exercises** *).*

4.2 GEOMETRIC TRANSFORMATIONS OF IMAGES

You need to think carefully to apply a geometric transformation to an image. The definition might suggest translating an $M \times N$ image by $(2, 2)$ by just changing the labels of the pixel locations from $1 \ldots M$ and $1 \ldots N$ to $3 \ldots M + 2$ and $3 \ldots N + 2$. Very little good will come from this approach. Instead, you transform the image and *put it in another image*. So in the example, you'd have a target image, and replace its pixel values with the image you were translating. This very minor point is a source of some irritations, because you may need to keep track of *where the transformed image goes*, otherwise it may disappear (not an exaggeration; example below).

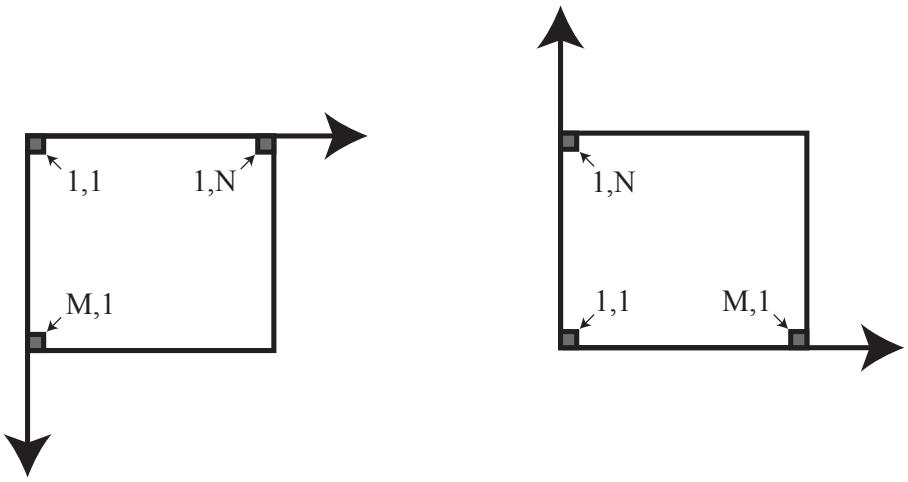

FIGURE 4.1: *The most common coordinate system for images, on the* **left** *mirrors that for arrays. The origin is at the top left corner, and we count in pixels. This is an $M \times N$ image. I will use the convention $\mathcal{I}_{ij}$ for points in this coordinate system, so the top right pixel is $\mathcal{I}_{1M}$. It is usual for pixel locations to be indexed starting at 1 (so $1 \leq i \leq M$ and $1 \leq j \leq N$). In some environments (notoriously, Python), the index starts at 0; in others (eg Matlab), it starts at 1. Keep track of this point, or you will lose some pixels. On the* **right**, *the origin is at the bottom left, and the coordinate axes are more familiar. Converting from one coordinate system to the other is straightforward,* but *not being consistent about the coordinate system you are working in is an important source of simple, annoying errors. I will always work in the coordinate system shown on the* **left**.

4.2.1 General Points

Transformations are only meaningful with respect to a coordinate system. The most common convention for image coordinate systems is strange at first glance. This coordinate system is shown in Figure 4.1 on the left. The inversion of the y-axis and of the order of coordinates is an annoying leftover from the way matrices are indexed, but has quite good properties that will become apparent later. It is quite usual to use this coordinate system, and I will do so in what follows. Readers should be aware that there are a variety of alternative conventions, and the choice of coordinate system has a significant effect on the expressions used to describe image transformations.

Transformations always take a source image $\mathcal{S}$ which is $s_M \times s_N$ to a target image $\mathcal{T}$ which is $t_M \times t_N$. I will need to refer to image values both at integer points – which I will write $\mathcal{S}_{ij}$ – and at points that are possibly not integer points – $\mathcal{S}(x, y)$. For points that are not integer points, care is required. If $1 \leq x \leq s_M$ and $1 \leq y \leq s_N$, then $\mathcal{S}(x, y)$ can be obtained by interpolation. Otherwise, some care is required.

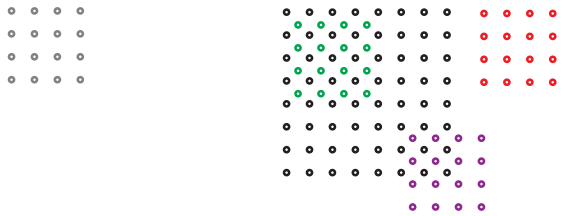

FIGURE 4.2: *When you transform an image, you must put it in a target image. The gray circles on the* **left** *represent the sample locations for a small image. On the* **right**, *this is placed at various different locations in the larger image (black samples). For translation to the green location, and using our current interpolation procedure, there will be no problem pasting the image, but the pasted region will be slightly smaller than the original. For the purple location, you will not see all pixels on the pasted image, because some pixels fall outside the target image range. This could be a desirable outcome. In the red location, you see no pixels from the translated image.*

As in Section 3.1, the correct general procedure is to scan the pixels of $\mathcal{T}$ and then modify them using interpolates of pixels from $\mathcal{S}$. This means it is important that transformations are invertible, and both $(u(x,y), v(x,y))$ and $(x(u,v), y(u,v))$ are known. If you require that the value of $\mathcal{S}(x,y)$ is known if $1 \leq x \leq s_M$ and $1 \leq y \leq s_N$, the image might shrink when you translate it. Figure 4.2 illustrates this effect. The source image has been translated to the green location. If you scan the target image (the bigger grid), and report a known value for $\mathcal{S}(x,y)$ only if $1 \leq x \leq s_M$ and $1 \leq y \leq s_N$, you will lose pixels (**exercises**).

You could mitigate this effect by *padding* the source image so that you know pixel values for $0 \leq x \leq s_M$ and $0 \leq y \leq s_N$. An easy way to do this is to attach a copy of the top row to the top of the image, and the leftmost column to the left of the image. More sophisticated mitigations are out of scope.

Scanning the target image can create inefficiencies. Figure 4.2 shows an example where the source image has been translated to lie outside the target image. Applying this transformation by scanning the whole target image is wasteful. It is usually a good idea to work out which values of u and v will lead to legitimate pixels, and scan only those values. Doing this efficiently requires some care, and is outside scope. An API will do all this for you.

4.2.2 Cropping, Pasting, Blending and Translation

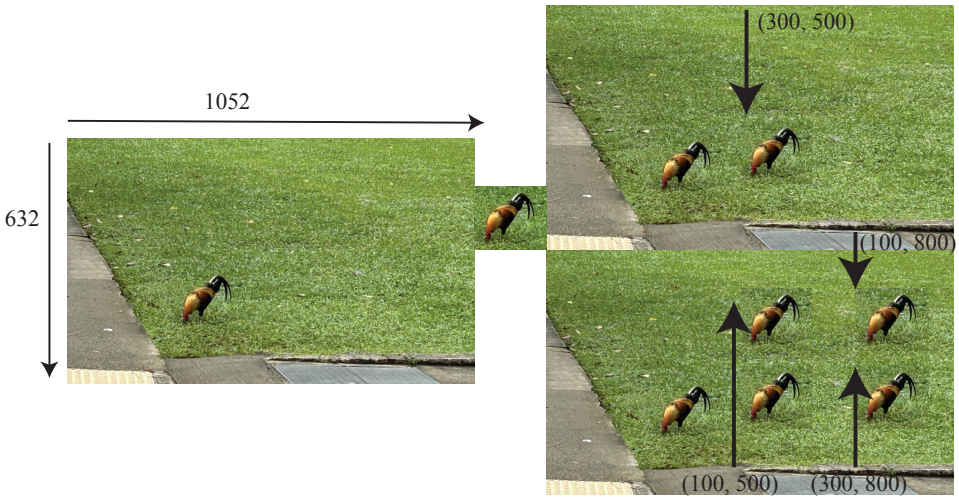

FIGURE 4.3: *The chicken in the* **left image** *has been cropped to yield the* **center image** *(which is* 162×187*), then translated and pasted to various points in the left image, to yield the images on the* **right**. *Note the choice of coordinate system strongly affects the value of translation. The chicken's origin is at the top left hand corner, yielding the translations shown in the overlay (left image scales are shown for reference). You should check you agree the translations indicated yield the chickens shown.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

Procedure: 4.1 *Cropping*

Cropping creates a smaller target image from a source image. One specifies a crop window in the $s_M \times s_N$ source image by $1 \leq x_n, x_x \leq M$ and $1 \leq y_n, y_x \leq N$. Here the vertices of the window are integers, and there is no interpolation. The target image is an $(x_x - x_n) \times (y_x - y_n)$ image. For $1 \leq i \leq x_x - x_n$ and $1 \leq j \leq y_n - y_x$, we have

$$\mathcal{T}_{ij} = \mathcal{S}_{i-x_n, j-y_n}$$

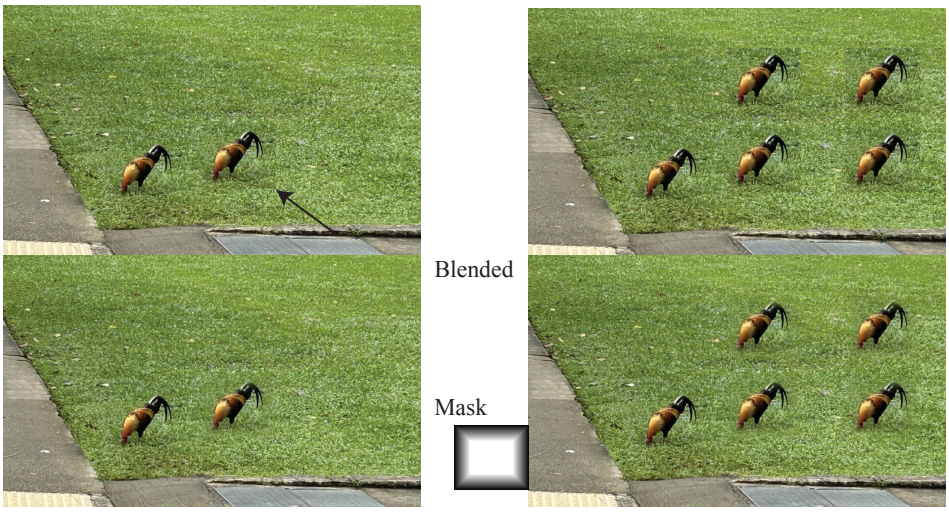

FIGURE 4.4: *The chickens of Figure 4.3 are simply pasted in the* **top row** *(as in that figure, reproduced here for comparison; the arrow on the* **left** *shows a problem with pasting not identified in that figure). In the* **bottom row**, *the chickens have been blended using the blending mask shown. Note the pasting is much less obvious.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

Procedure: 4.2 *Pasting*

Pasting replaces pixels in a target image with pixels from a source image using a transformation. For each of a collection of pixel locations u, v in $\mathcal{T}$:

- compute $x(u, v)$ and $y(u, v)$;
- obtain $p = \mathcal{S}(x, y)$ by interpolation, reporting `unknown` if x, y falls outside $\mathcal{S}$;
- and, if $p \neq$ `unknown`, set $\mathcal{T}_{uv} = p$.

It is a good idea, when pasting, to have a convenient representation of the pixel locations. This is easy if, for example, the transformation is a translation, but less so if it is a projective transformation. Pasting one image into another doesn't always yield good results, and you will often see visible lines on the outline of the transformed source image. This effect can be controlled quite well by blending using a mask.

Procedure: 4.3 *Blending*

Blending replaces pixels in a target image with a weighted sum of their original value and that of a pixel from a source image using a transformation and a specified set of weights. For each of a collection of pixel locations u, v in $\mathcal{T}$, where each has a weight $w(u, v)$:

- compute $x(u, v)$ and $y(u, v)$;
- obtain $p = \mathcal{S}(x, y)$ by interpolation, reporting `unknown` if x, y falls outside $\mathcal{S}$;
- and, if $p \neq$ `unknown`, set $\mathcal{T}_{uv} = wp + (1 - w)\mathcal{T}_{uv}$.

A good way to obtain blend weights is to have a map of blending weights, the same size as the source image. Typically, weights will be small at the boundary and bigger in the interior. Managing the collection of pixel locations in case of translation is particularly easy – you find the largest box of points that are in $\mathcal{T}$ and where the inverse translation maps them to points in $\mathcal{S}$ (Figure **??**). Notice that even in this case, you can come up with a transformation that appears to have no effect because translation can result in the source image ending up outside the window of the target image.

Procedure: 4.4 *Translating an image*

Apply the recipe for either pasting or blending, using a translation as the transformation.

Translation and pasting yield quite convincing composite images (Figure 4.3). However, close scrutiny of the multi-chicken image shows boundaries of the window where the translated chicken was pasted. These boundaries can be spotted because the grass on the left of the chicken is a little darker than the grass on which it was placed. Figure 4.4 indicates, blending can suppress problems at boundaries fairly effectively.

4.2.3 Scaling

Uniform scaling either makes the image bigger (upsampling, Section 3.1) or smaller (downsampling, Section 3.2).

Non-uniform scaling presents a combination of problems. If, say $s > 1$ and $t < 1$, we are upsampling in one direction and downsampling in the other. If t is relatively close to 1 (so there is not much downsampling), it is usually sufficient to ignore the upsampling, apply a gaussian smoother to the source, then resample with interpolation. If the downsampling is very aggressive, it may be better to smooth in one direction only, which is beyond scope.

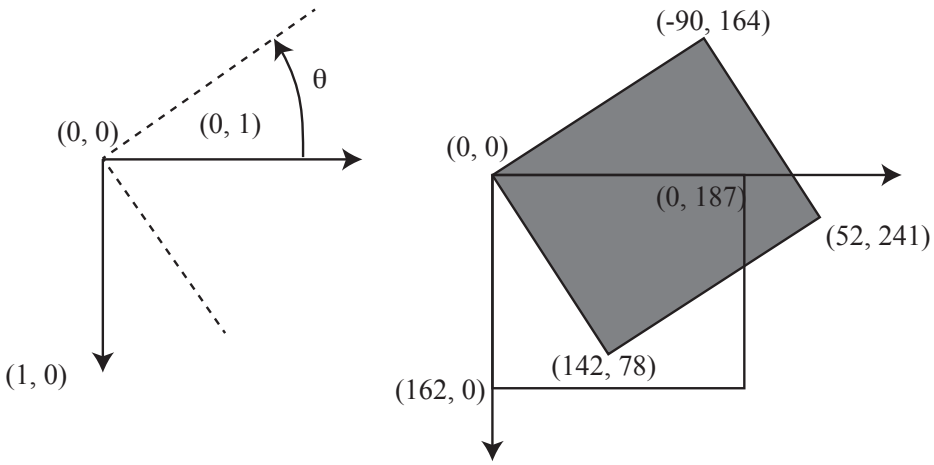

FIGURE 4.5: **Left** *shows the image coordinate system for reference, together with the result of rotating coordinate axes clockwise by θ (which in this example is 0.5 radians, about 30^0). Notice that a significant chunk of the source image ends up with negative coordinates.* **Right** *shows the original source rectangle from the cropped chicken of 4.3 (recall this is 162×187) as an open rectangle, and the rotated source rectangle in gray. The target image is then set up to enclose the whole result (implicitly translating the rotated source image) and pixels are then scanned into the target.*

4.2.4 Rotation, Affine and Projective Transformations

These transformations present problems. One is that the transformation applied by the API is sometimes not what you think (so you should check the manual). Another problem is caused by the fact that the rotated image usually spans more pixels in the coordinate directions than the source image (Figure 4.5).

Rotation is the transformation where it is most likely what you expect and what the API does might diverge. Rotations about the origin tend to cause images to disappear. For example, the grid of positive integer points by 180^0 anti-clockwise around the origin – all the grid points are still integer, but they are now all negative. As a result, most APIs rotate an image *about the center of the image*, rather than about the point 0, 0. You should interpret this as translating the image, rotating the image, then translating it back (**exercises**).

There are other things to watch out for in APIs:

- The API might create an empty target image whose horizontal and vertical spans are big enough to contain whole of the the transformed image, then apply the transformation to the source image and paste it in the target image. You will see all pixels in the source image, but there will be target pixels that are unknown – typically, these contain zero. This option is common.

- The API might create an image as in the previous option, but then crop it

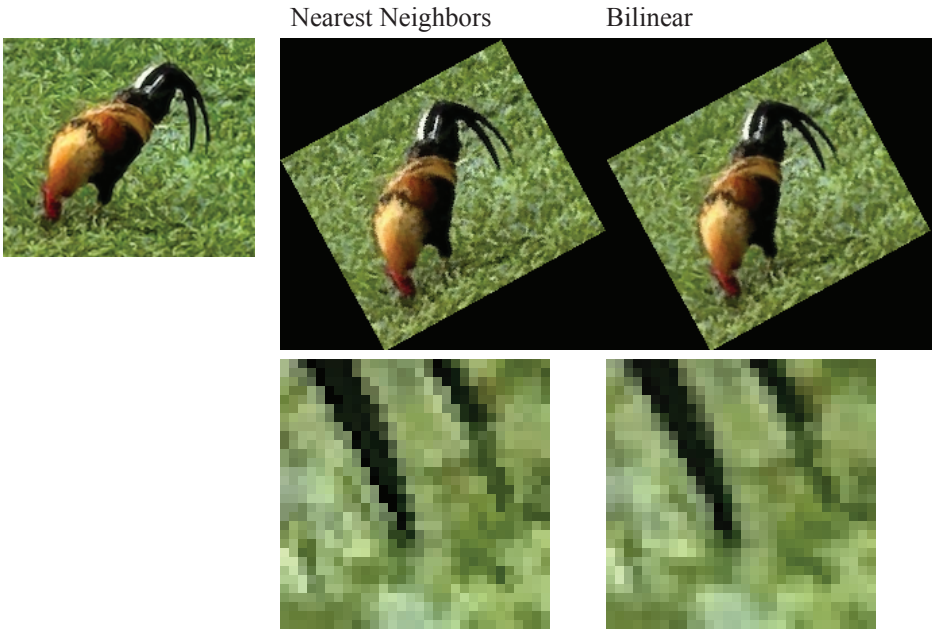

FIGURE 4.6: *The chicken of Figure 4.3, rotated by 0.5 radians as in Figure 4.5, showing the effect of different choices of interpolation. I have zoomed in on a section of the tail feathers to make the difference more apparent.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

to the largest rectangle that lies inside the known pixels. In this case, every pixel comes from the source image, but you will miss some pixels. This option is also common.

- The API will usually provide a way to choose the interpolate. This choice has a real effect on the results (Figure 4.6).

Procedure: 4.5 *Transforming an image*

To apply a transformation $\mathcal{F}$ to an image $\mathcal{I}$:

- The source image $\mathcal{S}$ is either $\mathcal{I}$ or a padded version of $\mathcal{I}$.
- Write $\mathbf{x}_1 = (1,1)^T$, $\mathbf{x}_2 = (s_M,1)^T$, $\mathbf{x}_3 = (s_M,s_N)^T$ and $\mathbf{x}_4 = (1,s_N)^T$ for the four vertices of the source image.
- Compute $\mathbf{u}_i = \mathcal{F}(\mathbf{x}_i)$ for the result of applying the transformation to these vertices. Now write u_n, u_x for the smallest (resp. largest) value of the first component of these points; similarly, v_n, v_x for the smallest (resp. largest) value of the second component of these points.
- $\mathcal{T}$ is now a $(\text{ceil}(u_x - u_n) + 1) \times (\text{ceil}(v_x - v_n) + 1)$ image.
- Predict the range of $\mathcal{T}$ in which $\mathcal{S}$ will land.
- For each i, j in the that range of $\mathcal{T}$
  - Write $(x(i,j), y(i,j))^T = \mathcal{F}^{-1}((i,j)^T)$.
  - If $1 \leq x \leq s_M$ and $1 \leq y \leq s_N$ then

    $$\mathcal{T}_{ij} \leftarrow \mathcal{S}(x(i + u_n - 1, j + v_n - 1), y(i + u_n - 1, j + v_n - 1))$$

    interpolating as required.

Optionally, crop $\mathcal{T}$ to the size of the largest axis aligned rectangle inside the transformed source (**exercises**).

The choice of interpolate has a real effect (Figure 4.6).

Affine transformations follow the recipe for the rotation. However, an affine transformation may involve a component of scaling, which might be non-uniform. One way to see this is to apply a singular value decomposition to $\mathcal{A}$ which will yield

$$\mathcal{A} = \mathcal{U}\Sigma\mathcal{V}^T$$

where $\mathcal{U}$ and $\mathcal{V}$ are rotations. But Σ is diagonal, and may be non-uniform. As long as the values on the diagonal of Σ are not too different, and the smallest is not too small, then one can apply a gaussian smoother to the source, and resample with interpolation. A robust smoothing strategy is firmly beyond scope, however.

Projective transformations follow the same general recipe as rotations, but smoothing is now tricky. For a general projective transformation, there might be singular points, caused by a divide-by-zero. For geometric reasons, these projective transformations do not arise in cases interesting to us (Section 22.3), and should be seen as evidence of a problem elsewhere. Nasty smoothing problems occur because at some pixels a projective transformation may upsample an image and at different pixels downsample the image. For this effect, look at Figure 4.8 and consider what

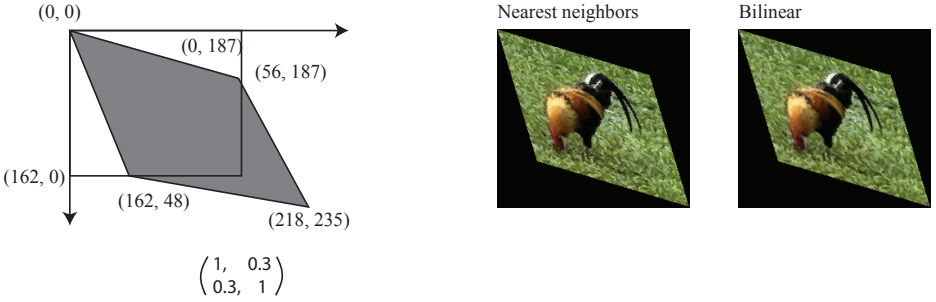

FIGURE 4.7: *In the standard image coordinate system, the affine transformation whose matrix is shown at the* **left** *is applied to the original chicken crop of Figure 4.3 (recall this is* 162×187*; the unfilled rectangle). The gray diamond indicates the result. The target image is then set up to enclose the whole result, and pixels scanned into the target. In this case, the source image was not smoothed, because there is relatively little downsampling (the diamond is not much smaller than the open rectangle).* **Center** *shows the result using nearest neighbors interpolation, and* **right** *shows the result using bilinear interpolation. Look closely at the tail feathers to see the difference.*

happens if the transform scales the image as well. It is relatively straightforward to predict at a given pixel whether downsampling is occuring, and the degree of downsampling (**exercises**), meaning a gaussian pyramid is useful. At a pixel in the target image, predict which location in the source image will be used; estimate the degree of smoothing required; then look at the relevant layer of the gaussian pyramid. This strategy is sometimes referred to as *MIP-mapping*.

> **Remember this:** *Transform a source image by: determining how big the transformed image will be; constructing a target image that will span the bits you want; then scanning the target, picking up pixels from the source image using the inverse transformation. There are several different possible choices of what pixels from the transformed image you want, and APIs usually implement most.*

4.3 ALIGNMENT AND DETECTION WITH GEOMETRIC TRANSFORMATIONS

Color photography is usually dated to the 1930's when it first became available to the public. In fact, James Clerk Maxwell described a method to capture a color photograph in an 1855 paper. The procedure likely looks straightforward to you: obtain three color filters, and take a picture of the scene through each of these filters. Capturing these *color separations* presented a number of technical challenges, and the first color photograph was taken by Thomas Sutton in 1861.

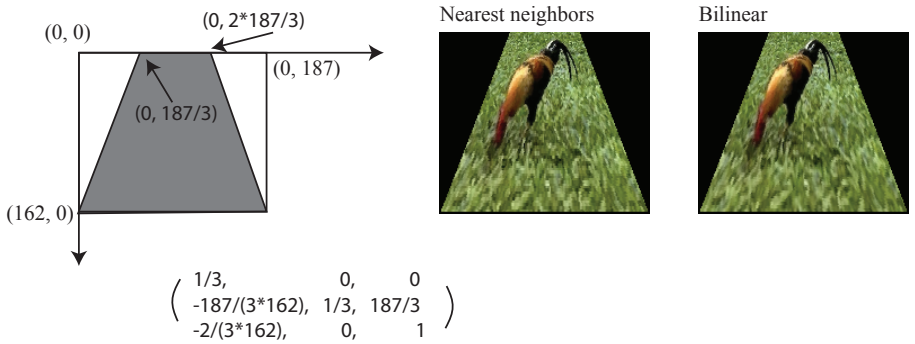

FIGURE 4.8: *In the standard image coordinate system, the projective transformation whose matrix is shown at the* **left** *is applied to the original chicken crop of Figure 4.3 (recall this is* 162×187*; unfilled rectangle). The gray region indicates the result. Note that the projective transformation has taken the rectangular source to a shape that is not even a parallelogram. The target image is then set up to enclose the whole result, and pixels scanned into the target. In this case, the source image was not smoothed, because there is relatively little downsampling (the gray region is not much smaller than the open rectangle).* **Center** *shows the result using nearest neighbors interpolation, and* **right** *shows the result using bilinear interpolation. Look closely at the tail feathers to see the difference.*

Actually displaying pictures obtained like this was tricky. One had to pass red light through the red separation, green through the green, and blue through the blue, then ensure all three resulting images lay on top of one another on screen. Turning them into the image files we are familiar with is also tricky, because each layer of the separation is typically a bit offset from the others (the camera moved slightly between photographs), and each layer has aged and been damaged slightly differently.

If you can score the similarity between two images, you can line up these pictures to produce a color image by sliding green and blue to line up with red. Similarly, you can detect things by sliding a window around an image and scoring the similarity between the window and the image. If they are very similar, then the image looks like the window at that location, and you may be able to declare a detection.

4.3.1 Scoring an Overlap with a Cost Function

You can use the sum of squared differences SSD to score the similarity between the overlapping parts of two images $\mathcal{R}$ and $\mathcal{B}$. The definition is in a box.

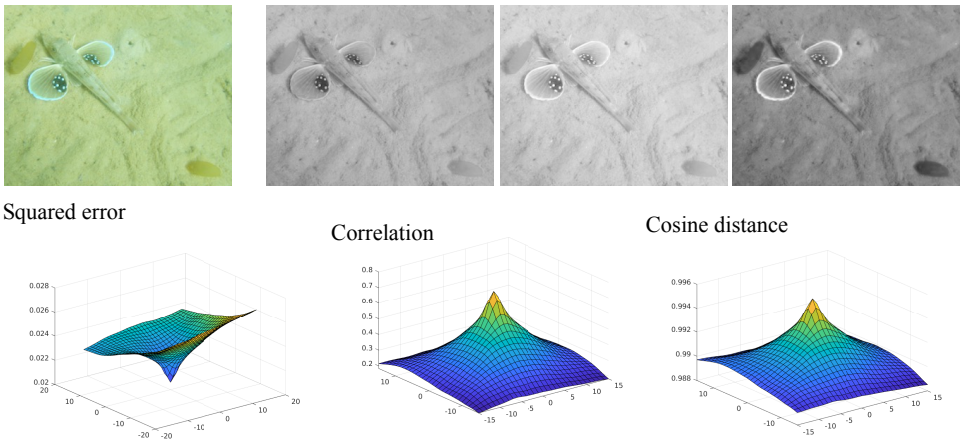

FIGURE 4.9: **Top left** *shows a Gurnard, flashing its pectoral fins in alarm.* **Top rest** *shows the color separations of this image (in red, green, blue order). The image is slightly blue-green (taken at about 5 meters depth, where water absorbs red light), and this shows as a darker red separation.* **Bottom** *shows how various cost functions react to registering red to blue. The correct alignment is at 0, 0 and the images are 257 by 323. Notice that: all the extrema are in the right place, but the correlation and cosine distance must be maximized, and the squared error minimized; the squared error changes relatively little from the best to the worst, because the blue image is rather unlike the red; both cosine distance and correlation are much more sensitive than SSD – they fall off much more quickly than the SSD rises.* Image credit: *Figure shows my photograph of a Gurnard, at Long Beach in Cape Town.*

Definition: 4.8 *The sum of squared differences or SSD*

The *sum of squared differences* or *SSD* scores the similarity between two images $\mathcal{U}$ and $\mathcal{V}$ of the same size ($N \times M$ pixels) by

$$\text{SSD}(\mathcal{U}, \mathcal{V}) = \sum (\mathcal{R}_{ij} - \mathcal{B}_{ij})^2 .$$

For different offsets, the number of overlapping pixels is different. Given an offset m, n, shift $\mathcal{B}$ by that offset. Write $\mathcal{B}_o$ for the set of pixels in this shifted version of $\mathcal{B}$ that overlap $\mathcal{R}$. Write $\mathcal{R}_o$ for the pixels in $\mathcal{R}$ that are overlapped by the shifted version of $\mathcal{B}$. Write N_o for the number of pixels in the overlap. Then use the cost function

$$C_{\text{reg}}(m, n; \mathcal{R}, \mathcal{B}) = \frac{1}{N_o} \text{SSD}(\mathcal{R}_o, \mathcal{B}_o)^2.$$

Notice that normalizing by N_o is important; if you don't, you will find that the best match occurs when the overlap is smallest.

The SSD assumes that the images to be registered are very close to the same when they are aligned. But color separations do not agree exactly when they overlap – if they did, the image would be a monochrome image. It is useful to have alternative cost functions that (a) will tend to be minimized or maximized when the images are correctly registered and (b) change quite quickly when they are not. Quite widely used alternatives are the cosine distance and the correlation coefficient.

Definition: 4.9 *The cosine distance*

The *cosine distance* scores the similarity between two images $\mathcal{U}$ and $\mathcal{V}$ of the same size ($N \times M$ pixels) by

$$C_{cos}(\mathcal{U}, \mathcal{V}) = \frac{\sum (\mathcal{A}_{ij} * \mathcal{B}_{ij})}{\sqrt{\sum \mathcal{A}_{ij}^2} \sqrt{\sum \mathcal{B}_{ij}^2}}.$$

Definition: 4.10 *The correlation coefficient*

The *correlation coefficient* scores the similarity between two images $\mathcal{U}$ and $\mathcal{V}$ of the same size ($M \times N$ pixels) by

$$C_{corr}(m,n) = \frac{\sum [(\mathcal{A}_{ij} - \mu_A) * (\mathcal{B}_{ij} - \mu_B)]}{\sqrt{\sum (\mathcal{A}_{ij} - \mu_A)^2} \sqrt{\sum (\mathcal{B}_{ij} - \mu_B)^2}}$$

$$\text{where} \quad \mu_A = \frac{1}{MN} \sum \mathcal{A}_{ij} \text{ and}$$

$$\text{where} \quad \mu_B = \frac{1}{MN} \sum_{overlap} \mathcal{B}_{ij}.$$

Annoyingly, the cosine distance is largest when best, even though it's called a distance. Some authors subtract this distance from one (its largest value) to fix this. The correlation coefficient is big for the best alignment. It corrects for the mean of each image.

Each is in the range -1 to 1, and neither scales with the size of the overlap neighborhood. Terminology in this area is severely confused. The cosine distance isn't a distance; it is sometimes referred to as *normalized correlation*; and sometimes as *correlation*. Several functions similar to correlation are referred to as correlation. Figure 4.9 shows how these cost functions behave when trying to register the red and blue separations of an image. These separations will be fairly similar, but not exactly the same.

4.3.2 Aligning Color Separations by Translation

Separations are *in register* if they lie over one another exactly and so form a color image. If they are out of register, objects will have slight, odd color halos. Early color separations tend not to be in register. A class assignment, now hallowed by tradition in computer vision, but likely to have originated with A. Efros in 2010, uses the pictures of Sergei Mikhailovich Prokudin-Gorskii (1863-1944). Prokudin-Gorskii traveled the Russian empire and took color photographs of many scenes. He left Russia in 1918. His negatives survived and ended up in the Library of Congress. A digitized version of the collection is available online at `https://www.loc.gov/collections/prokudin-gorskii/about-this-collection/` (look for the glass slides, which give the R, G and B separations of each image). The assignment asks students to register the color separations for some of these images.

There is a natural strategy: write a function that is smallest when the G (respectively B) separation is in register with the R separation; now search for the best value of the cost function obtained by small translations of the G (respectively B) separation.

The search is easy when the separations are at relatively low resolution. The offsets will be relatively small (a few pixels or so). It is then practical to simply evaluate the cost function at a grid of translations, and choose the best (fussier readers might interpolate, exercises). The remaining issue is the cost function. Section 4.3.1 describes a number of possible cost functions.

This assignment requires care when one works with the high resolution version of the scans. These are quite big, and there can be moderately large offsets. Simply looking at each offset in turn will be hideously expensive (dealt with in Section 15.1.3).

4.3.3 Elementary Object Detection, or Find the Chicken

Object detection is the problem of determining whether an object appears in an image *and* where it is if it is there. There are a wide range of variants, explored in much greater detail in Chapter 22.3; differences hinge on how one interprets the word "object", an alarmingly rich question.

A very simple object detector can be built out of the mosaic procedure. Assume $\mathcal{A}$ is an image which might contain an object, and $\mathcal{B}$ is a *template* – an example image of the object to be detected. For every offset m, n where $\mathcal{B}$ lies inside $\mathcal{A}$, compute the cost function and store values in an array (the *score array*). Notice that if the values are small, then at that offset, the overlapping bits of $\mathcal{B}$ and $\mathcal{A}$ "look like" one another. If they "look like" one another sufficiently (test the cost function against a threshold), declare that the object is present. Figure 4.10 shows what the arrays look like for a variety of cost functions.

This detector will tend to overcount objects rather significantly. Shifting a template by one or two pixels will not tend to change the cost function by much. This means if the cost function is below threshold at m, n, it is likely to be below threshold at neighboring points in the score array, too. This could mean you find many instances of the object nearly on top of one another. A straightforward procedure called *non-maximum suppression* deals with this. Find the smallest below threshold value in the score array. Record an object present at that location, then

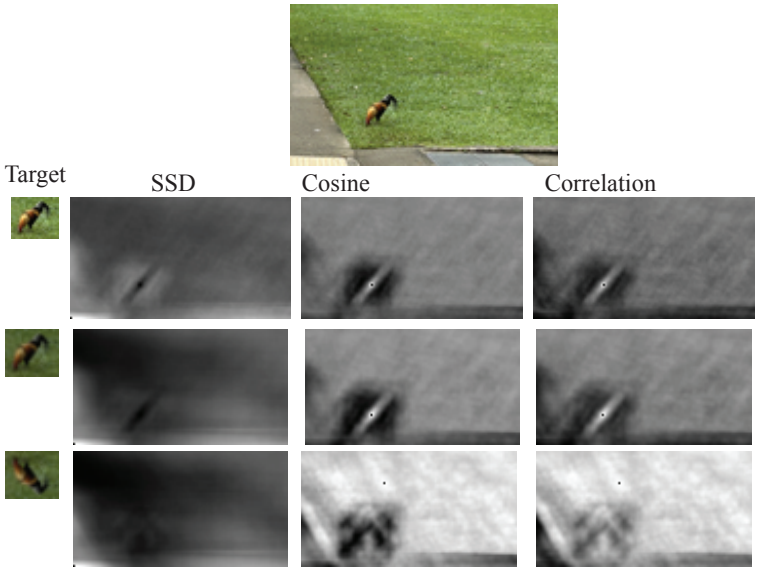

FIGURE 4.10: *Translation and an image matching cost function yield an elementary detector. Model the object – here, the chicken – using an image window (***leftmost column***), then translate this window to each location in the image (***top***) and compute the cost of the overlap. If the underlying image looks a lot like the chicken, you will get a good value of the cost function (***other columns***. For SSD, a good value is small – and so dark – for others it is large – and so light. This elementary detector has serious problems. In the* **second row***, the chicken template is darker than the original image, and so SSD matches are not particularly good. Cosine distance and correlation are less affected. But chickens don't stay in a fixed configuration, and if the chicken moves* **third row***, all scores fall off.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*

suppress that location and all nearby values (nearby might mean, for example, all values in a $k \times k$ window centered on the current best value in score array) by setting all to a large value. Repeat this procedure until there are no more below threshold values in the score array.

There are other good reasons this isn't a good object detector. Look at Figure 4.10. The detector will only find chickens if they are in the same configuration as the template, and on a grass background, and with the same lighting. Some of this can be fixed with straightforward procedures. For some specialized applications, where very little computing is available, and where relatively few pixels lie on the object, a detector built like this can be useful, but outside these applications different procedures are used. A large family of modern detectors are built on this framework, with some crucial modifications: the cost function for evaluating the match between an image window and the concept "chicken" is much more sophisticated than just comparing image pixels with template pixels and the search procedure is more elaborate and more efficient (Chapter **??**).

Remember this: *Color separations can be registered by using translations and a cost function that checks how well they are registered. There are numerous useful cost functions. This procedure can be adapted to produce a simple detector which is not particularly reliable.*

4.4 YOU SHOULD

4.4.1 remember these definitions:

4.4.2 remember these facts:

4.4.3 remember these procedures:

4.4.4 be able to:

- Remember the form of translations, rotations, Euclidean transformations, uniform and non-uniform scaling, affine transformations and projective transformations.
- Explain the main difficulties in transforming an image, and how they are resolved.
- Compel an API to produce the transformation result you want, and understand the difference between options.
- Register color separations with translations.

EXERCISES

QUICK CHECKS

4.1. Which transformations preserve angles?

4.2. Which transformations preserve lengths?

4.3. Could there be a family of transformations that preserves lengths, but not angles? Why?

4.4. Assume that the 2×2 matrix $\mathcal{N}$ has the property $\mathcal{N}^T\mathcal{N} = \mathcal{I}$ and $\det(\mathcal{N}) = -1$. Check that there is some rotation $\mathcal{R}$ such that $\mathcal{N} = \mathcal{R}\mathsf{diag}\left((1, -1)\right)$.

4.5. What happens if you apply $\mathsf{diag}\left((1, -1)\right)$ to an image?

4.6. Figure 4.1 shows two image coordinate systems. What transformation takes the coordinates of a point in the left-hand coordinate system to the coordinates of the same point in the right-hand coordinate system?

4.7. Write $\mathcal{R}$ for a rotation matrix. Show that the transformation that takes $\mathbf{x}$ to $\mathcal{R}(\mathbf{x} - \mathbf{t}) + \mathbf{t}$ is a rotation about the point $\mathbf{t}$.

4.8. Section 4.2.4 has " As long as the values on the diagonal of Σ are not too different, and the smallest is not too small, then one can apply a gaussian smoother to the source, and resample with interpolation." Explain.

4.9. Is the transformation that takes (x, y) to $((50x)/(x - 100), (50y)/(x - 100))$ a projective transformation?

4.10. For pixels near $(25, 25)$, does the transformation that takes (x, y) to $((50x)/(x - 100), (50y)/(x - 100))$ upsample or downsample an image?

4.11. For pixels near $(75, 75)$, does the transformation that takes (x, y) to $((50x)/(x - 100), (50y)/(x - 100))$ upsample or downsample an image?

4.12. Section 4.3.1 says: " Notice that normalizing by N_o is important; if you don't, you will find that the best match occurs when the overlap is smallest." Explain.

4.13. Explain why you don't need to normalize the cosine distance by the size of the overlap (Section 4.3.1).

LONGER PROBLEMS

4.14. Recall that if θ is the angle between two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, then

$$\cos\theta = \frac{\mathbf{v}_1^T\mathbf{v}_2}{\sqrt{\mathbf{v}_1^T\mathbf{v}_1}\sqrt{\mathbf{v}_2^T\mathbf{v}_2}}$$

(a) Show that translation preserves the angle between two vectors.

(b) Show that rotation preserves the angle between two vectors.

(c) Show that uniform scaling preserves the angle between two vectors.

(d) Show that uniform scaling preserves the ratio of any two lengths.

4.15. Write p_{ij} for the i, j'th component of a 3×3 matrix $\mathcal{P}$ that is invertible. This is a projective transformation of the image plane, which maps a point $(u, v)^T$ to

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p_{11}x + p_{12}y + p_{13}}{p_{31}x + p_{32}y + p_{33}} \\ \frac{p_{21}x + p_{22}y + p_{23}}{p_{31}x + p_{32}y + p_{33}} \end{bmatrix}.$$

(a) Show that if $p_{11} = p_{12}$, and all other $p_{ij} = 0$ *except* p_{33}, then the transformation is in fact a uniform scaling transformation.

(b) Show that if $p_{31} = p_{32} = 0$, $p_{33} = 1$, and the 2×2 matrix

$$\mathcal{Q} = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

is orthonormal (ie $\mathcal{Q}\mathcal{Q}^T$ is the identity) then the transformation is in fact a euclidean transformation.

4.16. Write $\mathbf{x}_1 = (1,1)^T$, $\mathbf{x}_2 = (s_M, 1)^T$, $\mathbf{x}_3 = (s_M, s_N)^T$ and $\mathbf{x}_4 = (1, s_N)^T$ for the four vertices of the source image. Compute $\mathbf{u}_i = \mathcal{F}(\mathbf{x}_i)$ for the result of applying the transformation $\mathcal{F}$ to these vertices. Now write u_n, u_x for the smallest (resp. largest) value of the first component of these points; similarly, v_n, v_x for the smallest (resp. largest) value of the second component of these points.

(a) Show that if $\mathcal{F}$ is an affine transformation, the vertices $\mathbf{u}_i$ are the vertices of a parallelogram.

(b) Show that if $\mathcal{F}$ is an affine transformation, the parallelogram identified by $\mathbf{u}_i$ lies inside the axis aligned rectangle whose first coordinate u is in the range $u_n \leq u \leq u_x$ and whose second coordinate v is in the range $v_n \leq v \leq v_x$.

(c) Recall that a set $\mathcal{S}$ is convex if, for any $\mathbf{x} \in \mathcal{S}$ and $\mathbf{y} \in \mathcal{S}$, and any $0 \leq t \leq 1$, $t\mathbf{x} + (1-t)\mathbf{y} \in \mathcal{S}$. Show that an affine transformation maps a convex set to a convex set.

4.17. Write $\mathbf{x}_1 = (1,1)^T$, $\mathbf{x}_2 = (s_M, 1)^T$, $\mathbf{x}_3 = (s_M, s_N)^T$ and $\mathbf{x}_4 = (1, s_N)^T$ for the four vertices of the source image. Assume $\mathcal{F}$ is a projective transformation, given by

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p_{11}x + p_{12}y + p_{13}}{p_{31}x + p_{32}y + p_{33}} \\ \frac{p_{21}x + p_{22}y + p_{23}}{p_{31}x + p_{32}y + p_{33}} \end{bmatrix}..$$

Compute $\mathbf{u}_i = \mathcal{F}(\mathbf{x}_i)$ for the result of applying the transformation $\mathcal{F}$ to the four vertices.

(a) Show that, if the line given by $p_{31}x + p_{32}y + p_{33} = 0$ does *not* intersect the original image, then $\mathbf{u}_i$ are the vertices of a quadrilateral. What happens if the line does intersect the original image?

(b) Show that, if the line given by $p_{31}x + p_{32}y + p_{33} = 0$ does *not* intersect the original image, then the quadrilateral whose vertices are $\mathbf{u}_i$ is convex. Show that this means it lies within the axis aligned rectangle whose first coordinate u is in the range $u_n \leq u \leq u_x$ and whose second coordinate v is in the range $v_n \leq v \leq v_x$.

4.18. Construct a projective transformation that downsamples the image at some pixels and upsamples at others, using Figure 4.8 as a guide.

PROGRAMMING EXERCISES

4.19. Write a code that applies the transformation of Figure 4.8 to an image of your choice. Compare the effects of using a bilinear interpolate and a nearest neighbors interpolate.

4.20. A digitized version of the Prokudin-Gorskii photographs is available at `https://www.loc.gov/collections/prokudin-gorskii/about-this-collection/` (look for the glass slides). These give the R, G and B separations of each image. You must write a program that registers separations.

(a) First, find a 64×64 RGB image on the internet or make one. Separate this into three color separations, then write a code that aligns these separations

by searching translations for the one with the smallest SSD. You should not need to search a large range of offsets. You can tell whether you have the right answer in two ways: first, you shifted the layers with respect to one another, so you know the right shift to register them; second, if you look at the registered picture, the colors should be pure.

(b) Now take six of the Prokudin-Gorskii slides and reduce the size of each separation to 64×64. Use your code to register the separations. The easiest way to align the parts is to exhaustively search over a window of possible displacements (say [-15,15] pixels independently for the x and y axis), score each one using some image matching metric, and take the displacement with the best score. Investigate the effects of using a different metric on the alignment.

(c) Now use a Gaussian pyramid and a coarse to fine search to align the same six slides at full resolution. Can you improve the result by considering alignments at a subpixel resolution (i.e. an offset of, say, 1.5 pixels rather than just an integer number of pixels)? Doing so will involve some thought about interpolation.

4.21. Build and evaluate a simple chicken detector, along the lines of Section 4.3.3. Can you improve its behavior by using more than one template for a chicken?