- 16. If you are using MATLAB or Octave, then use the `tic` and `toc` timer functions to compare the use of `nlfilter` and `colfilt` functions. If you are using the ipython enhanced shell of Python, try the `%time` function
- 17. Use `colfilt` (MATLAB/Octave) or `generic_filter` (Python) to implement the geometric mean and alpha-trimmed mean filters.
- 18. If you are using MATLAB or Octave, show how to implement the root-mean-square filter.
- 19. Can unsharp masking be used to reverse the effects of blurring? Apply an unsharp masking filter after a 3 × 3 averaging filter, and describe the result.
- 20. Rewrite the Kuwahara filter as a single function that can be applied with either `colfilt` (MATLAB/Octave) or `generic_filter` (Python).

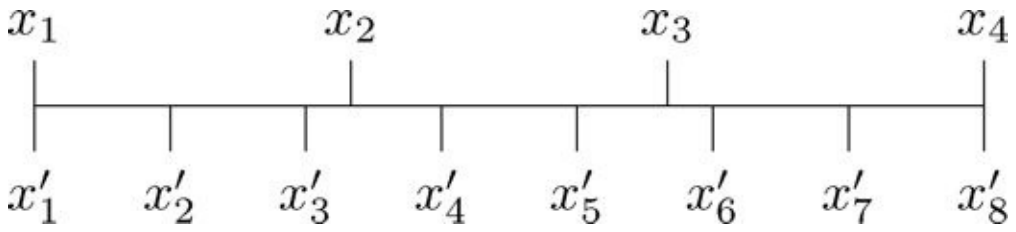
6 Image Geometry

There are many situations in which we might want to change the shape, size, or orientation of an image. We may wish to enlarge an image, to fit into a particular space, or for printing; we may wish also to reduce its size, say for inclusion on a web page. We might also wish to rotate it: maybe to adjust for an incorrect camera angle, or simply for affect. Rotation and scaling are examples of *affine transformations*, where lines are transformed to lines, and in particular parallel lines remain parallel after the transformation. Non-affine geometrical transformations include warping, which we will not consider.

6.1 Interpolation of Data

We will start with a simple problem: suppose we have a collection of 4 values, which we wish to enlarge to 8. How do we do this? To start, we have our points x_1, x_2, x_3, and x_4, which we suppose to be evenly spaced, and we have the values at those points: $f(x_1)$, $f(x_2)$, $f(x_3)$, and $f(x_4)$. Along the line $x_1 \ldots x_4$ we wish to space eight points $x'_1, x'_2, \ldots, x'_8$. Figure 6.1 shows how this would be done.

Figure 6.1: Replacing four points with eight



Suppose that the distance between each of the x_i points is 1; thus, the length of the line is 3. Thus, since there are seven increments from x'_i to x'_8, the distance between each two will be 3/7 ≈ 0.4286. To obtain a relationship between x and x' we draw Figure 6.1 slightly differently as shown in Figure 6.2. Then

$$x' = \frac{1}{3}(7x - 4),$$
$$x = \frac{1}{7}(3x' + 4).$$

As you see from Figure 6.1, none of the x'_i coincides exactly with an original x_j, except for the first and last. Thus we are going to have to "guess" at possible function values $f(x'_i)$. This guessing at function values is

called *interpolation*. Figure 6.3 shows one way of doing this: we assign $f(x'_i) = f(x_j)$, where x_j is the original point closest to x'_i. This is called *nearest neighbor interpolation*.

Figure 6.2: Figure 6.1 slightly redrawn

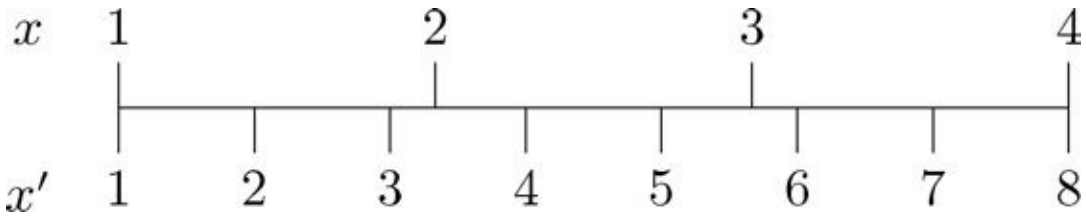
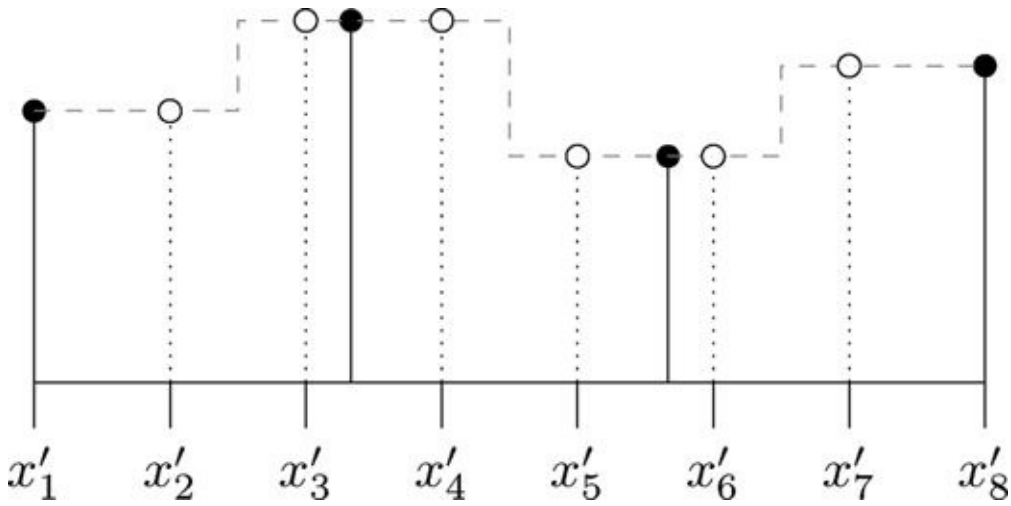
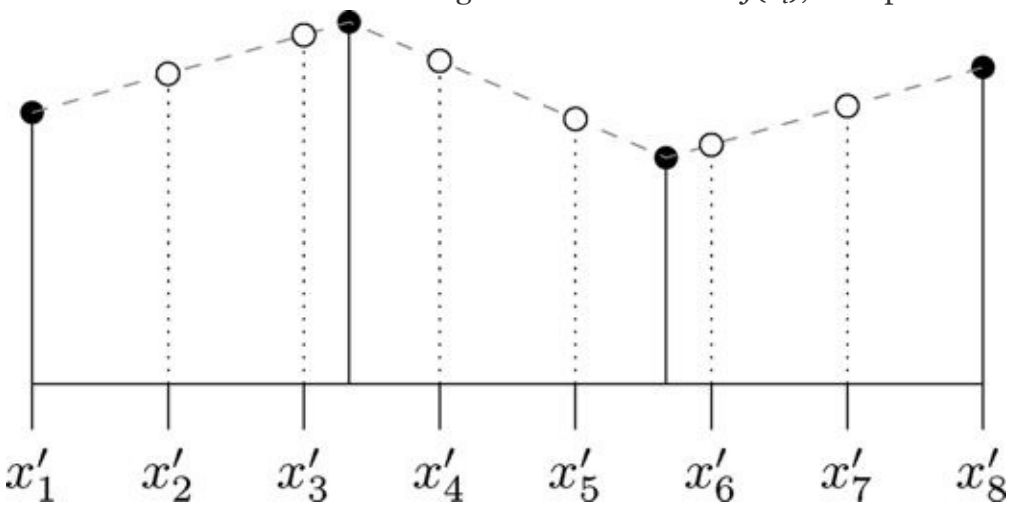


Figure 6.3: Nearest neighbor interpolation

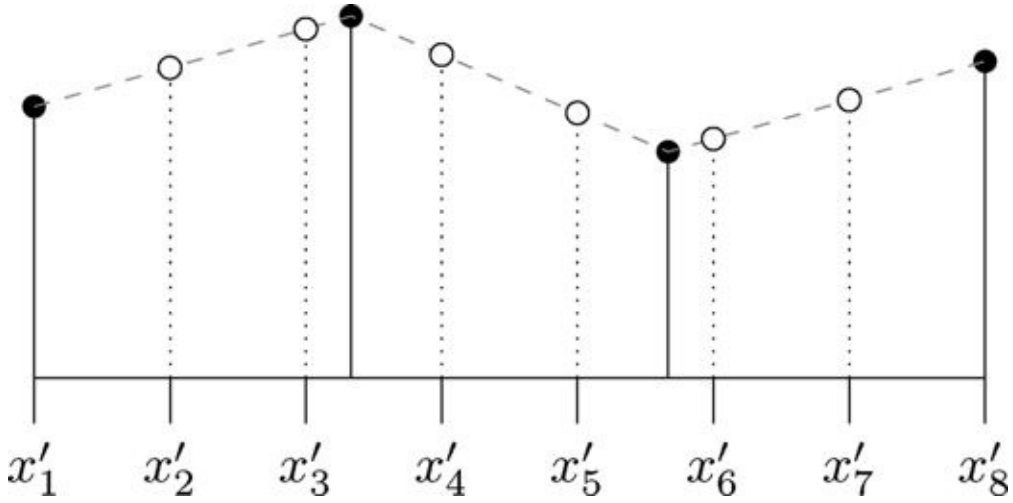


The closed circles indicate the original function values $f(x_i)$; the open circles, the interpolated values



Another way is to join the original function values by straight lines, and take our interpolated values as the values at those lines. Figure 6.4 shows this approach to interpolation; this is called *linear interpolation*.

Figure 6.4: Linear interpolation



To calculate the values required for linear interpolation, consider the diagram shown in Figure 6.5.

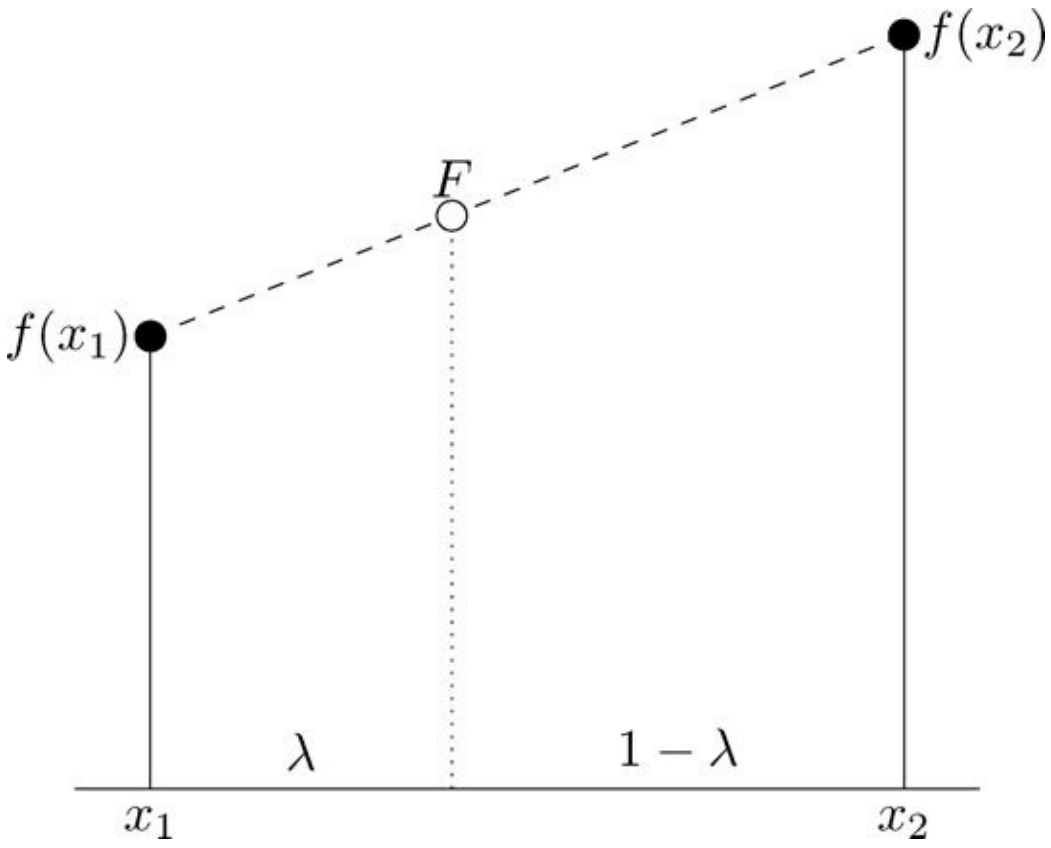
In this figure we assume that $x_2 = x_1 + 1$, and that F is the value we require. By considering slopes:

$$\frac{F - f(x_1)}{\lambda} = \frac{f(x_2) - f(x_1)}{1}.$$

Solving this equation for F produces:

$$F = \lambda f(x_2) + (1 - \lambda) f(x_1). \tag{6.1}$$

Figure 6.5: Calculating linearly interpolated values



As an example of how to use this, suppose we have the values $f(x_1) = 2$, $f(x_2) = 3$, $f(x_3) = 1.5$ and $f(x_4) = 2.5$. Consider the point x'_4. This is between x_2 and x_3, and the corresponding value for λ is 2/7. Thus

$$
\begin{aligned}
f(x'_4) &= \frac{2}{7} f(x_3) + \frac{5}{7} f(x_2) \\
&= \frac{2}{7}(1.5) + \frac{5}{7}(3) \\
&\approx 2.5714.
\end{aligned}
$$

For x'_7, we are between x_3 and x_4 with $\lambda = 4/7$. So:

$$
\begin{aligned}
f(x'_7) &= \frac{4}{7} f(x_4) + \frac{3}{7} f(x_3) \\
&= \frac{4}{7}(2.5) + \frac{3}{7}(1.5) \\
&\approx 2.0714.
\end{aligned}
$$

6.2 Image Interpolation

The methods of the previous section can be applied to images. Figure 6.6 shows how a 4 × 4 image would be interpolated to produce an 8 × 8 image. Here the large open circles are the original points, and the smaller closed circles are the new points.

To obtain function values for the interpolated points, consider the diagram shown in Figure 6.7.

We can give a value to $f(x', y')$ by either of the methods above: by setting it equal to the function values of the closest image point, or by using linear interpolation. We can apply linear interpolation first along the top row to obtain a value for $f(x, y')$, and then along the bottom row to obtain a value for $f(x + 1, y')$. Finally, we can interpolate along the y' column between these new values to obtain $f(x', y')$. Using the formula given by

Figure 6.6: Interpolation on an image

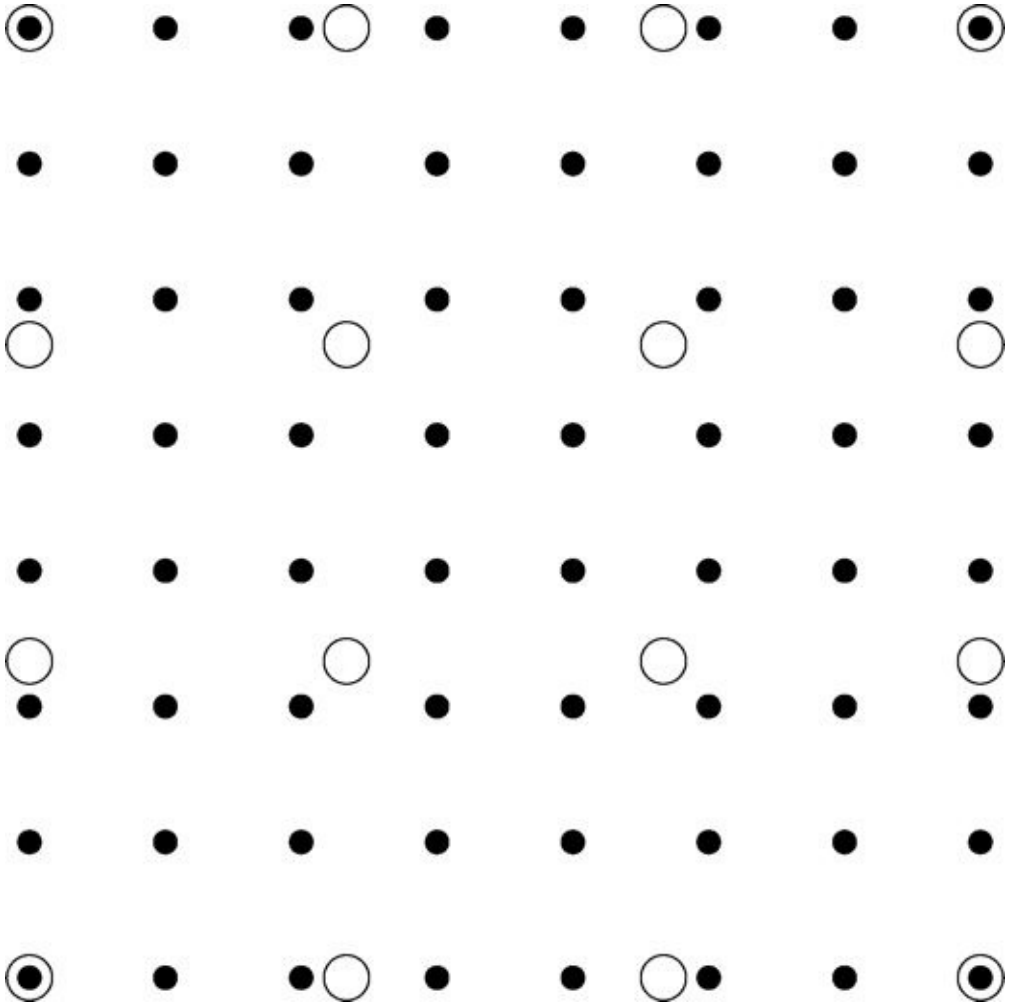
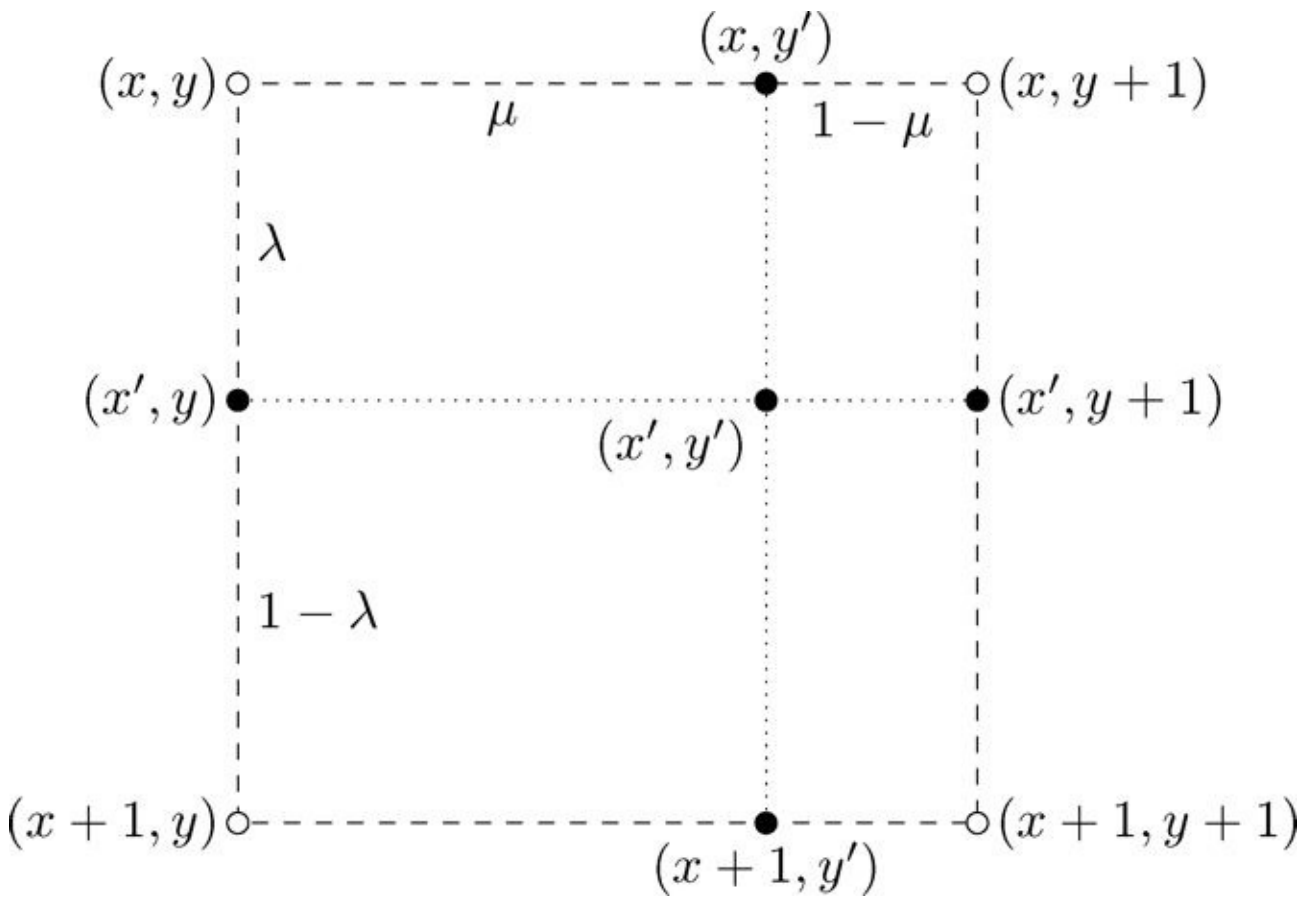


Figure 6.7: Interpolation between four image points



Equation 6.1, then

$$f(x, y') = \mu f(x, y+1) + (1-\mu) f(x, y)$$

and

$$f(x+1, y') = \mu f(x+1, y+1) + (1-\mu) f(x+1, y).$$

Along the y' column we have

$$f(x', y') = \lambda f(x+1, y') + (1-\lambda) f(x, y')$$

and substituting in the values just obtained produces

$$\begin{aligned} f(x', y') &= \lambda(\mu f(x+1, y+1) + (1-\mu) f(x+1, y)) + (1-\lambda)(\mu f(x, y+1) \\ &\quad + (1-\mu) f(x, y)) \\ &= \lambda\mu f(x+1, y+1) + \lambda(1-\mu) f(x+1, y) + (1-\lambda)\mu f(x, y+1) \\ &\quad + (1-\lambda)(1-\mu) f(x, y) \end{aligned}$$

This last equation is the formula for *bilinear interpolation*.

Now image scaling can be performed easily. Given our image, and either a scaling factor (or separate scaling factors for x and y directions), or a size to be scaled to, we first create an array of the required size. In our example above, we had a 4 × 4 image, given as an array (x, y), and a scale factor of two, resulting in an array (x', y') of size 8 × 8. Going right back to Figures 6.1 and 6.2, the relationship between (x, y) and (x', y') is

$$(x', y') = \left(\frac{1}{3}(7x - 4), \frac{1}{3}(7y - 4) \right),$$

$$(x, y) = \left(\frac{1}{7}(3x' + 4), \frac{1}{7}(3y' + 4) \right).$$

Given our (x', y') array, we can step through it point by point, and from the corresponding surrounding points from the (x, y) array calculate an interpolated value using either nearest neighbor or bilinear interpolation.

There's nothing in the above theory that requires the scaling factor to be greater than one. We can choose a scaling factor *less* than one, in which case the resulting image array will be *smaller* than the original. We can consider Figure 6.6 in this light: the small closed circles are the *original* image points, and the large open circles are the smaller array on which we are to find interpolated values.

MATLAB has the function `imresize` which does all this for us. It can be called with

```
imresize(A,k,'method')
```

where `A` is an image of any type, `k` is a scaling factor, and `'method'` is either `'nearest'` or `'bilinear'` (or another method to be described later). Another way of using `imresize` is

```
imresize(A,[m,n],'method')
```

where `[m,n]` provide the size of the scaled output. There is a further, optional parameter allowing you to choose either the size or type of low pass filter to be applied to the image before reducing its size—see the help file for details.

Let's try a few examples. We shall start by taking the head of the cameraman, and enlarging it by a factor of four:

```
>> c = imread('cameraman.png');
>> head = c(33:96,90:153);
>> imshow(head)
>> head4n = imresize(head,4,'nearest');imshow(head4n)
>> head4b = imresize(head,4,'bilinear');imshow(head4b)
```

MATLAB/Octave

Python has a `rescale` function in the `transform` module of `skimage`:

```
In :  c = io.imread('cameraman.png')
In :  head = c[32:96,89:153]
In :  io.imshow(head)
In :  head4n = tr.rescale(head,2,order=0)
In :  head4n = tr.rescale(head,2,order=1)
```

Python

The `order` parameter of the `rescale` method provides the order of the interpolating polynomial: 0 corresponds to nearest neighbor and 1 to bilinear interpolation.

The head is shown in Figure 6.8 and the results of the scaling are shown in Figure 6.9.

Figure 6.8: The head

Nearest neighbor interpolation gives an unacceptable blocky effect; edges in particular appear very jagged. Bilinear interpolation is much smoother, but the trade-off here is a certain blurriness to the result. This is unavoidable: interpolation can't predict values: we can't create data from nothing! All we can do is to guess at values that fit best with the original data.

6.3 General Interpolation

Although we have presented nearest neighbor and bilinear interpolation as two different methods, they are in fact two special cases of a more general approach. The idea is this: we wish to interpolate a value $f(x')$ for $x_1 \le x' \le x_2$, and suppose $x' - x_1 = \lambda$. We define an interpolation function $R(u)$, and set

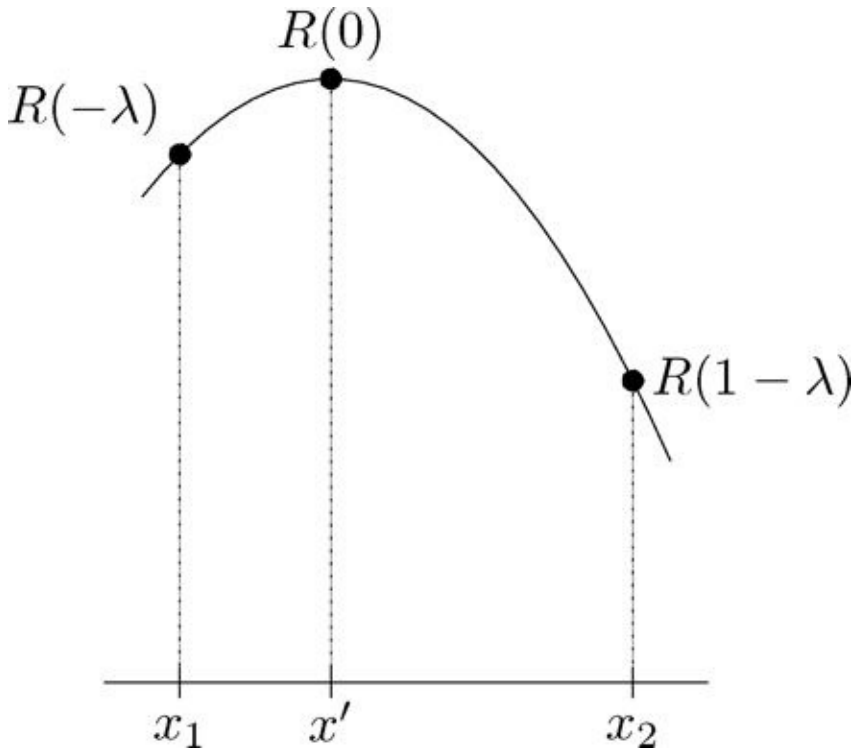
$$f(x') = R(-\lambda)f(x_1) + R(1-\lambda)f(x_2). \tag{6.2}$$

Figure 6.9: Scaling by interpolation

(a) Nearest neighbor scaling (b) Bilinear interpolation

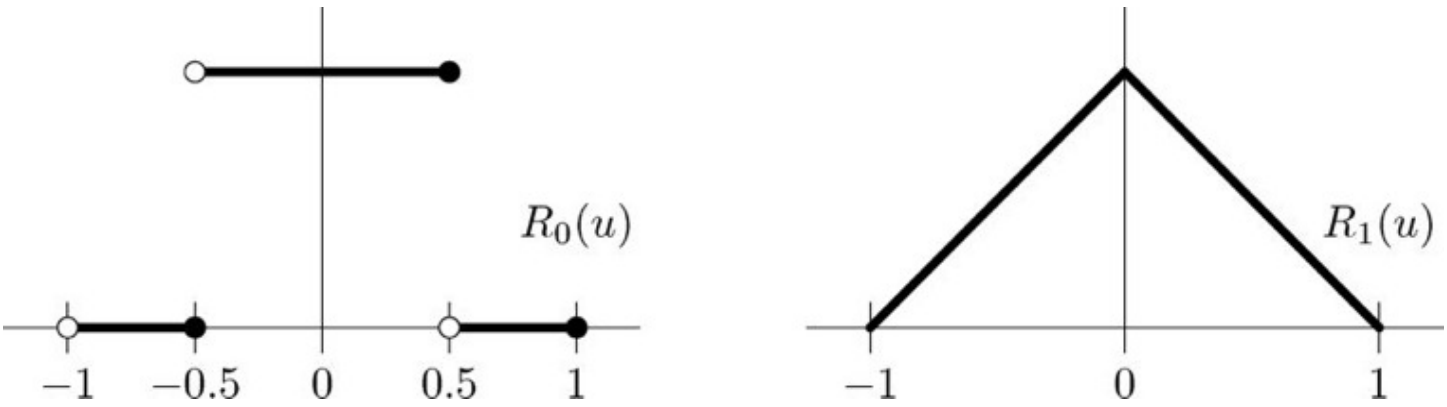
Figure 6.10 shows how this works. The function $R(u)$ is centered at x', so x_1 corresponds

Figure 6.10: Using a general interpolation function



with $u = -\lambda$, and x_2 with $u = 1 - \lambda$. Now consider the two functions $R_0(u)$ and $R_1(u)$ shown in Figure 6.11. Both these functions are defined on the interval $-1 \le u \le 1$ only. Their formal definitions are:

Figure 6.11: Two interpolation functions



$$R_0(u) = \begin{cases} 0 & \text{if } u \le -0.5 \\ 1 & \text{if } -0.5 < u \le 0.5 \\ 0 & \text{if } u > 0.5 \end{cases}$$

and

$$R_1(u) = \begin{cases} 1+u & \text{if } u \le 0 \\ 1-u & \text{if } u \ge 0 \end{cases}$$

The function $R_1(u)$ can also be written as $1 - |x|$. Now substituting $R_0(u)$ for $R(u)$ in Equation 6.2 will produce nearest neighbor interpolation. To see this, consider the two cases $\lambda < 0.5$ and $\lambda \ge 0.5$ separately. If $\lambda < 0.5$, then $R_0(-\lambda) = 1$ and $R_0(1 - \lambda) = 0$. Then

$$f(x') = (1)f(x_1) + (0)f(x_2) = f(x_1).$$

If $\lambda \geq 0.5$, then $R_0(-\lambda) = 0$ and $R_0(1-\lambda) = 1$. Then

$$f(x') = (0)f(x_1) + (1)f(x_2) = f(x_2).$$

In each case $f(x')$ is set to the function value of the point closest to x'.

Similarly, substituting $R_1(u)$ for $R(u)$ in Equation 6.2 will produce linear interpolation. We have

$$\begin{aligned} f(x') &= R_1(-\lambda)f(x_1) + R_1(1-\lambda)f(x_2) \\ &= (1-\lambda)f(x_1) + \lambda f(x_2) \end{aligned}$$

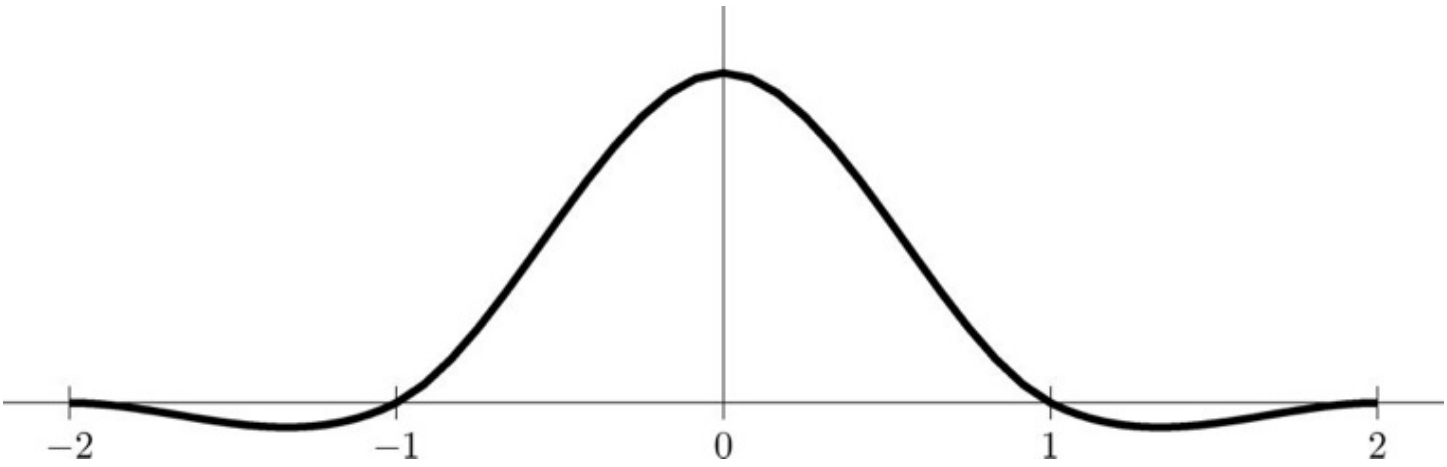
which is the correct equation.

The functions $R_0(u)$ and $R_1(u)$ are just two members of a family of possible interpolation functions. Another such function provides *cubic interpolation*; its definition is:

$$R_3(u) = \begin{cases} 1.5|u|^3 - 2.5|u|^2 + 1 & \text{if } |u| \leq 1, \\ -0.5|u|^3 + 2.5|u|^2 - 4|u| + 2 & \text{if } 1 < |u| \leq 2. \end{cases}$$

Its graph is shown in Figure 6.12. This function is defined over the interval $-2 \leq u \leq 2$,

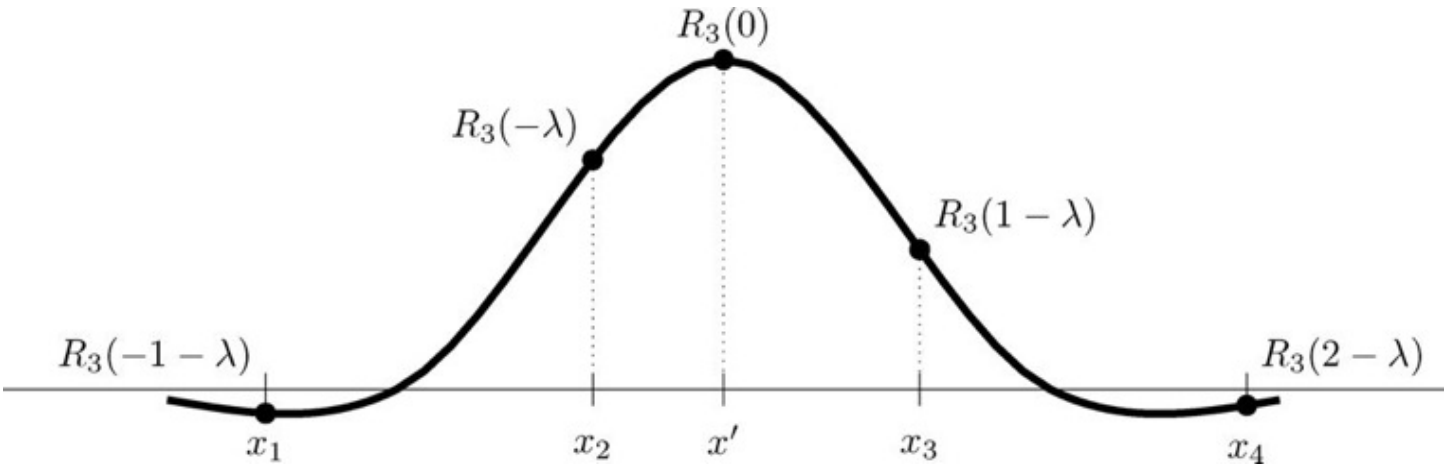
Figure 6.12: The cubic interpolation function $R_3(u)$



and its use is slightly different from that of $R_0(u)$ and $R_1(u)$, in that as well as using the function values $f(x_1)$ and $f(x_2)$ for x_1 and x_2 on either side of x', we use values of x further away. In fact the formula we use, which extends Equation 6.2, is:

$$f(x') = R_3(-1-\lambda)f(x_1) + R_3(-\lambda)f(x_2) + R_3(1-\lambda)f(x_3) + R_4(2-\lambda)f(x_4)$$

Figure 6.13: Using $R_3(u)$ for interpolation



where x' is between x_2 and x_3, and $x - x_2 = \lambda$. Figure 6.13 illustrates this. To apply this interpolation to images, we use the 16 known values around our point (x', y'). As for bilinear interpolation, we first interpolate along the rows, and then finally down the columns, as shown in Figure 6.14. Alternately, we could first interpolate down the columns, and then across the row. This means of image interpolation by applying cubic interpolation in both directions is called *bicubic interpolation*. To perform bicubic interpolation on an

Figure 6.14: How to apply bicubic interpolation

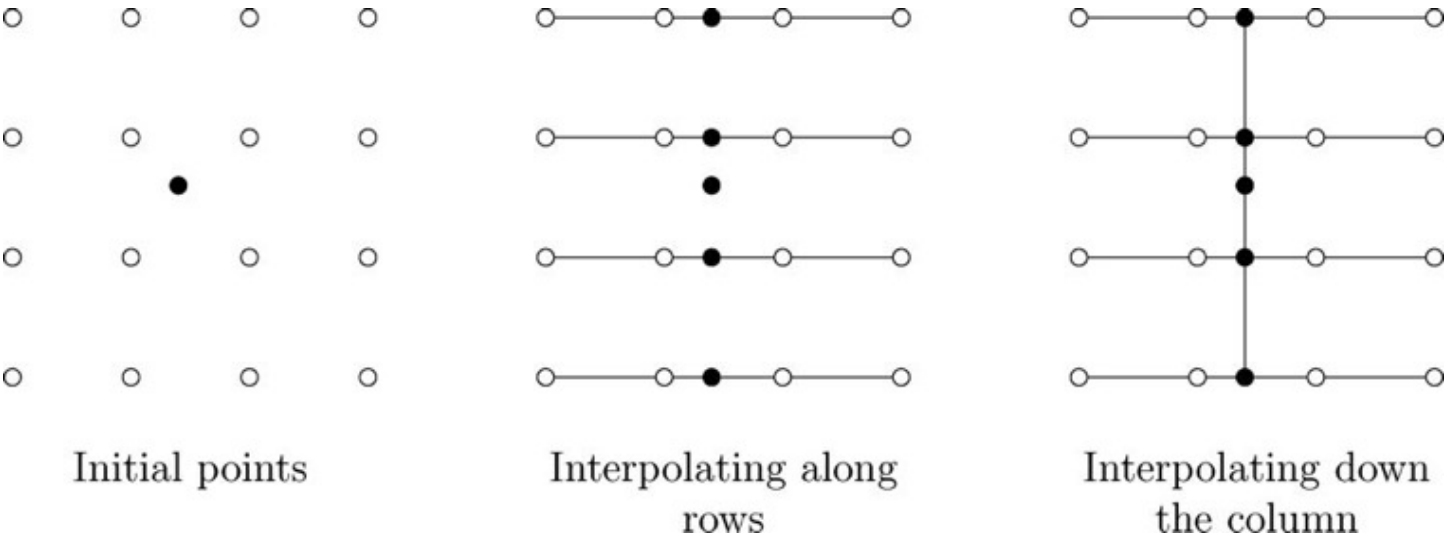


image with MATLAB, we use the `'bicubic'` method of the `imresize` function. To enlarge the cameraman's head, we enter (for `MATLAB` or Octave):

```
>> head4c = imresize(head,4,'bicubic');imshow(head4c)
```

MATLAB/Octave

In Python the `order` parameter of `rescale` must be set to 3 for bicubic interpolation:

```
In :  head4c = tr.rescale(head,4,order=3)
```

Python

and the result is shown in Figure 6.15.

Figure 6.15: Enlargement using bicubic interpolation

6.4 Enlargement by Spatial Filtering

If we merely wish to enlarge an image by a power of 2, there is a quick and dirty method which uses linear filtering. We give an example. Suppose we take a simple 4 × 4 matrix:

```
>> m = magic(4)

m =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

MATLAB/Octave

Our first step is to create a *zero-interleaved* version of this matrix. This is obtained by interleaving rows and columns of zeros between the rows and columns of the original matrix. Such a matrix will be double the size of the original, and will contain mostly zeros. If m_2 is the zero-interleaved version of m, then it is defined by:

$$m_2(i,j) = \begin{cases} m((i+1)/2,(j+1)/2) & \text{if } i \text{ and } j \text{ are both odd,} \\ 0 & \text{otherwise.} \end{cases}$$

This can be implemented very simply:

```
>> [r,c] = size(m);
>> m2 = zeros(2*r,2*c);
>> m2(1:2:1*r,1:2:2*c) = m

ans =

    16     0     2     0     3     0    13     0
     0     0     0     0     0     0     0     0
     5     0    11     0    10     0     8     0
     0     0     0     0     0     0     0     0
     9     0     7     0     6     0    12     0
     0     0     0     0     0     0     0     0
     4     0    14     0    15     0     1     0
     0     0     0     0     0     0     0     0
```

MATLAB/Octave

In Python, interleaving is also easily done:

```
In :  m = array([[16,2,3,13],[5,11,10,8],[9,7,6,12],[4,14,15,1]]);
In :  r,c = m.shape
In :  m2 = zeros((2*r,2*c))
In :  m2[::2,::2] = m
```

Python

We can now replace the zeros by applying a spatial filter to this matrix. The spatial filters

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

implement nearest neighbor interpolation and bilinear interpolation, respectively. We can test this with a few commands:

```
>> imfilter(m2,[1 1 0;1 1 0;0 0 0])

ans =

    16    16     2     2     3     3    13    13
    16    16     2     2     3     3    13    13
     5     5    11    11    10    10     8     8
     5     5    11    11    10    10     8     8
     9     9     7     7     6     6    12    12
     9     9     7     7     6     6    12    12
     4     4    14    14    15    15     1     1
     4     4    14    14    15    15     1     1

>> format bank
>> imfilter(m2,[1 2 1;2 4 2;1 2 1]/4)

ans =

   16.00    9.00    2.00    2.50    3.00    8.00   13.00    6.50
   10.50    8.50    6.50    6.50    6.50    8.50   10.50    5.25
    5.00    8.00   11.00   10.50   10.00    9.00    8.00    4.00
    7.00    8.00    9.00    8.50    8.00    9.00   10.00    5.00
    9.00    8.00    7.00    6.50    6.00    9.00   12.00    6.00
    6.50    8.50   10.50   10.50   10.50    8.50    6.50    3.25
    4.00    9.00   14.00   14.50   15.00    8.00    1.00    0.50
    2.00    4.50    7.00    7.25    7.50    4.00    0.50    0.25
```

MATLAB/Octave

(The `format bank` provides an output with only two decimal places.) In Python, the filters can be applied as:

```
In : ndi.convolve(m2,array([[0,0,0],[0,1,1],[0,1,1]]),mode='constant')
In : ndi.convolve(m2,array([[1,2,1],[2,4,2],[1,2,1]])/4.0,mode='constant')
```

Python

We can check these with the commands

```
>> m2b=imresize(m,[8,8],'nearest');m2b

>> m2b=imresize(m,[7,7],'bilinear');m2b
```

MATLAB/Octave

In the second command we only scaled up to 7×7, to ensure that the interpolation points lie exactly half-way between the original data values. The filter

$$\frac{1}{64}\begin{pmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{pmatrix}$$

can be used to approximate bicubic interpolation.

We can try all of these with the cameraman's head, doubling its size.

```
>> hz = uint8(zeros(size(head)*2));
>> hz(1:2:2nd,1:2:end) = head;
>> imshow(hz)
>> imshow(imfilter(hz, [1 1 0;1 1 0;0 0 0])
>> imshow(imfilter(hz,[1 2 1;2 4 2;1 2 1]/4)
>> bfilt=[1 4 6 4 1;4 16 24 16 4;6 24 36 24 6;4 16 24 16 4;1 4 6 4 1]/64;
>> imshow(imfilter(hz,bfilt))
```

MATLAB/Octave

or in Python with

```
In :  r,c = head.shape
In :  hz = np.zeros((2*r,2*c)).astype('uint8')
In :  hz[::2,::2] = head
In :  ne = array([[0,0,0],[0,1,1],[1,1,1]])
In :  bi = array([[1,2,1],[2,4,2],[1,2,1]])/4.0
In :  bc = array
    ([[1,4,6,4,1],[4,16,24,16,4],[6,24,36,24,6],[4,16,24,16,4],[1,4,6,4,1]])
    /64.0
In :  io.imshow(ndi.correlate(hz,ne))
```

Python

and similarly for the others. The results are shown in Figure 6.16. We can enlarge more by simply taking the result of the filter, applying a zero-interleave to it, and then another filter.

Figure 6.16: Enlargement by spatial filtering

6.5 Scaling Smaller

The business of making an image smaller is also called *image minimization*; one way is to take alternate pixels. If we wished to produce an image one-sixteenth the size of the original, we would take out only those pixels (i, j) for which i and j are both multiples of four. This method is called image *subsampling* and corresponds to the `nearest` option of `imresize`, and is very easy to implement.

However, it does not give very good results at high frequency components of an image. We shall give a simple example; we shall construct a large image consisting of a white square with a single circle on it. The `meshgrid` command provides row and column indices which span the array:

```
>> [x,y] = meshgrid(-255:256);
>> z = sqrt(x.^2+y.^2);
>> t = 1 - (z>254.5 & z<256);
>> imshow(t)
```

MATLAB/Octave

or

```
In :  r = range(-256,256)
In :  [x,y] = np.meshgrid(r,r)
In :  z = sqrt(x**2 + y**2)
In :  t = 1-((z>254.5) & (z<256))*1
In :  io.imshow(t)
```

Python

Now we can resize it by taking out most pixels:

```
>> t2 = imresize(t,0.25,'nearest');
```

MATLAB/Octave

or

```
In :  t2 = tr.rescale(t,0.25,order=0)
```

Python

and this is shown in Figure 6.17(a). Notice that because of the way that pixels were removed, the resulting circle contains gaps. If we were to use one of the other methods:

```
>> t3 = imresize(t,0.25,'bicubic');
```

MATLAB/Octave

or

```
In :  t3 = tr.rescale(t,0.25,order=3)
```
Python

a low pass filter is applied to the image first. The result is shown in Figure 6.17(b). The image in Figure 6.17(b) can be made binary by thresholding (which will be discussed in greater detail in Chapter 9). In this case

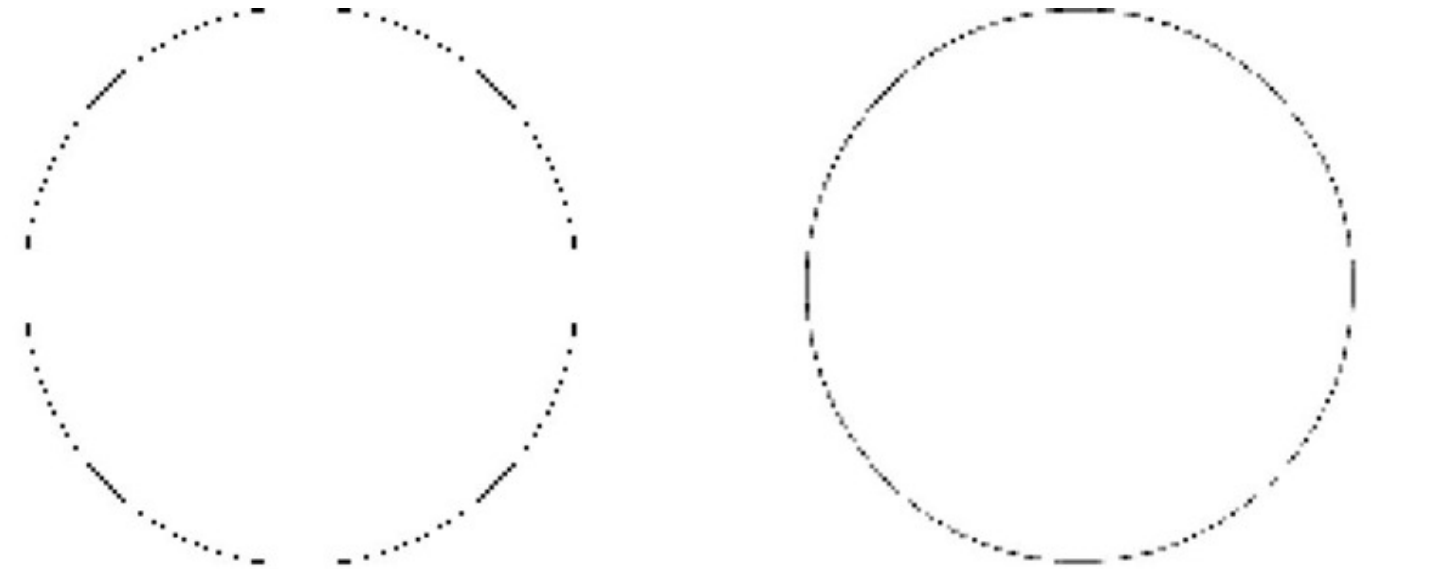
```
>> t4 = imresize(t,0.25,'bicubic')>0.9;
```
MATLAB/Octave

or

```
In :  t4 = tr.rescale(t,0.25,order=3)>0.9
```
Python

does the trick.

Figure 6.17: Minimization



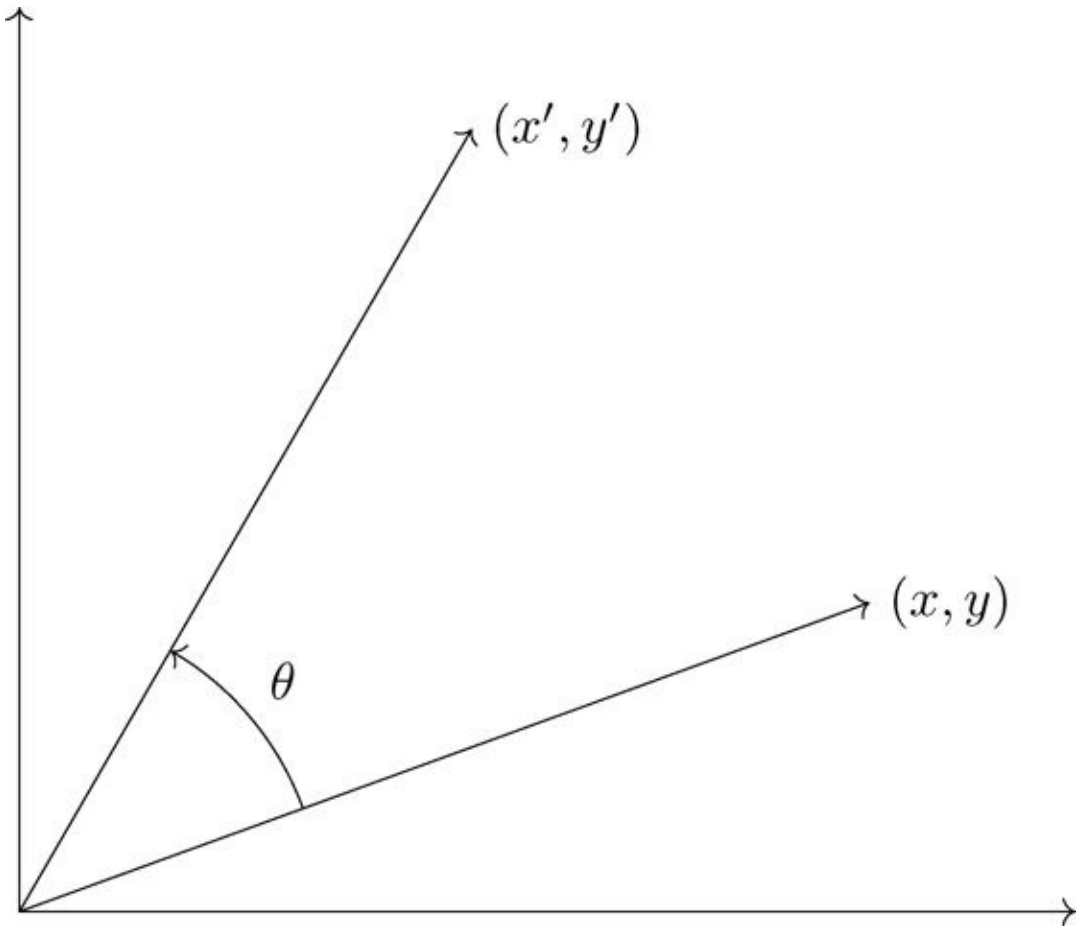
(a) Nearest neighbor minimization (b) Bicubic interpolation for minimization

6.6 Rotation

Having done the hard work of interpolation for scaling, we can easily apply the same theory to image rotation. First recall that the mapping of a point (x, y) to another (x', y') through a counter-clockwise rotation of θ as shown in Figure 6.18 is obtained by the matrix product

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Figure 6.18: Rotating a point through angle θ

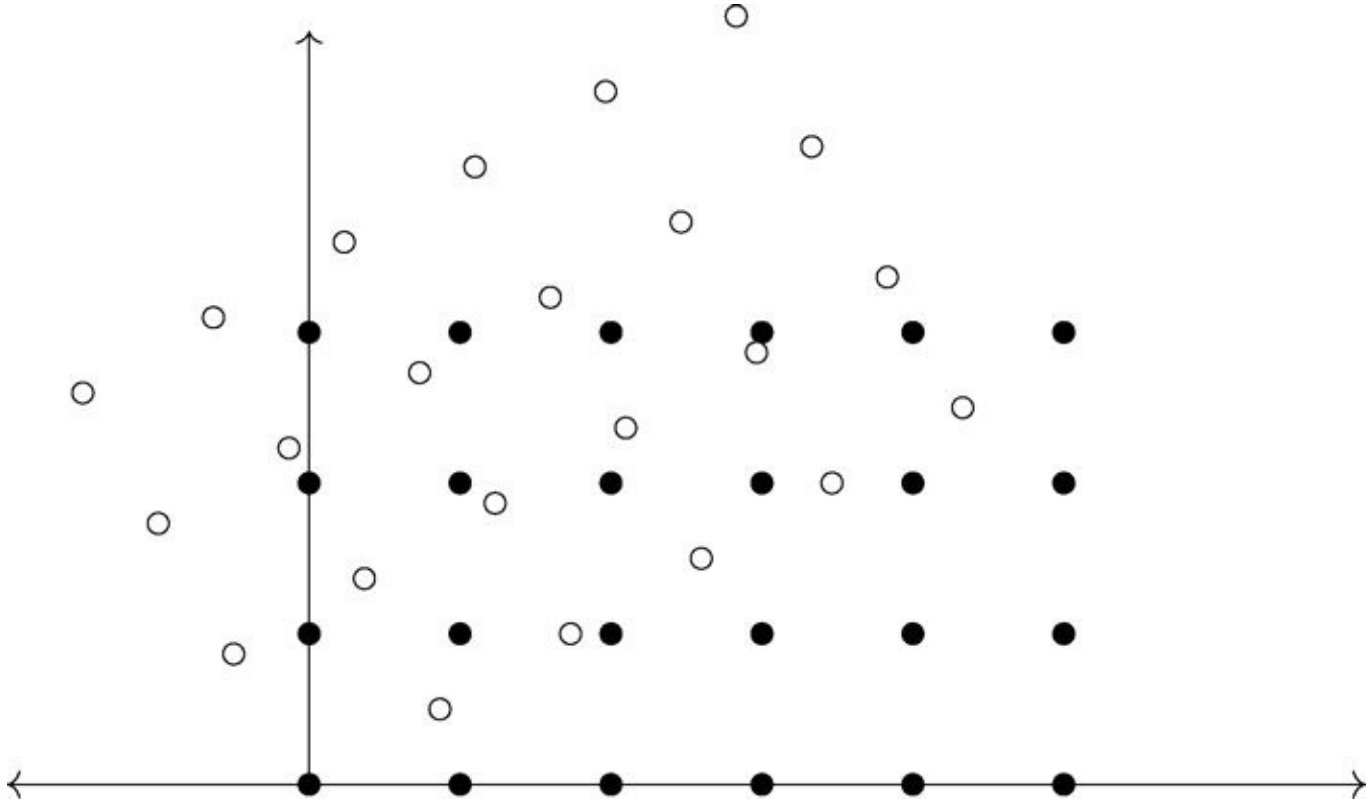


Similarly, since the matrix involved is orthogonal (its inverse is equal to its transpose), we have:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}.$$

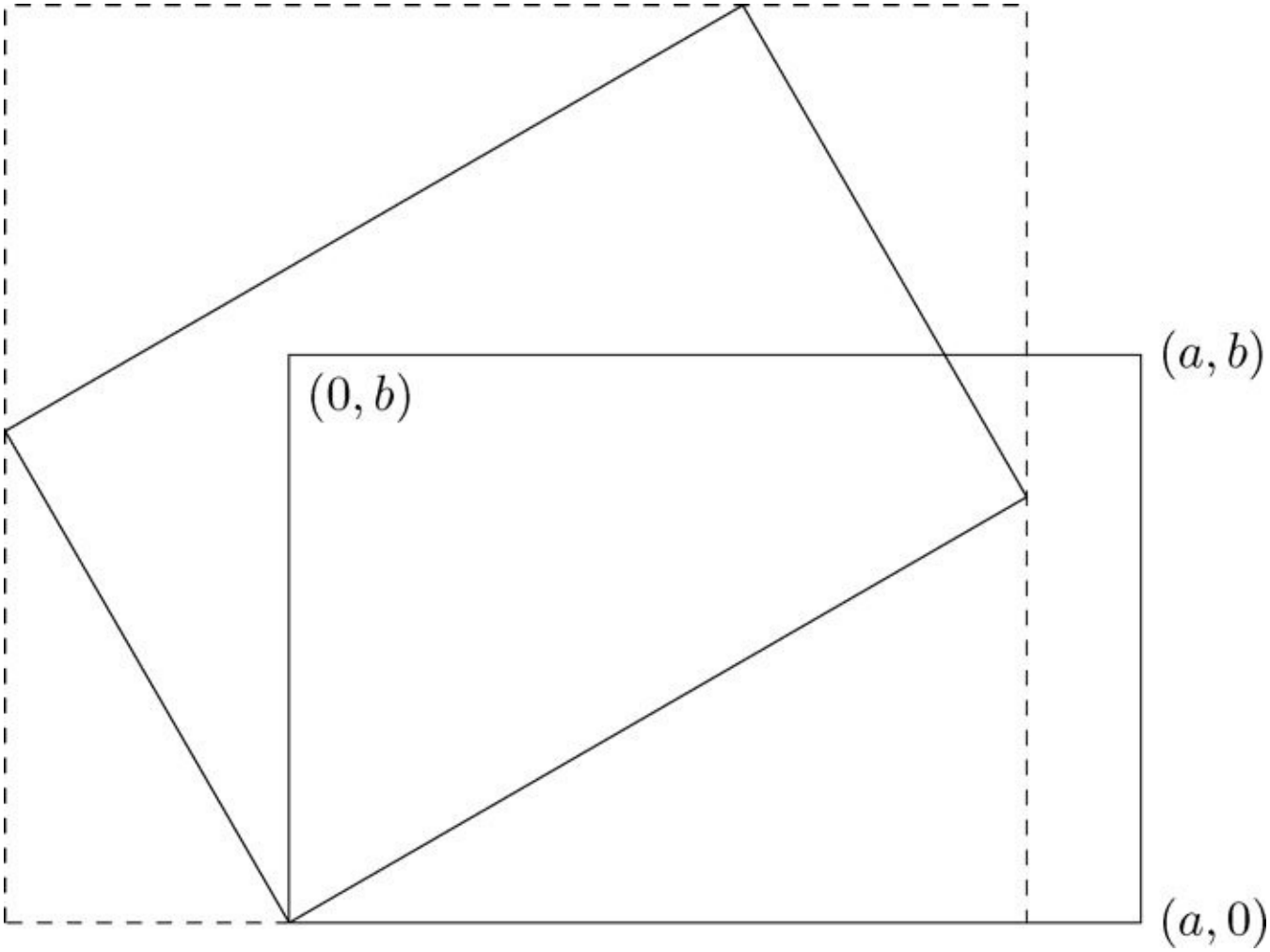
Now we can rotate an image by considering it as a large collection of points. Figure 6.19 illustrates the idea. In this figure, the dark circles indicate the original position; the light

Figure 6.19: Rotating a rectangle



points their positions after rotation. However, this approach won't work for images. Since an image grid can be considered as pixels forming a subset of the Cartesian (integer valued) grid, we must ensure that even after rotation, the points remain in that grid. To do this we consider a rectangle that includes the rotated image, as shown in Figure 6.20. Now consider

Figure 6.20: A rectangle surrounding a rotated image



all integer-valued points (x', y') in the dashed rectangle. A point will be in the image if, when rotated back, it lies within the original image limits. That is, if

$$\begin{array}{rcl} 0 \leq & x' \cos\theta + y' \sin\theta & \leq a \\ 0 \leq & -x' \sin\theta + y' \cos\theta & \leq b \end{array}$$

If we consider the array of 6 × 4 points shown in Figure 6.19, then the points after rotation by 30° are shown in Figure 6.21 This gives us the position of the pixels in our rotated image,

but what about their value? Take a point (x', y') in the rotated image, and rotate it back into the original image to produce a point (x'', y''), as shown in Figure 6.22. Now the gray value at (x'', y'') can be found by interpolation using surrounding gray values. This value is then the gray value for the pixel at (x', y') in the rotated image.

Figure 6.21: The points on a grid after rotation

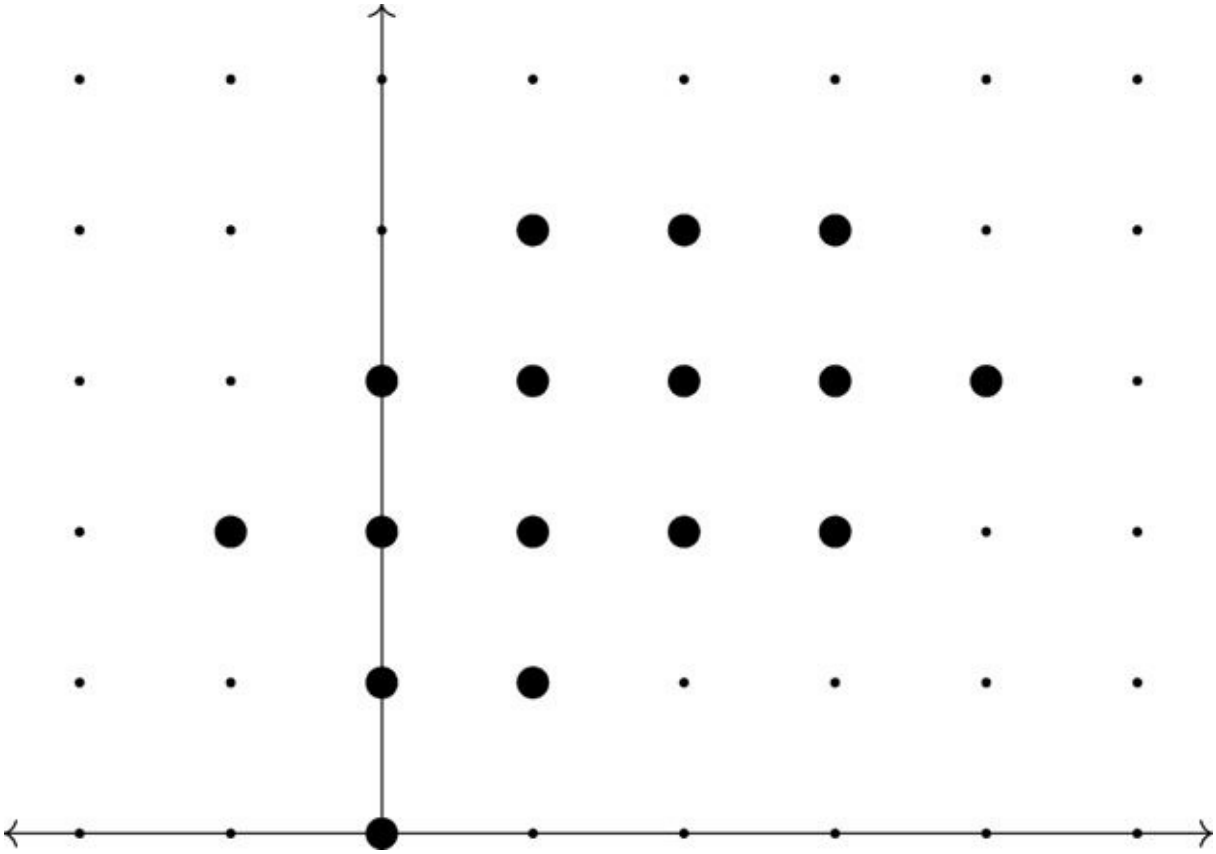


Figure 6.22: Rotating a point back into the original image

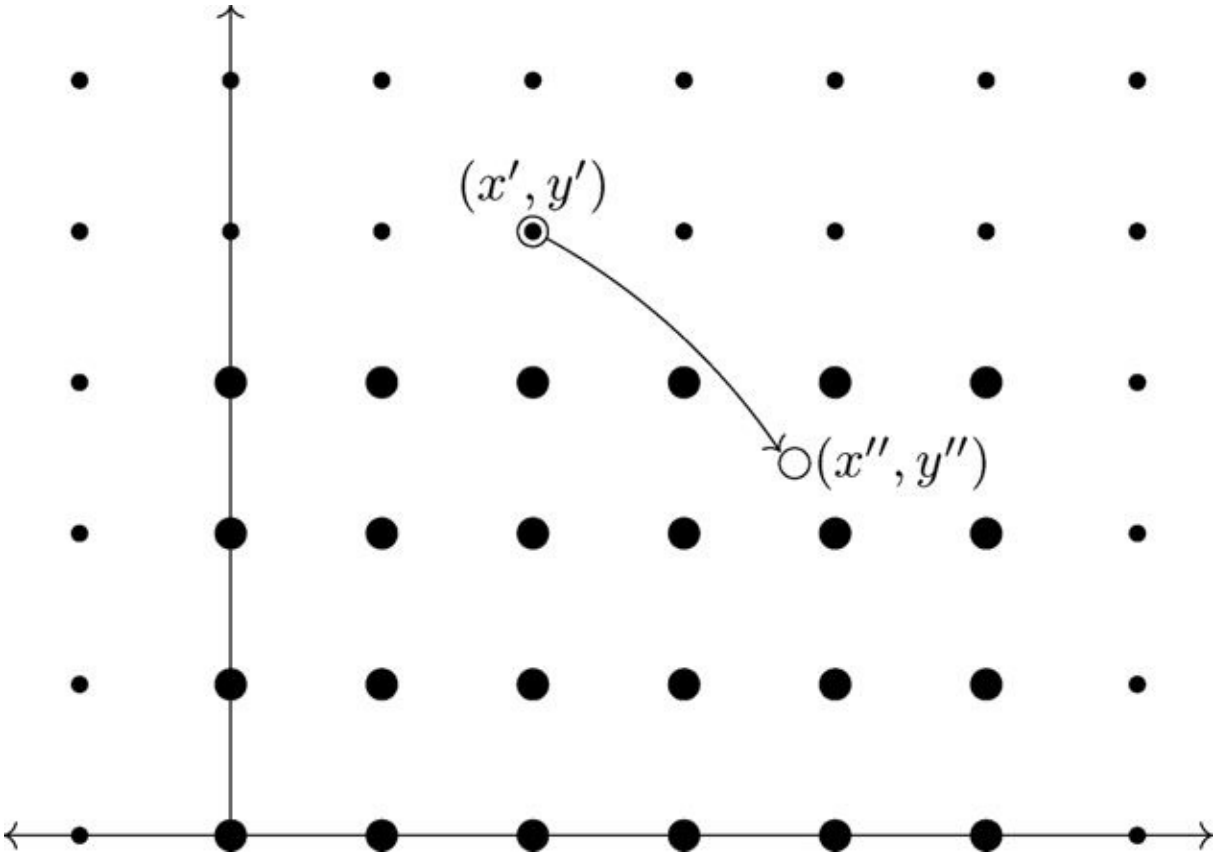


Image rotation in MATLAB is obtained using the command `imrotate`; it has the syntax

```
imrotate(image,angle,'method')
```

where `method`, as with `imresize`, can be any one of `nearest, bilinear`, or `bicubic`. Also as with `imresize`, the `method` parameter may be omitted, in which case nearest neighbor interpolation is used. Python has the `rotate` method in the `transform` module of `skimage`, with syntax

```
transform.rotate(image,angle,[resize],[order],[mode])
```

Here the last three parameters are optional; `order` (default is 1) is the polynomial order of the interpolation.

For an example, let's take our old friend the cameraman, and rotate him by 60°. We shall do this twice; once with nearest neighbor interpolation, and once using bicubic interpolation. With MATLAB or Octave, the commands are:

```
>> cr = imrotate(c,60);
>> imshow(cr)
>> crc = imrotate(c,60,'bicubic');
>> figure,imshow(crc)
```

MATLAB/Octave

and in Python the commands are

```
In :  cr = tr.rotate(c,60,order=0)
In :  io.imshow(cr)
In :  crc = tr.rotate(c,60,order=3)
In :  f = figure(), f.show(io.imshow(crc))
```

Python

The results are shown in Figure 6.23. There's not a great deal of observable difference between the two images; however, nearest neighbor interpolation produces slightly more jagged edges.

Figure 6.23: Rotation with interpolation

(a) Nearest neighbor

(b) Bicubic interpolation

Notice that for angles that are integer multiples of 90° image rotation can be accomplished far more efficiently with simple matrix transposition and reversing of the order of rows and columns. These commands can be used:

MATLAB/Octave	Python	Result
`flipud`	`np.flipud`	Flips a matrix in the up/down direction
`fliplr`	`np.fliplr`	Flips a matrix in the left/right direction
`rot90`	`np.rot90`	Rotates a matrix by 90°

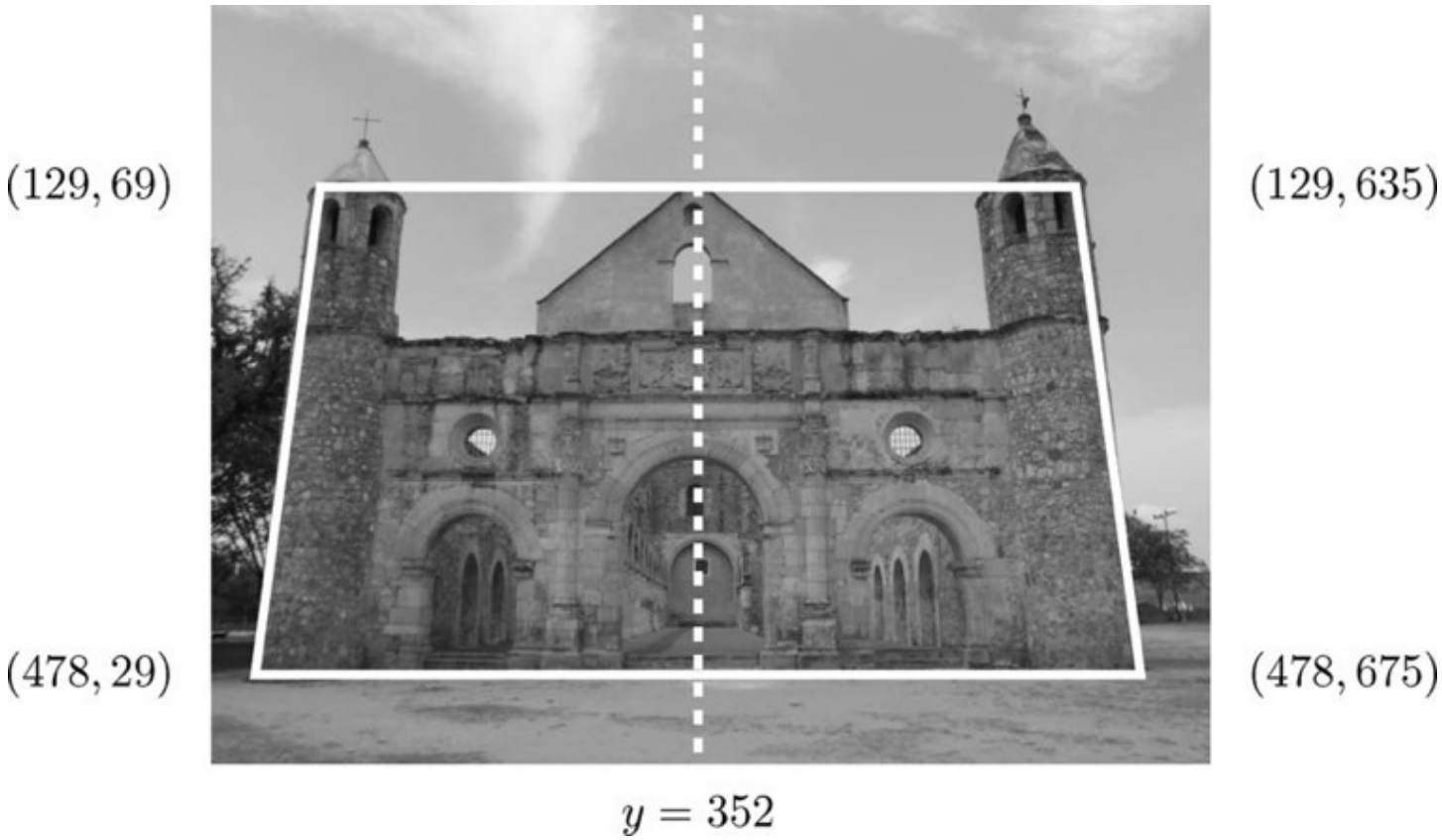
In fact, `imrotate` uses these simpler commands for these particular angles.

6.7 Correcting Image Distortion

This is in fact a huge topic: there are many different possible distortions caused by optical aberrations and effects. So this section will investigate just one: *perspective distortion*, which is exemplified by the image in Figure 6.24. Because of the position of the camera lens relative to the building, the towers appear to be leaning inward. Fixing this requires a little algebra. First note that the building is contained within a symmetric trapezoid, the corners of which can be found by carefully moving a cursor over the image and checking its coordinates.

Figure 6.24: Perspective distortion

Figure 6.25: The corners of the building

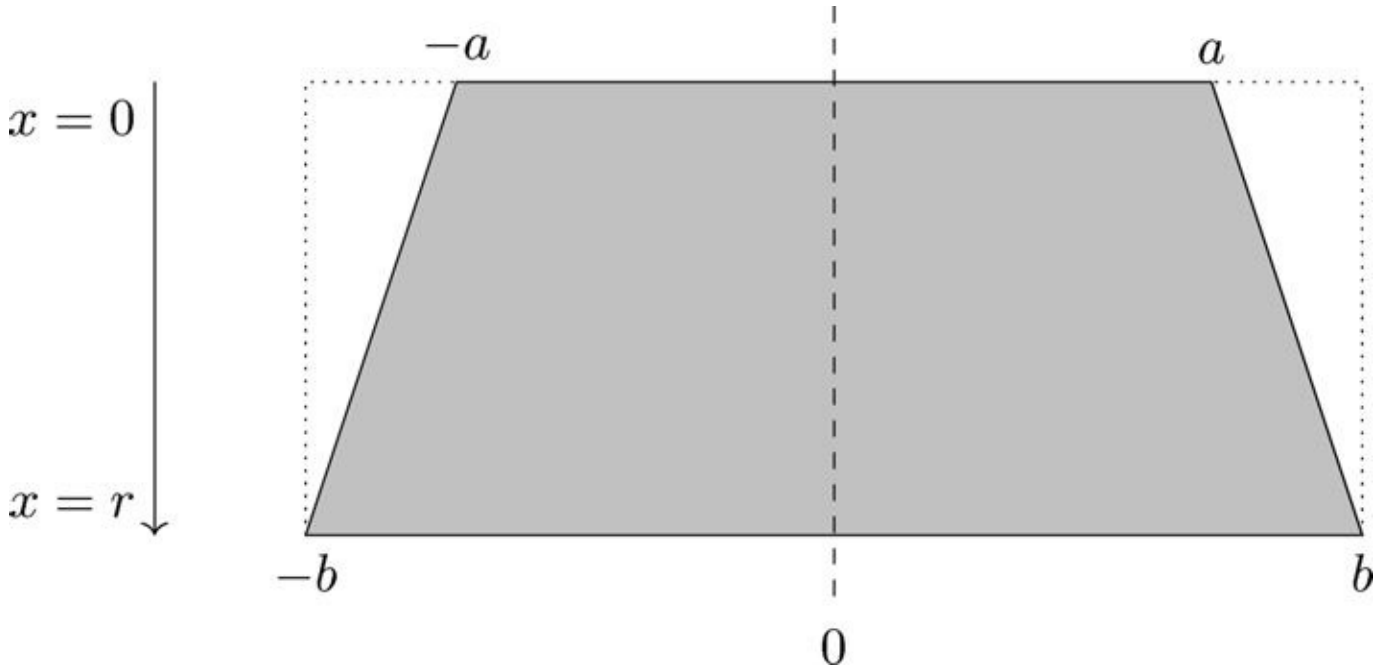


The trapezoid can be seen to be centered at y = 352. The problem now is to warp the trapezoid into a rectangle. This warping is built in to many current picture processing software tools; the user can implement such a warp by drawing lines over the edges and then dragging those lines in such a way to fix

the distortion. Here we investigate the mathematics and the background processing behind such an operation.

In general, suppose a symmetric trapezoid of height r is centered on the y axis, with horizontal lengths of $2a$ and $2b$, as shown in Figure 6.26.

Figure 6.26: A general symmetric trapezoid



To warp this trapezoid to a rectangle, each horizontal line must be stretched by an amount str(x), where str(r) = 1 and str(0) = b/a. Since the stretching amount is a linear function of x, the values given mean that

$$\begin{aligned}\text{str}(x) &= \frac{\frac{b}{a}-1}{-r}(x-r)+1 \\ &= \frac{a-b}{ar}x+\frac{b}{a}.\end{aligned}$$

If (x_1, y_1) and (x_2, y_2) are points on one of the slanted sides of the trapezoid, then finding the equation of line through them and substituting $x = 0$ and $x = r$ will produce the values of a and b:

$$a = y_1 - x_1\frac{y_2-y_1}{x_2-x_1}, \qquad b = y_1 + (r-x_1)\frac{y_2-y_1}{x_2-x_1}.$$

Consider the trapezoid in Figure 6.25, with a shift by subtracting 352 from all the y values, as shown in Figure 6.27.

The previous formulas can now be applied to $(x_1, y_1) = (478, 323)$ and $(x_2, y_2) = (129, 283)$ to obtain $a \approx 268.21$ and $b \approx 331.71$. To actually perform the stretching, the transformation from the old to new image is given by

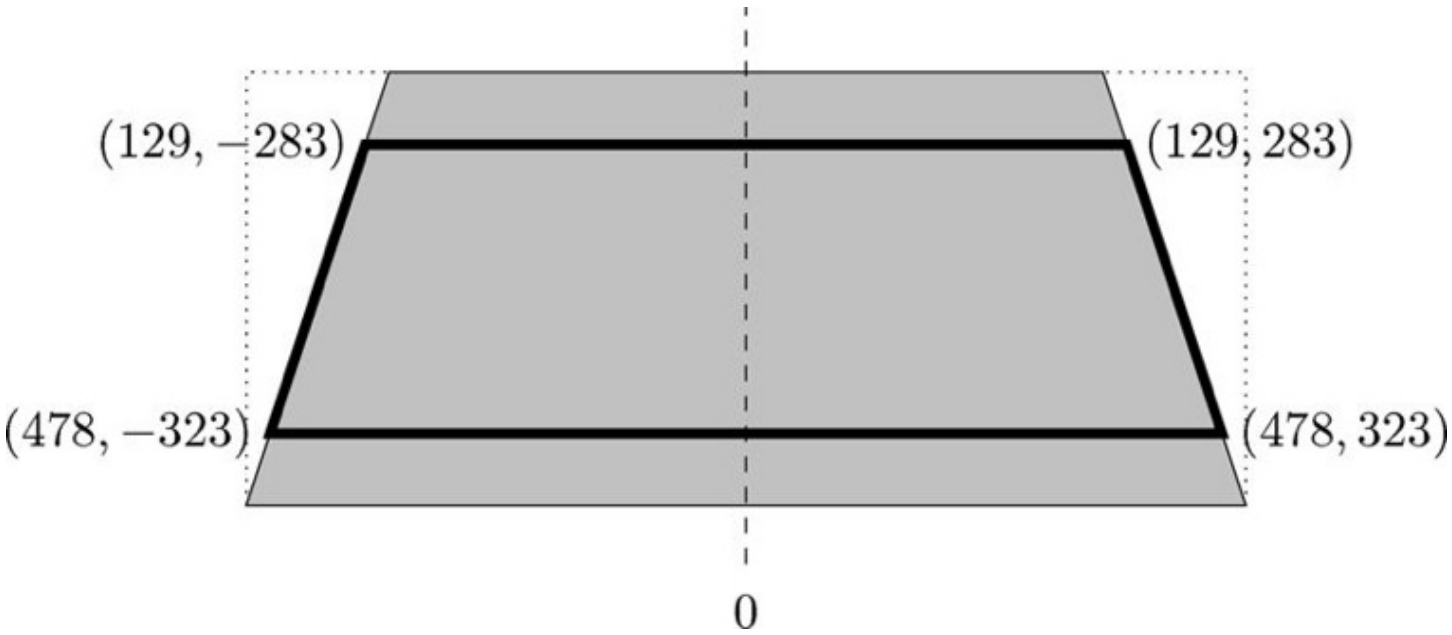
$$(x, y) \to (x, (y-352)\text{str}(x)+352).$$

In other words, a pixel at position (x, y) in the new image will take the value of the pixel at place

$$\left(x, \frac{y-352}{\text{str}(x)} + 352\right)$$

in the old image.

Figure 6.27: The trapezoid around the building



However, this involves a division, which is in general a slow operation. To rewrite as a multiplication, we would use replace pixel at (x, y) in the new image would take the place of a pixel at place

$$(x, \text{sq}(x, y - 352) + 352)$$

where sq(x) ("sq" stands for "squash") is a linear function for which

$$\begin{aligned} \text{sq}(0) &= a/b \\ \text{sq}(r) &= 1. \end{aligned}$$

This means that

$$\text{sq}(x) = \frac{b-a}{br}(x) + \frac{a}{b}.$$

Here's how this can be implemented in MATLAB or Octave, assuming that the image has been read as `im`:

```
>> [r,c] = size(im);
>> z = zeros(r,c);
>> x1 = 129; y1=283; x2=478; y2=323;
>> a = y1-x1*(y2-y1)/(x2-x1)
>> b = y1+(r-x1)*(y2-y1)/(x2-x1)
>> [y,x] = meshgrid(1:c,1:r);
>> sq = floor((y-352).*((b-a)/(b*r)*x+a/b)+352);
>> for i = 1:r,...
>    for j = 1:c,...
>      z(i,j) = im(i,sq(i,j));
>    end;...
>  end;
>> im2 = uint8(z)
```

MATLAB/Octave

and in Python as

```
In :  r,c = im.shape
In :  x1, y1, x2, y2 = 129.0, 283.0, 478.0, 323.0
In :  a = y1-x1*(y2-y1)/(x2-x1)
In :  b = y1+(r-x1)*(y2-y1)/(x2-x1)
In :  z = np.zeros_like(im)
In :  x,y = np.mgrid[0:r,0:c]
In :  sq = np.floor((y-352)*((b-a)/(b*r)*x+a/b)+352)
In :  for i in range(r):
...:      for j in range(c):
...:          z[i,j] = im[i,sq[i,j]]
...:
In :  im2 = uint8(z)
```

Python

and the result `im2` is shown in Figure 6.28. Note that the Python commands used `mgrid` which can be used as an alternative to `meshgrid` and with a more concise syntax.

Figure 6.28: The image corrected for perspective distortion

Exercises

1. By hand, enlarge the list

$$1\ 4\ 7\ 4\ 3\ 6$$

to lengths

(a) 9
(b) 11
(c) 2

by

(a) nearest neighbor interpolation
(b) linear interpolation

Check your answers with your computer system.

- 2. By hand, enlarge the matrix $\begin{pmatrix} 8 & 6 & 13 & 9 \\ 1 & 13 & 1 & 15 \\ 5 & 4 & 7 & 7 \\ 5 & 10 & 3 & 7 \end{pmatrix}$ to sizes

(i) 7×7
(ii) 8×8

(iii) 10×10

by

(a) nearest neighbor interpolation,
(b) bilinear interpolation.

Check your answers with your computer system.

- 3. Use zero-interleaving and spatial filtering to enlarge the cameraman's head by a factor of four in each dimension, using the three filters given. Use the following sequence of commands:

```
>> head2 = zeroint(head);
>> head2n = imfilter(head2,filt);
>> head4 = zeroint(head2n);
>> head4n = imfilter(head4,filt);
>> imshow(head4n/255)
```

MATLAB/Octave

where `filt` is a filter. Compare your results to those given by `imresize`. Are there any observable differences?

- 4. Take another small part of an image: say the head of the seagull in `seagull.png`. This can be obtained with:

```
>> g = imread('seagull.png');
>> head = g(110:173,272:367);
```

MATLAB/Octave

or with

```
In :  g = io.imread('seagull.png')
In :  head  = g[109:173,271:367]
```

Python

Enlarge the head to four times as big using both `imresize` with the different parameters, and the zero-interleave method with the different filters. As above, compare the results.

- 5. Suppose an image is enlarged by some amount k, and the result decreased by the same amount. Should this result be exactly the same as the original? If not, why not?
- 6. What happens if the image is decreased first, and the result enlarged?
- 7. Create an image consisting of a white square with a black background. Rotate the image by 30° and 45°. Use (a) rotation with nearest neighbor interpolation, and (b) rotation with bilinear interpolation Compare the results.
- 8. For the rotated squares in the previous question, rotate back to the original orientation. How close is the result to the original square?
- 9. In general, suppose an image is rotated, and then the result rotated back. Should this result be exactly the same as the original? If not, why not?

- 10. Write a function to implement image enlargement using zero-interleaving and spatial filtering. The function should have the syntax

```
imenlarge(image,n,filt)
```

where `n` is the number of times the interleaving is to be done, and `filt` is the filter to use. So, for example, the command

```
imenlarge(head,2,bfilt);
```

would enlarge an image to four times its size, using the 5 × 5 filter described in Section 6.4.

7 The Fourier Transform

7.1 Introduction

The Fourier Transform is of fundamental importance to image processing. It allows us to perform tasks that would be impossible to perform any other way; its efficiency allows us to perform other tasks more quickly. The Fourier Transform provides, among other things, a powerful alternative to linear spatial filtering; it is more efficient to use the Fourier Transform than a spatial filter for a large filter. The Fourier Transform also allows us to isolate and process particular image "frequencies," and so perform low pass and high pass filtering with a great degree of precision.

Before we discuss the Fourier Transform of images, we shall investigate the one-dimensional Fourier Transform, and a few of its properties.

7.2 Background

Our starting place is the observation that a periodic function may be written as the sum of sines and cosines of varying amplitudes and frequencies. For example, in Figure 7.1 we plot a function and its decomposition into sine functions.

Some functions will require only a finite number of functions in their decomposition; others will require an infinite number. For example, a "square wave," such as is shown in Figure 7.2, has the decomposition

$$f(x) = \sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x + \frac{1}{7}\sin 7x + \frac{1}{9}\sin 9x + \cdots \tag{7.1}$$

In Figure 7.2 we take the first five terms only to provide the approximation. The more terms of the series we take, the closer the sum will approach the original function.

This can be formalized; if $f(x)$ is a function of period $2T$, then we can write

$$f(x) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{n\pi x}{T} + b_n \sin\frac{n\pi x}{T}\right)$$