Interpolation

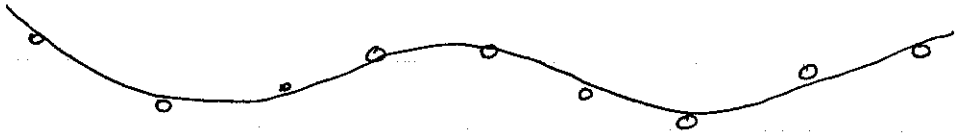
For many image processing problems, we have a relatively sparse, numerical data set that we wish to display on a regular grid of output pixels. In other words, given a few known points in an image, how can we estimate the values between the known points? This specific problem is a special case of the process of interpolation, that is, reconstructing a continuous signal from discrete data points.

In this way, interpolation can be seen as the inverse function to sampling, where continuous data are represented by a set of discrete points.

Let's first look at the general problem, and then some specific cases for our own image processing application.

Say we have the following set of points:

We can imagine a smooth curve connecting these dots together:



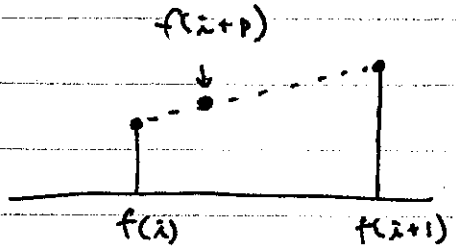
But how do we know the answer is right? And how would we do this in two dimensions? Many algorithms exist.

Nearest-neighbor algorithm

Suppose we have a sequence $f(i)$ and want to find the value at $f(i+p)$, $0 < p < 1$. The simple, "natural" thing to do might be to interpolate linearly:

$$f(i+p) = (1-p)f(i) + p f(i+1)$$

Graphically,



Sometimes we need a faster algorithm which still retains the correct value at the sample points. This might be done with the nearest-neighbor algorithm:

$$\text{if } p \le \tfrac{1}{2}, \quad f(i+p) = f(i)$$
$$\text{if } p > \tfrac{1}{2}, \quad f(i+p) = f(i+1)$$

This algorithm uses only logical operations, not computations, and is much faster than linear interpolation. A quicker method of writing the algorithm might be:

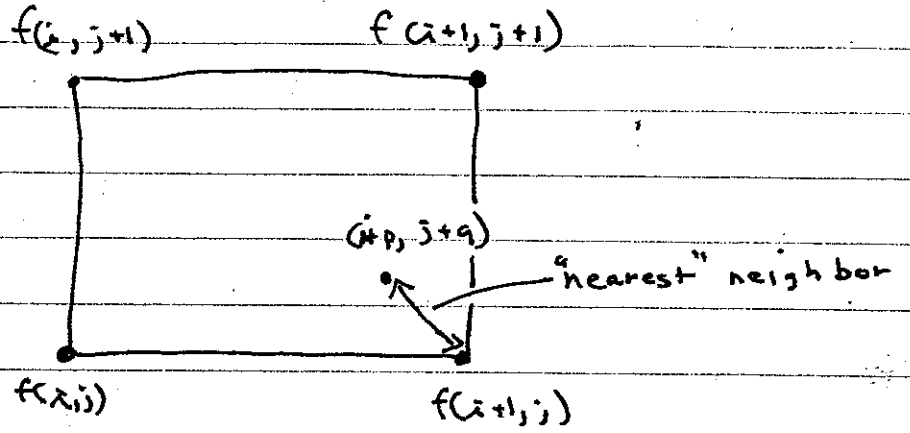
$$f(i+p) = f(\text{nint}(i+p))$$

where nint() is the nearest integer value.

The extension to 2-D is straight-forward for this algorithm:

$$f(i+p, j+q) = f(\text{nint}(i+p), \text{nint}(j+q))$$

We can picture this as follows:



Twisted-plane (sometimes called bilinear) interpolation

So what do we do if we need more accuracy than nearest-neighbor interpolation allows? In one-D, we had a simple linear interpolation scheme. Its 2-D analog is:

$$f(i+p, j+q) = (1-p)(1-q)\,f(i,j) + p(1-q)\,f(i+1, j) + q(1-p)\,f(i, j+1) + pq\,f(i+1, j+1)$$

We could visualize linear interpolation in 1-D as a string stretched between points on the line, in 2-D the model is a sheet tacked at the four corners:

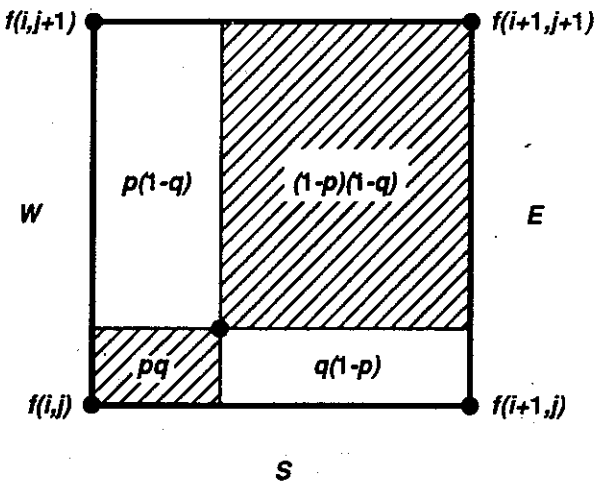


Figure 7-15 The "twisted plane" through the four corner samples has a height at $(i + p, j + q)$ that is a weighted mean of the corner heights; the weight for any corner is equal to the area opposite that corner.

We can also think of bilinear interpolation as follows: First implement 2 one-d linear interpolations of $f(i,j)$ and $f(i+1,j)$ to get $f(i+p,j)$. Similarly obtain $f(i+p,j+1)$. Then do an orthogonal linear interpolation between $f(i+p,j)$ and $f(i+p,j+1)$ to get $f(i+p,j+q)$.

Twisted plane in 2-D is actually more accurate than simple 1-D linear interpolation because the slopes are measured at two locations in each direction, rather than just one.

Six-point formula

If curvature is significant, we need to consider the second derivatives $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$, or, equivalently, the second differences. A common approach is to add the two points located at $(i-1,j)$ and $(i,j-1)$. The resulting 6-point formula is

~~$f(i+p,j+q) =$~~

$$f(i+p,j+q) = \tfrac{1}{2}q(q-1)f(i,j-1) + \tfrac{1}{2}p(p-1)f(i-1,j) + (1+pq-p^2-q^2)f(i,j) + \tfrac{1}{2}p(p-2q+1)f(i+1,j) + \tfrac{1}{2}q(q-2p+1)f(i,j+1) + pq\,f(i+1,j+1)$$

Cubic-splines

When is is more important to have a smooth curve than it is to have absolute accuracy, we may choose ~~=~~ spline interpolation to fit a smooth curve between points. In two-D this means less "noise" in the high-frequency part of the spectrum. A popular smoothing polynomial is of order 3, it uses data at four points to interpolate between the innermost 2 points:

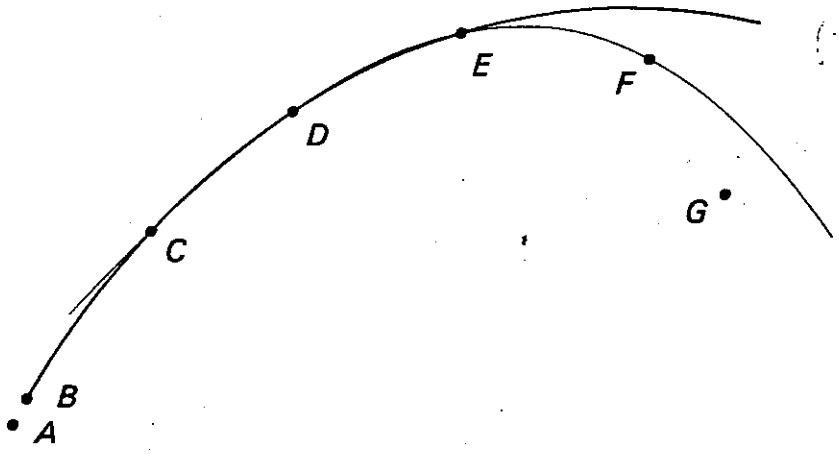


Figure 7-16 Cubic spline interpolation.

This algorithm has the advantage of smoothness, reasonable accuracy, and relatively few computations.

Midpoint Interpolation

Often we can formulate an interpolation requirement, in terms of obtaining the midpoints in a sequences - clearly this can be extended to permit multiple passes aimed at obtaining solutions spaced by $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$. This allows us, say, to zoom an image by a factor of 2, or permits us to faithfully reproduce the spectrum of the product of two bandlimited signals.

In two dimensions, sampling along midpoints in one dimension does not depend on behavior in the other dimension. This follows from the notion that if a two-D function is bandlimited, any one-D cut through it will also be bandlimited. If the samples are adequately spaced along the cut, any intermediate value may be determined from those samples alone. This may be seen by considering the image as a superposition of corrugations - clearly each corrugation may be interpolated along a cut only considering value along the cut. Doing this for each corrugation

frequency and direction means we can do the same to the total function.

We can represent linear midpoint interpolation by the following notation:

$$\{ 0.5 \uparrow 0.5 \}$$

which means the value at the arrow is equal to the sum of the points on either side divided by 2. Looking at higher moments and differences, if we need more accuracy we could use

$$\frac{1}{16} \{ -1 \quad 9 \uparrow 9 \quad -1 \}$$

or

$$\frac{1}{256} \{ 3 \quad -25 \quad 150 \uparrow 150 \quad -25 \quad 3 \}$$

These sequences have unit sum, and the 1st, 2nd, and 3rd order moments are zero, respectively, as well as lower moments (other than 0, of course).

Sinc interpolation

For band-limited functions, sinc interpolation is very common. It is defined according to:

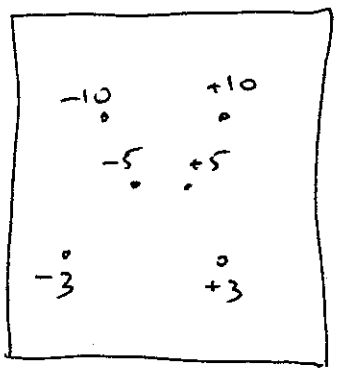
$$f(x,y) = \sum_i \sum_j s(i,j) \ \text{sinc}(x-i, y-j)$$

Here we can choose the range of i, j in the sum depending on our requirements for accuracy. For approximate work, perhaps only four points are needed in each dimension, for greater accuracy more points may be taken. Since sinc() falls off

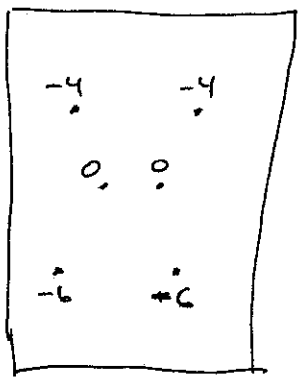
as $1/x$, the power associated with each coefficient falls as $1/x^2$ so increasing the number of coefficients rapidly increases accuracy.

Very sparse data and morphing

We have seen that selecting tie points in the morphing problem leads to an interpolation problem. Recall the figure of x and y displacements for a few known vectors in an image morph:



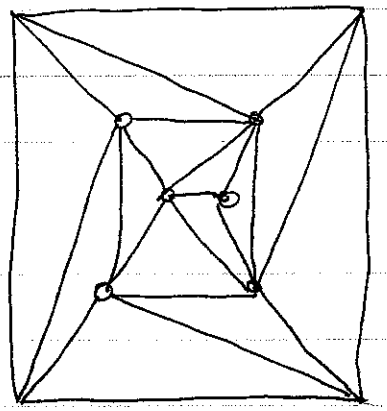
x displacements



Y displacements

How might we determine the proper x and y shifts for other spots in this image?

One approach, which works well in most cases, is to divide the matrix up into triangles with vertices defined at the tie points. Because we want all points in the matrix to be part of a triangle, we also will define new tie points at the corners of the matrix. Applying this to the x displacement matrix, we obtain, say



Clearly other sets of triangles will work, also. But now we have a set of triangles with known values at the vertices, and we can interpolate between these to set the intermediate values. Note that we had to assign values for the shift at the image corners, but we can assume zero, or the value of the closest point, or any other reasonable choice.

In our case matlab has a built-in function to do this interpolation, called griddata. It looks like

```
out = griddata(x, y, z, xi, yi);
```

where x, y, and z are vectors of the location (x,y) and value z of the known points, and xi and yi are matrices containing the x and y values for which we want to obtain the interpolated values of z. The results are placed in 'out', an array of the same dimensions as xi and yi.

xi and yi may be readily obtained using the matlab command meshgrid.