CS322 Lecture Notes: Interpolation

Steve Marschner
Cornell University

12 February 2007

A problem that arises all the time is that you have the values of some function at a set of points, but you need to know values everywhere. The points could be:

- from some measurement. For instance, an air traffic radar gives you position of an aircraft at discrete times, but you may well be interested in the location of an aircraft at the time of some event that happened between times.
- from some expensive computation. For instance, maybe you are running two simulations, one computing fluid flow around an aircraft and one simulating the control surface hydraulics. Each may proceed in its independent sequence of time steps, but they need to communicate with one another at common instants in time.
- from some table you built of values for a function that's easy enough to evaluate but takes longer than you'd like. For a hypothetical example, maybe you are computing CMY values for a laser printer, which were computed by inverting the measured CMY to RGB map. You have a procedure for computing the values but it's too slow to use for millions of pixels in an image, so you build a table and interpolate instead.

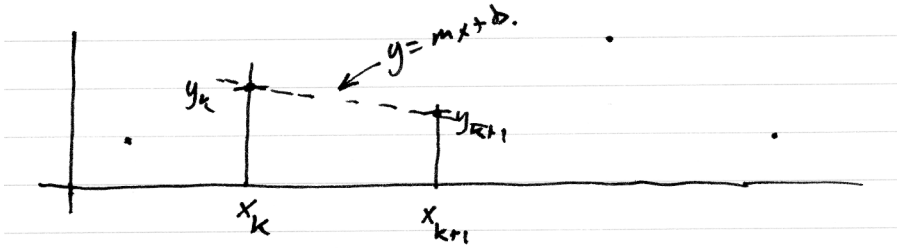
The basic approach to interpolation is to fit some function to some values (maybe all of them; maybe only some values near where you are evaluating), then evaluate that function at the desired point(s). Polynomials (or sometimes rational functions, which I won't talk about today) are the near-universal choice. What order you pick, and which points you fit it to, and how, determines the method.

Another important distinction is the dimension of the problem. We'll look at a lot of 1D examples, and interpolation in 1D (particularly if the one dimension is time) is common. But for spatial and other multidimensional problems (e.g. color), we often need to interpolate in higher dimensiont, which gets more interesting (and more expensive) for regular grids and distinctly more difficult for non-regular data.

1 linear interpolation: 1D

The simplest place to start is with first order. (The order of an interpolation scheme, by one definition, is one less than the number of points involved in defining an output value.)

Given points (x_i, y_i) for $i = 1 \ldots n$, with the x_i increasing, we can construct the *linear interpolant*, which is a piecewise linear function that has one piece between each pair of successive input points.



We could just find the m and b from the two xs and ys, but it's simpler just to write:

$$y(x) = m(x - x_k) + y_k$$

which amounts to shifting the x origin to x_k. Then it's clear that if $y(x_{k+1})$ is supposed to be y_{k+1} we want

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \quad \text{or} \quad m = \frac{\Delta y}{\Delta x}.$$

That is, the value is a weighted average between the points on either side, with a weight that favors the closer one.

So there is an algorithm: Store (x_i, y_i) in a table, and when a value for x walks in the door, use binary search to find k, then interpolate between $y - k$ and y_{k+1}, using $\alpha = (x - x_k)/(x_{k+1} - x_k)$ and $(1 - \alpha) = (x_{k+1} - x)/(x_{k+1} - x_k)$

2 Interpolating polynomials

We have talked about polynomial fitting: a degree n polynomial to $m > k + 1$ points. The exactly determined case $m = n + 1$ is computing the *interpolating polynomial* of a set of points.

Interpolating polynomials that fit a lot of points (and are therefore of high order) are generally not very useful, at least directly, because they tend to have too much variation. In low order they are often useful, though, often as pieces of other methods.

The matrix form of the problem asks for the coefficients of the polynomial

$$p(x) = a_n x^n + \ldots + a_1 x + a_0$$

Writing $p(x_i) = y_i$ for the $n+1$ points results in a matrix equation:

$$\begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ \vdots & & & & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ vdots \\ y_m \end{bmatrix}$$
$$\mathbf{V} \qquad\qquad \mathbf{x} \quad = \quad \mathbf{y}$$

The square matrix $\mathbf{V}$ is known as a *Vandermonde* matrix.

A different approach is due to Lagrange. Let's say we are looking for the quadratic interpolant that goes through (x_1, y_1), (x_2, y_2), and (x_3, y_3). First I can write down a polynomial of degree 2 that has the value 1 at x_1 and 0 at the other two points:

$$p_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}$$

I can construct p_2 and p_3 similarly—for p_k, leave out $(x - x_k)$ in the numerator and include $(x_k - x_i)$ for $i \neq k$ in the denominator.

These three polynomials are building blocks we can use to construct a polynomial with any desired values y_i:

$$p(x) = y_1 p_1(x) + y_2 p_2(x) + y_3 p_3(x)$$

(You may recognize the pattern $(x - x_1)/(x_2 - x_1)$ from the linear interpolation section. The way we constructed the linear interpolant is basically using the Lagrange polynomials for degree 1.)

These formulas let you evaluate the interpolant without having to actually compute the coefficients of the polynomial. This can be important for stability—you should *not* assume that you necessarily need to get your hands on the coefficients of the interpolating polynomial.

3 Linear interpolation: 2D

Now suppose you have some values of a function $z = f(x, y)$.

- For example: a digital elevation map: you want to generate nice smooth contours for hiking maps based on grids of elevation as a function of latitude and longitude that can be downloaded from USGS. This is going to require computing values at points that are between the grid locations where you are given elevation.

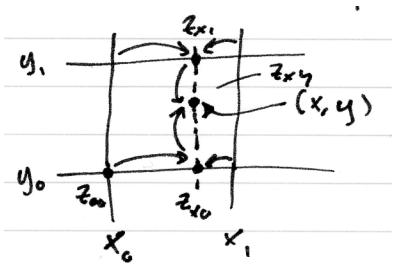
- For example: Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) medical data. These scanners produce maps of density (or other physical properties) as a function of 3D position, generally for distinct slices through a body. We may need to shift and rotate one slice relative to the next, due to motion of the patient between scans. Without

interpolation we are limited to shifting by an integer multiple of the grid spacing, but with the ability to evaluate between the points, we can shift it exactly as much as is needed.

There are two distinct generalizations of linear interpolation to 2D: bliniear and what I'll call "truly linear."

3.1 Bilinear interpolation

Bilinear interpolation is by far the more common. The idea is to interpolate along one dimension using values that were themselves interpolated along the other dimension.



(Here I have renamed the points x_0, x_1 rather than x_k, x_{k+1} to keep the indices from getting out of hand.)

If I had values at (x, y_0) and (x, y_1) then I could linearly interpolate along the vertical line. No problem—just generate them by interpolating along the horizontals!

$$\begin{aligned} z_{x0} &= (1-\alpha)z_{00} + \alpha z_{10} & \alpha &= (x - x_0)/(x_1 - x_0) \\ z_{x1} &= (1-\alpha)z_{01} + \alpha z_{11} & & \\ z_{xy} &= (1-\beta)z_{x0} + \beta z_{x1} & \beta &= (y - y_0)/(y_1 - y_0) \end{aligned}$$

Note that it does not matter whether I interpolate across and then down or down and then across (i.e. on x first or y first). Either way I end up with

$$z_{xy} = (1-\alpha)(1-\beta)z_{00} + (1-\alpha)\beta z_{01} + \alpha(1-\beta)z_{10} + \alpha\beta z_{11}$$

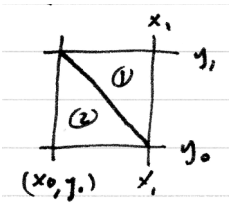
This is *bilinear interpolation*. It results in a piecewise function that is *not* piecewise linear—of course it can't be, because it matches the data at four different points, and three points uniquely determine the linear function. It has a piece for each cell in the grid of data points, but the interpolation defined over that rectangle is not linear. Look at this most recent equation, remembering that α is a linear function of x and β is a linear function of y. The full expression for z_{xy} is going to contain a constant term, an x term, a y term, and an xy term. Because of the presence of this last term is is not linear.

This kind of function is called *bilinear* because it is linear as a function of x when y is held fixed and also linear as a function of y when x is held fixed.

3.2 Truly linear interpolation

We can't have a piecewise linear interpolant where the pieces are the cells of our grid. This would require a linear function in 2D that matches with prescribed values at 4 points. We saw in HW2 that this is already a fully determined problem for 3 points.

However, if we cut the squares along their diagonals and define a linear interpolant for each half, we can make it work:



- (1) has a linear function defined by $(x_1, y_0) \mapsto z_{10}$, $(x_1, y_1) \mapsto z_{11}$, and $(x_0, y_1) \mapsto z_{01}$.
- (2) has a linear function defined by $(x_0, y_0) \mapsto z_{00}$, $(x_1, y_0) \mapsto z_{10}$, and $(x_0, y_1) \mapsto z_{01}$.

We saw in HW2 how to construct this linear (affine, really) interpolant. It is a 3-vector (coefficients for x, y, and 1).

Of course this will work for any set of triangles, not only for triangles that are made by cutting the squares of a grid in half.

When your data naturally comes with a triangulation, this approach is generally best. When the grid is naturally rectilinear, bilinear is probably most convenient. When there is no organization at all, you can build a triangulation (hint: see the `delaunay` function in Matlab) or use the method from the next section.

4 Local least squares

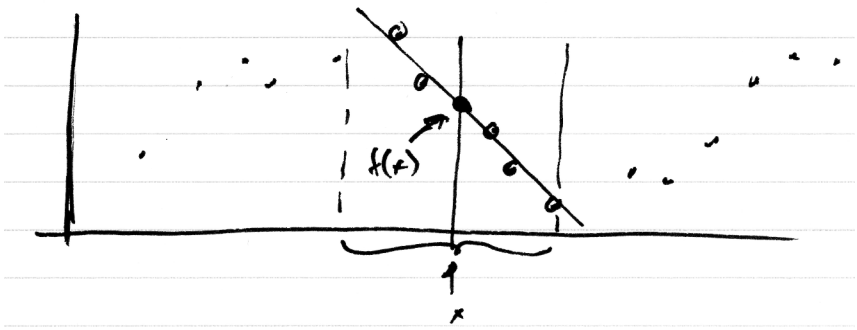
Linear interpolation extends to more than one dimension only when a grid or a triangulation is available. But what if we are handed data points with no particular placement? This problem is known as *scattered data interpolation*.

If they would all fit a single polynomial, this would be no problem. But we know this is a bad idea.

If the came with a triangulation we could use that and do truly linear interpolation. But we may prefer not to build one.

If we tolerate inexactness at the data points (maybe they are approximate anyway), we could fit a low-order polynomial by least squares. But this may not be expressive enough for the whole function.

Idea: fit a low-order polynomial *locally* to just a few data points near where we are trying to get a value.



If we just do this literally, the function will have discontinuities as points drop into and out of the interval around x. But we can use a weighting function $w(x - x_i)$ to reduce the weight of points gradually to zero.

(Read more about local least squares in the notes associated with Project 1.)

Sources

- Press, Teukolsky, Vetterling, and Flannery, *Numerical Recipes in C*, Second edition. Cambridge Univ. Press, 1992. Chapter 3.
- Our textbook: Moler, *Numerical Computing in Matlab*. Chapter 3.