CS 542G: Interpolation, Radial Basis Functions

Robert Bridson

September 10, 2008

1 Interpolation

Given a set of data points that sample an unknown function, the interpolation problem is estimating the function at other points. For example, the water depth in a channel can be measured at a set of discrete points, but to determine the shape of the channel requires interpolation between those points. When the points have no special structure, this is often referred to as the "scattered data interpolation" problem, which comes up in many, many applications; we will use it as a core problem for the first part of this course both because it's interesting and important in its own right, and because it motivates a lot of interesting and even more important numerical algorithms that get used everywhere.

We can set this problem up more formally by saying we are given the locations of sample points $(x_1, x_2, \ldots, x_n)$ along with the function values at those points $(f_1, f_2, \ldots, f_n)$; we then want to determine a function $f(x)$ which satisfies $f(x_i) = f_i$ for all i (the interpolation conition) and is somehow "plausible".

Obviously there are infinitely many functions that satisfy the interpolation condition, such as the function

$$f(x) = \begin{cases} f_i & : \quad x = x_i \text{ for any } i \\ 0 & : \text{otherwise} \end{cases}$$

This one, however, is probably not plausible for any application. Plausibility is ill-defined, and may ultimately depend significantly on the application—if we have any prior knowledge about typical functions for the problem at hand—but if nothing else is known a good first attempt is to define plausibility in terms of continuity or smoothness, since much of the real world is smooth at a macroscopic level.

However, this is still a bit abstract. One of the classic things to do in applied math and scientific computing is start with a model problem. In this case, we'll begin with interpolation in one dimension, which is significantly simpler to deal with, and hope to draw some wisdom from that experience to tackle the more general problem.

1.1 Interpolation in 1D

We'll try it out with all the x_i being in one dimension, i.e. being just real numbers. In this case, we can assume without loss of generality that they are sorted in fact, with $x_1 < x_2 < \ldots < x_n$. The interpolation problem is, in some sense, plotting the points (x_i, f_i) on a graph, and then drawing a nice curve through them all.

We'll first tackle this with a different conceptual approach, though, what I might call the "operational mindset", phrasing the problem so that the answer is an algorithm, a sequence of operations you would do:

Given the data and an arbitrary point x, figure out a good estimate for $f(x)$

How should we do this?

1.1.1 Nearest-Neighbour Interpolation

The simplest sensible answer is something called **nearest-neighbour interpolation**: find the closest sample point to x, say x_i, and use the known value there, f_i. Implicit behind this is the idea that data points which are closer are more relevant, and so the nearest data point has the best estimate.

Of course, if you plot what this algorithm does for all x, you immediately realize the function it produces is piecewise constant, with possible jump discontinuities at the midpoints between samples. It's easy and has a degree of common sense (and works for the original problem in any dimension!), but obviously for many applications this won't be good enough.

However, the assumption (that nearby data points are the most relevant) bears examination. Why is this the case? This boils down to results in calculus that for smooth functions (which we are assuming) values near to a point give you good information about what the function does. The most important result is the Taylor series, which says for a function with enough derivatives at a point x,

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 + \cdots + \frac{1}{k!}f^{(k)}(x)\Delta x^k + O(\Delta x^{k+1})$$

The error term at the end, expressed in $O()$ notation, is sometimes expressed more exactly with the $k+1$'st derivative if it exists, and as long as that isn't too large and Δx is small enough, should be very close to zero. (Raising a small number to a high power makes it nearly zero.) Nearest-neighbour interpolation is based on the zero'th order Taylor series: $f(x + \Delta x) \approx f(x)$ when Δx isn't too big, and the error probably increases (in an $O()$ notation sense) with Δx, so sample points further away are probably worse estimates.

1.1.2 Polynomial Interpolation

We can't directly generalize this to more terms in the Taylor series, since the data we're given just has function values, not derivatives. However, it gives us a guide that polynomials (such as the Taylor polynomial) are useful for approximating smooth functions. There are more sophisticated theorems that can make this precise (how closely polynomials can approximate a function), but that will come later.

For now, let's see if we can use any old polynomial for interpolation. In particular, if we write down a degree $n-1$ polynomial,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots a_{n-1} x^{n-1},$$

we see it has n coefficients, a_0 up to a_{n-1}. If we require this polynomial to interpolate the n data points, $p(x_1) = f_1, \dots, p(x_n) = f_n$, then we get n equations for those n unknown coefficients—sounds good! We won't go through it, but you can in fact prove there is a unique solution to this problem (in particular, it turns out this is a full-rank linear system of equations; if you search the web for "Lagrange Interpolation" you can actually find an explicit formula for the interpolating polynomial, expressed slightly differently).

Is this any good? It's certainly smooth, in one sense, since polynomials are infinitely differentiable (C^∞), and in fact all derivatives after the $n-1$'st are zero. If you try this out on a few points (say $n \leq 4$) you will see it work pretty well. However, when you get to a lot of points, the approach can fail dramatically.

Heuristically what's going on is that the derivative of the polynomial is an $n-2$ degree polynomial, which in general may have $n-2$ roots; these roots are the locations of local minima and maxima of the original polynomial, where the slope changes sign. Thus a typical degree $n-1$ polynomial is pretty wiggly. It also has terms like x^{n-1}, which for n large can explode if x is bigger than 1 in magnitude. Very often, for n bigger than 5 say, the interpolating polynomial looks nothing like the data, with gigantic oscillations between sample points.

1.1.3 Piecewise Polynomials

The usual solution to this problem is to restrict polynomials to only be low degree, say three or less. Of course, a degree three polynomial can't generally interpolate more than four points, so we use **piecewise polynomials**. That is, on each interval we build a different polynomial of low degree, and then stitch them together into a single function. In some contexts like graphics this could be called a **spline**. The nearest neighbour interpolant is an example of this, using degree zero polynomials (constants); we can get something at least continuous by using higher degrees.

Linear interpolation is what you get with degree one polynomials, i.e. straight lines connecting up

the data points. In each interval $[x_i, x_{i+1}]$ we can define a line which interpolates the endpoints, i.e. goes through the points (x_i, f_i) and (x_{i+1}, f_{i+1}). Because the line to the left of any given data point x_i matches the value of the line to the right (both are f_i) the result is continuous, though not in general differentiable at the sample points. Linear interpolation has some nice properties that we'll explore more next lecture: for example, between data points the function is as straight as possible (you can't get straighter than a line) so we don't have to worry about the wiggles which afflict full polynomial interpolation.

Using higher degree polynomials like quadratics or cubics in each interval gives us more freedom. For example, it's possible with piecewise cubics to make the interpolating function differentiable (even twice differentiable) at the sample points, which visually makes it very nice and smooth; this is very important in some geometric contexts like computer graphics, but actually often is not very important generally in scientific computing. More commonly, we can use that extra freedom to more closely model the values of a smooth function, reducing the error similar to how more terms in a Taylor series can help. Obviously, though, we don't want to take this too far, or we get back to the problems of full polynomial interpolation.[1]

2 Going Back to Higher Dimensions

Now that we have a reasonable handle on interpolation in 1D, our model problem, let's see if it helps out in higher dimensions.

2.1 Tensor-Product Interpolation

The first special case is when the data points are arranged (at least logically) on a regular grid. We can then use so-called **tensor-product interpolation**, which largely amounts to interpolating the data one dimension at a time. For simplicity of notation let's assume the grid is lined up with the axes, and just do a 2D example. First, we can use 1D interpolation to fill in any values on the grid lines aligned with the x-axis. Second, at any point in the grid, we can draw a line parallel to y-axis and pick up the interpolated values on every grid line, which forms another 1D interpolation problem (now along y). You might write the first step of this procedure formally as:

$$f(x, y_j) = \text{interp}(x, (x_1, f_{1j}), \ldots (x_m, f_{mj}))$$

[1] If you're interested, you might want to find out about Essentially Non-Oscillatory (ENO) interpolation, where in each interval we construct several different polynomials which fit the endpoints and different sets of neighbouring points, and then pick the least wiggly of the bunch.

in other words we can interpolate at any x and at the y values that show up in the grid, by doing 1D interpolation with respect to x and the grid data points with that y value. The second step does the same in y, using interpolated values:

$$f(x, y) = \text{interp}(y, (y_1, f(x, y_1))_{1j}), \ldots (y_n, f(x, y_n)))$$

Extending this to any number of dimensions is relatively simple.

However, this fails if the data isn't lined up nicely on a grid. For high dimensions, it also can be pretty expensive: a k-dimensional regular grid with n data points along each side needs n^k storage. This exponential dependence on the dimension of the problem (k) is sometimes referred to as "the curse of dimensionality".

2.2 Polynomial Interpolation

Even though it doesn't work so well in 1D, we also can take a peek at full polynomial interpolation. The curse of dimensionality is at work here too. In particular, the notion of a degree p polynomial in more than one dimension is a bit different: there are a lot more than $p + 1$ coefficients. Just as an example, a quadratic polynomial in 2D has coefficients for all of the following terms: 1, x, y, x^2, xy, and y^2. That's six coefficients (compared to three for 1D). A cubic has ten coefficients in 2D, and it keeps increasing quadratically with degree. In fact, a degree p polynomial in k dimensions has $O(p^k)$ coefficients. At a minimum, we'll probably hit the problem that the number of data points we're given doesn't match up to the number of coefficients of any polynomial.

2.3 Triangulation

One of the reasons interpolation with piecewise polynomials works so nicely in 1D is that the data points naturally split the space up into intervals, and we can define a separate polynomial on each interval. The tensor-product interpolation approach in higher dimensions is one natural generalization of this, and in fact that's where it gets its name—the grid elements are tensor products of intervals in the set theory sense.

Another, more flexible way of generalizing intervals to higher dimensions is **triangulation**. Here we partition the space into triangles (2D), tetrahedra (3D), or more general simplices (4D+), with the sample points furnishing the vertices of these elements. This can always be done, no matter how irregularly scattered the data points are, and we will look at how to do this well later on in the course. Once we have partitioned space this way, we can define a low degree piecewise polynomial in each triangle

(or tetrahedron, or simplex) that interpolates the data—in particular, linear interpolation makes perfect sense.

However, the curse of dimensionality strikes here too. Triangulation in 2D is relatively easy, triangulation in 3D is doable but remains a source of tough research problems, and triangulation in higher dimensions looks really difficult. The combinatorial complexity of the graphs also grows potentially exponentially with dimension: in 2D the number of triangles is linear in the number of vertices, but in 3D it can be quadratic...

So let's go back to the model problem, and try to think about it in a different way that's easier to generalize to higher dimensions.

3 A Function Space Approach to 1D Interpoation

We'll bring a different mindset to 1D interpolation, what might be called a "function space" approach, or more generally "non-constructive": instead of phrasing the problem operationally, looking for an algorithm to provide a definite answer, we'll just try to define the interpolant (the function that interpolates the data) as something satisfying a number of criteria and worry about actually solving for it later. This is how we started, in fact, saying our interpolating function should pass through the data and be as plausible, or as smooth, as possible. Let's make that precise!

The first step on this path is to set this up in terms of an n-dimensional function space, one we've already discussed: for example, the space of degree $n-1$ polynomials, or the space of piecewise linear functions (linear on each interval). We can then define the interpolant to be the function from this space that interpolates the data, i.e. satisfies $f(x_i) = f_i$ for $i = 1, \ldots, n$. If we introduce a basis for the space (i.e. a set of n functions from the space that are linearly independent) we can express any function as a linear combination of the basis functions, and solve for the coefficients in that linear combination which make the interpolation condition hold true. However, this presupposes we already know the right finite dimensional function space, which is what we're having a hard time with in higher dimensions.

Instead, we'll start with an infinite dimensional function space, such as the space of continuous functions with a well-behaved weak derivative. (This can be set up to have a precise technical definition, which isn't too important right now; think continuous functions who are differentiable except at a few points, such as the piecewise linear functions we've already dealt with.) We'll then look for function in this space which interpolates the data and is smooth as possible, in a sense that we need to precisely define.

What we don't want is a function that's excessively wiggly. Being wiggly has something to do with the derivative of the function being unnecessarily large—so we'll try defining "as smooth as possible" to mean "has a small a derivative as possible". Of course, the derivative of the function isn't a single value, but function of x, so for this to be sensible we need to boil it down to a single number somehow. Perhaps the obvious choice is to use the largest value of $|f'(x)|$ over all x, but it will be considerably more convenient mathematically to use a value that takes the derivative everywhere into account:

$$\int_{-\infty}^{\infty} \big(f'(x)\big)^2 \, dx$$

This is known as the L^2 norm of the derivative. For some functions, it's infinite or undefined; we'll rule those out from the problem.

We now have a minimization problem:

> Find f, a weakly differentiable function, that interpolates the data $f(x_i) = f_i$ and minimizes, over all such functions, the L^2 norm of its derivative.

The classic calculus approach to problems like this (minimize a quantity) would be take the derivative and set it zero, but that isn't so easy here since we're minimizing with respect to an entire function, not a simple value. How do you differentiate with respect to a function?

3.1 The Calculus of Variations

There is a whole topic devoted to questions like this, the calculus of variations. We'll take a classic trick from this subject to walk through solving the problem at hand.

First suppose that $f(x)$ is the solution. Introduce an arbitrary weakly differentiable function $g(x)$ that satisfies $g(x_i) = 0$ for each i, i.e. that is zero at the data points but could be nonzero in between. Then the function $f + g$ also satisfies the interpolation condition, but it can't have a smaller value for the L^2 norm of its derivative $f' + g'$ than f does. In fact, for any real number ϵ the function $f + \epsilon g$ interpolates and can't have a smaller L^2 norm of its derivative, $f' + \epsilon g'$.

Now build a regular function $h(\epsilon)$ that maps real numbers to real numbers, the L^2 norm of the derivative of $f + \epsilon g$:

$$h(\epsilon) = \int_{-\infty}^{\infty} \big(f'(x) + \epsilon g'(x)\big)^2 \, dx$$

(If you look closely at this, you should be able to see that h is actually just a quadratic in ϵ, with fixed coefficients involving integrals.) From our assumption, this function h has to have a minimum value at

$\epsilon = 0$. Since it's a regular function of one variable, we can do the usual calculus trick here and say the derivative at $\epsilon = 0$ is zero:

$$h'(0) = 0$$

Expanding $h'(\epsilon)$ out gives:

$$h'(\epsilon) = \frac{\partial}{\partial \epsilon} \int_{-\infty}^{\infty} \left(f'(x) + \epsilon g'(x)\right)^2 dx$$

Since the integral is over x, not ϵ, we can switch the order of integration and differentiation:

$$\begin{aligned} h'(\epsilon) &= \int_{-\infty}^{\infty} \frac{\partial}{\partial \epsilon} \left(f'(x) + \epsilon g'(x)\right)^2 dx \\ &= \int_{-\infty}^{\infty} 2\left(f'(x) + \epsilon g'(x)\right) g'(x) dx \end{aligned}$$

Evaluating at $\epsilon = 0$ and setting this to zero gives

$$\int_{-\infty}^{\infty} 2f'(x)g'(x)dx = 0$$

Now, the powerful thing about this equation is that it should be true for any g: this somehow is telling us a lot about f. It's not clear what it's telling us yet, though, since we don't know much about $g'(x)$: to get rid of the $g'(x)$ term we use **integration by parts**:

$$\begin{aligned} 0 &= \int_{-\infty}^{\infty} 2f'(x)g'(x)dx \\ &= -\int_{-\infty}^{\infty} 2f''(x)g(x)dx + \left[2f'(x)g(x)\right]_{-\infty}^{\infty} \end{aligned}$$

Note that for the integral of $(f'(x))^2$ to be finite, we must have $f'(x) = 0$ at infinity, so the last term actually vanishes.

We're left with the equation

$$-\int_{-\infty}^{\infty} 2f''(x)g(x)dx = 0$$

which should be true for any $g(x)$ that satisfies $g(x_i) = 0, i = 1, \ldots, n$. From there, we can argue that, except for the sample points $x = x_i$ we must have that $f''(x) = 0$.

This is a very simple differential equation, stating that the second derivative of $f(x)$ is zero between samples, and it has an equally simple solution: $f(x)$ must be linear in each interval. We already insisted that we were only looking at continuous functions, so we have in fact derived the piecewise linear spline we knew about before. Crucially, however, we derived it from a pretty general statement that didn't have anything to do with intervals or polynomials, and that's going to make it easy to generalize to higher dimensions.