Polynomial approximation and Splines

1. Weierstrass approximation theorem

The basic question we’ll look at today is how to approximate a complicated function \( f(x) \) with a simpler function \( P(x) \)

\[
f(x) \approx P(x)
\]

for example, where \( P(x) \) is a polynomial of degree \( d \). There are a variety of approaches. We have the following theorem:

**Weierstrass Approximation Theorem** *For every continuous function \( f(x) \) on \([a, b]\) and every \( \epsilon > 0 \) there is a polynomial \( P(x) \) such that \( |f(x) - P(x)| < \epsilon \) for each \( x \in [a, b] \).*

This is a useful idea for computer graphics and computer aided design. Instead of storing a lot of data (function values, bitmap images), just store a few numbers (polynomial coefficients), and generate the approximate function values when you need them.

In general, a polynomial of degree \( n \) has \( n + 1 \) coefficients, so we’ll fit \( P \) to \( f \) using \( n + 1 \) equations in some way. There are various ways to do that.

2. Taylor polynomials

The idea here is to match the value of \( f \) and the values of \( n \) of its derivatives at a single point \( x_0 \). That gives us our \( n + 1 \) equations. By Taylor’s theorem,

\[
P(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k.
\]

The problem is that this can be a great approximation near \( x_0 \), but a lousy one far away. It works well for \( f \)’s like polynomials or exponential functions, but not for other functions. In Figures 1-3 I’ve plotted the \( n = 3, 5, 7 \) approximations to \( f_1(x) = \sin x \).

This method is a good thing to try if you only care about \( x \approx x_0 \). It is often not a good choice for interpolating between widely separated values \( x_0 \) and \( x_1 \). These won’t prove the theorem, even for infinitely differentiable \( f \), as the following example shows

**Example:** Let \( f_2(x) = \begin{cases} e^{-1/x^2}, & x > 0 \\ 0, & x \leq 0 \end{cases} \).

Then \( f_2 \) is infinitely differentiable, but one can show that \( \frac{f_2(x)}{x^n} \to 0 \) when \( x \to 0 \). This implies that \( f_2^{(k)}(0) = 0 \) for every \( k \). Therefore when \( x_0 = 0 \), the Taylor polynomials are all \( P(x) = 0 \).
3. LAGRANGE POLYNOMIALS

The idea here is that we get our \(n + 1\) equations by matching the values of \(f\) at \(n + 1\) points \(x_0, \ldots, x_n\) (called nodes)

\[
P(x) = \sum_{k=0}^{n} f(x_k) \prod_{i=0, i \neq k}^{n} \frac{x - x_i}{x_k - x_i}.
\]
Figure 3. Degree 7 Taylor approx. to $y = \sin x$, about $x_0 = 0$

For example, if you substitute $x = x_j$ then the $k \neq j$ terms in the sum all = 0, and the $k = j$ term gives $f(x_j)$.

Figure 4 plots the Lagrange interpolation to the above $f_2$ with nodes -1.5, -1, -.5, 0, .5, 1, 1.5, 2. We’re using 8 nodes, so the fit is with a degree $n = 7$ polynomial. It does a lot better than Taylor’s $P(x) = 0$, but it is still pretty wiggly (ie it is easy to overfit using this method).

4. Hermite polynomials

The idea here is that we get $n + 1$ equations by matching the values and first derivatives of $f$ at $\frac{n+1}{2}$ points. For example, with a cubic $P$ we match four pieces of data at two points: $f(x_0)$, $f'(x_0)$, $f(x_1)$, $f'(x_1)$. There’s a formula for this, which I’ll write down but won’t prove:

$$P(x) = f(x_0) \frac{(x-x_0)^2(x_1-x_0+2x)}{(x_1-x_0)^3} + f(x_1) \frac{(x-x_0)^2(3x_1-x_0-2x)}{(x_1-x_0)^3}$$

$$+ f'(x_0) \frac{(x-x_0)(x_1-x^2)}{(x_1-x_0)^2} - f'(x_1) \frac{(x-x_0)^2(x_1-x)}{(x_1-x_0)^2}$$

5. Splines

The above can still be pretty wiggly. To get more practical approximations, split up $[a,b]$ into $k$ pieces $[x_0,x_1], \ldots, [x_{k-1},x_k]$, and approximate on each using a different polynomial $P_j$, $j = 1, \ldots, k$. If we use cubics to do this, then we’re trying to match $4k$ parameters via $4k$ equations. There are several ways of doing this.

5.1. Hermite spline: Use Hermite interpolation to match $f(x_k)$ and $f'(x_k)$ at the beginning and end of each interval. Put another way, we impose the constraints $f(x_0) = P_1(x_0)$, $f'(x_0) = P'_1(x_0)$, $f(x_k) = P_k(x_k)$, $f'(x_k) = P'_k(x_k)$, and for $j = 1$
Figure 4. A degree 7 Lagrange interpolation for $y = f_2(x)$

to $k - 1$: $f(x_j) = P_j(x_j) = P_{j+1}(x_j)$ and $f'(x_j) = P'_j(x_j) = P'_{j+1}(x_j)$. Because we have a nice formula for Hermite interpolation, this is easy to work out.

In Figure 5 I’ve plotted this for the example $f_2$ with $k = 3$, and nodes at $x_0 = -2$, $x_1 = 0.25$, $x_2 = 1$, and $x_3 = 2$. This means we’re fitting 12 parameters this time. It doesn’t quite handle the sharp turn, but away from there it is excellent.

Figure 5. A 3-window Hermite spline for $y = f_2(x)$
5.2. **Cubic spline:** With this approach, we won’t worry about matching the \( f'(x_j) \), for \( j = 1, \ldots, k - 1 \). That saves us \( 2(k - 1) \) constraints. Instead just make the \( k - 1 \) derivatives match: \( P_j'(x_j) = P_{j+1}'(x_j) \), and then make the \( k - 1 \) second derivatives also match: \( P_j''(x_j) = P_{j+1}''(x_j) \). The only derivative information we use is at the very beginning and end, but we get smoother solutions because the 2nd derivatives no longer jump.

There isn’t a simple formula for the coefficients any more – now one needs to do some numerical linear algebra to figure them out (they solve a “tridiagonal system”), but that is easy to do numerically, and it typically gives excellent looking results.

This approach is widely used in computer graphics and computer aided design.

6. **Bezier curves**

This works fine for functions. But what about approximating curves? Now use parametric curves \( x = P(t) \) and \( y = Q(t) \). Using splines for each can certainly be made to work easily enough. But if you’re trying to use this to draw, you want something geometric that is easy to adjust. In that setting, what people do is break the curve up into \( k \) arcs, and use cubic Hermite polynomials for \( P \) and \( Q \). We need to specify values and derivatives for both \( x \) and \( y \) at each of \( k + 1 \) values of \( t \). We do that in a geometric way by giving guidepoints, i.e. \((x_j, y_j)\) and \((x_j + x'_j, y_j + y'_j)\). If you use a sophisticated computer drawing program, you’ll find these somewhere in it.

7. **Bernstein polynomials**

Let’s go back to the original problem. How does one prove the Weierstrass Approximation theorem (say on \([0, 1]\))? The slickest proof is using the Bernstein polynomials

\[
P(x) = \sum_{k=0}^{n} \binom{n}{k} f\left(\frac{k}{n}\right) x^k (1-x)^{n-k}.
\]

The idea here is that if \( N \) has a Binomial\((n, x)\) distribution, then \( f(x) = f(E[N/n]) \) and \( P(x) = E[f(N/n)] \). So they should be close, by the law of large numbers. In fact, chasing through the standard proof of the law of large numbers, one can turn this into a rigorous proof.

8. **Orthogonal polynomials**

There are other classes of polynomials used for approximation, for example the Legendre polynomials or the Chebyshev polynomials. Calculations for them are based on an orthogonality property, very much like what happens for Fourier series. So we’ll focus on those instead.