

Review I: Interpolation

Varun Shankar

January 1, 2016

1 Introduction

In this document, we review interpolation by polynomials. Unlike many reviews, we will not stop there: we will discuss how to differentiate and integrate these interpolants, allowing us to approximate derivatives and integrals of unknown functions.

2 Interpolation versus Approximation

Let $X = \{x_k\}_{k=0}^N$ be some set of points (or *nodes*) at which we are provided samples of some function $f(x)$. Thus, we are given $y_k = f(x_k)$. The goal of most approximation techniques is to construct a *proxy* for $f(x)$ using the supplied y_k values. If this proxy is “reasonable”, we can use it to give us more information about the function f , despite having been given only discrete samples of the function f . Such a proxy is in general called an *approximant*. This technique of generating proxies is called *function approximation*.

An *interpolant* is an approximant that exactly agrees with the y_k values at the x_k nodes. This is not always reasonable, especially if you are supplied with some spurious y_k values.

Let us formalize things a bit further. Let $I_X f$ be the interpolant to the function f on the set of nodes X . Then, at $X = \{x_k\}_{k=0}^N$, we require that

$$I_X f = f|_X, \tag{1}$$

$$\implies I_X f(x_k) = f(x_k) = y_k, k = 0 \dots N. \tag{2}$$

This notation is quite abstract, but therein lies its utility. This formula will make more sense when we discuss specific interpolants.

A note on indices

With some interpolants, it is common to use indices $k = 0 \dots N$. On the other hand, for implementation purposes, it is often more useful to think of the indices as $k = 1 \dots N$. We will freely switch between the two representations.

Note that certain index representations make more sense for certain programming languages. C/C++ have array indices starting from 0, while Matlab has indices starting from 1. Fortran lets you start with any index you want.

3 Polynomial Interpolation

Let us focus on the most powerful interpolation tool in 1D: polynomial interpolation. This is a bold statement; everyone has his/her own favorite interpolation technique (mine are RBFs). This issue is further confounded by some myths surrounding polynomial interpolation, which we will now attempt to dispel.

The goal is to find a polynomial p that interpolates f at the $N + 1$ points x_k . In other words, we seek an interpolant of the form

$$p(x) = \sum_{k=0}^N a_k x^k, \quad (3)$$

$$p(x_k) = y_k, \quad (4)$$

where a_k are the unknown interpolation coefficients. To find these coefficients, we simply need to enforce the interpolation conditions $p(x_k) = y_k$. This gives rise to a linear system of the form

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \dots & x_0^N \\ 1 & x_1 & x_1^2 & x_1^3 & \dots & x_1^N \\ 1 & x_2 & x_2^2 & x_2^3 & \dots & x_2^N \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 & \dots & x_N^N \end{bmatrix}}_V \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \quad (5)$$

The above matrix V is called the *Vandermonde* matrix. To solve for the unknown a_k values, we need to solve the above dense linear system. The obvious choice is some variant of Gaussian elimination (some form of LU decomposition). The cost of doing so is $O(N^3)$.

Of course, as you should already know, the above Vandermonde matrix is typically disastrously ill-conditioned, for most common choices of the nodes x_k . In fact, the condition number $\kappa(V) \sim e^N, N \rightarrow \infty$. This is problematic! At $N \approx 37$, this puts $\kappa(V) \approx 10^{16}$. Since computers can only store 16 digits of precision when using double-precision arithmetic, this means our answer will

not even be correct to one digit of precision. The only way to really compute a polynomial interpolant for large N is to somehow avoid construction of the Vandermonde matrix V altogether. However, for small N (say 2 to 6), the monomial form of the polynomial interpolant is fine.

Given a fixed node set X in 1D and some fixed set of data, there is a unique polynomial interpolant through that data. That said, it is possible to write the same interpolant in many different forms. There are two important forms: the Lagrange form, and the Newton form. We will focus on the Lagrange form and a form derived from the Lagrange form. The Lagrange form of the polynomial interpolant is given by

$$p(x) = \sum_{k=0}^N y_k \ell_k(x), \quad (6)$$

$$\ell_k(x) = \frac{\prod_{\substack{j=0, \\ j \neq k}}^N (x - x_j)}{\prod_{\substack{j=0, \\ j \neq k}}^N (x_k - x_j)}. \quad (7)$$

The *Lagrange basis* ℓ_k has the so-called Kronecker delta property

$$\ell_k(x_j) = \begin{cases} 1 & k = j \\ 0 & k \neq j \end{cases}$$

Notice that we never form a matrix in this scenario. The basis is computed from the node locations, and the “coefficients” are the function samples y_k themselves. At this point, most texts remark that the Lagrange form

1. is numerically unstable
2. is expensive to evaluate ($O(N^2)$ per evaluation).
3. has to be recomputed when new nodes and samples are added.
4. is worse than the Newton form.

These are all true statements! However, it would be premature to abandon the Lagrange basis at this point. We will now improve the Lagrange interpolation formula to ameliorate the above difficulties.

3.1 Barycentric Lagrange Interpolation

To define the improved Lagrange interpolation formula, first define

$$\ell(x) = (x - x_0)(x - x_1) \dots (x - x_N), \quad (8)$$

and the *barycentric weights* by

$$w_k = \frac{1}{\prod_{j \neq k} (x_k - x_j)}, k = 0, \dots, N. \quad (9)$$

Then, $\ell_k(x) = \ell(x) \frac{w_k}{x - x_k}$, and the interpolant becomes

$$p(x) = \ell(x) \sum_{k=0}^N \frac{w_k}{x - x_k} y_k. \quad (10)$$

This is the first form of the barycentric formula. It is easy to show that adding a new node x_{N+1} now costs only $O(N)$ flops. This formula can be made even more stable, yielding

$$p(x) = \frac{\sum_{k=0}^N \frac{w_k}{x - x_k} y_k}{\sum_{k=0}^N \frac{w_k}{x - x_k}}, \quad (11)$$

which is the second form of the barycentric formula. This form is used in practice in most tools for polynomial interpolation.

3.2 Weights for specific point distributions

If the nodes are equispaced on any interval, the weights can be easily computed as $w_k = (-1)^k \binom{N}{k}$. However, recall that polynomial interpolation on equispaced nodes is highly unstable, susceptible to the *Runge phenomenon*; this manifests as exponentially growing weights w_k . This, to many, was the final nail in the coffin for polynomial interpolation. However, that issue too is surmountable if one is allowed to change the node set X .

If X is chosen to be the set of *Chebyshev zeros*, the weights are given by $w_k = (-1)^k \sin\left(\frac{2k+1}{2n+2}\pi\right)$. This set of points does not include the endpoints of the interval.

If X is chosen to be the set of *Chebyshev extrema*, the weights are given by $w_k = (-1)^k \delta_k$, where $\delta_k = \frac{1}{2}$ if $k = 0$ or $k = N$, and $\delta_k = 1$, otherwise. These points include the endpoints of the interval.

Use barycentric interpolation with any Chebyshev nodes¹, and suddenly polynomial interpolation is stable for thousands of points. Contrast that with the degree 37 polynomial interpolant allowed by Vandermonde matrices!

¹Confusingly, when people say “Chebyshev nodes”, they can alternatively mean either zeros or extrema.

4 Differentiating and integrating polynomial interpolants

Since the monomial form is stable for small values of N , and the barycentric form for all values of N , we will use a simple rule of thumb: use monomials when it is safe to do so, barycentric form when necessary. However, regardless of how we compute the interpolant, recall that there is a *unique* polynomial interpolant to the data.

For convenience, we will work with the monomial or Lagrange forms, rather than the barycentric form. Consider again the polynomial interpolant to the data $y_k, k = 0 \dots N$, given by

$$p(x) = \sum_{k=0}^N a_k x^k. \quad (12)$$

Now, assuming the data y_k is sampled from some function $f(x)$, it is natural to attempt an approximation to the derivative of f using the derivative of p . Say we want $\frac{\partial f}{\partial x}$. This can be approximated by

$$\frac{\partial f}{\partial x} \approx \frac{\partial p}{\partial x} = \sum_{k=0}^N a_k \frac{\partial}{\partial x} x^k, \quad (13)$$

$$= \sum_{k=0}^N a_k k x^{k-1}. \quad (14)$$

It is also possible (though a little more painful) to differentiate the Lagrange basis. Why would one do this instead of the approach above? Recall: when we use the Lagrange form of the interpolating polynomial, there is no linear system to solve, and the “interpolation coefficients” are the function samples y_k . Thus, we have

$$\frac{\partial f}{\partial x} \approx \frac{\partial p}{\partial x} = \sum_{k=0}^N y_k \frac{\partial}{\partial x} \ell_k(x). \quad (15)$$

We will revisit the idea of differentiating the Lagrange form shortly.

In general, the above procedure applies for any general linear operator L in place of the first derivative. For example, $L = \frac{\partial^2}{\partial x^2} + 7$. Note that integrals are also linear operators. Consider approximating the definite integral of a function over the interval $[x_i, x_{i+1}]$. We have

$$\int_{x_i}^{x_{i+1}} f dx \approx \int_{x_i}^{x_{i+1}} p dx = \sum_{k=0}^N a_k \int_{x_i}^{x_{i+1}} x^k dx, \quad (16)$$

$$= \sum_{k=0}^N a_k \left[\frac{x_{i+1}^{k+1}}{k+1} - \frac{x_i^{k+1}}{k+1} \right]. \quad (17)$$

Alternatively, one could use the Lagrange form again, giving us

$$\int_{x_i}^{x_{i+1}} f dx \approx \int_{x_i}^{x_{i+1}} p dx = \sum_{k=0}^N y_k \int_{x_i}^{x_{i+1}} l_k(x) dx, \quad (18)$$

with this approach doing away with the need for computing interpolation coefficients.

Step back and think for a moment on the power of this approach. The insights here will be used over and over again for the rest of this course.

Error Analysis

Let f be a function in $C^{N+1}[a, b]$, and p be the polynomial interpolant to that function. Then, for each $x \in [a, b]$, there exists a point $\xi_x \in [a, b]$ such that

$$f(x) - p(x) = \frac{f^{(N+1)}(\xi_x)}{(N+1)!} \prod_{k=0}^N (x - x_k). \quad (19)$$

This formula applies for points *not* in X , the set of interpolation nodes. At the interpolation nodes, the error is exactly zero (which is the definition of interpolation).

At Chebyshev zeros, we can show that the product term is minimized. We can now get a sharper error bound (indeed, the best possible)

$$|f(x) - p(x)| \leq \frac{1}{2^N (N+1)!} \max_{-1 \leq t \leq 1} |f^{(N+1)}(t)|. \quad (20)$$

We will use both these error estimates over the course of this semester.

5 Looking ahead

Next, we will revisit finite differences and quadrature from the perspective of polynomial interpolation. We will compare the Taylor series approach of generating finite differences to the polynomial approach, and decide which approach is more general/powerful. We will skip that particular discussion for quadrature, and jump straight to the generation of quadrature from polynomial interpolation.