Polynomial Interpolation

Varun Shankar

January 16, 2019

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 and 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

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

$$p(x_k) = y_k, \tag{4}$$

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

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

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. We will discuss some choices for doing so later in the semester, but one typically uses Gaussian Elimination or one of its variants. The cost of doing so is $O(N^3)$.

Once we computed the polynomial, we should be able to evaluate it wherever we want (within the range of X). Let $X^e = \{x_j\}_{j=0}^{M}$ be the set of evaluation points. Then, we can evaluate the polynomial to obtain approximations to $f|_{X^e}$ as follows:

$$f|_{X^e} \approx p|_{X^e} = \sum_{k=0}^{N} a_k (x_j^e)^k, j = 0, \dots, M. \tag{6}$$

This can be concisely written as a matrix-vector product that produces an $M+1$ vector:

$$f|_{X^e} = \begin{bmatrix} y_0^e \\ y_1^e \\ y_2^e \\ \vdots \\ y_M^e \end{bmatrix} \approx \begin{bmatrix} 1 & x_0^e & (x_0^e)^2 & \dots & (x_0^e)^N \\ 1 & x_1^e & (x_1^e)^2 & \dots & (x_1^e)^N \\ 1 & x_2^e & (x_2^e)^2 & \dots & (x_2^e)^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_M^e & (x_M^e)^2 & \dots & (x_M^e)^N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix} \tag{7}$$

The matrix here is simply referred to as an evaluation matrix.

Conditioning of a matrix

When solving linear systems on paper, we only worry about the matrix being singular (zero determinant), since this means an inverse does not exist. On a computer, where each real number is only stored to a finite number of digits of precision, we have to worry about the matrix *appearing singular in finite-precision arithmetic*. One way to quantify how close a matrix is to singular is the **condition number**, defined for a matrix A as

$$\kappa(A) = \|A^{-1}\|\|A\|, \tag{8}$$

where $\|A\|$ is some measure of the "magnitude" of a matrix. The ideal condition number is 1. In finite precision arithmetic, since we can only store 16 digits of precision, the worst condition number is 10^{16}. We'll discuss this in greater detail later.

The above Vandermonde matrix is typically disastrously ill-conditioned, for most common choices of the nodes x_k. In fact, the condition number $\kappa(V) \sim e^N, N \to \infty$. 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

$$p(x) = \sum_{k=0}^{N} y_k \ell_k(x), \tag{9}$$

$$\ell_k(x) = \frac{\prod\limits_{\substack{j=0,\\ j\neq k}}^{N} (x - x_j)}{\prod\limits_{\substack{j=0,\\ j\neq k}}^{N} (x_k - x_j)}. \tag{10}$$

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

$$\ell(x) = (x - x_0)(x - x_1)\dots(x - x_N), \tag{11}$$

and the *barycentric weights* by

$$w_k = \frac{1}{\prod\limits_{j\neq k} (x_k - x_j)}, k = 0, \dots, N. \tag{12}$$

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

$$p(x) = \ell(x) \sum_{k=0}^{N} \frac{w_k}{x - x_k} y_k. \tag{13}$$

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

$$p(x) = \frac{\sum_{k=0}^{N} \frac{w_k}{x-x_k} y_k}{\sum_{k=0}^{N} \frac{w_k}{x-x_k}}, \tag{14}$$

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

3.2 The Weierstrass Approximation Theorem

Suppose f is a continuous real-valued function defined on the real interval $[a, b]$. Then, there exists a polynomial p such that for all $x \in [a, b]$, $|f(x) - p(x)| < \epsilon$, for every $\epsilon > 0$.

This theorem tells us that in 1D, for any function defined on any real interval, there is a polynomial interpolant to that function. The ϵ stuff tells us that we can make the approximation error arbitrarily small using polynomials, typically accomplished by increasing the number of points. But there's a catch, and it's a big one, as we'll see soon.

3.3 Error analysis

Before proceeding, let's discuss the error associated with polynomial interpolation. Let f be a function in $C^{N+1}[a, b]$; this means the function and its $N + 1$ derivatives are continuous. Let p_N be the polynomial of degree at most N interpolating that function. Then, for each $x \in [a, b]$, there exists a point $\xi_x \in [a, b]$ such that

$$f(x) - p_N(x) = \frac{f^{(N+1)}(\xi_x)}{(N+1)!} \prod_{k=0}^{N} (x - x_k). \tag{15}$$

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). You can see this by plugging $x = x_k$ in the above formula.

This formula by itself is actually very useful as a practical tool. Consider the following:

1. We see that the error depends on the magnitude of the derivative of the *unknown function* f. If we know that our function's $N + 1$st derivative is very large, then polynomial interpolation may be of low accuracy.

2. You should be able to see that to bring the error down for an arbitrary x, we can move the x_k values closer together, introduce more of them, and use a polynomial of higher degree. This will make that product term shrink. This is called *convergence*.
3. It is quite reasonable to ask for the values x_k that *minimize* the above error term. We'll discuss this soon.

3.4 The Runge Phenomenon

Consider the Runge function defined as

$$f(x) = \frac{1}{1 + 25x^2}, x \in [-1, 1]. \tag{16}$$

When we attempt to interpolate this function at equidistant points $x_k = \frac{2k}{N} - 1$, $k = 0, \ldots, N$, the polynomial interpolant oscillates at the ends of $[-1, 1]$. In fact, as the degree of the polynomial (and the number of points) is increased, the error $f(x) - p(x)$ grows without bound! In other words, polynomial interpolation on equispaced points *diverges*!

At this point, you should pause and ask yourself why this is the case. Didn't Weierstrass guarantee convergence when using polynomial interpolation? The answer is yes, but the key here is to realize that Weierstrass says nothing about the set of points or the function's derivatives.

1. In the case of Runge's function, the magnitude of the $N + 1$ derivative grows rapidly as N grows. This means that constant term grows rapidly as well.
2. The equidistant points themselves cause the product term $\prod_{k=0}^{N} (x - x_k)$ to decay slowly with increasing N.
3. This implies that the full error term itself grows rapidly as N increases, since the product is not decaying.

Fortunately, there are several workarounds. If you have the freedom to choose the points at which you sample $f(x)$, picking points that cluster towards the ends of the interval solves this issue. Alternatively, if you don't have this freedom to pick points, you can change to *piecewise polynomial interpolation*, or *least-squares polynomial approximation*. We'll discuss all of these possibilities eventually, but for now, we'll focus on (global) polynomial interpolation.

3.5 Chebyshev zeroes and extrema

Chebyshev zeroes are the zeroes/roots of the Chebyshev polynomials, while Chebyshev extrema are the extrema of these polynomials. Chebyshev polyno-

mials are defined as:

$$T_0(x) = 1, \tag{17}$$
$$T_1(x) = x, \tag{18}$$
$$T_{N+1}(x) = 2xT_N(x) - T_{N-1}(x). \tag{19}$$

These are *orthogonal polynomials*, special families of polynomials that are useful for a variety of applications. The Chebyshev zeroes are given by:

$$x_k = \cos\left(\frac{2k-1}{2N}\pi\right), k = 1, \ldots, N. \tag{20}$$

The Chebyshev extrema are given by:

$$x_k = \cos\left(\frac{k}{N-1}\pi\right), k = 0, \ldots, N-1. \tag{21}$$

These points are not equispaced. Rather, they tend to cluster towards the ends of the interval $[-1, 1]$. They can also be transformed to other intervals easily. Roots of other orthogonal polynomials (Legendre, Hermite, Laguerre) also have this clustering property. However, Chebyshev zeros (which do not include -1 and 1) and Chebyshev extrema (which do), have an important property.

Chebyshev nodes (of both types) minimize the product term from the polynomial error estimate. One can show that the error is upper bounded by:

$$|f(x) - p(x)| \leq \frac{1}{2^N(N+1)!} \max_{-1 \leq t \leq 1} |f^{(N+1)}(t)|. \tag{22}$$

Notice that as N increases, the error decreases *geometrically* or *exponentially* provided the derivative stays bounded or grows more slowly. This means that for every point you add (for every higher degree), you gain another two digits of precision. This is called *spectral convergence*, and this is why polynomial interpolation in Chebyshev points is a phenomenal tool for function approximation in 1D.

3.6 Barycentric 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 *Chebshev 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 nodes[1] , and suddenly polynomial interpolation is stable for thousands of points. Contrast that with the polynomial interpolant allowed by Vandermonde matrices!

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

$$p(x) = \sum_{k=0}^{N} a_k x^k. \tag{23}$$

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

$$\frac{\partial f}{\partial x} \approx \frac{\partial p}{\partial x} = \sum_{k=0}^{N} a_k \frac{\partial}{\partial x} x^k, \tag{24}$$

$$= \sum_{k=0}^{N} a_k k x^{k-1}. \tag{25}$$

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

$$\frac{\partial f}{\partial x} \approx \frac{\partial p}{\partial x} = \sum_{k=0}^{N} y_k \frac{\partial}{\partial x} \ell_k(x). \tag{26}$$

[1] Confusingly, when people say "Chebyshev nodes", they can alternatively mean either zeros or extrema.

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

$$\int_{x_i}^{x_{i+1}} f dx \approx \int_{x_i}^{x_{i+1}} p dx = \sum_{k=0}^{N} a_k \int_{x_i}^{x_{i+1}} x^k dx, \tag{27}$$

$$= \sum_{k=0}^{N} a_k \left[\frac{x_{i+1}^{k+1}}{k+1} - \frac{x_i^{k+1}}{k+1} \right]. \tag{28}$$

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

$$\int_{x_i}^{x_{i+1}} f dx \approx \int_{x_i}^{x_{i+1}} p dx = \sum_{k=0}^{N} y_k \int_{x_i}^{x_{i+1}} l_k(x) dx, \tag{29}$$

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 in this course.

5 Looking ahead

Next, we will discuss 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.