Topic: Interpolation

Nick Battista

Date Created: 5/22/2014
Date Modified: 7/3/2014

1 BACKGROUND

Ever felt interested in learning a new party trick? Have someone give you two lists of data. Tell them you can find a function that will run through all the points and that it will be a unique polynomial that minimizes the error for those points.

Say you're given two lists,$\{x_i\}_{i=0}^N$ and $\{y_i\}_{i=0}^N$, the problem becomes finding a polynomial $p \in \pi_N$ such that $\forall\, i = 0, 1, ..., N$,

$$p(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + ... + a_N x_i^N = y_i.$$

Theorem 1. *There is a* **unique** *polynomial* $\in \pi_n$ *with this property.*

Proof : Let $p, q \in \pi_n$, then

$$(p-q)(x_i) = p(x_i) - q(x_i) = 0,$$

since by assumption they are interpolating the same data. Hence we can factor out all roots of the above difference, i.e.,

$$(p-q)(x) = (x-x_0)(x-x_1)\cdots(x-x_N)h(x),$$

where $h(x)$ is a term coming from the assumption that p and q are different polynomials. However, since both $p, q \in \pi_n$, their difference must exist in the same space, and hence $h(x) = 0$. □

Now let's take a look at a few different ways to interpolate such data!

2 Interpolation Schemes

Essentially to find an interpolating polynomial, if we're given $N+1$ data points, we need $N+1$ equations. Makes sense. Even though we will present 3 different methods to find the polynomial, thanks to the theorem above, we know they will all arrive at the same unique polynomial.

2.1 Standard (Monomial) Interpolation

Since we know we're looking for an N^{th} order polynomial, why don't we do the natural thing and just plug in all the data into a general polynomial to arrive at a system of $N+1$ equations for $N+1$ unknowns. In this way, we are looking at a *monomial* basis, i.e., each basis function has the form x^p, where $p = 1, 2, 3, ..., n$. Let's take a look,

$$\begin{aligned} p(x_0) &= a_0 + a_1 x_0 + a_2 x_0^2 + ... + a_N x_0^N = y_0, \\ p(x_0) &= a_0 + a_1 x_1 + a_2 x_1^2 + ... + a_N x_1^N = y_0, \\ &\vdots \\ p(x_N) &= a_0 + a_1 x_N + a_2 x_N^2 + ... + a_N x_N^N = y_0. \end{aligned}$$

Written in matrix form we have,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ 1 & x_2 & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^2 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \\ a_N \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix}$$

Note that the matrix in the above system is a Vandermonde matrix. As you probably guessed, since this is the most simplistic approach there has to be drawbacks. Although this method works great for small systems, unfortunately as the number of points increases (or $N > 4$...), the system becomes ill-conditioned.

2.2 Newton Interpolation

Sometimes the best way to read a book is one word at a time. This in essence is what Newton interpolation is like. You successively build up the interpolating polynomial data point by data point. Let's start!

For $\{x_0\}$,

$$p(x_0) = y_0$$

For $\{x_0, x_1$,

$$p(x) = y_0 + c_1(x - x_0)$$

Now we must find the value of c. We have the following two equations by our assumptions above,

$$\begin{aligned} p(x_0) &= y_0 = y_0 \\ p(x_1) &= y_1 = y_0 + c_1(x_1 - x_0) \end{aligned}$$

and hence we find that $c_1 = \frac{y_1 - y_0}{x_1 - x_0}$. Therefore our interpolating polynomial for 2-pts is:

$$p(x) = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}.$$

Adding in other points, we could work similarly; however, Newton interpolation has a clever relation for finding all these unknowns coefficients as we build up the interpolating polynomial. We define $f(x_j) = y_j \ \forall \ j = 1, 2, ..., n$. The general Newton Interpolation polynomial takes the following form,

$$p(x) = f(x_0) + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_n(x - x_0)(x - x_1)\cdots(x - x_n)$$

In a similar fashion to the monomial case, since we have a series of $n-$equations and $n-$unknowns, we can construct the following matrix system to solve for the unknown coefficients

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & x_1 - x_0 & 0 & \cdots & \vdots \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N - x_0 & (x_N - x_0)(x_N - x_1) & \cdots & \prod_{j=0}^{N-1}(x_j - x_N) \end{bmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \\ c_N \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{N-1}) \\ f(x_N) \end{pmatrix}$$

As compared to the monomial polynomial basis case, which gives rise to a Vandermonde matrix system, we now have the pleasure to invert a lower-triangular matrix to find the coefficients for our unique interpolating polynomial.

Rather than inverting the above matrix, there is another way to find the coefficients, $\{c_j\}$, which is more desirable in a numerical way (i.e., less expensive). Defining $c_j = f[x_0, x_1, ..., x_j]$,we can solve for the $\{c_j\}$ coefficients by constructing a *Newton Divided Difference table* below:

$$
\begin{array}{lll}
x_0 & f[x_0] = f(x_0) & \\
 & & f[x_0,x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0} \\
x_1 & f[x_1] = f(x_1) & \\
 & & f[x_1,x_2] = \frac{f(x_2)-f(x_1)}{x_2-x_1} \\
x_2 & f[x_2] = f(x_2) & \\
 & \vdots & \\
\vdots & & \\
 & \vdots & \\
x_N & f[x_N] = f(x_N) &
\end{array}
\qquad
f[x_0,x_1,x_2] = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2-x_0}
\qquad
\begin{array}{l}
\\ \\ \\ \vdots \\ \\ \vdots \\ \\ \vdots
\end{array}
$$

For completeness, the coefficients we care about align the top of our table above, i..e, the coefficients in *blue*,

$$p(x) = f[x_0] + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + \ldots + f[x_0,x_1,\ldots,x_N](x-x_0)(x-x_1)\cdots(x-x_n)$$

The recursive formula for the j^{th} polynomial looks like

$$f[x_0,x_1,\ldots,x_j] = \frac{f[x_1,x_2,\ldots,x_j] - f[x_0,x_1,\ldots,x_{j-1}]}{x_j - x_0}.$$

2.2.1 'Newton' Interpolation with Derivatives (Hermite Interpolation)

In all the previous sections we were concerned with the problem of having been given $n-$data points, $\{x_j\}_{j=0}^N$, and $n-$function values, $\{f(x_j)\}_{j=0}^N$, to construct a polynomial that interpolates all the data. (Duh). However, what if we are given more information, like values of derivatives of order 1 or higher? After all, more information should give us a better interpolation, right?!

The long and the short of it is that we can use the *Newton Divided Differences* table to construct an interpolating polynomial; however, there are going to differences. (math pun!)
Let's just state how to handle higher derivatives in this syntax. Suppose we have a point x_i that has $k-$derivatives. Then the data we're given as k identical copies of x_i, and we have

$$
\begin{aligned}
f[x_i] &= f(x_i) \\
f[x_i, x_i] &= f'(x_i) \\
f[x_i, x_i, x_i] &= \frac{1}{2!} f''(x_i) \\
f[x_i, x_i, x_i, x_i] &= \frac{1}{3!} f'''(x_i) \\
&\vdots \\
f[x_i, x_i, ..., x_i] &= \frac{1}{(k-1)!} f^{(k-1)}(x_i)
\end{aligned}
$$

Let's check out an example.

- **Example:** Given $\{x_0, x_0, x_1, x_2\}$ with respective values $\{f(x_0), f'(x_0), f(x_1), f(x_2)\}$, we form the divided differences table,

$$
\begin{array}{llll}
x_0 & f[x_0] & & \\
 & & f'(x_0) & \\
x_0 & f[x_0] & & f[x_0, x_0, x_1] = \frac{f[x_0, x_1] - f'(x_0)}{x_1 - x_0} \\
 & & f[x_0, x_1] & \\
x_1 & f[x_1] & & f[x_0, x_1, x_2] \\
 & & f[x_1, x_2] & \\
x_2 & f[x_2] & &
\end{array}
\qquad f[x_0, x_0, x_1, x_2] = \frac{f[x_0, x_0, x_1] - f[x_0, x_0, x_1]}{x_2 - x_0}
$$

The interpolating polynomial is then

$$
p(x) = f[x_0] + f'(x_0)(x - x_0) + f[x_0, x_0, x_1](x - x_0)^2 + f[x_0, x_0, x_1, x_2](x - x_0)^2(x - x_1)
$$

2.3 Lagrange Interpolation

If you're still awake youve probably noticed how the previous two interpolation schemes resulted in a dense matrix system (with monomial basis) and a lower-triangular matrix system (through Newton interpolation) to find the interpolating polynomial coefficients. What's everyone's favorite linear system to solve? That's right- a diagonal one.

How can we achieve this? We just have to be a tidbit creative in how we choose our polynomial basis. Remember the polynomial we find is going to be unique. What we do know is that for every x_j, we need $p(x_j) = f(x_j) = y_j$. Moreover we can write the desired interpolating polynomial as

$$p(x) = \sum_{j=0}^{N} b_j L_j(x),$$

where each polynomial basis function has the property that

$$L_j(x_k) = \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases}$$

Using this logic we arrive at the following linear system:

$$\begin{bmatrix} L_0(x_0) & & & & \\ & L_1(x_1) & & & \\ & & \ddots & & \\ & & & L_{N-1}(x_{N-1}) & \\ & & & & L_N(x_N) \end{bmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{N-1}) \\ f(x_N) \end{pmatrix}$$

However, this linear system is more superfluous than a math graduate student practicing integer arithmetic for funsies. (*That's what Matlab is for!*) since The matrix above is nothing more than our best friend, $I_{N\times N}$. Hence the coefficients are $b_k = f(x_k)\ \forall\ k$. The million dollar question (okay, more like 2 dollar question), remains how to choose those basis polynomials. Lucky for us, there's a simple trick. We can force the basis polynomials to satisfy the necessary relation. Let's build up the idea with a very simple example.

- **Example:** Suppose we're given 3−pts $\{x_0, x_1, x_2\}$. Now let's chat about how to find $L_0(x)$. Well we know we want $L_0(x_0) = 1, L_0(x_1) = 0$, and $L_0(x_2) = 0$. To satisfy the last two conditions we can impose $L_0(x)$ to have a form as follows,

  $$L_0(x) = A(x - x_1)(x - x_2),$$

  where A is a constant we'll turn our sights till next.

  Welp, we're basically done. To satisfy, $L_0(x_0) = 1$, we just need A to have a form as $A = \frac{1}{(x_0-x_1)(x_0-x_2)}$. Hence

  $$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}.$$

  Similarly, $L_1(x)$ and $L_2(x)$ take analogous forms. Hence we have for three points the basis polynomials take the forms,

$$L_0(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)},$$

$$L_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)},$$

$$L_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Now just like solving linear systems in higher dimensions with Gaussian elimination, this process works in a completely analogous way for $N-$pts.

If you didn't skip the example, hopefully you'll see how easy writing these funky looking basis functions is. In general for $N-$pts, they take a form such as

$$L_j(x) = \prod_{j \neq k} \frac{(x-x_k)}{(x_j-x_k)}.$$

Hence our interpolating polynomials takes the complicated-but-cute form of,

$$p(x) = \sum_{j=0}^{N} \prod_{j \neq k} y_j \frac{(x-x_k)}{(x_j-x_k)}.$$

3 Error Analysis

In this section we'll dive into the error analysis behind polynomial interpolation. We'll also briefly touch upon how we know that adding more data should give us a more accurate interpolation (*Weierstrauss Theorem*) as well as where error accumulates for uniform points and why choosing *Chebyshev* node points to interpolate at is sometimes the bees knees.

3.1 Error Polynomial

Yup, it's that time! Time for everyone's favorite part- error analysis. Remember we're given the task of finding an interpolating polynomial for data $\{x_n\}_{j=0}^{N}$ with respective function values, $\{f(x_n)\}_{j=0}^{N}$.

First let's definite our error, $e(x)$,

$$e(x) = f(x) - p(x),$$

where $f(x)$ is the true function that interpolates the data and $p(x) \in \pi_N$. Now suppose we add another point into the data, t and $f(t)$ and construct another interpolating polynomial, $q(x) \in \pi_{N+1}$.

Using the Newton interpolation idea, we can construct $q(x)$ from the previous polynomial $p(x) \in \pi_N$, i.e.,

$$q(x) = p(x) + f[x_0, x_1, ..., x_N, t](x - x_0)(x - x_1) \cdots (x - x_N).$$

Next we definite a function, $d(t)$, that finds the difference between $f(x)$ and $q(x)$,

$$d(x) = f(x) - q(x).$$

Note that this function has *at least* $N+2$ roots, therefore taking $N+1$ derivatives of $d(x)$, we note that it has *at least* 1 root, call it ξ,

$$d^{N+1}(\xi) = f^{N+1}(\xi) - q^{N+1}(\xi) = 0$$

Hence we have

$$\begin{aligned} f^{N+1}(\xi) &= q^{N+1}(\xi) \\ &= p^{N+1}(\xi) + f[x_0, x_1, ..., x_N, t] \frac{d^{N+1}}{dx^{N+1}} \prod_{j=0}^{N} (x - x_j). \\ &= f[x_0, x_1, ..., x_N, t]\ (N+1)! \end{aligned}$$

since $p(x) \in \pi_N$ and $\frac{d^{N+1}}{dx^{N+1}} \prod_{j=0}^{N} (x - x_j) = (N+1)!$. Hence we find

$$f[x_0, x_1, ..., x_N, t] = \frac{f^{N+1}(\xi)}{(N+1)!}.$$

Therefore, computing the error at t, we see

$$e(t) = f(t) - p(t)$$

. Now since $q(t) = f(t)$, we obtain

$$e(t) = q(t) - p(t),$$

and hence see that

$$e(t) = \left[p(t) - f[x_0, x_1, ..., x_N, t](x - x_0)(x - x_1) \cdots (x - x_N)\right] - p(t) = \frac{f^{N+1}(\xi)}{(N+1)!}(x - x_0)(x - x_1) \cdots (x - x_N).$$

As much as it makes sense that the more information we're given, the more accuracy we can hope to achieve, it still important to recognize that proof is needed to uphold such a claim. Luckily for us, this is exactly what the *Weierstrauss Theorem* explores.

Theorem 2. *If $f(x)$ is continuous on $[a,b]$, then $\lim_{n\to\infty} \min_{p\in\pi_n} ||f(x)-p(x)||_\infty = 0$. (Or anotherwards, $\forall\, \epsilon > 0, \exists\, p(x)$ such that $||f(x)-p(x)||_\infty < \epsilon$.)*

This theorem helps us exert that the more points, the better our interpolation is!

Unfortunately, it does tell us what points to use, if using different points within the same interval will achieve the same amount of accuracy, or many other concerns. Furthermore contrary to what may be considered common sense, thinking that using *uniformly spaced* interpolation points will give the most accurate interpolation, turns out to be false. Using uniformly spaced points leads to a lot of error accumulating near the boundaries of the interval, even for the smoothest of functions! (i.e., C^∞).

What points should one choose instead? Well there are many options- Gauss-Lobatto, Chebyshev, Gauss-Laguerre, Gauss-Hermite,..., each usually coming from some form of the zeroes of orthogonal polynomials arising from Sturm-Liouville eigenvalue ODEs. We'll consider *Chebyshev nodes* as the saving grace to our error troubles.

Let's first recall a few factoids about Chebyshev Polynomials, $T_n(x)$, and Chebyshev Nodes, $\{\hat{x}_j\}$.

- **Polynomials:** $T_n(x) = \cos\left(n\cos^{-1}x\right)$ for $x\in[-1,1]$.
- **Recurrence Relation:** $T_{n+1} = 2xT_n(x) - T_{n-1}$.
- **Orthogonality:** $\displaystyle\int_{-1}^{1} T_n(x)T_m(x)\frac{ds}{\sqrt{1-x^2}} = \begin{cases} 0 & \text{if } n\neq m \\ \pi & \text{if } n=m=0 \\ \frac{\pi}{2} & \text{if } n=m\neq 0 \end{cases}$
- **Notes (roots):** The roots of the Chebyshey polynomials are cleverly enough called the Chebyshev nodes.

  $$\cos(n\cos^{-1}\hat{x}_k) = 0 \quad\Rightarrow\quad \hat{x}_k = \cos\left(\frac{2k-1}{2n}\pi\right),$$

  for $k = 1,2,3,\ldots,n$.

The above information is cute, but let's take a good hard (but leisurely) look at error analysis with the Chebyshev nodes! Recall for a general polynomial interpolation, the error is

$$e(x) = \frac{f^{N+1}(\xi)}{(N+1)!}(x-x_0)(x-x_1)\cdots(x-x_N),$$

for some $\xi \in [a,b]$ given $(N+1)$ data points to interpolate. Now the above relation was still general and did not assume anything about the nodes being chosen. Let's now see what happens when we choose Chebyshev!
First we recall that the polynomial in the error relation is a *monic* polynomial, i.e., a polynomial with a leading coefficient of 1 on the highest order term. Our only hope of improving the error function is by tweaking that monic polynomial somehow, since the function $f(x)$ is just what it is. Our train of thought should be to chose points in an intelligent way to decrease the norm over that monic polynomial on the specified domain.

We now implore a theorem that lassos a nice relation between Chebyshev polynomials (which are also *monic*) and monic polynomials.

Theorem 3. *Consider* $\hat{x}_k = \cos\left(\frac{2k-1}{2n}\pi\right)$, *for* $k = 1,2,3,\ldots,n$. *Then the monic polynomial*

$$T_{N+1}(x) = \prod_{k=0}^{N}(x-\hat{x}_k)$$

of degree $N+1$ *has the smallest possible uniform norm in* $[-1,1]$ *in the sense that* $||T_{N+1}|| < ||p_{N+1}||$ *for any other monic polynomial,* p_{N+1}, *of degree* $N+1$. *Furthermore,*

$$||T_{N+1}|| = 2^{-N}.$$

The proof is left for funsies for the reader. (*Hint:* Do it by contradiction and look at the number of sign changes.)

Now we can easily snap our fingers to call on that theorem to give us a nice result for error analysis with Chebyshev points.

$$\begin{aligned}
\max_{-1\le x\le 1}||e(x)|| &= \max_{-1\le x\le 1}\left|\frac{f^{N+1}(x)}{(N+1)!}(x-x_0)(x-x_1)\cdots(x-x_N)\right|,\\
&= \frac{1}{(N+1)!}\max_{-1\le x\le 1}|(x-x_0)(x-x_1)\cdots(x-x_N)|\max_{-1\le x\le 1}|f^{N+1}(x)|,\\
&= \frac{2^{-N}}{(N+1)!}\max_{-1\le x\le 1}|f^{N+1}(x)|.
\end{aligned}$$

Note: The above error relation is the best we can do for any general function, $f(x)$, that we are trying to interpolate. Moreover, even if we are interpolating on a general domain, $[a,b]$, we can always transform that domain into $[-1,1]$ and find an analogous error expression.