Lecture 7: Function Interpolation

CSC 338: Numerical Methods

Ray Wu

University of Toronto

March 1, 2023

Approximation vs Interpolation

Difference between approximation and interpolation

1. Approximation: Given a set of data points (x_i, y_i) find a function that fits the data. If data is precise enough, may want to consider an interpolant.
2. Function interpolation: for a complicated function, find a simpler function that approximates it.

What's the difference?

- Once we specify the data points for function interpolation, they are identical – so we have the freedom to choose the data points intelligently.
- Additionally, we may be able to consider the global error.

Polynomial interpolation: Motivation

Why interpolate, and why interpolate with polynomials?

- prediction: to calculate the function value at some intermmediate point between the data points.
- manipulation: to find derivatives, integrals, etc of the function.
- Polynomials are easy to calculate, and easy to manipulate – derivative and integral rules are easy.
- Polynomials are universal – can approximate any continuous function (Stone-Weierstrass Theorem).

Interpolation: General considerations

- We assume a linear form for interpolating functions, in other words,

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x) \tag{1}$$

  c_j are the unknown coefficients, and ϕ_j are the basis functions.
- Assume that ϕ_j are linearly independent, which means that if v is zero on the entire interval, then c_j must all be zero.
- Assume that the number of basis functions is equal to the number of data points.

System of Equations

The following system of linear equations arise:

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \dots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \dots & \phi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \dots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

Polynomial Interpolation – Monomial Basis

- The "easy" way to represent a polynomial of degree n is

$$p_n(x) = \sum_{i=0}^{n} c_i x^i. \tag{3}$$

  Basis functions: $\phi_i = x^i$.

- Which equations must c_i satisfy?

$$\forall i, y_i = \sum_{i=0}^{n} c_i x^i \tag{4}$$

  Set up system of linear equations to find c_i:

Polynomial Interpolation – system of linear equations

- System of linear equations $Ac = y$ is given by

$$\begin{bmatrix} 1 & x_0^1 & x_0^2 & \dots & x_0^n \\ 1 & x_1^1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n^1 & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{5}$$

- A is known as the Vandermonde matrix.
- As long as the x_i are distinct, the determinant is nonzero, and hence, A is nonsingular which implies that there is a unique interpolating polynomial.
- For a large number of data points, Vandermonde matrix A is frequently ill-conditioned.
- Alternatives to Vandermonde matrix: Lagrange and Newton basis

Polynomial interpolation: Lagrange Basis

- The Lagrange polynomials $L_j(x)$ satisfy:

$$L_j(x_i) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

- Then, $c_j = y_j$ (easy to solve linear system of equations)
- How to determine such polynomials?

Lagrange Polynomials

- Recall that we want $L_j(x)$ to be
  1. zero on every data point that is **not** x_j
  2. one at x_j.
- To make it zero on every data point not x_j, consider the polynomial

$$p(x) = (x - x_0)(x - x_1)\dots(x - x_{j-1})(x - x_{j+1})\dots(x - x_n) \tag{7}$$

- To make it one on x_j, calculate the value at $x = x_j$ and divide by it:

$$L_j(x) = p(x)/p(x_j) \tag{8}$$

Evaluating Lagrange basis

- Suppose now we actually want to evaluate the interpolating polynomial using Lagrange basis at a certain point x.
- First, compute the **barycentric weights**:

$$w_j = \frac{1}{\prod_{i \neq j}(x_j - x_i)} \tag{9}$$

- Next, calculate

$$\psi(x) = \prod (x - x_i) \tag{10}$$

- Finally, calculate

$$p(x) = \psi(x) \sum \frac{w_j y_j}{(x - x_j)} \tag{11}$$

Newton Basis

- Newton's basis is "in-between" monomial and Lagrange basis: coefficients c_j only depend on themselves and the past
- In other words, the linear system we solve is **triangular**.
- Can use forward/backward substitution.
- Two advantages of Newton's basis:
  - Can add a data point without changing the rest of the interpolant
  - Easy to use divided differences to come up with error estimates in polynomial interpolation.
- Newton basis:

$$\phi_j(x) = \prod_{i=0}^{j-1}(x - x_i) \tag{12}$$

Calculating the interpolant

- Recall the form of $p(x)$:

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_1)(x - x_0) + \cdots + c_n(x - x_{n-1})...(x - x_0) \tag{13}$$

- Determine c_i iteratively:
  - Since $p(x_0) = f(x_0)$, then $c_0 = f(x_0)$.
  - Since $p(x_1) = f(x_1)$, then $c_1 = \frac{f(x_1) - f(x_0)}{x - x_0}$.
  - Next, use the condition that $p(x_2) = f(x_2)$ to determine c_2. Note that c_0 and c_1 have already been determined. With some algebra, we can show that

$$c_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} \tag{14}$$

  - This process continues until all the coefficients are determined.
  - The coefficients c_j are known as **divided differences**.

Divided Differences

- Divided differences are defined recursively as follows:

$$f[x_i] = f(x_i)$$
$$f[x_i, \dots, x_j] = \frac{f[x_{i+1}, \dots, x_j] - f[x_i, \dots, x_{j-1}]}{x_j - x_i}$$

- Calculate them by listing out a table
- The polynomial is then given by

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, x_1, \dots, x_n](x - x_0)\dots(x - x_{n-1}) \tag{15}$$

Monomial, Lagrange, and Newton

- All three methods yield the same result.
- Monomial is the simplest method.
- Lagrange is is the most stable - leads to decoupled equations.
- Newton basis allows for the addition of new points without recalculating the entire polynomial.

Error estimates of polynomial interpolation

- First, we come up with an expression for the error: if f is the original function and p_n the interpolating polynomial, then

$$e_n(x) = f(x) - p_n(x). \tag{16}$$

- Assume we are not on a data point (otherwise, error is zero...). Treat $(x, f(x))$ as a new data point, then

$$f(x) = p_{n+1}(x) = p(x) + f[x_0, \ldots, x_n, x] \prod (x - x_j) \tag{17}$$

and we have

$$e_n(x) = f[x_0, \ldots, x_n, x] \prod (x - x_j) \tag{18}$$

Divided Differences and Derivatives

- Divided differences are approximations to derivatives, and there is a theorem that states

$$\exists \zeta \in [a, b] : f[x_0, x_1, \ldots, x_k] = \frac{f^{(k)}(\zeta)}{k!} \tag{19}$$

  It's like the mean value theorem for higher-order derivatives.
- Divided difference have a symmetrical definition, in other words, it doesn't matter what order you list x_i in.
- Hence, sub in $f[x_0, \ldots, x_n, x]$ into the theorem, take upper bounds on the product, and we know that there exists some ξ for which

$$|e_n(x)| \leq \frac{f^{(n+1)}(\xi)}{(n+1)!}(b-a)^{n+1} \tag{20}$$

- This error could be very large if the derivative becomes large.

What could go wrong with polynomial interpolation

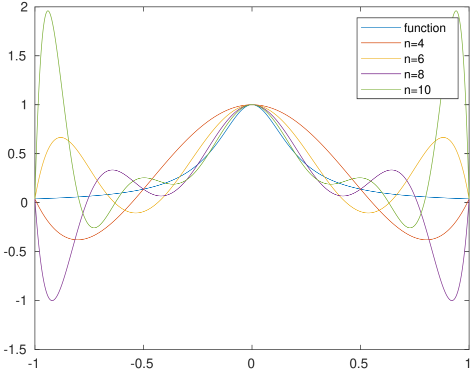

Figure 1: Polynomial interpolation of Runge's function, $f(x) = \frac{1}{1+25x^2}$. As the number of points increase, the approximation becomes **less** accurate.

Chebyshev Interpolation

- So far, we have assumed uniform choice of points.
- This assumption is more strict than necessary – we have the freedom to choose whatever points we wish.
- **Chebyshev interpolation** minimizes the error term

$$\min_{x_j} \max_{s \in [a,b]} \left| \prod (s - x_j) \right| \tag{21}$$

- Assume that $a = -1$, $b = 1$.
- Shift and scale them to the correct interval

$$t_i = a + \frac{b-a}{2}(x_i - 1) \tag{22}$$

- **Chebyshev points** minimize the above expression, defined by

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right) \tag{23}$$

Interpolating with Chebyshev points

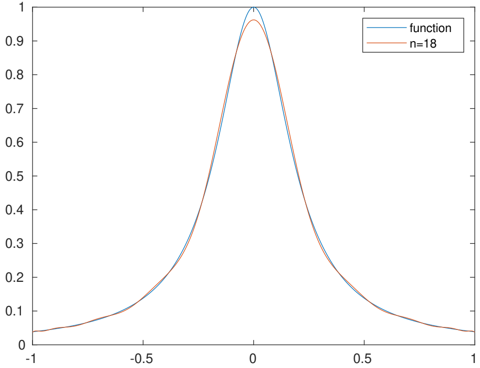

Figure 2: Polynomial interpolation of Runge's function, $f(x) = \frac{1}{1+25x^2}$. The use of Chebyshev points allows the interpolant to fit the original function better.

Discussion of Chebyshev points

- A more detailed explanation of Chebyshev points go beyond this course.
- For now, use of Chebyshev points is the only natural context in which the number of points n can become large.
- Lagrange interpolation should be used in this case.
- Chebyshev points cannot work for every function, e.g. on $[-1, 1]$

$$\frac{\exp(3(x+1))\sin(100(x+1))}{1+20(x+1)^2} \tag{24}$$

- Additionally, unique polynomial implies that if you chage the data point a little bit, your polynomial can change a lot.
- No locality for polynomial interpolation

Failure of Chebyshev interpolation

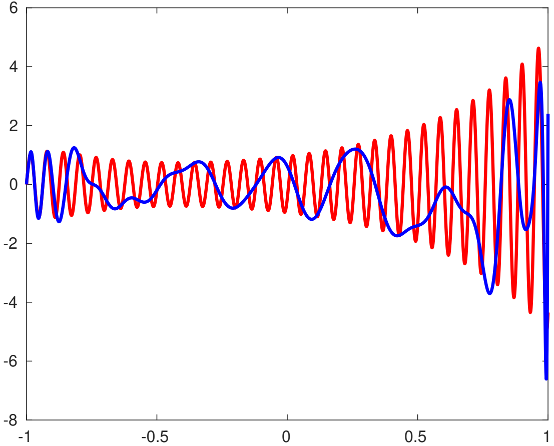

Figure 3: An example where Chebyshev interpolation cannot save us

Piecewise polynomial interpolation

- Let's look again at the error term:

$$|e_n(x)| \leq \frac{f^{(n+1)}(\xi)}{(n+1)!}(b-a)^{n+1} \tag{25}$$

  This term may not be small if the $n+1$st derivative is not small.
- Additionally, higher-order polynomials tend to oscillate, which may not be something we want.
- If we have data, we cannot force it to be at the "chebyshev points".
- No locality: if you change one data point, the entire interpolant will be changed (possibly significantly).

Piecewise Polynomials

- Divide and conquer: partition the domain into n intervals.
- Piecewise polynomials split up the domain $[a, b]$ into smaller segments, to reduce the error.
- Partition at the points $a = t_0 < t_1 < \cdots < t_r = b$.
- Interpolate each segment with a low-degree polynomial.
- Enforce desireable conditions upon the piecewise polynomials.

Linear Splines

- We pick $t_i = x_i$, and let $r = n$.
- Interpolate each subinterval with a line (draw the straight line from (x_k, y_k) to (x_{k+1}, y_{k+1})).
- Each partition can be written as

$$s_i(x) = a_i(x - x_i) + b_i \tag{26}$$

  - $b_i = y_i$, $a_i = (y_{i+1} - y_i)/(x_{i+1} - x_i)$
- Define $h = \max|t_i - t_{i-1}|$. Error on each subinterval is given by

$$e_n(x) \leq \frac{h^2}{8} \max_{\zeta \in [a,b]} |f''(\zeta)| \tag{27}$$

- Important parts: $\mathcal{O}(h^2)$ error, assumes second derivative is bounded.

Cubic splines

- Cubic splines assume every subinterval has a cubic function on it.
- We have $4n$ free parameters, need $4n$ equations to determine them.
- $2n$ equations are interpolation conditions:

$$s_i(x_i) = f(x_i) \text{ and } s_i(x_{i+1}) = f(x_{i+1}) \tag{28}$$

- $2n - 2$ equations are continuity conditions

$$s_i'(x_i) = s_{i+1}'(x_i) \text{ and } s_i''(x_i) = s_{i+1}''(x_i) \tag{29}$$

- This leaves two more equations, which give rise to various specific splines:

Types of Cubic Splines

The three types of Cubic splines in this course:

1. Free boundary/natural spline: $s_1''(x_0) = 0$ and $s_n''(x_n) = 0$
   - Since the second derivatives of the original function are not necessarily zero on the endpoints, this destroys the fourth-order convergence of the method, and near the endpoints the method is only second-order convergent.
2. Clamped boundary conditions: $s_1'(x_0) = f'(x_0)$ and $s_n'(x_n) = f'(x_n)$
   - Not an ideal choice if we do not have information about the derivative on the endpoints. Otherwise, keeps fourth-order accuracy.
3. Not-a-knot: $s_1'''(x_1) = s_2'''(x_1)$ and $s_{n-1}'''(x_{n-1}) = s_n'''(x_{n-1})$.
   - Ideal if we do not know information about $f'(x)$ on the boundary; keeps 4th order convergence.
   - "not-a-knot" means that s_1 and s_2 (s_{n-1} and s_n likewise) are really just one cubic polynomial, hence, the knot is gone.

Cubic Spline Algorithm

How do we compute the coefficients?

- Each subinterval has the following cubic equation:

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{30}$$

- The derivatives are given by the following:

$$s_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \tag{31}$$
$$s_i''(x) = 2c_i + 6d_i(x - x_i) \tag{32}$$
$$s_i'''(x) = 6d_i \tag{33}$$

- Interpolation conditions on left endpoints immediately give us

$$a_i = y_i \tag{34}$$

Cubic Spline Algorithm

- Denote $h_i = x_{i+1} - x_i$
- Next, we consider the interpolation conditions on the right endpoints

$$y_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 \tag{35}$$

- Sub in $a_i = y_i$, and divide by h_i, rearrange:

$$b_i + c_i h_i + d_i h_i^2 = \frac{y_{i+1} - y_i}{h_i} \tag{36}$$

- Smoothness conditions (first derivative):

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1} \tag{37}$$

- Smoothness conditions (second derivative):

$$2c_i + 6d_i h_i = 2c_{i+1} \tag{38}$$

Cubic Spline Algorithm

- Use second derivative smoothness conditions to eliminate d_i

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \tag{39}$$

- Next, we use the right interpolation conditions to eliminate the b_i's:

$$b_i + c_i h_i + \frac{c_{i+1} - c_i}{3} h_i = \frac{y_{i+1} - y_i}{h_i} \tag{40}$$

- Rearrange:

$$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}) \tag{41}$$

- Now, if we can determine the c_i's, all the coefficients are determined.

Cubic Spline Algorithm

- Reduce the counter by 1 from the smoothness condition:

$$b_{i-1} + 2c_{i-1}h_{i-1} + 3d_{i-1}h_{i-1}^2 = b_i \tag{42}$$

- Sub in expressions for b_i and d_i:

$$\frac{y_i - y_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{3}(2c_{i-1} + c_i) + 2c_{i-1}h_{i-1} + \tag{43}$$
$$(c_i - c_{i-1})h_{i-1} = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1})$$

- Now, we have our equations for only c_i's.
- Rearrange once again:

$$h_{i-1}c_{i-1} + 2(h_i + h_{i-1})c_i + h_i c_{i+1} = g_i \tag{44}$$

where g_i is from combining the constants together.

Cubic Spline Matrix

Assuming that we have a natural spline, we have $c_0 = c_n = 0$, then the matrix that arises from the cubic spline is

$$A = \begin{bmatrix} 2(h_0 + h_1) & h_1 & & & \\ h_1 & 2(h_1 + h_2) & h_2 & & \\ & \ddots & \ddots & \ddots & \\ & & h_{n-3} & 2(h_{n-3} + h_{n-2}) & h_{n-2} \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{bmatrix}$$

Then, we can write

$$Ac = g \tag{45}$$

Other cubic Splines

- For Clamped boundary conditions, we look at b_0 and b_n. Sub them in to the relevant equations, and simplification leads to a tridiagonal matrix.
- For not-a-knot boundary conditions, we set $d_0 = d_1$, and $d_{n-1} = d_{n-2}$. Once again we end up with a tridiagonal matrix.

Cubic Spline Matrix

Let's look at the matrix in a bit more detail:

$$A = \begin{bmatrix} 2(h_0+h_1) & h_1 & & & \\ h_1 & 2(h_1+h_2) & h_2 & & \\ & \ddots & \ddots & \ddots & \\ & & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} \\ & & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix}$$

- Matrix is nonsingular as a consequence of diagonal dominance.
- Tridiagonal, symmetric positive definite.
- Represents a global coupling of the unknonwns.
- However, thanks to diagonal dominance, in A^{-1}, the elements away from the diagonal are exponentially decreasing.
- Almost local behavior, unlike linear spline.

Error in the Cubic Spline interpolant

- What is the error in the cubic spline interpolant?
- Assume for simplicity, $h_i = h$.
- Number of subintervals: $n = (b - a)/h$.
- Each subinterval: $\mathcal{O}(h^4)$ accuracy as a result of substituting in the endpoints of the interval.
- Max error: maximum on every interval, still $\mathcal{O}(h^4)$.
- What assumptions about the function?
- Fourth derivative exists and is bounded.

Summary of MATLAB commands

- `p = polyfit(x, y, n)`: fits an n-degree polynomial through the data points specified by `x` and `y`. The coefficients are stored in the vector `p`.
- `y = polyval(p, x)`: evaluates the polynomial with coefficients defined by `p` at the points `x`.
- `y = interp1(x0, y0, x)`: evaluates the linear spline defined by the points `x0` and `y0` at `x`
- `y = spline(x0, y0, x)`: evaluates the cubic spline defined by the points `x0` and `y0` at `x`