Chapter 3 - *Interpolation*

3.1 The Interpolating Polynomial

Interpolation is the process of defining a function that "connects the dots" between specified (data) points.

In this chapter, we focus on two closely related interpolants, the cubic spline and the shape-preserving cubic spline called "`pchip`".

Two distinct points uniquely determine a straight line.

Restated in more mathematical terms, any pair of points (x_1, y_1) and (x_2, y_2) with $x_1 \neq x_2$ determine a unique polynomial in x of degree less than two whose graph passes through the two points.

There may be different formulas for the polynomial, but they all describe the same straight line.

This idea generalizes to more than two points.

For example, any three points (x_1, y_1), (x_2, y_2), and (x_3, y_3) with $x_1 \neq x_2 \neq x_3$ determine a unique polynomial in x of degree less than three whose graph passes through the three points.

In general, given n points (x_k, y_k), $k = 1, 2, \ldots, n$, with distinct x_k, there is a unique polynomial in x of degree less than n whose graph passes through the n points.

It is easiest to remember that the number of data points n is also the number of polynomial coefficients.

Note: Some of the leading coefficients might be zero, so the degree might actually be less than $n - 1$.

Again, there may be many different ways to express the polynomial, but they are all equivalent algebraically, and they all plot the same curve.

This polynomial is called the interpolating polynomial because it passes through the given points; i.e.,

$$P_n(x_k) = y_k, \ k = 1, 2, \ldots, n.$$

Note: Sometimes other polynomials of lower degree are used to fit data; e.g., the line of best fit.

These are not *interpolating* polynomials!

The most direct form in which to express $P_n(x)$ is the *Lagrange form*:

$$P_n(x) = \sum_{k=1}^{n} \left(\prod_{\substack{j=1 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \right) y_k.$$

Note: There are n terms in the sum and $n-1$ terms in each product $\leftrightarrow$ a polynomial of degree less than n.

If $P_n(x)$ is evaluated at $x = x_K$, all the products except the one where $k = K$ are 0.

Furthermore, the Kth product is equal to 1
$\leftrightarrow$ the sum evaluates to y_K just as it should!

Consider the following data set:

x	0	1	2	3
y	-5	-6	-1	16

Then the Lagrange interpolating polynomial is

$$P_4(x) = \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)}(-5) + \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}(-6)$$
$$+ \frac{(x-0)(x-1)(x-3)}{(2-0)(2-1)(2-3)}(-1) + \frac{(x-0)(x-1)(x-2)}{(3-0)(3-1)(3-2)}(16).$$

Each term is of degree 3, so $P_4(x)$ is of degree 3.

(Because the coefficient of x^3 does not vanish, the degree is 3. Verify!)

If we substitute $x = 0, 1, 2,$ or 3, three of the terms vanish, and the fourth produces the corresponding value from the data set.

Polynomials are usually not written in their Lagrange form.

They are usually written in their power form; e.g., the previous Lagrange polynomial can be written as

$$x^3 - 2x - 5.$$

Of course, a polynomial in Lagrange form can always be written out in power form if you like.

But if we want to obtain the power form of an interpolating polynomial directly,

$$P_n(x) = c_1 x^{n-1} + c_2 x^{n-2} + \ldots + c_{n-1} x + c_n,$$

its coefficients c_k can (in principle!) be computed by solving a system of linear equations:

$$\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \ldots & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

The coefficient matrix of this linear system has a special structure: It is known as a Vandermonde matrix, $\mathbf{V}$.

Its elements are

$$v_{ij} = x_i^{n-j}.$$

Note: Depending on the definition being used, the columns of a Vandermonde matrix are sometimes written in the opposite order.

But in MATLAB, polynomial coefficient vectors are always assumed to be in decreasing order; i.e., the coefficient of the highest power is the first element.

The MATLAB function `vander` automatically generates Vandermonde matrices.

See `vanderDemo.m`

It can be shown that Vandermonde matrices are nonsingular as long as the points x_k are distinct.

However, it can also be shown that a Vandermonde matrix can often be very badly conditioned!

Consequently, using the power form of a polynomial and the Vandermonde matrix to solve for the coefficients will only work well for problems involving **a few well-spaced and well-scaled data points**.

It is dangerous to use as a general-purpose approach!

There are several (external) MATLAB functions that implement different interpolation algorithms.

All of them are called as follows:

```
P = polyinterp(x_k,y_k,x)
```

The first 2 input arguments `x_k` and `y_k` are vectors of the same length that contain the data.

The third input argument `x` is a vector of points where you would like the interpolant to be evaluated.

The output `P` is the same length as `x` and has elements `P(i) = polyinterp(x_k,y_k,x(i))`.

Of course, if you evaluate `P` at `x_k`, you get `y_k`.

The interpolation function `polyinterp` is based on the Lagrange interpolating polynomial.

See `polyinterpDemo.m`

Newton Interpolation

We have seen two extreme cases of representations of polynomial interpolants:

1. The Lagrange form, which allows you to *write out* $P_n(x)$ *directly* but is *very complicated*.

2. The power form, which is *easy to use* but requires the *solution of a typically ill-conditioned Vandermonde linear system*.

Newton interpolation provides a trade-off between these two extremes.

The Newton interpolating polynomial takes the form

$$P_n(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + \ldots + c_n \prod_{k=1}^{n-1} (x - x_k).$$

Example

Interpolate the points $(-2, -27)$, $(0, -1)$, $(1, 0)$ using Newton interpolation.

Let

$$P_3(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2).$$

Then,

$$\begin{aligned}
-27 &= c_1, \\
-1 &= c_1 + c_2(0 + 2) \implies c_2 = \frac{26}{2} = 13, \\
0 &= c_1 + c_2(1 + 2) + c_3(1 + 2)(1 - 0) \implies c_3 = \frac{27 - 13(3)}{3} = -4.
\end{aligned}$$

Thus,

$$P_3(x) = -27 + 13(x + 2) - 4(x + 2)(x).$$

Once the coefficients c_k, $k = 1, 2, \ldots, n$, have been computed, the Newton polynomial can be efficiently evaluated using Horner's method.

Note also that Newton interpolation can be done incrementally; i.e., the interpolant can be created as points are being added.

So we fit a straight line to two points, then add a point and fit a quadratic to three points, then add a point and fit a cubic to four points, etc.

Final notes:

- The coefficients c_k can be obtained *recursively* in $\mathcal{O}(n^2)$ operations using *divided differences*.

  These computations are less prone to overflow and underflow than the previous methods.

- In theory, any order of the interpolation points x_k is OK, but the conditioning depends on this ordering!

  Left-to-right ordering is not necessarily the best!

  Two better ideas are to order the points in increasing distance from their mean or from a specified point at which the interpolant will be evaluated.

Another Example

We will also be making use of the following data set in the remainder of this chapter.

See `polyinterpDemo2.m`

Here we see the primary difficulty with high-degree polynomial interpolation at equally spaced points.

Even with only six equally spaced points, the interpolant shows an unnatural-looking amount of variation (overshoots, wiggles, etc.), especially in the first and last subintervals.

Consequently, high-degree polynomial interpolation at equally spaced points is hardly ever used for data and curve fitting.

In this course, its primary application is in the derivation of other numerical methods.

Parametric Interpolation

None of the techniques described so far can be used to generate curves like the letter "S".

That's because the letter "S" is not a function (a vertical line intersects "S" more than once).

One way to get around this problem is to describe the curve in terms of a *parameter* t.

We connect the points (x_0, y_0), (x_1, y_1), . . . , (x_n, y_n) *in that order* by using a parameter $t \in [t_0, t_n]$ with $t_0 < t_1 < \ldots < t_n$ such that

$$x_k = x_k(t_k),\ y_k = y_k(t_k), \qquad k = 0, 1, \ldots, n.$$

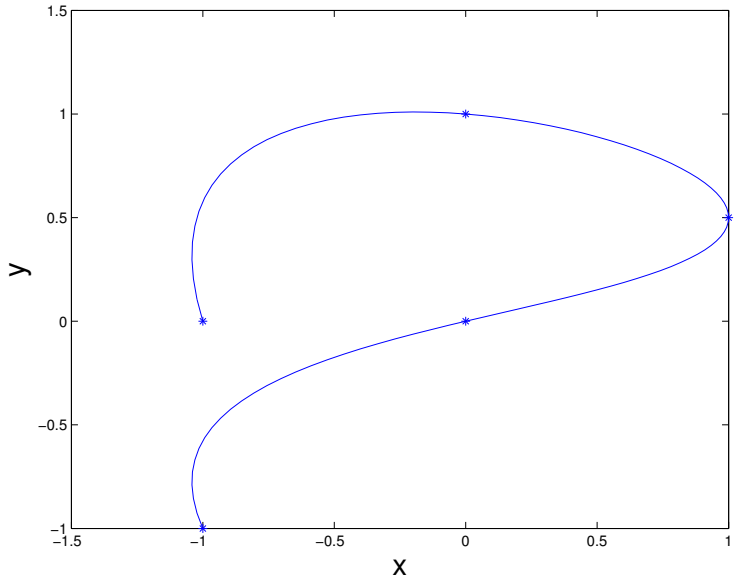
For example, suppose we had the following set of 5 data points, and we parameterized it by a parameter t of 5 equally spaced points in $[0, 1]$.

i	0	1	2	3	4
t_i	0	0.25	0.5	0.75	1
x_i	-1	0	1	0	1
y_i	0	1	0.5	0	-1

We could construct a pair of Lagrange polynomials to interpolate $x(t)$ and $y(t)$.

The data and the interpolant are shown in the figure.

See `parametricInterpolation.m`



3.2 Piecewise Linear Interpolation

This is the perhaps the most intuitive form of interpolation, even if you're still not sure what all the words mean.

Piecewise linear interpolation is simply connecting data points by straight lines.

"Linear interpolation" means to use straight-line interpolants.

We say it is "piecewise" interpolation because you normally need different straight lines to connect different pairs of points.

This is the default behaviour in MATLAB's `plot` routine.

If your data points are sufficiently close together, you don't notice the jaggedness associated with piecewise linear interpolation.

See `plotDemo.m`

Here is an outline of the algorithm used to produce a piecewise linear interpolant.

The steps of this algorithm are used as a basis for more sophisticated piecewise polynomial interpolants (or splines).

First, the interval index k is determined such that

$$x_k \leq x \leq x_{k+1}.$$

Second, we define a local variable $s := x - x_k$.

Third, we compute the first divided difference

$$\delta_k := \frac{y_{k+1} - y_k}{x_{k+1} - x_k}.$$

Finally, we construct the interpolant

$$\begin{aligned} P(x) &= y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k) \\ &= y_k + \delta_k s. \end{aligned}$$

This formula should look familiar!

This is the Newton form of the (linear) interpolating polynomial.

It can be generalized to higher-degree interpolants by using higher-order divided differences; i.e., divided differences of divided differences.

So we have constructed the straight line that passes through (x_k, y_k) and (x_{k+1}, y_{k+1}).

The points x_k are sometimes called breakpoints.

Note: $P(x)$ *is* a continuous function of x, but its first derivative $P'(x)$ is not!

$\rightarrow P'(x) = \delta_k$ on each subinterval and jumps at the breakpoints.

3.3 Piecewise Cubic Hermite Interpolation

Many of the most effective interpolants are based on piecewise cubic polynomials.

Let $h_k := x_{k+1} - x_k$ be the length of the kth subinterval.

Then

$$\delta_k = \frac{y_{k+1} - y_k}{h_k}.$$

Let $d_k := P'(x_k)$.

Note: If $P(x)$ is piecewise linear, then d_k is not really defined because $d_k = \delta_{k-1}$ on the left of x_k, but $d_k = \delta_k$ on the right of x_k. Usually $\delta_{k-1} \neq \delta_k$!

But for higher-order interpolants (like cubics), it is possible to force the interpolant to be smooth at the breakpoints.

This is done by forcing the derivative at the right end of one piecewise cubic to agree with the derivative at the left end of the next piecewise cubic.

Why not? A cubic polynomial has 4 degrees of freedom (i.e., the 4 coefficients c_1, c_2, c_3, c_4).

We can specify 4 pieces of information, and as long as the 4×4 system of linear equations is non-singular, we can obtain unique values for the 4 unknowns c_i and hence uniquely specify the cubic polynomial.

Note: Until now, we have specified that the 4 pieces of information should all be function (data) values (x_k, y_k); i.e., $P(x)$ is the unique cubic that passes through the 4 data points.

This is not necessary!

In fact what we will do now (in order to enforce smoothness) is to specify function values and slopes (first derivatives) at the end-points of each subinterval to define the piecewise cubic polynomial.

For example, suppose $P(x) = c_1x^3 + c_2x^2 + c_3x + c_4$.

We can specify say $P(0) = 0$, $P(1) = 3$, $P'(0) = 1$, and $P'(1) = 2$.

This leads to 4 equations:

$$\begin{array}{rcrcrcrcl} & & & & & & c_4 & = & 0, \\ c_1 & + & c_2 & + & c_3 & + & c_4 & = & 3, \\ & & & & c_3 & & & = & 1, \\ 3c_1 & + & 2c_2 & + & c_3 & & & = & 2. \end{array}$$

These can be solved to obtain $c_4 = 0$, $c_3 = 1$, $c_2 = 5$, and $c_1 = -3$.

Hence we have a cubic polynomial that agrees with the two data points and the two slopes we specified.

Our equations were simplified because we considered the interval $[0, 1]$.

Consider the following cubic polynomial on the interval $x_k \le x \le x_{k+1}$ expressed in terms of local variables $s = x - x_k$ and $h = h_k$:

$$\begin{aligned} P(x) =& \frac{3hs^2 - 2s^3}{h^3} y_{k+1} + \frac{h^3 - 3hs^2 + 2s^3}{h^3} y_k \\ &+ \frac{s^2(s-h)}{h^2} d_{k+1} + \frac{s(s-h)^2}{h^2} d_k. \end{aligned}$$

This is a cubic polynomial in s (and hence in x) that satisfies 4 interpolation conditions: 2 on function values and 2 on (possibly unknown) derivative values.

$$\begin{aligned} P(x_k) &= y_k, \quad & P(x_{k+1}) &= y_{k+1}, \\ P'(x_k) &= d_k, \quad & P'(x_{k+1}) &= d_{k+1}. \end{aligned}$$

Interpolants for derivatives are known as Hermite or osculatory interpolants because of the higher-order contact at the breakpoints.

(*Osculari* is Latin for "to kiss".)

If we know both function values and first derivatives at a set of points, then a piecewise cubic Hermite interpolant can be fit to those data.

But if we are not given the derivative values, we need to define the slopes d_k somehow.

We will study 2 possible ways to do this called `pchip` and `spline` in MATLAB.

3.4 Shape-Preserving Cubic Spline (`pchip`)

`pchip` stands for "piecewise cubic Hermite interpolating polynomial."

Unfortunately, the catchy name does not precisely specify how the interpolant is defined – there are many ways to have a "piecewise cubic Hermite interpolating polynomial."

The key features of the `pchip` interpolant in MATLAB is that it is *shape preserving* and "visually pleasing".

The key idea is to determine the slopes d_k so that the interpolant does not oscillate too much.

We will not study the `pchip` spline in mathematical detail; rather we will focus on its qualitative properties and performance.

3.5 Cubic Spline

The final piecewise cubic interpolating function we consider in this course is a cubic spline.

The term "spline" originates from the name of an instrument used in drafting.

A real spline is a thin, flexible wooden or plastic instrument that is passed through given data points; it is used to define a smooth curve in between the points.

Physically, the spline takes its shape by naturally minimizing its own potential energy subject to it passing through the data points.

Mathematically, the spline must have a continuous second derivative (curvature) and pass through the data points.

The breakpoints of a spline are also referred to as *knots* or *nodes*.

Note: Splines extend far beyond the one-dimensional, cubic, interpolatory spline we are studying.

There are multidimensional, high-order, variable-knot, and approximating splines.

The first derivative $P'(x)$ of our piecewise cubic function is defined by different formulas on either side of a knot x_k.

However, both formulas are designed to give the same value d_k at x_k, so $P'(x)$ is continuous.

We have no such guarantee for the second derivative; however, we choose continuity of the second derivative to be a defining condition for the cubic spline.

Applying the preceding approach to each interior knot x_k, $k = 2, 3, \ldots, n-1$, gives $n-2$ equations involving the n unknowns d_k.

A different approach is necessary near the ends of the interval.

One effective strategy is known as "not-a-knot".

The idea is to use a single cubic on the first two subintervals $(x_1 \leq x \leq x_3)$ and on the last two subintervals $(x_{n-2} \leq x \leq x_n)$.

The idea is if there is only one cubic on each of the first and last pairs of intervals, it is as if x_2 and x_{n-1} are not there – they are not treated like other knots.

e.g., on the first pair of intervals, we can pretend there are two different cubic polynomials $P_1(x)$, $P_2(x)$ and impose the condition

$$P_1'''(x_2+) = P_2'''(x_2-).$$

Together with the continuity of the cubic spline and its first two derivatives, this forces $P_1(x) \equiv P_2(x)$.

With the two end conditions, we now have n linear equations in n unknowns:

$$\mathbf{A}\mathbf{d} = \mathbf{r},$$

where $\mathbf{d} = (d_1, d_2, \ldots, d_n)^T$ is the vector of slopes.

The slopes can now be computed by

$$\mathbf{d} = \mathbf{A} \backslash \mathbf{r}.$$

Because most of the elements of $\mathbf{A}$ are zero, it is appropriate to store A in a *sparse* data structure.

The $\backslash$ operator can then take advantage of the tridiagonal structure and solve the linear equations in time and storage proportional to n, the number of data points.

Bézier Curves

Applications in computer graphics and computer-aided design (CAD) require the rapid generation of smooth curves that can be quickly and conveniently modified.

For reasons of aesthetics and computational expense, we do not want the entire shape of the curve to be affected by small local changes.

This rules out interpolating polynomials or splines!

In 1962, Bézier and de Casteljau independently developed a method for doing this while working on CAD systems for the French car companies Renault and Citroën, respectively. Other studies were carried out by Ferguson (Boeing) and Coons (Ford).

However, the Renault software was described in several publications by Bézier, and hence his name has become associated with this theory.

Bernstein polynomials

To understand Bézier curves, we start with the Bernstein polynomials of degree n on the interval $[0, 1]$:

$$B_i^{(n)}(x) = \binom{n}{i} (1-x)^{n-i} x^i,$$

where the binomial coefficient is

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}.$$

In computer graphics, the most popular Bézier curve is cubic ($n = 3$).

This leads to the polynomials

$$B_0^{(3)}(x) = (1-x)^3, \qquad B_1^{(3)}(x) = 3(1-x)^2 x,$$
$$B_2^{(3)}(x) = 3(1-x)x^2, \qquad B_3^{(3)}(x) = x^3.$$

Construction of Bézier curves

Four points $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$ (in 2 or 3 dimensions) now define the cubic Bézier curve:

$$\mathbf{B}(x) = \sum_{i=0}^{3} \mathbf{c}_i B_i^{(3)}(x).$$

The points $\mathbf{c}_i$ are known as control points.

$\mathbf{B}(x)$ starts at $\mathbf{c}_0$ going toward $\mathbf{c}_1$ and arrives at $\mathbf{c}_3$ coming from $\mathbf{c}_2$.

$\mathbf{B}(x)$ only interpolates $\mathbf{c}_0$ and $\mathbf{c}_3$; i.e., it does not generally pass through $\mathbf{c}_1$ or $\mathbf{c}_2$ — these points only provide directional information.

In fact, $\mathbf{B}(x)$ is tangent to the line connecting $\mathbf{c}_0$ and $\mathbf{c}_1$ (and to the line connecting $\mathbf{c}_2$ and $\mathbf{c}_3$), but it is "more tangent" the further $\mathbf{c}_1$ is away from $\mathbf{c}_0$ (and the further $\mathbf{c}_2$ is away from $\mathbf{c}_3$).

Example

For a non-interactive demo see

bezierDemo.m

To see the effects on the curve from changing the control points, try using nodes $(0, 0)$, $(1, 0)$ with control points at

- $(0.25, 0.25)$ and $(0.75, 0.25)$
- $(1, 1)$ and $(0.5, 0.5)$
- $(2, 2)$ and $(2, -1)$
- $(0.5, 0.5)$ and $(2, -1)$

A more interactive demo can be downloaded from MATLAB Central:

Yet another Bézier curve demo

Summary of observations

When interpolating data, there is a tradeoff between smoothness of the interpolant and a somewhat subjective property that we might call local monotonicity or "shape preservation".

At one extreme, we have the piecewise linear interpolant: It has hardly any smoothness. It is continuous, but there are jumps in its first derivative.

On the other hand, it preserves the local monotonicity of the data. It never overshoots the data, and it is increasing, decreasing, or constant on the same intervals as the data.

At the other extreme, we have the full-degree polynomial interpolant. It is infinitely differentiable. But it often fails to preserve shape, particularly near the ends of the interval.

The `pchip` and `spline` interpolants are in between these two extremes.

The `spline` is smoother than `pchip`. The `spline` has 2 continuous derivatives, whereas `pchip` has only 1.

A discontinuous second derivative implies discontinuous curvature. The human eye can detect large jumps in curvature! This might be a factor in choosing `spline` over `pchip`.

On the other hand, `pchip` is guaranteed to preserve shape, whereas `spline` might not.

The best spline is often in the eye of the beholder!