7 Interpolation

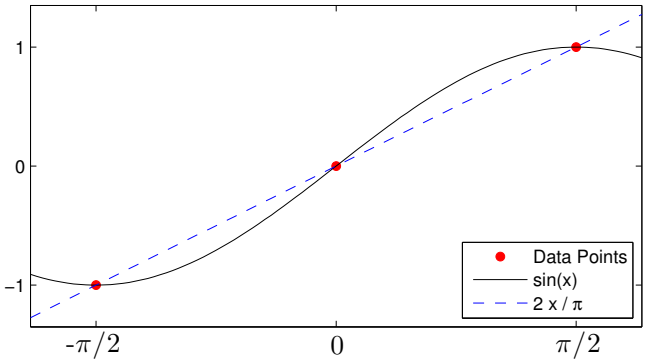
It is both helpful and convenient to express relations in data with functions. This allows for estimating the dependent variables at values of the independent variables not given in the data, taking derivatives, integrating, and even solving differential equations. In this chapter, we will look at one of the common classes of such methods: interpolants. An interpolant of a set of points is a function that passes through each of the data points. For example, given the points $(0, 0)$, $(\pi/2, 1)$, and $(-\pi/2, -1)$, both

$$f(x) = \sin(x)$$

and

$$g(x) = \frac{2x}{\pi}$$

would be interpolating functions, as shown in the plot below.



There are many applications where we would prefer to describe data with an interpolant. In this chapter, we will consider interpolation by a single polynomial, as well as piecewise-polynomial interpolation.

7.1 Interpolation by a single polynomial

Given n points: (x_i, y_i), $i = 1, 2, ..., n$, a degree $n - 1$ polynomial can be found that passes through each of the n points (assuming the x_i's are distinct of course). Such a polynomial would then be an interpolant of the data points. Since the interpolating polynomial is degree $n - 1$, it must be of the form

$$p(x) = c_1 + c_2 x + c_3 x^2 + ... + c_n x^{n-1}.$$

Thus, there are n unknowns $(c_1, \ldots, c_n)$ that, if we could determine them, would give us the interpolating polynomial.

Requiring $p(x)$ to interpolate the data leads to n equations. That is, if $p(x)$ is to pass through data point (x_i, y_i), then it must hold that $y_i = p(x_i)$. Thus, for each i $(1 \leq i \leq n)$, we get the equation

$$y_i = c_1 + c_2 x_i + c_3 x_i^2 + ... + c_n x_i^{n-1}.$$

Since all the x_i's and y_i's are known, each of these n equations is linear in the unknown c_j's. Thus, we have a square linear system that can be solved to determine the unknown coefficients:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & ... & x_1^{n-1} \\ 1 & x_2 & x_2^2 & ... & x_2^{n-1} \\ 1 & x_3 & x_3^2 & ... & x_3^{n-1} \\ : & : & : & ... & : \\ 1 & x_n & x_n^2 & ... & x_n^{n-1} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ : \\ c_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ : \\ y_n \end{pmatrix} \tag{7.1}$$

Solving this linear system will determine the polynomial. Consider the following example:

Example 49. *Find a polynomial that interpolates the five points*

$$(0,1), (1,2), (2,2), (3,6), (4,9).$$

```
% x and y values of the 5 points to interpolate:
px = [0 1 2 3 4];
py = [1 2 2 6 9];
n = length(px);

% build nxn matrix column by column
A = zeros(n,n);
for i=1:n
  A(:,i) = px.^(i-1);
end

% now solve for the coefficients:
c = A\py';

% output matrix and coefficients:
A
c

% plot the points and the polynomial at the points x
x = linspace(-1,5,100);
y = c(1) + c(2)*x + c(3)*x.^2 + c(4) *x.^3 + c(5)*x.^4;
plot(px,py,'rs',x,y,'k')
```

This code will produce the output:

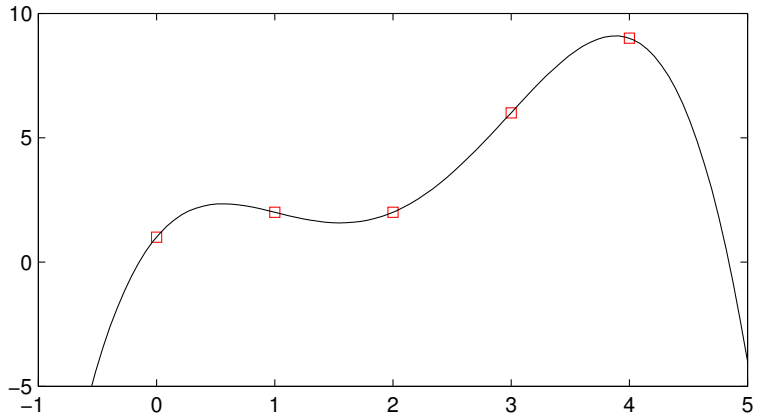
```
A =
     1     0     0     0     0
     1     1     1     1     1
     1     2     4     8    16
     1     3     9    27    81
     1     4    16    64   256

c =
   1.000000000000000
   5.666666666666661
  -7.583333333333332
   3.333333333333333
  -0.416666666666667
```

Thus we have found our interpolating polynomial for the 5 points:

$$p(x) = 1.0000 + 5.6667x - 7.5833x^2 + 3.3333x^3 - 0.4167x^4.$$

Plotting it along with the original data points gives the plot below, from which we can see that the curve does indeed pass through each point.



7.1.1 Lagrange interpolation

The matrix from equation (7.1) is ill-conditioned for n more than a few. Note this is a famous matrix called the Vandermonde matrix. For example, if the $x_i's$ are equally spaced on [0,10], then we can calculate condition numbers to be

```
n=7: cond(A)=1.2699e+07
n=8: cond(A)=2.9823e+08
n=9: cond(A)=7.2134e+09
n=10: cond(A)=1.7808e+11
```

```
n=11: cond(A)=4.4628e+12
n=12: cond(A)=1.1313e+14
n=13: cond(A)=2.8914e+15
n=14: cond(A)=7.2577e+16
```

On the one hand, it is probably not a good idea to try to fit data to a single higher-order polynomial at equally spaced or randomly selected points, since such polynomials tend to be oscillatory near the two ends of the interval of interest. However, if one does wish to construct a single higher-order polynomial, then there are different ways to build interpolating polynomials that do not require an ill-conditioned linear solve. One such way is called 'Lagrange interpolation', and since the interpolating polynomial of degree $n-1$ is unique for n data points, we will be guaranteed that it produces the same answer as we get from solving the linear system above (if no numerical error is present).

To define Lagrange interpolation, we first must define the Lagrange basis. As a point of reference, the *monomial basis* for degree $n-1$ polynomials is $\{1,\ x,\ ...,\ x^{n-1}\}$, and our task with the linear system was to determine the coefficients c_i, which in turn defined the interpolating polynomial $p(x)$. In the Lagrange case, there are also n basis functions, and they are defined as follows, for each $i = 1, 2, ..., n$:

$$\begin{aligned} l_i(x) &= \frac{(x-x_n)(x-x_{n-1})\cdots(x-x_{i+1})(x-x_{i-1})\cdots(x-x_1)}{(x_i-x_n)(x_i-x_{n-1})\cdots(x_i-x_{i+1})(x_i-x_{i-1})\cdots(x_i-x_1)} \\ &= \frac{\Pi_{j=1,j\neq i}^{n}(x-x_j)}{\Pi_{j=1,j\neq i}^{n}(x_i-x_j)} \end{aligned}$$

The important things to notice about these basic functions is:

- They are degree $n-1$ polynomials, so any linear combination of them is also a degree $n-1$ polynomial,
- $l_i(x_i) = 1$,
- $l_i(x_j) = 0$ if $j \neq i$.

We wish to find an interpolating polynomial that is a linear combination of the l_i's, which means we want to find the c_i's that define

$$p_L(x) = c_1 l_1(x) + c_2 l_2(x) + ... + c_n l_n(x),$$

where

$$p_L(x_i) = y_i, \quad i = 1, 2, ...n.$$

It turns out the finding the c_i's is very easy! Consider the first data point, (x_1, y_1) for which we want $y_1 = p_L(x_1)$. By construction of the l_i's, we have that $l_1(x_1) = 1$ and $l_j(x_1) = 0$ for j=2,3,...,n. Thus

$$p_L(x_1) = c_1 l_1(x_1) + 0 + ... + 0 = c_1,$$

which means $c_1 = y_1$. The same thing can be done for the other data points. This precisely and completely defines the **Lagrange interpolating polynomial**:

$$p_L(x) = y_1 l_1(x) + y_2 l_2(x) + ... + y_n l_n(x). \tag{7.2}$$

Again, since the degree $n-1$ interpolating polynomial of n points is unique, and since the Lagrange and monomial interpolating polynomials are both degree $n-1$, they must be equal.

Example 50. *Use the Lagrange interpolation method to find the interpolating polynomial of the points*

$$(0,1), (1,2), (2,2), (3,6), (4,9).$$

Note that in the previous example, using the monomial basis, we found the interpolating polynomial to be

$$p_M(x) = 1.0000 + 5.6667x - 7.5833x^2 + 3.3333x^3 - 0.4167x^4.$$

The Lagrange basis functions are

$$\begin{aligned}
l_1(x) &= \frac{(x-1)(x-2)(x-3)(x-4)}{(-1)(-2)(-3)(-4)} = \frac{(x-1)(x-2)(x-3)(x-4)}{24} \\
l_2(x) &= \frac{(x-0)(x-2)(x-3)(x-4)}{(1)(-1)(-2)(-3)} = \frac{(x-0)(x-2)(x-3)(x-4)}{-6} \\
l_3(x) &= \frac{(x-0)(x-1)(x-3)(x-4)}{(2)(1)(-1)(-2)} = \frac{(x-0)(x-1)(x-3)(x-4)}{4} \\
l_4(x) &= \frac{(x-0)(x-1)(x-2)(x-4)}{(3)(2)(1)(-1)} = \frac{(x-0)(x-1)(x-2)(x-4)}{-6} \\
l_5(x) &= \frac{(x-0)(x-1)(x-2)(x-3)}{(4)(3)(2)(1)} = \frac{(x-0)(x-1)(x-2)(x-3)}{24}
\end{aligned}$$

and the Lagrange interpolating polynomial is defined by

$$p_L(x) = l_1(x) + 2l_2(x) + 2l_3(x) + 6l_4(x) + 9l_5(x).$$

With some arithmetic, one can expand $p_L(x)$ to show that $p_L(x) = p_M(x)$.

The following MATLAB script creates the Lagrange basis functions, then creates the Lagrange interpolant.

```
xpts = [0,1,2,3,4];
ypts = [1,2,2,6,9];

% define the Lagrange basis functions
```

```
l1 = @(x) (x-xpts(2)).*(x-xpts(3)).*(x-xpts(4)).*(x-xpts(5)) ...
       ./ ( (xpts(1)-xpts(2)).*(xpts(1)-xpts(3)).*...
       (xpts(1)-xpts(4)).*(xpts(1)-xpts(5)));

l2 = @(x) (x-xpts(1)).*(x-xpts(3)).*(x-xpts(4)).*(x-xpts(5)) ...
       ./ ( (xpts(2)-xpts(1)).*(xpts(2)-xpts(3)).*...
       (xpts(2)-xpts(4)).*(xpts(2)-xpts(5)));

l3 = @(x) (x-xpts(1)).*(x-xpts(2)).*(x-xpts(4)).*(x-xpts(5)) ...
       ./ ( (xpts(3)-xpts(1)).*(xpts(3)-xpts(2)).*...
       (xpts(3)-xpts(4)).*(xpts(3)-xpts(5)));

l4 = @(x) (x-xpts(1)).*(x-xpts(2)).*(x-xpts(3)).*(x-xpts(5)) ...
       ./ ( (xpts(4)-xpts(1)).*(xpts(4)-xpts(2)).*...
       (xpts(4)-xpts(3)).*(xpts(4)-xpts(5)));

l5 = @(x) (x-xpts(1)).*(x-xpts(2)).*(x-xpts(3)).*(x-xpts(4)) ...
       ./ ((xpts(5)-xpts(1)).*(xpts(5)-xpts(2)).*...
       (xpts(5)-xpts(3)).*(xpts(5)-xpts(4)));


% Define the interpolating polynomial
p = @(x) ypts(1)*l1(x) + ypts(2)*l2(x) + ...
ypts(3)*l3(x) + ypts(4)*l4(x) + ypts(5)*l5(x)
```

7.2 Chebyshev interpolation

In this section, we discuss an optimal approach to efficiently construct and evaluate the polynomial interpolant $p(x)$ that is guaranteed to converge to $f(x)$. We recommend single polynomial interpolation to be performed this way.[1]

First, we emphasize that the x-points of data should be chosen appropriately whenever possible. If n is larger than a few, then typically it is a bad idea to interpolate data with a single polynomial at equally spaced or randomly selected points. In many cases, such a high order polynomial tends to be oscillatory near the two ends of the interval $[a, b]$ of interpolation. This is called the 'Runge phenomenon' and has been incorrectly associated with the high order of the interpolant in many textbooks. In fact, it is the *location* of the x-points that causes the poor behavior of the interpolant, *not the degree* itself. Instead, if we

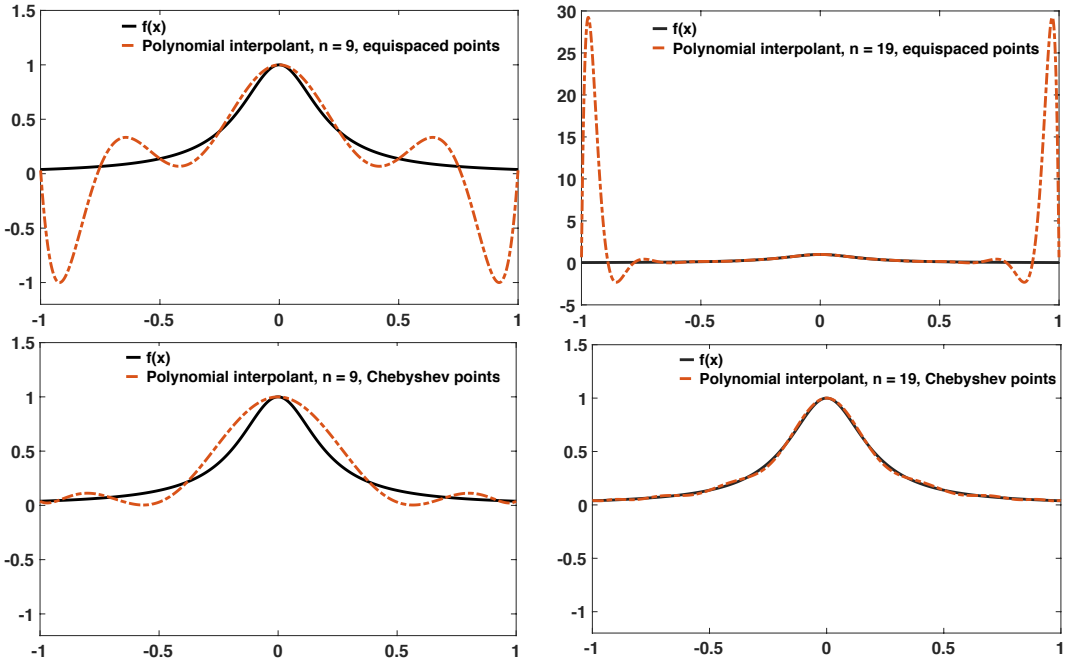
1 For in-depth discussion about polynomial approximations, see *Approximation Theory and Approximation Practice*, by L. N. Trefethen, SIAM, 2013.

choose the x-points at the Chebyshev points

$$x_i = -\frac{b-a}{2}\cos\frac{(i-1)\pi}{n-1} + \frac{b+a}{2}, \quad 1 \le i \le n \tag{7.3}$$

on the interval $[a,b]$ of interest, the interpolant $p(x)$ would be increasingly more accurate approximation to $f(x)$ as the number n of data points increases, as long as $f(x)$ is differentiable on $[a,b]$. This $p(x)$ is called a Chebyshev interpolant.

Example 51. *Approximate* $f(x) = \frac{1}{1+25x^2}$ *over the interval* $[-1,1]$, *using* 9 *and* 19 *equally spaced* x*-points, and also Chebyshev* x*-points (the corresponding* y*-points obtained by plugging the* x*'s into* $f(x)$*) to create a polynomial interpolant.*



As the number of data points n *increases, the polynomial interpolant* $p(x)$ *based on equispaced points goes away from* $f(x)$ *near the ends of the interval, whereas the interpolant based on Chebyshev points converges to* $f(x)$*. In addition, if the* y*-data* $f(x_i)$ *change slightly, the Chebyshev interpolant* $p(x)$ *will also change slightly. The unstable oscillatory behavior of* $p(x)$ *based on interpolation points far from Chebyshev (e.g., equispaced or random) is avoided in a reliable manner.*

Barycentric formula. To construct the polynomial interpolation $p(x)$ and efficiently evaluate the interpolant at any point of interest on $[a,b]$, we need to discuss an equivalent expression of the Lagrange interpolating polynomial $p(x) = y_1 l_1(x) + y_2 l_2(x) + \cdots + y_n l_n(x)$. This standard original formula, though mathematically convenient for analysis, is expensive for evaluating $p(x)$ at many different values of x if n is not very small. In fact, we can see from the expression

of the Lagrange basis $l_i(x)$ that it takes $O(n^2)$ flops to evaluate $p(x)$ at a single value of x, and hence $O(mn^2)$ flops to do so for m different values of x.

To derive a more efficient formula, define $\mu_i = \frac{1}{\Pi_{j=1,j\neq i}^n (x_i - x_j)}$, such that

$$l_i(x) = \frac{\Pi_{j=1,j\neq i}^n (x - x_j)}{\Pi_{j=1,j\neq i}^n (x_i - x_j)} = \mu_i \frac{l(x)}{x - x_i}, \quad \text{where} \quad l(x) = \Pi_{j=1}^n (x - x_j).$$

Therefore

$$p(x) = l(x) \left(\frac{y_1\mu_1}{x - x_1} + \frac{y_2\mu_2}{x - x_2} + \cdots + \frac{y_n\mu_n}{x - x_n} \right), \quad x \neq x_i. \tag{7.4}$$

Recall that the polynomial interpolant of degree no higher than $n-1$ going through $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ is unique. Let $f(x) \equiv 1$, and the interpolant $p(x) = y_1 l_1(x) + y_2 l_2(x) \ldots + y_n l_n(x) = l_1(x) + l_2(x) + \ldots + l_n(x)$. Since $f(x)$ itself is a polynomial of degree zero (less than $n-1$) going through these data points, $f(x) \equiv p(x)$ by the uniqueness; that is

$$l_1(x) + l_2(x) + \ldots + l_n(x) \equiv 1$$

for any x. Dividing $l_i(x) = \mu_i \frac{l(x)}{x - x_i}$ by this identity, with some work, we have $l_i(x) = \frac{\mu_i}{x - x_i} \Big/ \left(\frac{\mu_1}{x - x_1} + \frac{\mu_2}{x - x_2} + \ldots + \frac{\mu_n}{x - x_n} \right)$, and it follows that

$$p(x) = \frac{\frac{y_1\mu_1}{x - x_1} + \frac{y_2\mu_2}{x - x_2} + \ldots + \frac{y_n\mu_n}{x - x_n}}{\frac{\mu_1}{x - x_1} + \frac{\mu_2}{x - x_2} + \ldots + \frac{\mu_n}{x - x_n}}, \quad x \neq x_i. \tag{7.5}$$

The new formulas (7.4) and (7.5) are called the *modified Lagrange interpolating polynomial formula* and the *barycentric formula*, respectively. The virtue of (7.5) is that the quantities μ_i appear on the numerator and denominator, and therefore we can multiply or divide them by any nonzero number (called 'scaling') without changing the formula, whereas this is not possible for (7.4).

Using Chebyshev x-points $x_i = -\frac{b-a}{2} \cos \frac{(i-1)\pi}{n-1} + \frac{b+a}{2}$ for polynomial interpolation, one can show that after proper scaling,

$$\mu_1 = \frac{1}{2}, \quad \mu_i = (-1)^{i-1} \ (2 \leq i \leq n-1), \quad \mu_n = \frac{1}{2}(-1)^{n-1}. \tag{7.6}$$

It is simple to memorize these quantities: they are just the alternating sequence $1, -1, 1, -1, \ldots$, but the first and the last one need to be halved.

We recommend single polynomial interpolation to be performed by choosing the Chebyshev x-points (7.3), and evaluated by the barycentric formula (7.5) with μ_i defined in (7.6). Such a formula has two major strengths: 1) $p(x)$ is guaranteed to converge to $f(x)$ for any x on $[a, b]$, as long as $f(x)$ is differentiable

on this interval, and the error $\max_{a\le x\le b}|f(x)-p(x)|$ goes to zero exponentially with n if $f(x)$ is can be differentiated infinitely many times; 2) evaluation of $p(x)$ for a single value of x takes only $O(n)$ flops (instead of $O(n^2)$ flops needed by the original Lagrange interpolation formula), which is essentially optimal.

A MATLAB code for the Chebyshev interpolation using the barycentric formula is given as follows:

```
function pxeval = chebinterp(func,a,b,n,xeval)
% for a function f(x) differentiable on [a,b], we construct the
% polynomial interpolant p(x) based on n Chebyshev points, and
% evaluate p(x) at each element of 'xeval'

% Chebyshev x-points (x1,x2,...,xn)
xs = -(b-a)/2*cos((0:(n-1))/(n-1)*pi)+(a+b)/2;
% corresponding y-points (y1,y2,...,yn) = (f(x1),f(x2),...,f(xn))
ys = func(xs);
% mu = [1/2 -1 1 -1 ... 1/2*(-1)^(n-1)]
mus = ones(1,n);    mus(2:2:end) = -1;
mus([1 end]) = mus([1 end])/2;

pxeval = zeros(size(xeval));
% use the barycentric formula to evaluate p(x) at the 'xeval'
% $x$-points; for simplicity, assume that all elements of 'xeval'
% are not equal to any elements of the Chebyshev $x$-points
for k = 1 : length(xeval)
    denom = mus./(xeval(k)-xs);
    numer = ys.*denom;
    pxeval(k) = sum(numer)/sum(denom);
end
```

We illustrate the accuracy and efficiency of the Chebyshev interpolation on an example similar to the previous one. We approximate $f(x)=\frac{1}{1+2500x^2}$ on $[-1,1]$ by the interpolant $p(x)$ of degree 499, 999 and 1999, respectively, and evaluate $p(x)$ at $100,000$ uniformly distributed random x-values on $[-1,1]$.

```
func = @(x)1./(1+2500*x.^2);
a = -1;  b = 1;
% uniformly distributed evaluation points on [a,b]
xeval = rand(100000,1)*(b-a)+a;
ts1 = tic; px1 = chebinterp(func,a,b,500,xeval); te1 = toc(ts1);
ts2 = tic; px2 = chebinterp(func,a,b,1000,xeval); te2 = toc(ts2);
ts3 = tic; px3 = chebinterp(func,a,b,2000,xeval); te3 = toc(ts3);
err1 = norm(px1-func(xeval),'inf');
err2 = norm(px2-func(xeval),'inf');
err3 = norm(px3-func(xeval),'inf');
fprintf('inf-norm of |f(x)-p(x)|: %.3e, %.3e, %.3e.\n', ...
          err1,err2,err3);
```

```
fprintf('Timing: %.3f, %.3f, %.3f secs.\n',te1,te2,te3);

inf-norm of |f(x)-p(x)|: 9.267e-05, 4.210e-09, 7.550e-15.

Timing: 0.275, 0.457, 0.853 secs.
```

We see that $f(x) = \frac{1}{1+2500x^2}$ on $[-1, 1]$ can be approximated by the Chebyshev interpolant of degree 1999 to near machine epsilon. Also, the time needed to evaluate $p(x)$ grows roughly linearly with n, the number of data points.

7.3 Piecewise interpolation

In some applications, it is inconvenient or even impossible to choose the x-points of data at Chebyshev points to construct polynomial interpolation. For example, we measure the temperature at a place several times during a 24-hour period and want to interpolate the measured temperatures to get a smooth temperature function for any time during this period. It would be much more convenient to sample such a time-dependent function at equally spaced time intervals, but as we have seen in the previous section, a single polynomial $p(x)$ based on equispaced data points would usually a bad approximation to the true function. In this case, piecewise interpolation is a natural solution.

The first step to construct such an interpolation is to divide the entire interval $[a, b]$ into several intervals, $[x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n]$ (where $x_1 = a$ and $x_n = b$ as usual). Piecewise linear interpolation is defined, on interval i, to be

$$p_i(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i).$$

The interpolating polynomial $p(x)$ is defined so that $p(x) = p_i(x)$ on interval i.

This may seem simplistic, but often it is reasonably accurate. It avoids oscillations, and even satisfies the following error estimate.

Theorem 52. *Suppose we choose n data points, (x_i, y_i), $i = 1, ..., n$, with max spacing h in the x-points, from a function $f \in C^2([x_1, x_n])$ (i.e. $y_i = f(x_i)$ at the data points). Then the difference between the function and the piecewise linear interpolant satisfies*

$$\max_{x_1 \le x \le x_n} |f(x) - p(x)| \le h^2 \max_{x_1 \le x \le x_n} |f''(x)|.$$

Proof. Consider the maximum error on an arbitrary interval i,

$$\begin{aligned}
&\max_{x_i\le x\le x_{i+1}} |f(x)-p_i(x)| = \max_{x_i\le x\le x_{i+1}} \left| f(x) - \left(y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(x-x_i)\right)\right| \\
= &\max_{x_i\le x\le x_{i+1}} \left| f(x) - \left(f(x_i) + \frac{f(x_{i+1})-f(x_i)}{h}(x-x_i)\right)\right|.
\end{aligned}$$

From Taylor's theorem, we can write

$$f(x) = f(x_i) + f'(x_i)(x-x_i) + \frac{f''(c(x))}{2}(x-x_i)^2,$$

where c depends on x and is between x_i and x. Inserting this into the error definition, we have

$$\begin{aligned}
\max_{x_i\le x\le x_{i+1}} |f(x)-p_i(x)| \quad = \quad &\max_{x_i\le x\le x_{i+1}} \Bigg| f'(x_i)(x-x_i) \\
&+\frac{f''(c(x))}{2}(x-x_i)^2 - \frac{f(x_{i+1})-f(x_i)}{h}(x-x_i)\Bigg|.
\end{aligned} \tag{7.7}$$

Again using Taylor series, we have that

$$\begin{aligned}
f(x_{i+1}) &= f(x_i) + f'(x_i)(x_{i+1}-x_i) + \frac{f''(c_0)}{2}(x_{i+1}-x_i)^2 \\
&= f(x_i) + f'(x_i)h + \frac{f''(c_0)}{2}h^2,
\end{aligned}$$

where c_0 is between x_i and x_{i+1}. Some arithmetic on this equation reveals that

$$\frac{f(x_{i+1})-f(x_i)}{h} = f'(x_i) + \frac{f''(c_0)}{2}h,$$

and now using this identity to simplify the error equation (7.7) further, we obtain

$$\begin{aligned}
&\max_{x_i\le x\le x_{i+1}} |f(x)-p_i(x)| \\
= &\max_{x_i\le x\le x_{i+1}} \left| f'(x_i)(x-x_i) + \frac{f''(c(x))}{2}(x-x_i)^2 - \frac{f(x_{i+1})-f(x_i)}{h}(x-x_i)\right| \\
= &\max_{x_i\le x\le x_{i+1}} \left| f'(x_i)(x-x_i) + \frac{f''(c(x))}{2}(x-x_i)^2 - \left(f'(x_i) + \frac{f''(c_0)}{2}h\right)(x-x_i)\right| \\
= &\max_{x_i\le x\le x_{i+1}} \left| \frac{f''(c(x))}{2}(x-x_i)^2 - \left(\frac{f''(c_0)}{2}h\right)(x-x_i)\right|
\end{aligned}$$

Since x is in interval i, we have that both c and c_0 are also in the interval, and thus that $|x-x_i|\le h$. Thus we have the bound

$$\begin{aligned}
\max_{x_i\le x\le x_{i+1}} |f(x)-p_i(x)| &\le h^2 \left| \frac{f''(c(x))}{2} - \frac{f''(c_0)}{2}\right| \\
&\le h^2 \max_{x_i\le x\le x_{i+1}} |f''(x)| \\
&\le h^2 \max_{x_1\le x\le x_n} |f''(x)|.
\end{aligned}$$

Since this bound holds on an arbitrary interval i, it holds for every interval, and thus the theorem is proven. □

Example 53. *We now test the error rate in linear interpolation. Using the function $f(x) = e^x$ on $[-2, 2]$, we create linear interpolants of $f(x)$ using different numbers of points, and then calculate the maximum error in the linear interpolant. The MATLAB code to do this is as follows:*

```
format shorte
f = @(x) exp(x);

% true solution at 1000 points
x = linspace(-2,2,1000);
y = f(x);

for m = 1 : 8
    n = 10*2^m + 1;
    xpts = linspace(-2,2,n);
    ypts = f(xpts);
    yLI = interp1(xpts,ypts,x,`linear');
    maxerr(m)= max(abs(yLI - y));
end
maxerr'
ratios = log( maxerr(1:end-1)./maxerr(2:end) ) / log(2);
ratios'
```

which produces the output for error of

```
3.3456e-02
8.7800e-03
2.2495e-03
5.6933e-04
1.3782e-04
3.5046e-05
8.8090e-06
2.2109e-06
```

and ratios of errors of

```
1.9300e+00
1.9646e+00
1.9823e+00
2.0465e+00
1.9755e+00
1.9922e+00
1.9943e+00
```

The ratios are converging to 2, which is consistent with the predicted $O(h^2)$ error.

7.4 Piecewise cubic interpolation (cubic spline)

Often, a better choice of piecewise interpolant is a cubic spline interpolant. This type of interpolant will be a cubic function on each interval and will satisfy

1. On interval i,
$$p_i(x) = c_0^{(i)} + c_1^{(i)}x + c_2^{(i)}x^2 + c_3^{(i)}x^3$$
2. $p \in C^2([x_1, x_n])$ (i.e. it is smooth, not choppy like the piecewise linear interpolant)

Determining the $4(n-1)$ unknowns is done by enforcing that the polynomial p interpolates the data ($2n-2$ equations), enforcing $p \in C^2$ at nodes x_2 through x_{n-1} ($2n-4$ equations), and enforcing the 'not-a-knot' condition (2 equations). This defines a linear system of order $4(n-1)$ to determine the coefficient $c_j^{(i)}$'s.

If the data points come from a smooth function, then cubic splines can be much more accurate than piecewise linear interpolants.

Theorem 54. *Suppose we choose n data points, (x_i, y_i), $i = 1, ..., n$, with max point spacing h in the x-points, from a function $f \in C^4([x_1, x_n])$. Then the difference between the function and the cubic spline $p(x)$ satisfies*

$$\max_{x_1 \le x \le x_n} |f(x) - p(x)| \le Ch^4 \max_{x_1 \le x \le x_n} |f^{(4)}(x)|$$

MATLAB has a built-in function called `spline` to do this. We simply give it the data points, and MATLAB takes care of the rest as follows.

Example 55. *For the data points $(i, \sin(i))$ for $i = 0, 1, 2, ..., 6$, find a cubic spline. Then plot the spline along with the data points and the function $\sin(x)$.*

First, we create the data points and the spline.

```
>> x = [0:6]
x =
     0     1     2     3     4     5     6
>> y = sin(x);
>> cs = spline(x,y);
```

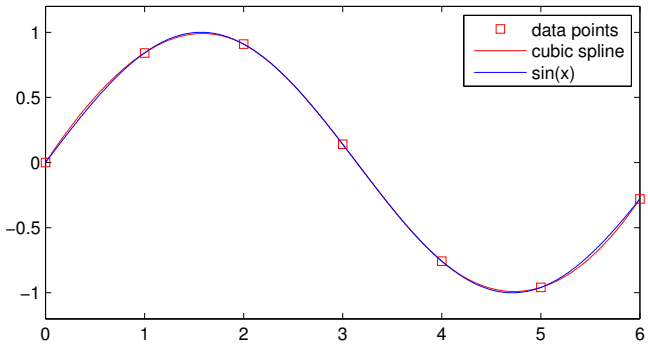
Next, we can evaluate points in the spline using MATLAB's `ppval`*. So for example, we can plug in $x = \pi$ via*

```
>> ppval(cs,pi)
ans =
   -1.3146e-04
```

Thus to plot the cubic spline, we just need to give it x-points to plot y values at. We do this below, and include in our plot the $\sin(x)$ curve as well.

```
>> xx = linspace(0,6,1000);
>> yy = ppval(cs,xx);
>> plot(x,y,'rs',xx,yy,'r',xx,sin(xx),'b')
>> legend('data points','cubic spline','sin(x)')
```

This produces the following plot:



From the plot, we see that with just 7 points, we are able to match the $\sin(x)$ *curve on a large interval quite well!*

Example 56. *For* $f(x) = \sin(x)$ *on* $[-1, 1]$*, calculate errors and rates for cubic spline and linear interpolants for varying numbers of points in MATLAB.*

```
% Create the function f(x) = sin(x)
format shorte;
f = @(x) sin(x);

% true solution at 1000 points
x = linspace(-1,1,1000);
y = f(x);

% cubic spline
for m = 1 : 8
    n = 10*2^m + 1;
    xpts = linspace(-2,2,n);
    ypts = f(xpts);
    yLI = interp1(xpts,ypts,x,`linear');
    CS = spline(xpts,ypts);
    yCS = ppval(CS,x);

    maxerrCS(m)= max( abs( yCS - y));
    maxerrLI(m)= max( abs( yLI - y));
end

ratiosCS = log( maxerrCS(1:end-1)./maxerrCS(2:end) ) / log(2);
ratiosLI = log( maxerrLI(1:end-1)./maxerrLI(2:end) ) / log(2);
```

```
[maxerrCS', [0;ratiosCS'], maxerrLI', [0;ratiosLI']]
```

which produces the output of [CS-error, CS-rates, LI-error, LI-rates]*, which correspond to the errors and order of cubic splines, and the errors and order of piecewise linear interpolants, respectively:*

```
3.3589e-06            0   3.9139e-03            0
2.1236e-07   3.9834e+00   1.0165e-03   1.9449e+00
1.3443e-08   3.9816e+00   2.5831e-04   1.9765e+00
8.4636e-10   3.9894e+00   6.5114e-05   1.9880e+00
5.3116e-11   3.9941e+00   1.6345e-05   1.9942e+00
3.2817e-12   4.0166e+00   4.0410e-06   2.0160e+00
2.0672e-13   3.9887e+00   1.0168e-06   1.9906e+00
1.2768e-14   4.0171e+00   2.5278e-07   2.0081e+00
```

We observe the expected convergence order of 4 *for cubic spline and* 2 *for piecewise linear interpolation. We also observe that the cubic spline is much more accurate.*

Consider now a less smooth function $f(x) = \sqrt{|\sin(10x)|}$*. This function is not even differentiable at some points in the interval, and its second derivative (where it exists) has vertical asymptotes. Repeating the same code above with this function produces the output*

```
9.7436e-01            0   8.1422e-01            0
5.3906e-01   8.5402e-01   6.0886e-01   4.1929e-01
3.5088e-01   6.1946e-01   4.0968e-01   5.7163e-01
2.1509e-01   7.0603e-01   2.5880e-01   6.6266e-01
1.3780e-01   6.4240e-01   1.6801e-01   6.2328e-01
8.8288e-02   6.4226e-01   9.4428e-02   8.3126e-01
6.9666e-02   3.4176e-01   5.3083e-02   8.3098e-01
3.0434e-02   1.1948e+00   1.9971e-02   1.4104e+00
```

Here, the approximations do not achieve their optimal convergence rates of 4 *and* 2*, but this is expected since the derivatives required in the theorems do not exist. Interestingly, the linear interpolant is more accurate here than the cubic spline.*

Chebyshev or piecewise interpolation? To interpolate an analytic function $f(x)$ and approximate it accurately by a polynomial $p(x)$ on a finite interval $[a, b]$, piecewise interpolants, including cubic splines, should be used only if it is inconvenient or impossible to choose the interpolation points at the Chebyshev points. If we can freely choose the interpolation points $x_1, x_2, \ldots, x_n$ and efficiently evaluate $f(x)$ at these points, Chebyshev interpolant is preferred because it is asymptotically a more accurate approximation to $f(x)$ based on the same number of interpolation points n. The error $\max_{x\in[a,b]} |f(x) - p(x)|$ is $\mathcal{O}(\rho^{-n})$ (some fixed $\rho > 1$) for Chebyshev interpolant, and it is $\mathcal{O}(n^{-p})$ (some fixed $p \geq 2$) for a piecewise interpolant (see an example in exercise problem 8).

7.5 Exercises

1. Consider the data points

```
x = [1    2    3    4    5    6    7    8    9    10]
y = [1.1  4.1  8.9  16.1 26   36   52   63   78   105]
```

   a) Find a single polynomial that interpolates the data points
   b) Check with a plot that your polynomial does indeed interpolate the data
   c) Would this interpolating polynomial likely give a good prediction of what is happening at $x = 1.5$? Why or why not?
   d) Create a cubic spline for the same data, and plot it on the same graph as the single polynomial interpolant. Does it do a better job fitting the data?

2. Find both the monomial and Lagrange interpolating polynomials for the data points $(0, 0)$, $(1, 1)$, $(2, 3)$, $(4, -1)$.

3. If an x value gets repeated in the data points, why is it a problem in both of the interpolating polynomial constructions we considered? Furthermore, should it create a problem (i.e. is it even possible to get an answer)?

4. Compare the classic Lagrange interpolation formula (7.2) and the barycentric formula (7.5) (with all μ_i's known). Verify that the former needs $O(n^2)$ flops to evaluate $p(x)$ for a single x, whereas the latter needs $O(n)$ flops.

5. Consider the Chebyshev points $x_i = -\cos\frac{(i-1)\pi}{6}$ $(1 \le i \le 7)$ and the quantities $\mu_i = \frac{1}{\Pi_{j=1,j\neq i}^{7}(x_i - x_j)}$. Verify by hand that $\mu_1 = \frac{8}{3}$, $\mu_2 = -\frac{16}{3}$, $\mu_3 = \frac{16}{3}$, and $\mu_4 = -\frac{16}{3}$. Use symmetry to find μ_5, μ_6 and μ_7 directly without evaluation. After scaling, are they consistent with the sequence $\frac{1}{2}, -1, 1, -1, \ldots, \frac{1}{2}(-1)^{n-1}$ defined in (7.6)?

6. Consider $f(x) = \ln x$ on $[1, 5]$. Give the expression of the barycentric formula (7.5) for the polynomial interpolation $p(x)$ based on $n = 4$ Chebyshev points. Then evaluate $p(x)$ at $x = 1.5$ and $x = 2.5$ and compare with $f(x)$.

7. Use the MATLAB code given to construct Chebyshev interpolants $p(x)$ for $f(x) = e^{-5x}\sin\frac{1}{x}\sin\frac{1}{\sin\frac{1}{x}}$ on $[0.1596, 0.3175]$ using $n = 200, 500, 800, 1100$ Chebyshev points, and evaluate $p(x)$ at $100,000$ uniformly distributed random evaluation x-points on this interval. How does $\max_{0.1596\le x\le 0.3175}|f(x) - p(x)|$ change with n? Also, does the timing seem roughly linear with n?

8. Consider the function $f(x) = \sin^2(x)$ on $[0, \pi]$.
   (a) Verify the convergence theorems for piecewise linear and cubic spline interpolation. Using 11, 21, 41, and 81 equally spaced points, calculate the errors in the approximations (error as defined in the theorems) and then

calculate convergence rates.

(b) By trial and error, find out how many equally spaced points are needed, so that the cubic spline $p(x)$ approximates $f(x)$ to near machine epsilon? You can use $100,000$ uniformly distributed random evaluation x-points on $[0, \pi]$ to evaluate $\max_{0 \le x \le \pi} |f(x) - p(x)|$.

(c) Use Chebyshev interpolation to approximate $f(x)$. Do the same test as in part (b) and see how many data points are needed to reach near machine epsilon accuracy. Do you prefer Chebyshev interpolation or cubic splines?