Chapter 3

Piecewise Polynomial Interpolation

An important lesson from Chapter 2 is that high-degree polynomial interpolants at equally-spaced points should be avoided. This can pose a problem if we are to produce an accurate interpolant across a wide interval $[\alpha, \beta]$. One way around this difficulty is to partition $[\alpha, \beta]$,

$$\alpha = x_1 < x_2 < \cdots < x_n = \beta$$

and then interpolate the given function on each subinterval $[x_i, x_{i+1}]$ with a polynomial of low degree. This is the *piecewise polynomial* interpolation idea. The x_i are called *breakpoints*.

We begin with piecewise linear interpolation working with both fixed and adaptively determined breakpoints. The latter requires a classical divide-and-conquer approach that we shall use again in later chapters.

Piecewise linear functions do not have a continuous first derivative, and this creates problems in certain applications. Piecewise cubic Hermite interpolants address this issue. In this setting, the value of the interpolant and its derivative is specified at each breakpoint. The local cubics join in a way that forces first derivative continuity.

Second derivative continuity can be achieved by carefully choosing the first derivative values at the breakpoints. This leads to the topic of splines, a very important idea in the area of approximation and interpolation. It turns out that cubic splines produce the smoothest solution to the interpolation problem.

3.1 Piecewise Linear Interpolation

Assume that $x(1{:}n)$ and $y(1{:}n)$ are given where $\alpha = x_1 < \cdots < x_n = \beta$ and $y_i = f(x_i)$, $i = 1{:}n$. If you connect the dots $(x_1, y_1), \ldots, (x_n, y_n)$ with straight lines, as in Figure 3.1, then the graph of a *piecewise linear* function is displayed. We already have considerable experience with such functions, for this is what `plot(x,y)` displays.

3.1.1 Set-Up

The piecewise linear interpolant is built upon the local linear interpolants

$$L_i(z) = a_i + b_i(z - x_i),$$

where for $i = 1{:}n-1$ the coefficients are defined by

$$a_i = y_i \quad \text{and} \quad b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

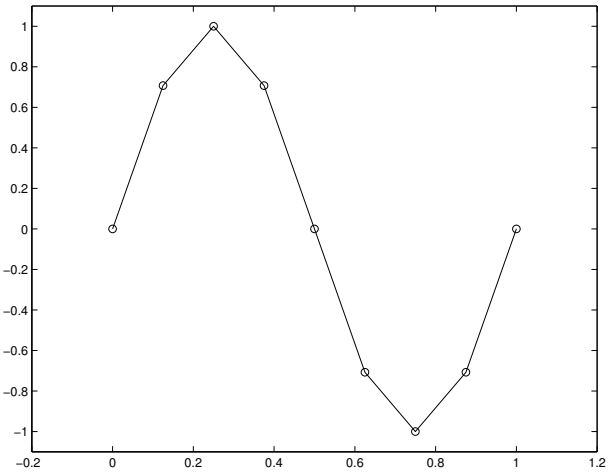

FIGURE 3.1 *A piecewise linear function*

Note that $L_i(z)$ is just the linear interpolant of f at the points $x = x_i$ and $x = x_{i+1}$. We then define

$$L(z) = \begin{cases} L_1(z) & \text{if} \quad x_1 \le z < x_2 \\ L_2(z) & \text{if} \quad x_2 \le z < x_3 \\ \vdots & \qquad\qquad \vdots \\ L_{n-1}(z) & \text{if} \quad x_{n-1} \le z \le x_n \end{cases} .$$

The act of setting up L is the act of solving each of the local linear interpolation problems. The $n-1$ divided differences $b_1, \ldots, b_{n-1}$ can obviously be computed by a loop,

```
for i=1:n-1
   b(i) = (y(i+1)-y(i))/(x(i+1)-x(i));
end
```

or by using pointwise division,

```
b = (y(2:n)-y(1:n-1)) ./ (x(2:n)-x(1:n-1))
```

or by using the built-in function `diff`:

```
b = diff(y) ./ diff(x)
```

Packaging these operations we obtain

```
  function [a,b] = pwL(x,y)
% Generates the piecewise linear interpolant of the data specified by the
% column n-vectors x and y. It is assumed that x(1) < x(2) < ... < x(n).
%
% a and b are column (n-1)-vectors with the property that for i=1:n-1, the
% line L(z) = a(i) + b(i)z passes though the points (x(i),y(i)) and (x(i+1),y(i+1)).

n = length(x);
a = y(1:n-1);
b = diff(y) ./ diff(x);
```

Thus,

```
z = linspace(0,1,9);
[a,b] = pwL(z,sin(2*pi*z));
```

sets up a piecewise linear interpolant of $\sin(2\pi z)$ on a uniform, nine-point partition of $[0, 1]$.

3.1.2 Evaluation

To evaluate L at a point $z \in [\alpha, \beta]$, it is necessary to determine the subinterval that contains z. In our problem `x` has the property that $x_1 < \cdots < x_n$ and so `sum(x<=z)` is the number of x_i that are to the left of z or equal to z. It follows that

```
if z == x(n);
   i = n-1;
else
   i = sum(x<=z);
end
```

determines the index i so that $x_i \leq z \leq x_{i+1}$. Notice the special handling of the case when z equals x_n. (Why?) A total of n comparisons are made because every component in `x` is compared to `z`.

A better approach is to exploit the monotonicity of the x_i and to use binary search. Here is the main idea. Suppose we have indices *Left* and *Right* so that $x_{Left} \leq z \leq x_{Right}$. If $mid = \text{floor}((Left + Right)/2)$, then by checking z's relation to x_{mid} we can halve the search space by redefining $Left$ or *Right* accordingly:

```
mid = floor((Left+Right)/2);
if z < x(mid)
   Right = mid;
else
   Left = mid;
end
```

Repeated application of this process eventually identifies the subinterval that houses z:

```
if z == x(n)
   i = n-1;
else
   Left = 1; Right = n;
   while Right > Left+1
      % z is in [x(Left),x(Right)].
      mid = floor((Left+Right)/2);
      if z < x(mid)
         Right = mid;
      else
         Left = mid;
      end
   end
   i = Left;
end
```

Upon completion, `i` contains the index of the subinterval that contains z. If $n = 10$ and $z \in [x_6, x_7]$, then here is the succession of `Left` and `Right` values produced by the binary search method:

Left	Right	mid
1	10	5
5	10	7
5	7	6
6	7	-

Roughly $\log_2(n)$ comparisons are required to locate the appropriate subinterval. If n is large, then this is much more efficient than the `sum(x<z)` method, which requires n comparisons.

For "random" z, we can do no better than binary search. However, if L is to be evaluated at an ordered succession of points, then we can improve the subinterval location process. For example, suppose we want to plot L on $[\alpha, \beta]$. This requires the assembly of the values $L(z_1), \ldots, L(z_m)$ in a vector where m is a typically large integer and $\alpha \leq z_1 \leq \cdots \leq z_m \leq \beta$. Rather than locate each z_i via binary search, it is more efficient to

exploit the systematic "migration" of the evaluation point as it moves left to right across the subintervals. Chances are that if i is the subinterval index associated with the current z-value, then i will be the correct index for the next z-value. This "guess" at the correct subinterval can be checked before we launch the binary search process.

```
  function i = locate(x,z,g)
% Locates z in a partition x.
% x is column n-vector with x(1) < x(2) <...<x(n) and
% z is a scalar with x(1) <= z <= x(n).
% g (1<=g<=n-1) is an optional input parameter
% i is an integer such that x(i) <= z <= x(i+1). Before the general
% search for i begins, the value i=g is tried.
if nargin==3
   % Try the initial guess.
   if (x(g)<=z) & (z<=x(g+1))
      i = g;
      return
   end
end
n = length(x);
if z==x(n)
   i = n-1;
else
   % Binary Search
   Left = 1; Right = n;
   while Right > Left+1
      % x(Left) <= z <= x(Right)
      mid = floor((Left+Right)/2);
      if z < x(mid)
         Right = mid;
      else
         Left = mid;
      end
   end
   i = Left;
end
```

This function makes use of the `return` command. This terminates the execution of the function. It is possible to restructure `locate` to avoid the `return`, but the resulting logic would be cumbersome. As an application of `locate`, here is a function that produces a vector of L-values:

```
  function LVals = pwLeval(a,b,x,zVals)
% Evaluates the piecewise linear polynomial defined by the column (n-1)-vectors
% a and b and the column n-vector x. It is assumed that x(1) < ... < x(n).
% zVals  is a column m-vector with each component in [x(1),x(n)].
% LVals is a column m-vector with the property that LVals(j) = L(zVals(j))
% for j=1:m where L(z)= a(i) + b(i)(z-x(i)) for x(i)<=z<=x(i+1).

m = length(zVals);  LVals = zeros(m,1);  g = 1;
for j=1:m
   i = locate(x,zVals(j),g);
   LVals(j) = a(i) + b(i)*(zVals(j)-x(i));
   g = i;
end
```

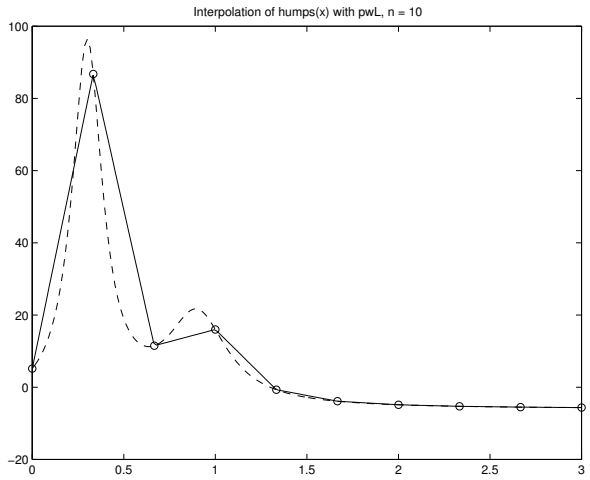

FIGURE 3.2 *Piecewise linear approximation*

The following script illustrates the use of this function, producing a sequence of piecewise linear approximations to the built-in function

$$\text{humps}(x) = \frac{1}{(x-.3)^2+.01} + \frac{1}{(x-.9)^2+.04} - 6.$$

```
% Script File: ShowPWL1
% Convergence of the piecewise linear interpolant to
% humps(x) on [0,3]
close all
z = linspace(0,3,200)';
fvals = humps(z);
for n = [5 10 25 50]
   figure
   x = linspace(0,3,n)';
   y = humps(x);
   [a,b] = pwL(x,y);
   Lvals = pwLEval(a,b,x,z);
   plot(z,Lvals,z,fvals,'--',x,y,'o');
   title(sprintf('Interpolation of humps(x) with pwL, n = %2.0f',n))
end
```

(See Figure 3.2 for the 10-point case.) Observe that more interpolation points are required in regions where `humps` is particularly nonlinear.

3.1.3 A Priori Determination of the Breakpoints

Let us consider how many breakpoints we need to obtain a satisfactory piecewise linear interpolant. If $z \in [x_i, x_{i+1}]$, then from Theorem 2,

$$f(z) = L(z) + \frac{f^{(2)}(\eta)}{2}(z - x_i)(z - x_{i+1}),$$

where $\eta \in [x_i, x_{i+1}]$. If the second derivative of f on $[\alpha, \beta]$ is bounded by M_2 and if $\bar{h}$ is the length of the longest subinterval in the partition, then it is not hard to show that

$$|f(z) - L(z)| \ \leq \ \frac{M_2 \bar{h}^2}{8}$$

for all $z \in [\alpha, \beta]$.

A typical situation where this error bound can be put to good use is in the design of the underlying partition upon which L is based. Assume that $L(x)$ is based on the uniform partition

$$\alpha = x_1 < x_2 < \cdots < x_n = \beta,$$

where

$$x_i = \alpha + \frac{i-1}{n-1}(\beta - \alpha).$$

To ensure that the error between L and f is less than or equal to a given positive tolerance δ, we insist that

$$|f(z) - L(z)| \;\leq\; \frac{M_2 \bar{h}^2}{8} \;=\; \frac{M_2}{8}\left(\frac{\beta-\alpha}{n-1}\right)^2 \;\leq\; \delta.$$

From this we conclude that n must satisfy

$$n \;\geq\; 1 + (\beta - \alpha)\sqrt{M_2/8\delta}.$$

For the sake of efficiency, it makes sense to let n be the smallest integer that satisfies this inequality:

```
  function [x,y] = pwLstatic(f,M2,alpha,beta,delta)
% Generates interpolation points for a piecewise linear approximation of
% prescribed accuracy.
%
% f is a handle that references a function f(x).
% Assume that f can take vector arguments.
% M2 is an upper bound for|f"(x)| on [alpha,beta].
% alpha and beta are scalars with alpha<beta.
% delta is a positive scalar.
%
% x and y  column n-vectors with the property that y(i) = f(x(i)), i=1:n.
% The piecewise linear interpolant L(x) of this data satisfies
% |L(z) - f(z)| <= delta for x(1) <= z <= x(n).

n = max(2,ceil(1+(beta-alpha)*sqrt(M2/(8*delta))));
x = linspace(alpha,beta,n)';
y = f(x);
```

The partition produced by `pwLstatic` *does not* take into account the sampled values of f. As a result, the uniform partition produced may be much too refined in regions where f'' is much smaller than the bound M_2.

3.1.4 Adaptive Piecewise Linear Interpolation

Suppose f is very nonlinear over just a small portion of $[\alpha, \beta]$ and very smooth elsewhere. (See Figure 3.2.) This means that if we use `pwLstatic` to generate the partition, then we are compelled to use a large `M2`. Lots of subintervals and (perhaps costly) f-evaluations will be required. Over regions where f is smooth, the partition will be overly refined.

To address this problem, we develop a recursive partitioning algorithm that "discovers" where f is "extra nonlinear" and that clusters the breakpoints accordingly. A definition simplifies the discussion. We say that the subinterval $[xL, xR]$ is *acceptable* if

$$\left| f\left(\frac{xL + xR}{2}\right) \;-\; \frac{f(xL) + f(xR)}{2} \right| \;\leq\; \delta$$

or if

$$xR - xL \leq h_{min},$$

where $\delta > 0$ and $h_{min} > 0$ are (typically small) refinement parameters. The first condition measures the discrepancy between the line that connects $(xL, f(xL))$ and $(xR, f(xR))$ and the function $f(x)$ at the interval midpoint $m = (xL+xR)/2$. The second condition says that sufficiently short subintervals are also acceptable where "sufficiently short" means less than h_{min} in length.

One more definition is required before we can describe the complete partitioning process. A partition $x_1 < \cdots < x_n$ is *acceptable* if each subinterval is acceptable. Note that if

$$xL = x_1^{(L)} < \cdots < x_n^{(L)} = m$$

is an acceptable partition of $[xL, m]$ and if

$$m = x_1^{(R)} < \cdots < x_n^{(R)} = xR$$

is an acceptable partition of $[m, xR]$, then

$$xL = x_1^{(L)} < \cdots < x_n^{(L)} < x_2^{(R)} < \cdots < x_n^{(R)} = xR$$

is an acceptable partition of $[xL, xR]$. This sets the stage for a recursive determination of an acceptable partition:

```
  function [x,y] = pwLadapt(f,xL,fL,xR,fR,delta,hmin)
% Adaptively determines interpolation points for a piecewise linear
% approximation of a specified function.
%
% f is a handle that references a function of the form y = f(u).
% xL and xR are real scalars and fL = f(xL) and fR = f(xR).
% delta and hmin are positive real scalars that determine accuracy.
%
% x and y are column n-vectors with the property that
%              xL = x(1) < ... < x(n) = xR
% and y(i) = f(x(i)), i=1:n. Each subinterval [x(i),x(i+1)] is
% either <= hmin in length or has the property that at its midpoint m,
% |f(m) - L(m)| <= delta where L(x) is the line that connects (x(i),y(i))
% and (x(i+1),y(i+1)).

if (xR-xL) <= hmin
   % Subinterval is acceptable
   x = [xL;xR];
   y = [fL;fR];
else
   mid  = (xL+xR)/2;
   fmid = f(mid);
   if (abs(((fL+fR)/2) - fmid) <= delta )
      % Subinterval accepted.
      x = [ xL;xR];
      y = [fL;fR];
   else
      % Produce left and right partitions, then synthesize.
      [xLeft,yLeft]   = pwLAdapt(f,xL,fL,mid,fmid,delta,hmin);
      [xRight,yRight] = pwLAdapt(f,mid,fmid,xR,fR,delta,hmin);
      x = [ xLeft;xRight(2:length(xRight))];
      y = [ yLeft;yRight(2:length(yRight))];
   end
end
```

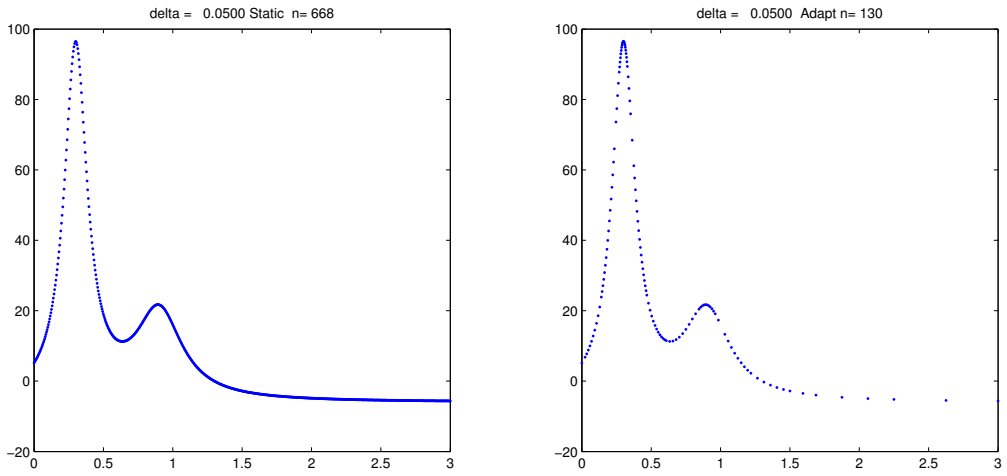

FIGURE 3.3 *Static versus adaptive approximation*

The idea behind the function is to check and see if the input interval is acceptable. If it is not, then acceptable partitions are obtained for the left and right half intervals. These are then "glued" together to obtain the final, acceptable partition.

The distinction between static and adaptive piecewise linear interpolation is revealed by running the following script:

```
% Script File: ShowpwL2
% Compares pwLstatic and pwLsdapt on [0,3] using the function
%
%   humps(x) = 1/((x-.3)^2 + .01)  +  1/((x-.9)^2+.04)
%
close all
% Second derivative estimate based on divided differences
z = linspace(0,1,101);
humpvals = humps(z);
M2 = max(abs(diff(humpvals,2)/(.01)^2));
for delta = [1 .5 .1 .05 .01]
   figure
   [x,y] = pwLstatic(@humps,M2,0,3,delta);
   subplot(1,2,1)
   plot(x,y,'.');
   title(sprintf('delta = %8.4f Static  n= %2.0f',delta,length(x)))
   [x,y] = pwLadapt(@humps,0,humps(0),3,humps(3),delta,.001);
   subplot(1,2,2)
   plot(x,y,'.');
   title(sprintf('delta = %8.4f  Adapt n= %2.0f',delta,length(x)))
   set(gcf,'position',[200 200 1200 500])
end
```

(See Figure 3.3.) The `humps` function is very nonlinear in the vicinity of $x = .3$. A second derivative bound is approximated with differences and used in `pwLstatic`. In the example approximately four times as many function evaluations are required when the static approach is taken.

Problems

P3.1.1 Generalize `locate` so that it tries $i = g + 1$ and $i = g - 1$ before resorting to binary search. (Take care to guard against subscript out-of-range.) Implement `pwLeval` with this modified subinterval locator and document the speed-up.

P3.1.2 Write a function `i = LocateUniform(alpha,beta,n,z)` that assumes $[\alpha, \beta]$ is partitioned into $n-1$ subintervals of equal length and returns the index of the interval that houses z.

P3.1.3 What happens if `pwLadapt` is applied to $\sin(x)$ with $[\alpha, \beta] = [0, 2\pi]$?

P3.1.4 Describe what would happen if `pwLadapt` is called with `delta = 0`.

P3.1.5 Describe why the number of recursive calls in `pwLadapt` is bounded if $|f''(x)|$ is bounded on $[\alpha, \beta]$.

P3.1.6 Modify `pwLadapt` so that a subinterval is accepted if $|f(p) - \lambda(p)|$ *and* $|f(q) - \lambda(q)|$ are less than or equal to `delta`, where $p = (2xL + xR)/3$, $q = (xL + 2xR)/3$, and $\lambda(x)$ is the line that connects `(xL, fL)` and `(xR, fR)`. Avoid redundant function evaluations.

P3.1.7 If `pwLadapt` is applied to the function $f(x) = \sqrt{x}$ on the interval [0,1], then a partition $x(1{:}n)$ is produced that satisfies

$$x_2 - x_1 \leq x_3 - x_2 \leq \cdots \leq x_n - x_{n-1}.$$

Why?

P3.1.8 Generalize `pwLadapt(f,xL,fL,xR,fR,delta,hmin)` to

```
function [x,y,eTaken] = pwLadapt(f,xL,fL,xR,fR,delta,hmin,eMax)
```

so that no more than `eMax` function evaluations are taken. The value of `eTaken` should be the actual number of function evaluations spent. Let $n = length(x)$. In a "successful" call, `x(n)` should equal `xR`, meaning that a satisfactory piecewise linear approximation was found extending across the entire interval $[xL, xR]$. If this is not the case, then the evaluation limit was encountered before xR was reached and `x(n)` will be less than `xR`. In this situation vectors returned define a satisfactory piecewise linear approximation across $[x(1), x(n)]$.

P3.1.9 Notice that in `pwLadapt` the vector `y` does not include all the computed function evaluations. So that these evaluations are not lost, generalize `pwLadapt` to

```
[x,y,xUnused,yUnused] = pwLadaptGen(f,xL,fL,xR,fR,delta,hmin,...)
```

where **(a)** the `x` and `y` vectors are identical to what `pwLadapt` computes and **(b)** `xUnused` and `yUnused` are column vectors that contain the x-values and function values that were computed, but not included in `x` and `y`. Thus, the `xUnused` and `yUnused` vectors should have the property that `yUnused(i) = feval(fname,xUnused(i))`, `i = 1:length(xUnused)`. You are allowed to extend the calling sequence if convenient. In that case, indicate the values that should be passed through these new parameters at the top-level call. `xUnused` and `yUnused` should be assigned the empty vector [ ] if `xR-xL<hmin`. The order of the values in `xUnused` is not important.

P3.1.10 Vectorize `locate` and `pwLeval`.

3.2 Piecewise Cubic Hermite Interpolation

Now let's graduate to piecewise cubic functions. With the increase in degree we can obtain a smoother fit to a given set of n points. The idea is to interpolate both f and its derivative with a cubic on each of the subintervals.

3.2.1 Cubic Hermite Interpolation

So far we have only considered the interpolation of function values at distinct points. In the *Hermite* interpolation problem, both the function and its derivative are interpolated. To illustrate the idea, we consider the interpolation of the function $f(z) = \cos(z)$ at the points $x_1 = 0$, $x_2 = \delta$, $x_3 = 3\pi/2 - \delta$, and $x_4 = 3\pi/2$ by a cubic $p_3(z)$. For small δ we notice that $p_3(z)$ seems to interpolate both f and f' at $z = 0$ and $z = 3\pi/2$. The interpolation shown in Figure 3.4 on the next page was obtained by running the following script:

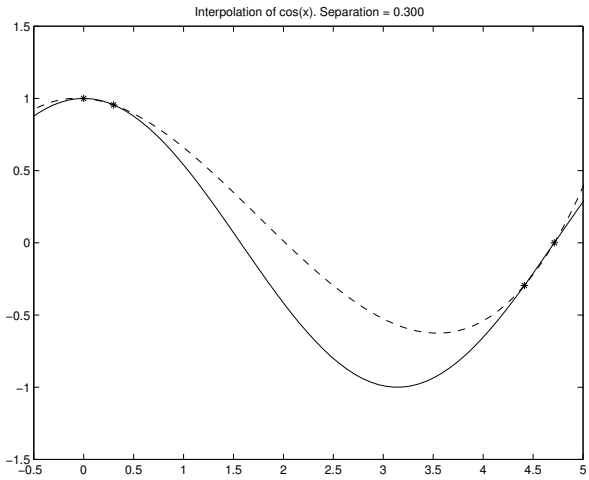

FIGURE 3.4 *A "nearly" Hermite interpolation*

```
% Script File: ShowHermite
% Plots a succession of cubic interpolants to cos(x).
% x(2) converges to x(1) = 0 and x(3) converges to x(4) = 3pi/2.
close all
z = linspace(-pi/2,2*pi,100);
CosValues = cos(z);
for d = [1 .5  .3  .1  .05 .001]
   figure
   xvals = [0;d;(3*pi/2)-d;3*pi/2];
   yvals = cos(xvals);
   c = InterpN(xvals,yvals);
   CubicValues = HornerN(c,xvals,z);
   plot(z,CosValues,z,CubicValues,'--',xvals,yvals,'*')
   axis([-.5 5 -1.5 1.5])
   title(sprintf('Interpolation of cos(x). Separation = %5.3f',d))
end
```

As the points coalesce, the cubic converges to a cubic interpolant of the cosine and its derivative at the points 0 and $3\pi/2$. This is called the *Hermite cubic interpolant.*

In the general cubic Hermite interpolation problem, we are given function values y_L and y_R and derivative values s_L and s_R and seek coefficients a, b, c, and d so that if

$$q(z) = a + b(z - x_L) + c(z - x_L)^2 + d(z - x_L)^2(z - x_R),$$

then

$$\begin{aligned} q(x_L) &= y_L & \qquad q(x_R) &= y_R \\ q'(x_L) &= s_L & \qquad q'(x_R) &= s_R. \end{aligned}$$

Each of these equations "says" something about the unknown coefficients. Noting that

$$q'(z) = b + 2c(z - x_L) + d(2(z - x_L)(z - x_R) + (z - x_L)^2),$$

we see that

$$\begin{aligned} a &= y_L & \qquad a + b\Delta x + c(\Delta x)^2 &= y_R \\ b &= s_L & \qquad b + 2c\Delta x + d(\Delta x)^2 &= s_R, \end{aligned}$$

where $\Delta x = x_R - x_L$. Expressing this in matrix-vector we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & \Delta x & (\Delta x)^2 & 0 \\ 0 & 1 & 2\Delta x & (\Delta x)^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_L \\ s_L \\ y_R \\ s_R \end{bmatrix}.$$

The solution to this triangular system is straightforward:

$$\begin{aligned} a &= y_L \\ b &= s_L \\ c &= \frac{y'_L - s_L}{\Delta x} \\ d &= \frac{s_R + s_L - 2y'_L}{(\Delta x)^2} \end{aligned}$$

where

$$y'_L = \frac{y_R - y_L}{\Delta x} = \frac{y_R - y_L}{x_R - x_L}.$$

Thus, we obtain

```
  function [a,b,c,d] = HCubic(xL,yL,sL,xR,yR,sR)
% Cubic Hermite interpolation
% (xL,yL,sL) and (xR,yR,sR) are x-y-slope triplets with xL and xR distinct.
% a,b,c,d are real numbers with the property that if
%            p(z) = a + b(z-xL) + c(z-xL)^2 + d(z-xL)^2(z-xR)
% then p(xL)=yL, p'(xL)=sL, p(xR)=yR, p'(xR)=sR.
a = yL; b = sL; delx = xR - xL;
yp = (yR - yL)/delx;
c  = (yp - sL)/delx;
d  = (sL - 2*yp + sR)/(delx*delx);
```

An error expression for the cubic Hermite interpolant can be derived from Theorem 3.

Theorem 3 *Suppose $f(z)$ and its first four derivatives are continuous on $[x_L, x_R]$ and that the constant M_4 satisfies*

$$|f^{(4)}(z)| \le M_4$$

for all $z \in [L, R]$. If q is the cubic Hermite interpolant of f at x_L and x_R, then

$$|f(z) - q(z)| \le \frac{M_4}{384} h^4,$$

where $h = x_R - x_L$.

Proof If $q_\delta(z)$ is the cubic interpolant of f at x_L, $x_L + \delta$, $x_R - \delta$, and x_R, then from Theorem 2 we have

$$|f(z) - q_\delta(z)| \le \frac{M_4}{24} |(z - x_L)(z - x_L - \delta)(z - x_R + \delta)(z - x_R)|$$

for all $z \in [x_L, x_R]$. We assume without proof[1] that

$$\lim_{\delta \to 0} q_\delta(z) = q(z)$$

and so

$$|f(z) - q(z)| \le \frac{M_4}{24} |(z - x_L)(z - x_L)(z - x_R)(z - x_R)|.$$

[1] But check out `ShowHermite`

The maximum value of the quartic polynomial on the right occurs at the midpoint $z = x_L + h/2$ and so for all z in the interval $[x_L, x_R]$ we have

$$|f(z) - q(z)| \leq \frac{M_4}{24}\left(\frac{h}{2}\right)^4 = \frac{M_4}{384}h^4. \ \square$$

Theorem 3 says that if the interval length is divided by 10, then the error bound is reduced by a factor of 10^4.

3.2.2 Representation and Set-Up

We now show how to glue a sequence of Hermite cubic interpolants together so that the resulting piecewise cubic polynomial $C(z)$ interpolates the data $(x_1, y_1), \ldots, (x_n, y_n)$, with the prescribed slopes $s_1, \ldots, s_n$. To that end we assume $x_1 < x_2 < \cdots < x_n$ and define the ith *local cubic* by

$$q_i(z) = a_i + b_i(z - x_i) + c_i(z - x_i)^2 + d_i(z - x_i)^2(z - x_{i+1}).$$

Define the piecewise cubic polynomial by

$$C(z) = \begin{cases} q_1(z) & \text{if} \quad x_1 \leq z < x_2 \\ q_2(z) & \text{if} \quad x_2 \leq z < x_3 \\ \vdots & \qquad\quad \vdots \\ q_{n-1}(z) & \text{if} \quad x_{n-1} \leq z \leq x_n \end{cases}.$$

Our goal is to determine $a(1{:}n-1)$, $b(1{:}n-1)$, $c(1{:}n-1)$, and $d(1{:}n-1)$ so that

$$\begin{matrix} C(x_i) &=& y_i \\ & & \\ C'(x_i) &=& s_i \end{matrix} \quad , \qquad i = 1{:}n$$

This will be the case if we simply solve the following $n-1$ cubic Hermite problems:

$$\begin{aligned} q_i(x_i) &= y_i \\ q_i'(x_i) &= s_i \\ q_i(x_{i+1}) &= y_{i+1} \\ q_i'(x_{i+1}) &= s_{i+1} \end{aligned}$$

The results of §3.2.1 apply:

$$a_i = y_i, \qquad b_i = s_i, \qquad c_i = \frac{y_i' - s_i}{\Delta x_i}, \qquad d_i = \frac{s_{i+1} + s_i - 2y_i'}{(\Delta x_i)^2},$$

where $\Delta x_i = x_{i+1} - x_i$ and

$$y_i' = \frac{y_{i+1} - y_i}{\Delta x_i} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

We could use `HCubic` to resolve the coefficients:

```
for i=1:n-1
   [a(i), b(i), c(i), d(i)] = HCubic(x(i),y(i),s(i),x(i+1),y(i+1),s(i+1))
end
```

But a better solution is to vectorize the computation, and this gives

```
  function [a,b,c,d] = pwC(x,y,s)
% Piecewise cubic Hermite interpolation.
%
% x,y,s  column n-vectors with x(1) < ... < x(n)
%
% a,b,c,d  column (n-1)-vectors that define a continuous, piecewise
% cubic polynomial q(z) with the property that for i = 1:n,
%
%             q(x(i)) = y(i) and q'(x(i)) = s(i).
%
% On the interval [x(i),x(i+1)],
%
%      q(z) = a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2  + d(i)(z-x(i))^2(z-x(i+1)).
n  = length(x);
a  = y(1:n-1);
b  = s(1:n-1);
Dx = diff(x);
Dy = diff(y);
yp = Dy ./ Dx;
c  = (yp - s(1:n-1)) ./ Dx;
d  = (s(2:n) + s(1:n-1) - 2*yp) ./ (Dx.* Dx);
```

If M_4 bounds $|f^{(4)}(x)|$ on the interval $[x_1, x_n]$, then Theorem 3 implies that

$$|f(z) - C(z)| \leq \frac{M_4}{384}\bar{h}^4$$

for all $z \in [x_1, x_n]$, where $\bar{h}$ is the length of the longest subinterval (i.e., $\max_i |x_{i+1} - x_i|$.)

3.2.3 Evaluation

The evaluation of $C(z)$ has two parts. As with any piecewise polynomial that must be evaluated, the position of z in the partition must be ascertained. Once that is accomplished, the relevant local cubic must be evaluated. Here is a function that can be used to evaluate C at a vector of z values:

```
  function Cvals = pwCeval(a,b,c,d,x,zVals)
% Evaluates the pwC defined by the column (n-1)-vectors a,b,c, and
% d and the column n-vector x. It is assumed that x(1) < ... < x(n).
% zVals  is a column m-vector with each component in [x(1),x(n)].
%
% CVals is a column m-vector with the property that CVals(j) = C(zVals(j))
% for j=1:m where on the interval [x(i),x(i+1)]
%
%   C(z)= a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2 + d(i)(z-x(i))^2(z-x(i+1))
m = length(zVals);
Cvals = zeros(m,1);
g=1;
for j=1:m
   i = Locate(x,zVals(j),g);
   Cvals(j) = d(i)*(zVals(j)-x(i+1)) + c(i);
   Cvals(j) = Cvals(j)*(zVals(j)-x(i)) + b(i);
   Cvals(j) = Cvals(j)*(zVals(j)-x(i)) + a(i);
   g = i;
end
```

Analogous to `pwLeval`, we use `Locate` to determine the subinterval that houses the jth evaluation point z_j. The cubic version of `HornerN` is then used to evaluate the appropriate local cubic. The following script file illustrates the use of `pwC` and `pwCeval`:

```
% Script File: ShowpwCH
% Convergence of the piecewise cubic hermite interpolant to
% exp(-2x)sin(10*pi*x) on [0,1].)
close all
z = linspace(0,1,200)';
fvals = exp(-2*z).*sin(10*pi*z);
for n = [4 8 16 24]
   x = linspace(0,1,n)';
   y = exp(-2*x).*sin(10*pi*x);
   s = 10*pi*exp(-2*x).*cos(10*pi*x)-2*y;
   [a,b,c,d] = pwC(x,y,s);
   Cvals = pwCeval(a,b,c,d,x,z);
   figure
   plot(z,fvals,z,Cvals,'--',x,y,'*');
   title(sprintf('Interpolation of exp(-2x)sin(10pi*x) with pwCH, n = %2.0f',n))
end
legend('e^{-2z}sin(10\pi z)','The pwC interpolant')
```

Sample output is displayed in Figure 3.5.

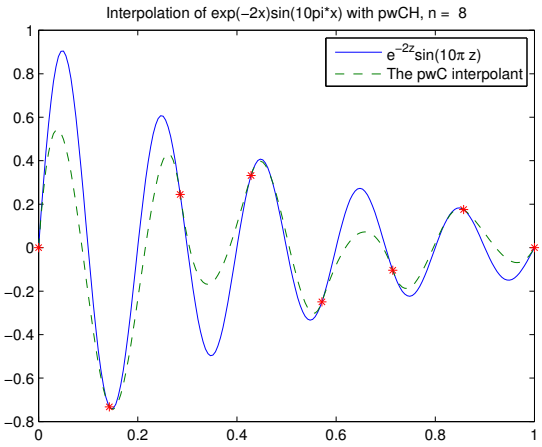

FIGURE 3.5 *Piecewise cubic Hermite interpolant of* $e^{-2x}\sin(10\pi x)$, $n = 8$

Problems

P3.2.1 Write a function `[a,b,c,d] = pwCstatic(f,fp,M4,alpha,beta,delta)` analogous to `pwLstatic`. It should produce a piecewise cubic Hermite approximation with uniform spacing. It should use the error result of Theorem 3 and the 4th derivative bound `M4` to determine the partition. The parameters `f` and `fp` should be handles that reference the function and its derivative respectively.

P3.2.2 Write a recursive function

```
function [x,y,s] = pwCAdapt(f,fp,L,fL,DfL,R,fR,DfR,delta,hmin)
```

analogous to `pwLadapt`. Use the same interval acceptance tests as in `pwLadapt`. The parameters `f` and `fp` should be handles that reference the function and its derivative respectively. Use both `pwLAdapt` and `pwCAdapt` to produce approximations to $f(x) = \sqrt{x}$ on the interval $[.001, 9]$. Fix $h_{min} = .001$. Print a table that shows the number of partition points computed by `pwLadapt` and `pwCadapt` for `delta` = .1, .01, .001, .0001, and .00001.

P3.2.3 Complete the following function:

```
function [R,fR] = stretch(L,fL,tol);
% L,fL are scalars that satisfy fL = exp(-L) and tol is a positive real.
% R,fR are scalars that satisfy fR = exp(-R) with the property that if q(z) is the cubic
% hermite interpolant of exp(-z) at z=L and z=R, then |q(z) - exp(-z)| <= tol on [L,R].
```

Make effective use of the error bound in Theorem 3 when choosing R. Hint: How big can you make R and still guarantee the required accuracy? Making effective use of `stretch` complete the following:

```
function [x,y] = pwCexp(a,b,tol)
% a,b are scalars that satisfy a < b and tol is a positive real.
% x,y are column n-vectors where a = x(1) < x(2) < ... < x(n) = b
% and y(i) = exp(-x(i)), i=1:n. The partition is chosen so that if C(z)
% is the piecewise cubic hermite interpolant of exp(-z) on this partition,
% then |C(z) - exp(-z)| <= tol for all z in [a,b]
```

P3.2.4 We want to interpolate a function f on $[a, b]$ with error less than *tol*. When is it cheaper to set up a piecewise linear interpolant $L(z)$ with a uniform partition than a piecewise cubic hermite interpolant $C(z)$ with a uniform partition? Your answer should make use of the following facts and assumptions:

- If ℓ is the linear interpolant of f on an interval $[\alpha, \beta]$, then on that interval the error is no bigger than $M_2(\beta - \alpha)^2/8$, where M_2 is an upper bound for $|f^{(2)}(z)|$. Assume that M_2 is known.
- If p is the cubic hermite interpolant of f on an interval $[\alpha, \beta]$, then on that interval the error is no bigger than $M_4(\beta - \alpha)^4/384$, where M_4 is an upper bound for $|f^{(4)}(z)|$. Assume that M_4 is known.
- A vectorized MATLAB implementation of the function f is available and it requires σn seconds to execute when applied to an n-vector. Assume that σ is known.
- A vectorized MATLAB implementation of the function f' is available and it requires τn seconds to execute when applied to an n-vector. Assume that τ is known.

P3.2.5 Consider the quartic polynomial $q(t)$ having the form

$$q(t) = a_1 + a_2 t + a_3 t^2 + a_4 t^2(t-1) + a_5 t^2(t-1)^2.$$

Given scalars v_0, s_0, v_1, s_1, v_τ, and τ, our goal is to determine the a_i so that

$$q(0) = v_0 \qquad q'(0) = s_0 \qquad q(1) = v_1 \qquad q'(1) = s_1 \qquad q(\tau) = v_\tau$$

We refer to this fourth degree Hermite interpolation problem as the "H4 problem" and to q as an "H4 interpolant." Note that its value is prescribed at three points and that at two of those points we also specify its slope. Complete the following function:

```
function A = H4(v0,s0,v1,s1,vtau,tau)
%
% Assume that the six inputs are length-n column vectors.
% A is an n-by-5 matrix with the property that if qi(t) is the quartic polynomial
%
%          qi(t) = A(i,1) + A(i,2)t + A(i,3)t^2 + A(i,4)t^2(t-1) + A(i,5)t^2(t-1)^2
%
% then qi(0) = v0(i), qi'(0) = s0(i), qi(1) = v1(i), qi'(1) = s1(i), and qi(tau(i)) = vtau(i)
% for i=1:n.
```

Your implementation should not involve any loops. Also develop a vectorized implementation for evaluation:

```
function Y = H4Eval(A,tval)
% Assume that A is an n-by-5 matrix and that tval is a length-m row vector.
% For i=1:n, let qi(t) be the quartic polynomial
%
%    qi(t) = A(i,1) + A(i,2)t + A(i,3)t^2 + A(i,4)t^2(1-t) + A(i,5)t^2(1-t)^2.
%
% Y is an n-by-m matrix with the property that Y(i,j) = qi(tval(j)) for
% i=1:n and j=1:m.
```

To test your implementations, write a script that plots in a single window the functions $f(t)$, $q_1(t)$, and $q_2(t)$ where $f(t) = e^{-t}\sin(5t)$ and q_1 and q_2 are H4 interpolants that satisfy

$$q_1(0) = f(0) \quad q_1'(0) = f'(0) \quad q_1(1) = f(1) \quad q_1'(1) = f'(1) \quad q_1(.5) = f(.5)$$

$$q_2(1) = f(1) \quad q_2'(1) = f'(1) \quad q_2(2) = f(2) \quad q_2'(2) = f'(2) \quad q_2(1.5) = f(1.5).$$

There should be just a single call to `H4` and `H4Eval`. In the same plot window, plot q_1 across [0,1] and q_2 across [1,2]. Print the coefficients of the two interpolants.

3.3 Cubic Splines

In the piecewise cubic Hermite interpolation problem, we are given n triplets

$$(x_1, y_1, s_1), \ldots, (x_n, y_n, s_n)$$

and determine a function $C(x)$ that is piecewise cubic on the partition $x_1 < \cdots < x_n$ with the property that $C(x_i) = y_i$ and $C'(x_i) = s_i$ for $i = 1{:}n$. This interpolation strategy is subject to a number of criticisms:

- The function $C(z)$ does not have a continuous second derivative: Its display may be too crude in graphical applications, because the human eye can detect discontinuities in the second derivative.
- In other applications where C and its derivatives are part of a larger mathematical landscape, there may be difficulties if $C''(x)$ is discontinuous. For example, trouble arises if C is a distance function.
- In experimental settings where the y_i are "instrument readings," we may not have the first derivative information required by the cubic Hermite process. Indeed, the underlying function f may not be known explicitly.

These reservations prompt us to pose the *cubic spline interpolation problem*:

> Given $(x_1, y_1), \ldots, (x_n, y_n)$ with $\alpha = x_1 < \cdots < x_n = \beta$, find a piecewise cubic interpolant $S(z)$ with the property that S, S', and S'' are continuous.

The function $S(z)$ that solves this problem is a *cubic spline interpolant.* This can be accomplished by *choosing* the appropriate slope values $s_1, \ldots, s_n$.

3.3.1 Continuity at the Interior Breakpoints

Assume that $S(z)$ is the cubic Hermite interpolant of the data (x_i, y_i, s_i) for $i = 1{:}n$. We ask the following question: Is it possible to choose $s_1, \ldots, s_n$ so that the second derivative of S is continuous? Let us look at what happens to S'' at each of the "interior" breakpoints $x_2, \ldots, x_{n-1}$. To the left of x_{i+1}, $S(z)$ is defined by the local cubic

$$q_i(z) = y_i + s_i(z - x_i) + \frac{y_i' - s_i}{\Delta x_i}(z - x_i)^2 + \frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2}(z - x_i)^2(z - x_{i+1}),$$

where $y_i' = (y_{i+1} - y_i)/(x_{i+1} - x_i)$ and $\Delta x_i = x_{i+1} - x_i$. The second derivative of this local cubic is given by

$$q_i''(z) \;=\; 2\frac{y_i' - s_i}{\Delta x_i} \;+\; \frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2}\left[4(z - x_i) + 2(z - x_{i+1})\right]. \tag{3.1}$$

Likewise, to the right of x_{i+1} the piecewise cubic $C(z)$ is defined by

$$q_{i+1}(z) = y_{i+1} + s_{i+1}(z - x_{i+1}) + \frac{y_{i+1}' - s_{i+1}}{\Delta x_{i+1}}(z - x_{i+1})^2 + \frac{s_{i+1} + s_{i+2} - 2y_{i+1}'}{(\Delta x_{i+1})^2}(z - x_{i+1})^2(z - x_{i+2}).$$

The second derivative of this local cubic is given by

$$q_{i+1}''(z) \;=\; 2\frac{y_{i+1}' - s_{i+1}}{\Delta x_{i+1}} \;+\; \frac{s_{i+1} + s_{i+2} - 2y_{i+1}'}{(\Delta x_{i+1})^2}\left[4(z - x_{i+1}) + 2(z - x_{i+2})\right]. \tag{3.2}$$

To force second derivative continuity at x_{i+1}, we insist that

$$q_i''(x_{i+1}) = \frac{2}{\Delta x_i}(2s_{i+1} + s_i - 3y_i')$$

and

$$q''_{i+1}(x_{i+1}) = \frac{2}{\Delta x_{i+1}}(3y'_{i+1} - 2s_{i+1} - s_{i+2})$$

be equal. That is,

$$\Delta x_{i+1}s_i \;+\; 2\left(\Delta x_i + \Delta x_{i+1}\right)s_{i+1} \;+\; \Delta x_i s_{i+2} \;=\; 3\left(\Delta x_{i+1}y'_i + \Delta x_i y'_{i+1}\right) \tag{3.3}$$

for $i = 1{:}n-2$. If we choose $s_1, \ldots, s_n$ to satisfy these equations, then $S''(z)$ is continuous.

Before we plunge into the resolution of these equations for general n, we acquire some intuition by examining the $n = 7$ case. The equations designated by (3.3) are as follows:

$$\begin{array}{rcrcl}
i=1 & \Rightarrow & \Delta x_2 s_1 + 2(\Delta x_1 + \Delta x_2)s_2 + \Delta x_1 s_3 & = & 3(\Delta x_2 y'_1 + \Delta x_1 y'_2) \\
i=2 & \Rightarrow & \Delta x_3 s_2 + 2(\Delta x_2 + \Delta x_3)s_3 + \Delta x_2 s_4 & = & 3(\Delta x_3 y'_2 + \Delta x_2 y'_3) \\
i=3 & \Rightarrow & \Delta x_4 s_3 + 2(\Delta x_3 + \Delta x_4)s_4 + \Delta x_3 s_5 & = & 3(\Delta x_4 y'_3 + \Delta x_3 y'_4) \\
i=4 & \Rightarrow & \Delta x_5 s_4 + 2(\Delta x_4 + \Delta x_5)s_5 + \Delta x_4 s_6 & = & 3(\Delta x_5 y'_4 + \Delta x_4 y'_5) \\
i=5 & \Rightarrow & \Delta x_6 s_5 + 2(\Delta x_5 + \Delta x_6)s_6 + \Delta x_5 s_7 & = & 3(\Delta x_6 y'_5 + \Delta x_5 y'_6).
\end{array}$$

Notice that we have five constraints and seven parameters and therefore two "degrees of freedom." If we move two of the parameters (s_1 and s_7) to the right hand side and assemble the results in matrix-vector form, then we obtain a 5-by-5 linear system

$$Ts(2{:}6) \;=\; T\begin{bmatrix} s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} \;=\; \begin{bmatrix} 3(\Delta x_2 y'_1 + \Delta x_1 y'_2) - \Delta x_2 s_1 \\ 3(\Delta x_3 y'_2 + \Delta x_2 y'_3) \\ 3(\Delta x_4 y'_3 + \Delta x_3 y'_4) \\ 3(\Delta x_5 y'_4 + \Delta x_4 y'_5) \\ 3(\Delta x_6 y'_5 + \Delta x_5 y'_6) - \Delta x_5 s_7 \end{bmatrix} \;=\; r,$$

where

$$T \;=\; \begin{bmatrix} 2(\Delta x_1 + \Delta x_2) & \Delta x_1 & 0 & 0 & 0 \\ \Delta x_3 & 2(\Delta x_2 + \Delta x_3) & \Delta x_2 & 0 & 0 \\ 0 & \Delta x_4 & 2(\Delta x_3 + \Delta x_4) & \Delta x_3 & 0 \\ 0 & 0 & \Delta x_5 & 2(\Delta x_4 + \Delta x_5) & \Delta x_4 \\ 0 & 0 & 0 & \Delta x_6 & 2(\Delta x_5 + \Delta x_6) \end{bmatrix}.$$

Matrices like this that are zero everywhere except on the diagonal, subdiagonal, and superdiagonal are said to be *tridiagonal*.

Different choices for the end slopes s_1 and s_n yield different cubic spline interpolants. Having defined the end slopes, the interior slopes $s(2{:}n-1)$ ar determined by solving an $(n-2)$-by-$(n-2)$ linear system. In each case that we consider here, the matrix of coefficients looks like

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & \cdots & 0 \\ \Delta x_3 & 2(\Delta x_2 + \Delta x_3) & \Delta x_2 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \Delta x_{n-2} & 2(\Delta x_{n-3} + \Delta x_{n-2}) & \Delta x_{n-3} \\ 0 & \cdots & 0 & t_{n-2,n-3} & t_{n-2,n-2} \end{bmatrix},$$

while the right-hand side r has the form

$$r = \begin{bmatrix} r_1 \\ 3(\Delta x_3 y_2' + \Delta x_2 y_3') \\ \vdots \\ 3(\Delta x_{n-2} y_{n-3}' + \Delta x_{n-3} y_{n-2}') \\ r_{n-2} \end{bmatrix}.$$

As we show in the next subsection, the values of t_{11}, t_{12}, and r_1 depend on how s_1 is chosen. The values of $t_{n-2,n-3}$, $t_{n-2,n-2}$, and r_{n-2} depend on how s_n is defined. Moreover, the T matrices that emerge can be shown to be nonsingular.

The following fragment summarizes what we have established so far about the linear system $Ts(2{:}n-1) = r$:

```
n=length(x);
Dx = diff(x);
yp = diff(y) ./ Dx;
T = zeros(n-2,n-2);
r = zeros(n-2,1);
for i=2:n-3
   T(i,i) = 2(Dx(i)+Dx(i+1));
   T(i,i-1) = Dx(i+1);
   T(i,i+1) = Dx(i);
   r(i) = 3(Dx(i+1)*yp(i) + Dx(i)*yp(i+1));
end
```

This sets up all but the first and last rows of T and all but the first and last components of r. How T and r are completed depends on the end conditions that are imposed on the spline.

3.3.2 The Complete Spline

The *complete spline* is obtained by setting $s_1 = \mu_L$ and $s_n = \mu_R$, where μ_L and μ_R are given real values. With these constraints, setting $i = 1$ and $i = n-2$ in (3.3) gives

$$\begin{aligned} \Delta x_2 \mu_L + 2(\Delta x_1 + \Delta x_2) s_2 + \Delta x_1 s_3 &= 3(\Delta x_2 y_1' + \Delta x_1 y_2') \\ \Delta x_{n-1} s_{n-2} + 2(\Delta x_{n-2} + \Delta x_{n-1}) s_{n-1} + \Delta x_{n-2} \mu_R &= 3(\Delta x_{n-1} y_{n-2}' + \Delta x_{n-2} y_{n-1}'), \end{aligned}$$

and so the first and last equations are given by

$$\begin{aligned} 2(\Delta x_1 + \Delta x_2) s_2 + \Delta x_1 s_3 &= 3(\Delta x_2 y_1' + \Delta x_1 y_2') - \Delta x_2 \mu_L \\ \Delta x_{n-1} s_{n-2} + 2(\Delta x_{n-2} + \Delta x_{n-1}) s_{n-1} &= 3(\Delta x_{n-1} y_{n-2}' + \Delta x_{n-2} y_{n-1}') - \Delta x_{n-2} \mu_R. \end{aligned}$$

Thus, the setting up of T and r and the resolution of s are completed with the fragment

```
T(1,1) = 2*(Dx(1) + Dx(2));
T(1,2) = Dx(1);
r(1) = 3*(Dx(2)*yp(1) + Dx(1)*yp(2)) - Dx(2)*muL;
T(n-2,n-2) = 2*(Dx(n-2) + Dx(n-1));
T(n-2,n-3) = Dx(n-1);
r(n-2) = 3*(Dx(n-1)*yp(n-2) + Dx(n-2)*yp(n-1)) - Dx(n-2)*muR;
s = [ muL; T \ r(1:n-2) ; muR];
```

assuming that `muL` and `muR` house μ_L and μ_R, respectively.

3.3.3 The Natural Spline

Instead of prescribing the slope of the spline at the endpoints, we can prescribe the value of its second derivative. In particular, if we insist that $\mu_L = q_1''(x_1)$, then from (3.1) it follows that

$$\mu_L = 2\frac{y_1' - s_1}{\Delta x_1} - 2\frac{s_1 + s_2 - 2y_1'}{\Delta x_1},$$

from which we conclude that

$$s_1 = \frac{1}{2}\left(3y_1' - s_2 - \frac{\mu_L}{2}\Delta x_1\right).$$

Substituting this result into the $i = 1$ case of (3.3) and rearranging, we obtain

$$(2\Delta x_1 + 1.5\Delta x_2)s_2 + \Delta x_1 s_3 \;=\; 1.5\Delta x_2 y_1' + 3\Delta x_1 y_2' + \frac{\mu_L}{4}\Delta x_1 \Delta x_2.$$

Likewise, by setting $\mu_R = q_{n-1}''(x_n)$, then (3.2) implies

$$\mu_R = 2\frac{y_{n-1}' - s_{n-1}}{\Delta x_{n-1}} + 4\frac{s_{n-1} + s_n - 2y_{n-1}'}{\Delta x_{n-1}},$$

from which we conclude that

$$s_n = \frac{1}{2}\left(3y_{n-1}' - s_{n-1} + \frac{\mu_R}{2}\Delta x_{n-1}\right).$$

Substituting this result into the $i = n - 2$ case of (3.3) and rearranging we obtain

$$\Delta x_{n-1}s_{n-2} + (1.5\Delta x_{n-2} + 2\Delta x_{n-1})s_{n-1} \;=\; 3\Delta x_{n-1}y_{n-2}' + 1.5\Delta x_{n-2}y_{n-1}' - \frac{\mu_R}{4}\Delta x_{n-2}\Delta x_{n-1}.$$

Thus, the setting up of T and r and the resolution of s are completed with the fragment

```
T(1,1) = 2*Dx(1) + 1.5*Dx(2);
T(1,2) = Dx(1);
r(1) = 1.5*Dx(2)*yp(1) + 3*Dx(1)*yp(2)) + Dx(1)*Dx(2)*muL/4;
T(n-2,n-2) = 1.5*Dx(n-2) + 2*Dx(n-1);
T(n-2,n-3) = Dx(n-1);
r(n-2) = 3*Dx(n-1)*yp(n-2) + 1.5*Dx(n-2)*yp(n-1) -Dx(n-2)*Dx(n-1)*muR;
stilde = T \ r;
s1 = (3*yp(1) - stilde(1) - muL*Dx(1)/2)/2;
sn = (3*yp(n-1) - stilde(n-2) + muR*Dx(n-1)/2)/2;
s = [s1; stilde; sn];
```

If $\mu_L = \mu_R = 0$, then the resulting spline is called *the natural spline.*

3.3.4 The Not-a-Knot Spline

This method for prescribing the end conditions is appropriate if no endpoint derivative information is available. It produces the *not-a-knot* spline. The idea is to ensure third derivative continuity at both x_2 and x_{n-1}. Note from (3.1) that

$$q_i'''(x) = 6\frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2},$$

and so $q_1'''(x_2) = q_2'''(x_2)$ says that

$$\frac{s_1 + s_2 - 2y_1'}{(\Delta x_1)^2} \;=\; \frac{s_2 + s_3 - 2y_2'}{(\Delta x_2)^2}.$$

It follows that this will be the case if we set

$$s_1 = -s_2 + 2y_1' + \left(\frac{\Delta x_1}{\Delta x_2}\right)^2, (s_2 + s_3 - 2y_2').$$

As a result of making the third derivative continuous at x_2, the cubics $q_1(x)$ and $q_2(x)$ are identical.

Likewise, $q_{n-2}'''(x_{n-1}) = q_{n-1}'''(x_{n-1})$ says that

$$\frac{s_{n-2} + s_{n-1} - 2y_{n-2}'}{(\Delta x_{n-2})^2} = \frac{s_{n-1} + s_n - 2y_{n-1}'}{(\Delta x_{n-1})^2}.$$

It follows that this will be the case if we set

$$s_n = -s_{n-1} + 2y_{n-1}' + \left(\frac{\Delta x_{n-1}}{\Delta x_{n-2}}\right)^2 (s_{n-2} + s_{n-1} - 2y_{n-2}').$$

Thus, the first and last equations for the not-a-knot spline are set up as follows:

```
q = Dx(1)*Dx(1)/Dx(2);
T(1,1)= 2*Dx(1) +Dx(2) + q;
T(1,2) = Dx(1) + q;
r(1) = Dx(2)*yp(1) + Dx(1)*yp(2)+2*yp(2)*(q+Dx(1));
q= Dx(n-1)*Dx(n-1)/Dx(n-2);
T(n-2,n-2) = 2*Dx(n-1) + Dx(n-2)+q;
T(n-2,n-3) = Dx(n-1)+q;
r(n-2) = Dx(n-1)*yp(n-2) + Dx(n-2)*yp(n-1) +2*yp(n-2)*(Dx(n-1)+q);
stilde = T\ r;
s1 = -stilde(1)+2*yp(1);
s1 = s1 + ((Dx(1)/Dx(2))^2)*(stilde(1)+stilde(2)-2*yp(2));
sn = -stilde(n-2) +2*yp(n-1);
sn=sn+((Dx(n-1)/Dx(n-2))^2)*(stilde(n-3)+stilde(n-2)-2*yp(n-2));
s=[s1;stilde;sn];
```

3.3.5 The Cubic Spline Interpolant

The function `CubicSpline` can be used to construct the cubic spline interpolant with any of the three aforementioned types of end conditions. Here is its specification:

```
  function [a,b,c,d] = CubicSpline(x,y,derivative,muL,muR)
% [a,b,c,d] = CubicSpline(x,y,derivative,muL,muR)
% Cubic spline interpolation with prescribed end conditions.
%
% x,y are column n-vectors. It is assumed that n >= 4 and x(1) < ... x(n).
% derivative is an integer (1 or 2) that specifies the order of the endpoint derivatives.
% muL and muR are the endpoint values of this derivative.
%
% a,b,c, and d are column (n-1)-vectors that define the spline S(z). On [x(i),x(i+1)],
%
%          S(z) =  a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2 + d(i)(z-x(i))^2(z-x(i+1).
%
% Usage:
%   [a,b,c,d] = CubicSpline(x,y,1,muL,muR)   S'(x(1))  = muL, S'(x(n))  = muR
%   [a,b,c,d] = CubicSpline(x,y,2,muL,muR)   S''(x(1)) = muL, S''(x(n)) = muR
%   [a,b,c,d] = CubicSpline(x,y)             S'''(z) continuous at x(2) and x(n-1)
%
```

Notice that a two-argument call is all that is required to produce the not-a-knot spline. The script `ShowSpline` examines various `CubicSpline` interpolants to the sine function.

Error bounds for the cubic spline interpolant are complicated to derive. The bounds are *not* good if the end conditions are improperly chosen. Figure 3.6 shows what can happen if the complete spline is used with end conditions that are at variance with the behavior of the function being interpolated. However, if the

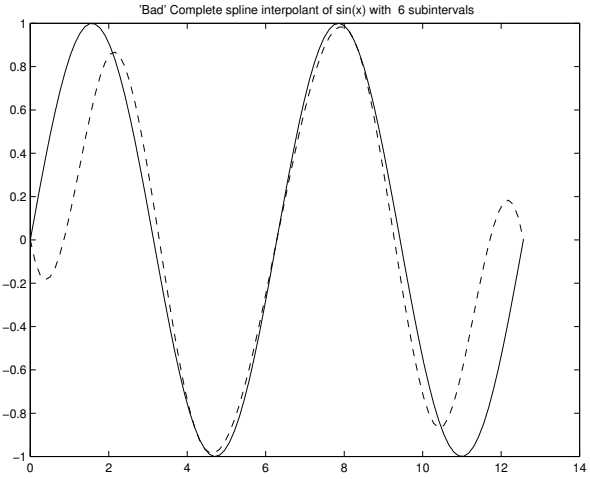

FIGURE 3.6 *Bad end conditions*

end values are properly chosen or if the not-a-knot approach is used, then the error bound has the form $M_4\bar{h}^4$ where $\bar{h}$ is the maximum subinterval length and M_4 bounds the 4th derivative of the function being interpolated. The script `ShowSplineErr` confirms this for the case of an "easy" $f(x)$. It produces the plots shown in Figure 3.7. Notice that the error is reduced by a factor of 10^4 if the subinterval length is reduced

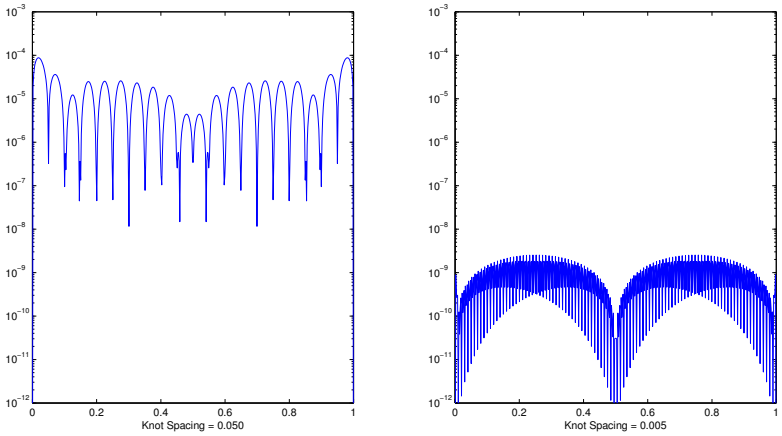

FIGURE 3.7 *Not-a-knot spline error*

by a factor of ten.

3.3.6 MATLAB Spline Tools

The MATLAB function `spline` can be used to compute not-a-knot spline interpolants. It can be called with either two or three arguments. The script

```
z = linspace(-5,5);
x = linspace(-5,5,9);
y = atan(x);
Svals = spline(x,y,z);
plot(z,Svals);
```

illustrates a three-argument call. It plots the $n = 9$ not-a-knot spline interpolant of the function $f(x) = \arctan(x)$ across the interval $[-5, 5]$. The first two arguments in the call to `spline` specify the interpolation points that define the spline S. The spline is then evaluated at `z` with the values returned in `Svals`. Thus $S(\texttt{x(i)}) = \texttt{y(i)}$ for `i=1:length(x)` and $S(\texttt{z(i)}) = \texttt{Svals(i)}$ for `i=1:length(z)`.

A 2-argument call to `spline` returns what is called the *pp-representation* of the spline. This type of reference is required whenever one has to manipulate the local cubics that make up the spline. The pp-representation of a spline is different from the four-vector representation that we have been using for piecewise cubics. For one thing, it is more general because it can accommodate piecewise polynomials of arbitrary degree.

To gain a facility with MATLAB's piecewise polynomial tools, let's consider the problem of constructing the pp-representation of the derivative of a cubic spline S. In particular, let's plot S' where S is a nine-point, equally spaced, not-a-knot spline interpolant of the arctangent function across the interval $[-5, 5]$. We start by constructing the pp-representation of S:

```
x = linspace(-5,5,9);
y = atan(x);
S = spline(x,y)
```

A two-argument call to `spline` such as this produces the pp-representation of the spline. The `ppval` function can be used to evaluate a piecewise polynomial in this representation:

```
z = linspace(-5,5);
Svals = ppval(S,z);
plot(z,Svals)
```

The call to `ppval` returns the value of the spline at `z`. The vector `Svals` contains the values of the spline on `z`. These values are then plotted.

The derivative of the spline is a piecewise *quadratic* polynomial, and by using the functions `unmkpp` and `mkpp` we can produce its pp-representation. A call to `unmkpp` unveils the four major components of the *pp*-representation:

```
[x,rho,L,k] = unmkpp(S)
```

The x-values are returned in `x`. The coefficients of the local polynomials are assembled in an `L`-by-`k` matrix `rho`. `L` is the number of local polynomials and `k-1` is their degree. So in our case, `x = linspace(-5,5,9)`, `L = 8`, and `k=4`. The coefficients of the i-th local cubic are stored in ith row of the `rho` matrix. In particular, the spline is defined by

$$S(z) = \rho_{i,4} + \rho_{i,3}(z - x_i) + \rho_{i,2}(z - x_i)^2 + \rho_{i,1}(z - x_i)^3$$

on the interval $[x_i, x_{i+1}]$. Thus, `rho(i,j)` is the ith local polynomial coefficient of $(x - x_i)^{k-j+1}$.

The function `mkpp` takes the breakpoints and the array of coefficients and produces the pp-representation of the piecewise polynomial so defined. Thus, to set up the *pp*-representation of the spline's derivative, we execute

```
drho = [3*rho(:,1) 2*rho(:,2) rho(:,3)];
dS = mkpp(x,drho);
```

The set-up of the three-column matrix `drho` follows from the observation that

$$S'(x) = \rho_{i,3} + 2\rho_{i,2}(x - x_i) + 3\rho_{i,1}(x - x_i)^2$$

on the interval $[x_i, x_{i+1}]$. Putting it all together, we obtain

```
% Script File: ShowSplineTools
% Illustrates the Matlab functions spline, ppval, mkpp, unmkpp
close all
% Set Up Data:
n = 9;
x = linspace(-5,5,n);
y = atan(x);
%   Compute the spline interpolant and its derivative:
S  = spline(x,y);
[x,rho,L,k] = unmkpp(S);
drho = [3*rho(:,1) 2*rho(:,2) rho(:,3)];
dS = mkpp(x,drho);
%   Evaluate S and dS:
z = linspace(-5,5);
Svals = ppval(S,z);
dSvals = ppval(dS,z);

%   Plot:
atanvals = atan(z);
figure
plot(z,atanvals,z,Svals,x,y,'*');
title(sprintf('n = %2.0f Spline Interpolant of atan(x)',n))
datanvals = ones(size(z))./(1 + z.*z);
figure
plot(z,datanvals,z,dSvals)
title(sprintf('Derivative of n = %2.0f Spline Interpolant of atan(x)',n))
```

Problems

P3.3.1 What can you say about the $n = 4$ not-a-knot spline interpolant of $f(x) = x^3$?

P3.3.2 Suppose $S(z)$ is the not-a-knot spline interpolant of (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4) where it is assumed that the x_i are distinct. Suppose $p(x)$ is the cubic interpolant at same four points. Explain why $S(z) = p(z)$ for all z.

P3.3.3 Let $S(z)$ be the natural spline interpolant of z^3 at $z = -3$, $z = -1$, $z = 1$, $z = 3$. What is $S(0)$?

P3.3.4 Given $\sigma > 0$, (x_i, y_i, s_i), and $(x_{i+1}, y_{i+1}, s_{i+1})$, show how to determine a_i, b_i, c_i, and d_i so that

$$g_i(x) = a_i + b_i(x - x_i) + c_i e^{\sigma(x-x_i)} + d_i e^{-\sigma(x-x_i)}$$

satisfies $g_i(x_i) = y_i$, $g_i'(x_i) = s_i$, $g_i(x_{i+1}) = y_{i+1}$, and $g_i'(x_{i+1}) = s_{i+1}$.

P3.3.5 Another approach that can be used to make up for a lack of endpoint derivative information is to glean that information from a four-point cubic interpolant. For example, if $q_L(x)$ is the cubic interpolant of (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4), then either of the endpoint conditions

$$\begin{aligned} q_1'(x_1) &= q_L'(x_1) \\ q_1''(x_1) &= q_L''(x_1) \end{aligned}$$

is reasonable, where $q_1(x)$ is the leftmost local cubic. Likewise, if $q_R(x)$ is the cubic interpolant of (x_{n-3}, y_{n-3}), (x_{n-2}, y_{n-2}), (x_{n-1}, y_{n-1}), and (x_n, y_n), then either of the right endpoint conditions

$$\begin{aligned} q_{n-1}'(x_n) &= q_R'(x_n) \\ q_{n-1}''(x_n) &= q_R''(x_n) \end{aligned}$$

is reasonable, where $q_{n-1}(x)$ is the rightmost local cubic.

Modify `CubicSpline` so that it invokes this strategy whenever the function call involves just three arguments, (i.e., `[a,b,c,d] = CubicSpline(x,y,derivative`.) The value of `derivative` should determine which derivative is to be matched at the endpoints.

(Its value should be 1 or 2.) Augment the script file `ShowSpline` so that it graphically depicts the splines that are produced by this method.

P3.3.6 Explain how MATLAB's spline tools can be used to compute

$$\int_{\alpha}^{\beta} [S''(x)]^2 dx,$$

where $S(x)$ is a cubic spline.

P3.3.7 Suppose $S(x)$ is a cubic spline interpolant of the data $(x_1, y_1), \ldots, (x_n, y_n)$ obtained using `spline`. Write a MATLAB function `d3 = MaxJump(S)` that returns the maximum jump in the third derivative of the spline `S` assumed to be in the *pp*-representation. Vectorize as much as possible. Use the `max` function.

P3.3.8 Write a MATLAB function `S = Convert(a,b,c,d,x)` that takes our piecewise cubic interpolant representation and converts it into pp form.

P3.3.9 Complete the following function:

```
  function [a,b,c,d] = SmallSpline(z,y)
% z is a scalar and y is 3-vector.
% a,b,c,d are column 2-vectors with the property that if
%
%     S(x) = a(1) + b(1)(x - z) + c(1)(x - z)^2 + d(1)(x - z)^3  on [z-1,z]
% and
%     S(x) = a(2) + b(2)(x - z) + c(2)(x - z)^2 + d(2)(x - z)^3  on [z,z+1]
% then
%                          (a) S(z-1) = y(1), S(z) = y(2), S(z+1) = y(3),
%                          (b) S''(z-1) = S''(z+1) = 0
%                          (c) S, S', and S'' are continuous on [z-1,z+1]
%
```

P3.3.10 In computerized typography the problem arises of finding an interpolant to points that lie on a path in the plane (e.g., a printed capital S). Such a shape cannot be represented as a function of x because it is not single valued. One approach is to number the points $(x_1, y_1), \ldots, (x_n, y_n)$ as we traverse the curve. Let d_i be the straight-line distance between (x_i, y_i) and (x_{i+1}, y_{i+1}), $i = 1{:}n-1$. Set $t_i = d_1 + \cdots + d_{i-1},\ i = 1{:}n$. Suppose $S_x(t)$ is a spline interpolant of $(t_1, x_1), \ldots, (t_n, x_n)$ and that $S_y(t)$ is a spline interpolant of $(t_1, y_1), \ldots, (t_n, y_n)$.

It follows that the curve $\Lambda = \{(S_x(t), S_y(t)) : t_1 \le t \le t_n\}$ is smooth and passes through the n points. Write a MATLAB function `[xi,yi] = SplineInPlane(x,y,m)` that returns in `xi(1:m)` and `yi(1:m)` the x-y coordinates of m points on the curve Λ. Use the MATLAB `Spline` function to determine the splines $S_x(t)$ and $S_y(t)$.

To test `SplineInPlane` write a script that solicits an arbitrary number of points from the plot window using `ginput`. It should echo your mouseclicks by placing an asterisk at each point. After all the points are acquired it should compute the splines S_x and S_y defined above and then plot the curve Λ. Use `hold on` so that the asterisks are also displayed.

Submit listings and sample output showing a personally designed letter "S". The number of input points used is up to you.

P3.3.11 Let $S(x)$ be the not-a-knot cubic spline interpolant of (0,0), (1,1), (2,8), (3,27). Explain why $S(3/2) = (3/2)^3$.

P3.3.12 Suppose `x` and `y` are column n-vectors with $x_1 < x_2 < \cdots < x_n$. If `z` is a column m-vector, then `sval = spline(x,y,z)` is a column m-vector with the property that `sval(i)` $= S(z_i)$, where S the not-a-knot spline interpolant of $(x_1, y_1), \ldots, (x_n, y_n)$.

Let

$S_1(x)$	be the not-a-knot spline interpolant of $\sin(x)$ at	$x_i = (i-1)/10,\ i = 1{:}21$
$S_2(x)$	be the not-a-knot spline interpolant of $\exp(x)$ at	$x_i = (i-1)/10,\ i = 1{:}21$
$S_3(x)$	be the not-a-knot spline interpolant of $\sin(x) \cdot \exp(x)$ at	$x_i = (i-1)/10,\ i = 1{:}21$
$S_4(x)$	be the not-a-knot spline interpolant of $2\sin(x) + 3\exp(x)$ at	$x_i = (i-1)/10,\ i = 1{:}21$

Write a vectorized MATLAB script that plots in a single window these four splines across the interval [0,2]. The plots should be based on one-hundred, equally-spaced evaluations. Avoid unnecessary function calls. You do not have to exploit any trigonometric or exponential identities.

P3.3.13 Produce a plot that shows that it is a bad idea to interpolate with the natural spline if the second derivative of the underlying function is not zero at the endpoints.

P3.3.14 Suppose $f(t)$ and its first two derivatives are defined everywhere. If f has period T (positive), then $f(t+T) = f(t)$ for all t. Consider the problem of interpolating such a function on an interval $[\tau, \tau + T]$ with a spline S having breakpoints

$$\tau = t_1 < \cdots < t_n = \tau + T.$$

It makes sense to require

$$\begin{aligned} S'(\tau) &= S'(\tau+T) \\ S''(\tau) &= S''(\tau+T), \end{aligned}$$

since $f'(\tau) = f'(\tau+T)$ and $f''(\tau) = f''(\tau+T)$. Moreover, we can then extend S periodically off the "base" interval $[\tau, \tau+T]$ and obtain a piecewise cubic interpolant that is continuous through the second derivative. **(a)** Modify `CubicSpline` so that a 3-argument call of the form

```
[a,b,c,d] = CubicSpline(x,y,0)
```

produces the periodic spline interpolant. In other words,

$$\begin{aligned} S(x_i) &= y_i \\ S'(x_1) &= S'(x_n) \\ S''(x_1) &= S''(x_n) \end{aligned}$$

where $i = 1{:}n$ and n is the length of `x`. Test your adaptation with the function

$$f(x) = \sin(2\pi x) - .3\cdot\cos(4\pi x) + .6\cdot\sin(6\pi x) + .2\cdot\cos(8\pi x)$$

by generating its periodic spline interpolant on `linspace(0,1,15)`. Print a table of the coefficients $a(1{:}14)$, $b(1{:}14)$, $c(1{:}14)$, and $d(1{:}14)$ and plot both f and the spline across $[0,1]$. **(b)** A not-a-knot spline interpolant of f across $[\tau, \tau+T]$ will in general not be periodic. However, we can make it "almost" periodic by choosing $t_2 = t_1 + \delta$ and $t_{n-1} = t_n - \delta$ for small δ. Write a function

```
  function s = Periodic(f,t1,T,n,del)
% f is a handle that references an available function f(t) that has period T and is defined
% everywhere. t1 is a real scalar, n is an integer >= 4, and del a positive scalar that
% satisfies del < T/2.
%
% s is the pp-form of the not-a-knot spline that interpolates f at t(1),...,t(n) where
%
%                        t1                                 if k=1
%                        t1 + del                           if k=2
%             t(k)  =    t1 + del + (k-2)*(T-2*del)/(n-3)   if k=3:n-2
%                        t1 + T-del                         if k=n-1
%                        t1 + T                             if k=n
%
%     A four-parameter reference of the form s = Periodic(f,t1,T,n) should
%     return the not-a-knot spline interpolant of f at linspace(t1,t1+T,n).
```

M-Files and References

Script Files

`ShowpwL1`	Illustrates `pwL` and `pwLeval`.
`ShowpwL2`	Compares pwLstatic and `pwLadapt`.
`ShowHermite`	Illustrates the Hermite interpolation idea.
`ShowpwC`	Illustrates `pwC` and `pwCeval`.
`ShowSpline`	Illustrates `CubicSpline`.
`ShowSplineErr`	Explores the not-a-knot spline interpolant error.
`ShowMatSplineTools`	Illustrates MATLAB spline tools.

Function Files

`Locate`	Determines the subinterval in a mesh that houses a given x-value
`pwL`	Sets up a piecewise linear interpolant.
`pwLeval`	Evaluates a piecewise linear function.
`pwLstatic`	A priori determination of a mesh for a pwL approximation.
`pwLadapt`	Dynamic determination of a mesh for a pwL approximation.
`HCubic`	Constructs the cubic Hermite interpolant.
`pwC`	Sets up a piecewise cubic Hermite interpolant.
`pwCeval`	Evaluates a piecewise cubic function.
`CubicSpline`	Constructs complete, natural, or not-a-knot spline.

References

R. Bartels, J. Beatty, and B. Barsky (1987). *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann, Los Altos, CA.

C. de Boor (1978). *A Practical Guide to Splines*, Springer, Berlin.