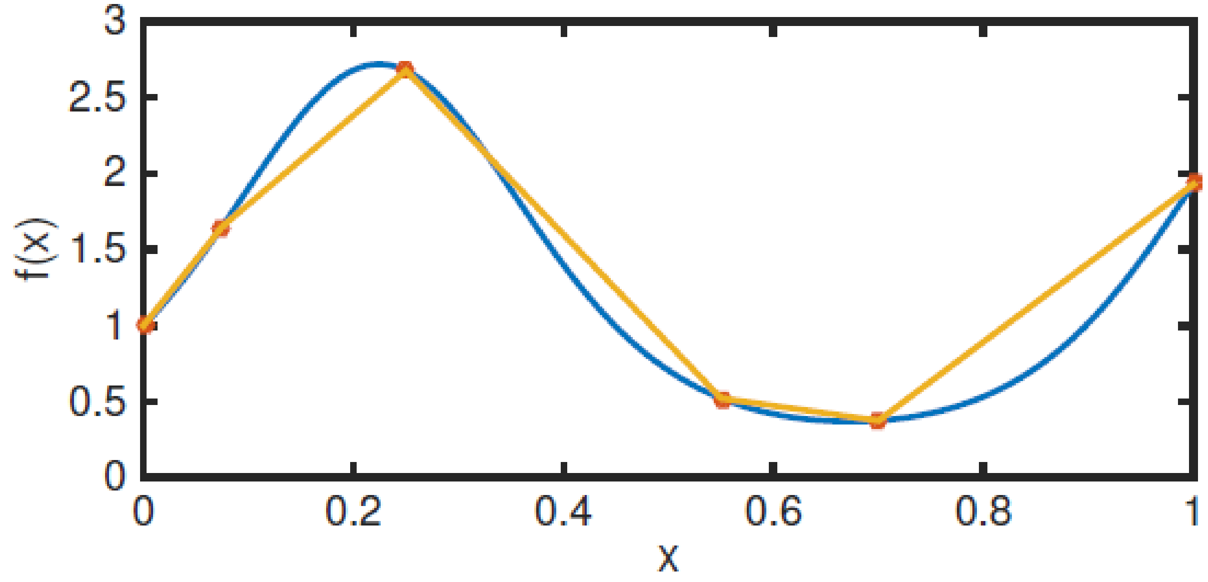
Chapter 5
Interpolation and Calculus



Interpolation

- We are interested in this on several levels.
- We can:
  1. Approximate data if we have just the right amount of data for the functions we want to use
  2. Approximate functions
  3. Develop error formulae for the difference between the interpolant and a given function
  4. Develop methods for approximating derivatives and integrals (calculus)
- We will mostly focus on the last three

Interpolation

- For this chapter and part of the course, we generate data from a function
- We assume that the function is known
- For the purposes of analysis, we assume that such a function is always present
- We will want to use our function to calculate values of the interpolant $p(x)$ anywhere on a continuous domain

Interpolation

- Interpolation is the process of creating a function $p(x)$ of the continuous variable x that passes through, or recovers, given data
- Consider the data to be $n+1$ distinct points (nodes) with $(t_0, y_0), (t_1, y_1), \dots, (t_n, y_n)$ with $t_0 < t_1 < \cdots < t_n$
- Note that the nodes $t_i, i = 0,1, \dots, n$ are distinct
- We require $p(t_i) = y_i, i = 0,1, \dots, n \Longrightarrow p(x)$ passes through the data
- The function that does this is the interpolant $p(x)$
- (The easiest and most common form is to recover only the function values themselves; more complicated versions may recover derivatives too)

Interpolation

- We already have a case where we have done something like this
- Consider the data to be n+1 distinct points with $(t_0, y_0), (t_1, y_1), \ldots, (t_n, y_n)$ with $t_0 < t_1 < \cdots < t_n$
- For a polynomial of degree n, there are n+1 constants to find
- Then we could create the Vandermonde system $\boldsymbol{Vc} = \boldsymbol{y}$ where
$$\boldsymbol{V} = [\boldsymbol{t}.\hat{}n\ \boldsymbol{t}.\hat{}(n-1)\ \ \ldots\ \boldsymbol{t}.\hat{}1\ \boldsymbol{t}.\hat{}0]$$
- and $\boldsymbol{y}$ and $\boldsymbol{t}$ are the column vectors of the data
- We could solve this system, but from a numerical point of view, we saw that $\boldsymbol{V}$ could be very poorly conditioned as n grows larger
- We return to practical issues for this in Chapter 9

Interpolation

- There are theoretical results that are of interest here however
- Consider the data to be n+1 distinct points with $(t_0, y_0), (t_1, y_1), \dots, (t_n, y_n)$ with $t_0 < t_1 < \cdots < t_n$
- One can prove that there is a unique polynomial of degree at most n that interpolates (passes through) the data
- Consider comparing a known function $f(x)$ with a polynomial $p(x)$
- *Weierstrass Approximation Theorem*: For a continuous f defined on $x \in [a, b]$, for each $\epsilon > 0$, there exists a polynomial p such that $|f(x) - p(x)| < \epsilon$ for all $x \in [a, b]$.
- That is, we can make a polynomial arbitrarily close to a function if we drive up the degree enough.

Interpolation

- Finally, one that is important for us is about the error between f and p
- Consider the data to be n+1 distinct points in $[a, b]$ with $(t_0, f(t_0)), (t_1, f(t_1)), \dots, (t_n, f(t_n))$ with $t_0 < t_1 < \cdots < t_n$
- Let $f(x)$ have at least $n + 1$ continuous derivatives
- There exists a number $\xi(x) \in (a, b)$ such that

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - t_i)$$

- This result quantifies how close the interpolant and the function are
- We will use this result, but we will not use it by making n large (at least not yet)

Interpolation

- The result

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - t_i)$$

suggests two strateges to lower the error.

- If the derivatives of *f* are small, the error may be small; but we can't choose all of our functions this way
- If the factors are $(x - t_i)$ are all small, and the derivative of *f* is not too large, we may be able to make the error small
- Large n creates a large denominator, works if other factors aren't too large
- We will not make n large (at least not yet)

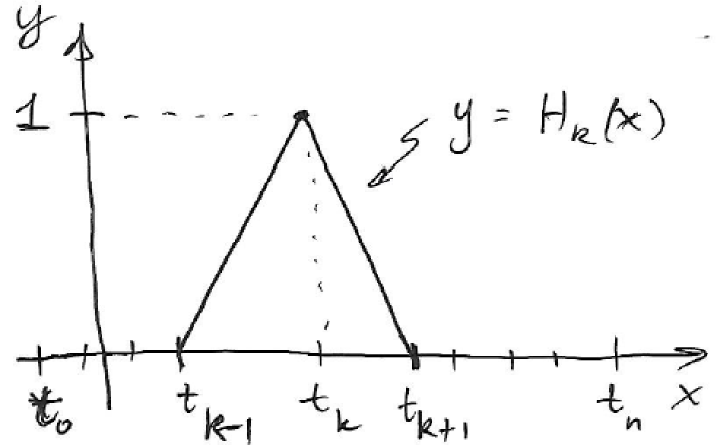
Piecewise linear interpolation

- This is great for one interval, but is not so easy to generalize for all subintervals

$$p(x) = y_k + \frac{y_{k+1} - y_k}{t_{k+1} - t_k}(x - t_k), \quad \text{for } x \in [t_k, t_{k+1}].$$

- Let's rewrite how we do the pcwise linear interpolant
- First, let's create a simple "cardinal function" which returns 1 at a node of interest, and 0 at every other node:

$$H_k(x) = \begin{cases} \frac{x - t_{k-1}}{t_k - t_{k-1}}, & \text{if } x \in [t_{k-1}, t_k], \\ \frac{t_{k+1} - x}{t_{k+1} - t_k}, & \text{if } x \in [t_k, t_{k+1}], \\ 0, & \text{otherwise,} \end{cases}$$



Piecewise linear interpolation

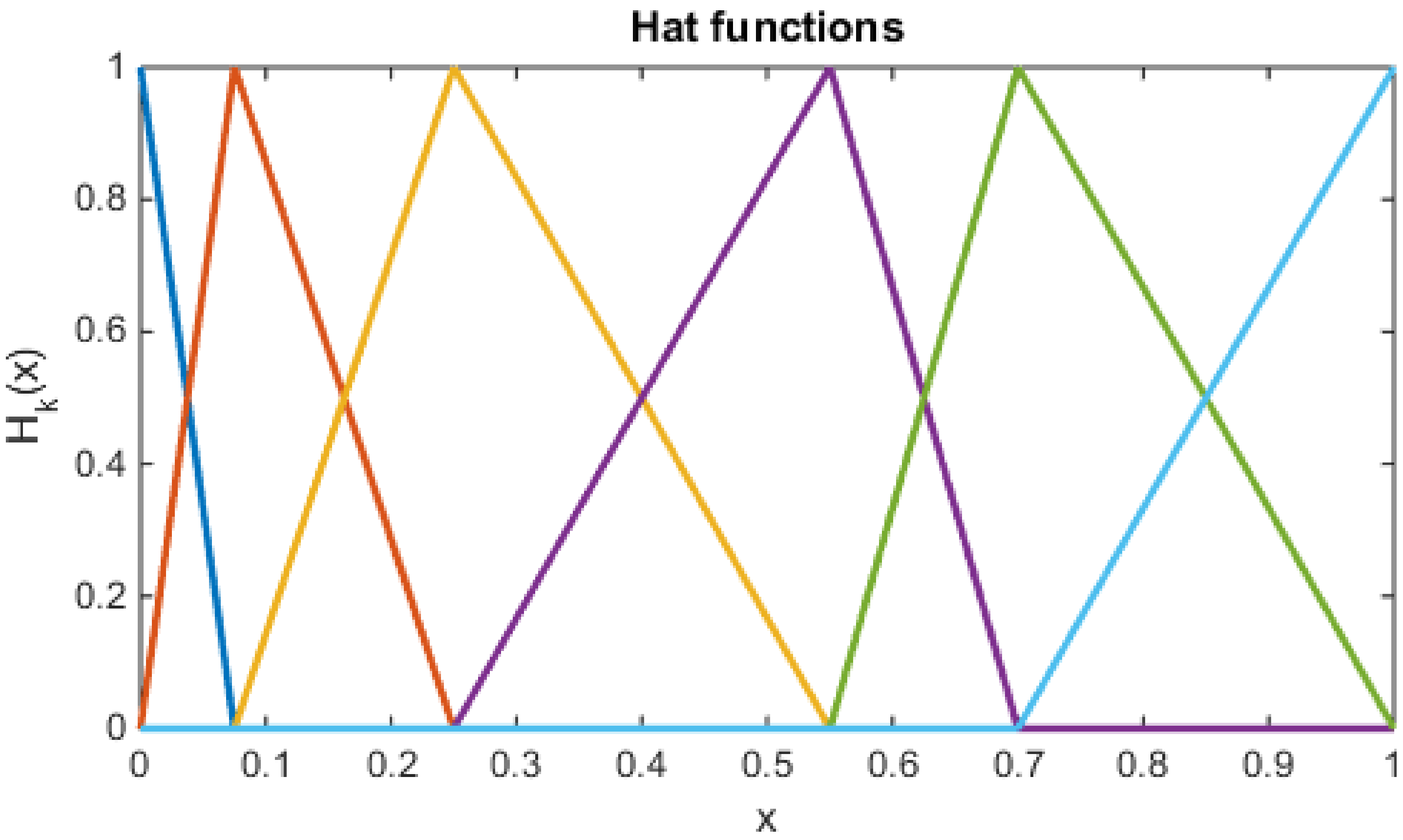
- This will allow us to draw hats or tents with a 1 at each node
- Here's a way to do it in Matlab:

```
t = [0 0.075 0.25 0.55 0.7 1]';
z = linspace(0,1,101);
for k = 0:5
    y = hatfun(z,t,k);
    plot(z,y,'LineWidth',2)
    % fplot (@(x) hatfun (x,t,k) ,[0 1])
    hold on
end
xlabel('x'),ylabel('H_k(x)'), title('Hat functions')
```

- Uses textbook hat function

Piecewise linear interpolation

- The nodes are the six given t values.
- Each color is one hat function
- Ends have half of one
- Let’s look at details



Piecewise linear interpolation

- Text Matlab function:

```
function H = hatfun(xi,x,k)
% HATFUN Evaluate a hat function (PL basis function).
% Input:
%   xi  evaluation points (vector)
%   x   interpolation nodes (vector, length n+1)
%   k   node index (integer, in [1,n+1])
% Output:
%   H   values of the kth basis function

n = length(x)-1;
% "Fictitious nodes" to deal with first, last functions
x = [ 2*x(1)-x(2); x(:); 2*x(n+1)-x(n) ];
k = k+2;  % adjust index to start with fictitious node as 1

H1 = max( 0, (xi-x(k-1))/(x(k)-x(k-1)) ); % upward slope
H2 = max( 0, (x(k+1)-xi)/(x(k+1)-x(k)) ); % downward slope
H = min(H1,H2);
```

Piecewise linear interpolation

- Text Matlab function:

```
function H = hatfun(xi,x,k)
% HATFUN Evaluate a hat function (PL basis function).
% Input:
%    xi   evaluation points (vector)
%    x    interpolation nodes (vector, length n+1)
%    k    node index (integer, in [1,n+1])
% Output:
%    H    values of the kth basis function

n = length(x)-1;
% "Fictitious nodes" to deal with first, last functions
x = [ 2*x(1)-x(2); x(:); 2*x(n+1)-x(n) ];
k = k+2;  % adjust index to start with fictitious node as 1

H1 = max( 0, (xi-x(k-1))/(x(k)-x(k-1)) ); % upward slope
H2 = max( 0, (x(k+1)-xi)/(x(k+1)-x(k)) ); % downward slope
H = min(H1,H2);
```

Sets max value starting from 0 index

Piecewise linear interpolation

- Text Matlab function:

```
function H = hatfun(xi,x,k)
% HATFUN Evaluate a hat function (PL basis function).
% Input:
%   xi   evaluation points (vector)
%   x    interpolation nodes (vector, length n+1)
%   k    node index (integer, in [1,n+1])
% Output:
%   H    values of the kth basis function

n = length(x)-1;
% "Fictitious nodes" to deal with first, last functions
x = [ 2*x(1)-x(2); x(:); 2*x(n+1)-x(n) ];
k = k+2;  % adjust index to start with fictitious node as 1

H1 = max( 0, (xi-x(k-1))/(x(k)-x(k-1)) ); % upward slope
H2 = max( 0, (x(k+1)-xi)/(x(k+1)-x(k)) ); % downward slope
H = min(H1,H2);
```

Add extra points with same spacing as ends in given data

Piecewise linear interpolation

- Text Matlab function:

```
function H = hatfun(xi,x,k)
% HATFUN Evaluate a hat function (PL basis function).
% Input:
%   xi   evaluation points (vector)
%   x    interpolation nodes (vector, length n+1)
%   k    node index (integer, in [1,n+1])
% Output:
%   H    values of the kth basis function

n = length(x)-1;
% "Fictitious nodes" to deal with first, last functions
x = [ 2*x(1)-x(2); x(:); 2*x(n+1)-x(n) ];
k = k+2;   % adjust index to start with fictitious node as 1

H1 = max( 0, (xi-x(k-1))/(x(k)-x(k-1)) ); % upward slope
H2 = max( 0, (x(k+1)-xi)/(x(k+1)-x(k)) ); % downward slope
H = min(H1,H2);
```

Shift index to start at 1

Piecewise linear interpolation

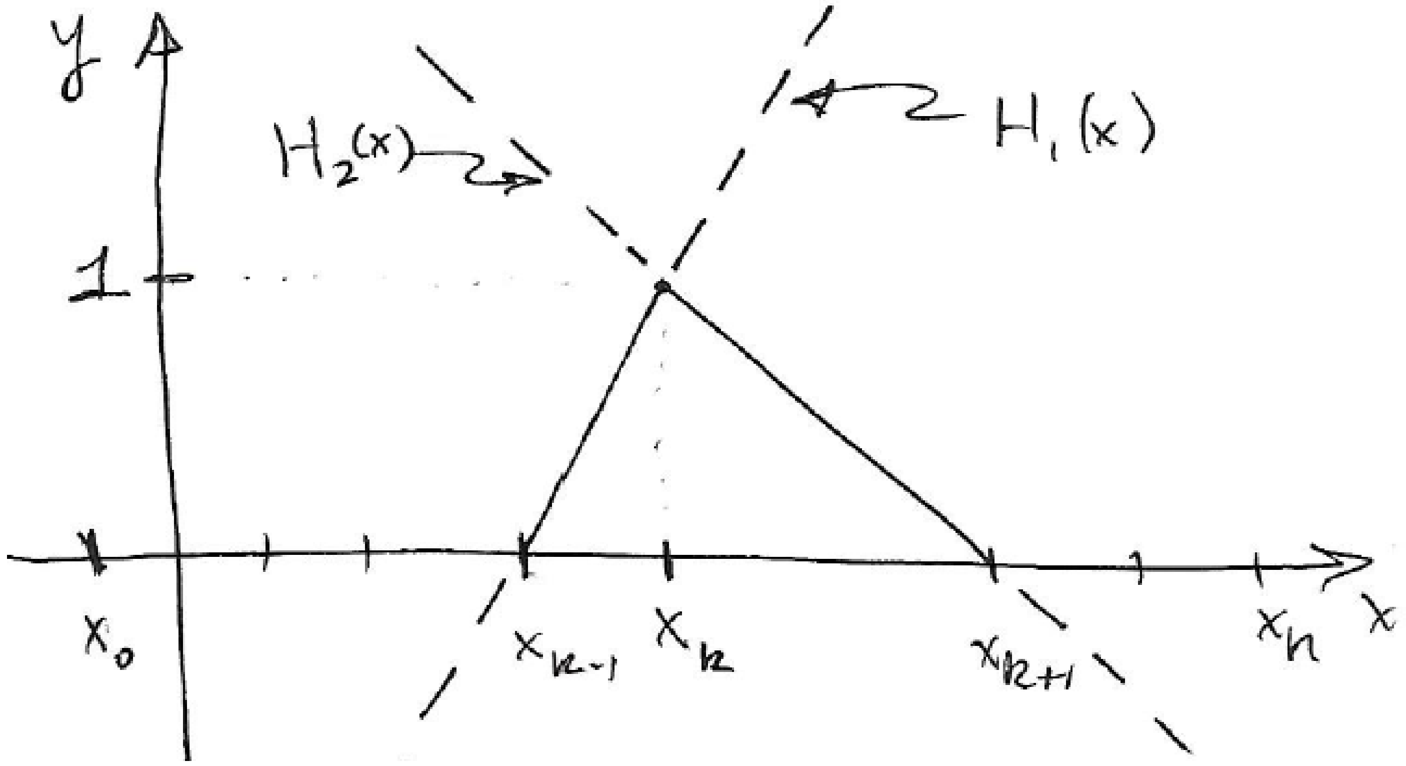
- Text Matlab function:

```
function H = hatfun(xi,x,k)
% HATFUN Evaluate a hat function (PL basis function).
% Input:
%    xi   evaluation points (vector)
%    x    interpolation nodes (vector, length n+1)
%    k    node index (integer, in [1,n+1])
% Output:
%    H    values of the kth basis function

n = length(x)-1;
% "Fictitious nodes" to deal with first, last functions
x = [ 2*x(1)-x(2); x(:); 2*x(n+1)-x(n) ];
k = k+2;  % adjust index to start with fictitious node as 1

H1 = max( 0,  (xi-x(k-1))/(x(k)-x(k-1)) ); % upward slope
H2 = max( 0,  (x(k+1)-xi)/(x(k+1)-x(k)) ); % downward slope
H = min(H1,H2);
```

Choose correct part of hat function from 4 possibilities

Piecewise linear interpolation

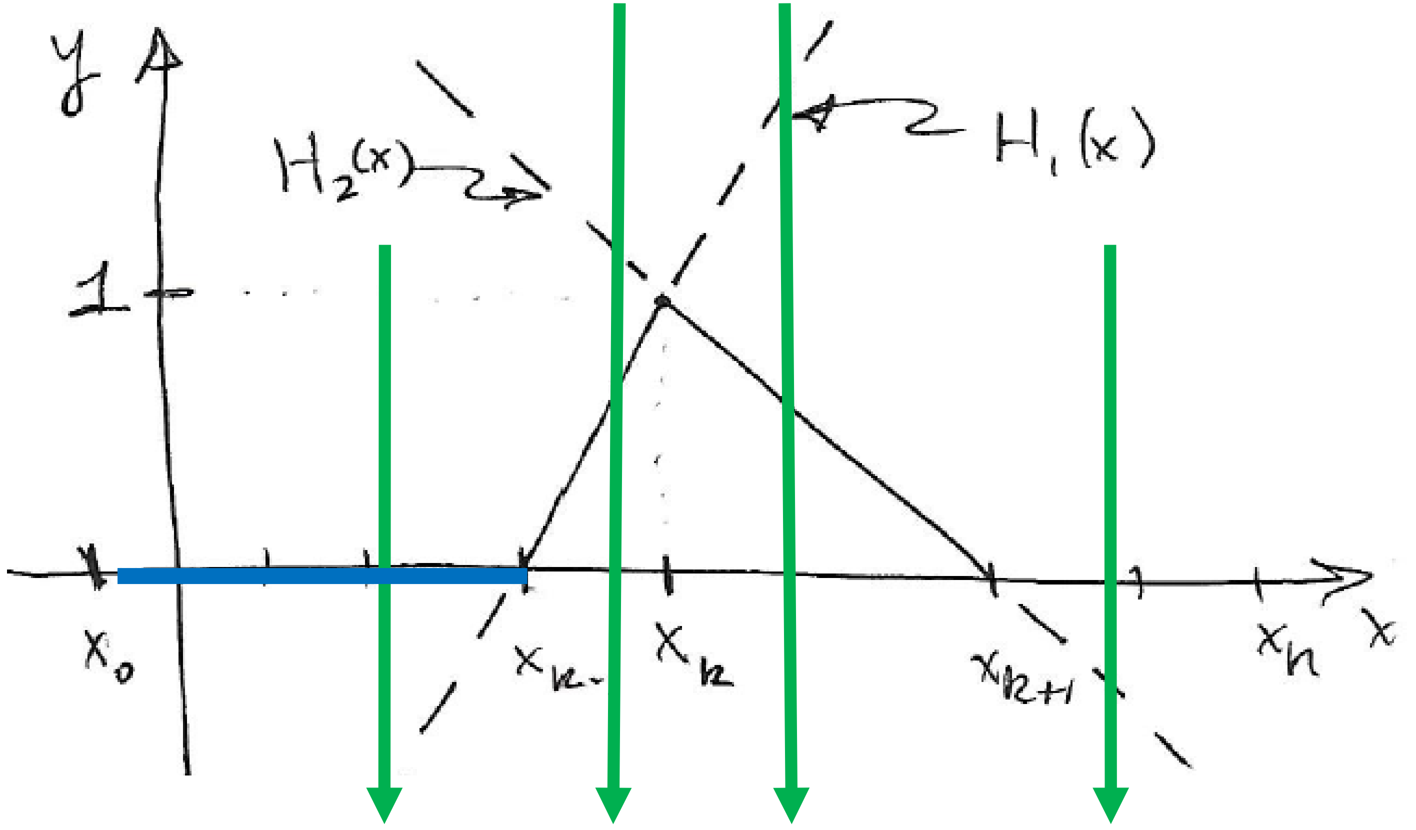
- Here's a sketch of parts of the hat function
- For node k, $H_1(x)$ and $H_2(x)$ are sketched
- We want the solid parts



Piecewise linear interpolation

- There are four possible choices according to each node k

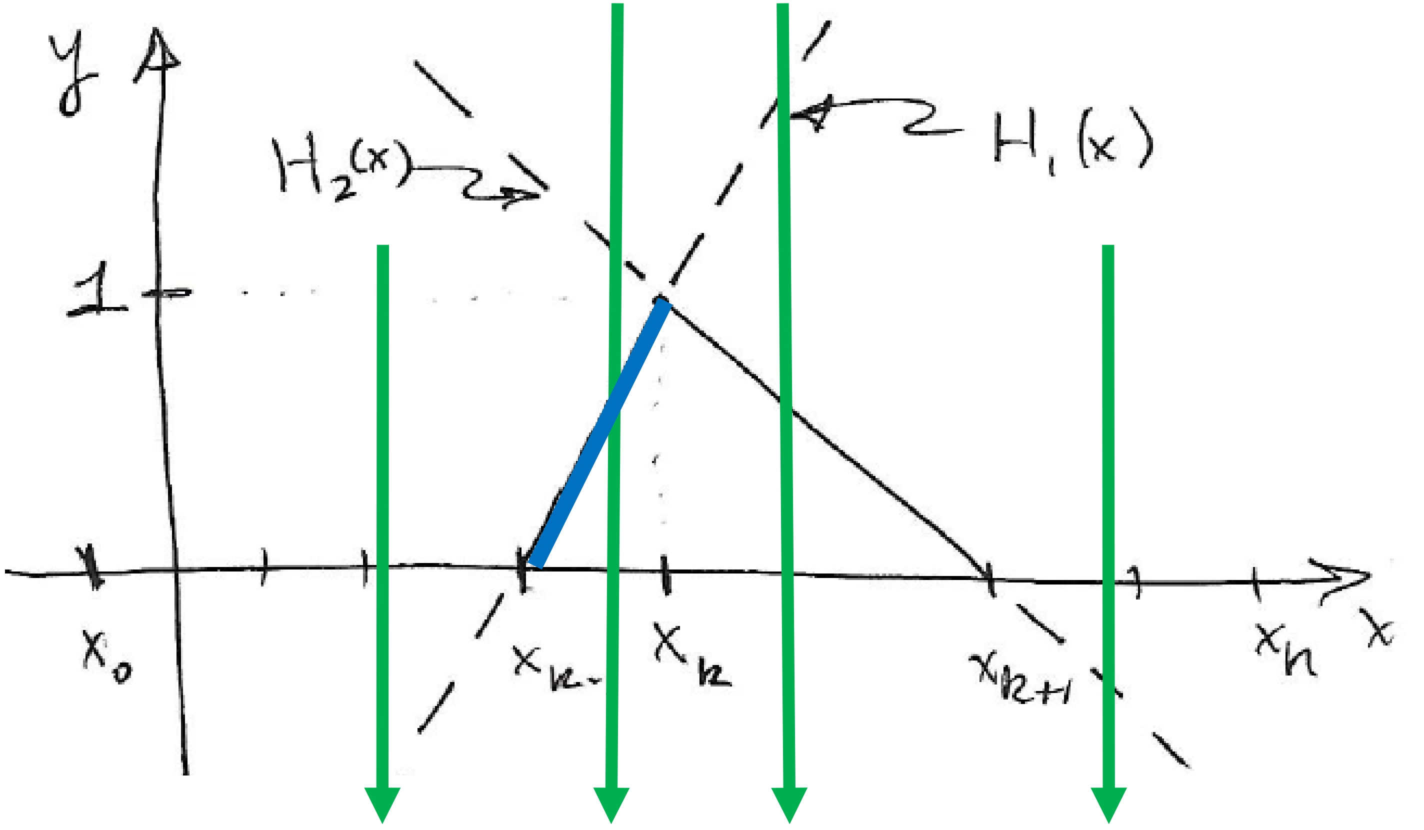
1. Choose 0 to left of previous node x_{k-1}.
2. Linear increase corresponding to left of node x_k
3. Linear decrease corresponding to right of node x_k
4. 0 to right of succeeding node x_{k+1}



Piecewise linear interpolation

- There are four possible choices according to each node k

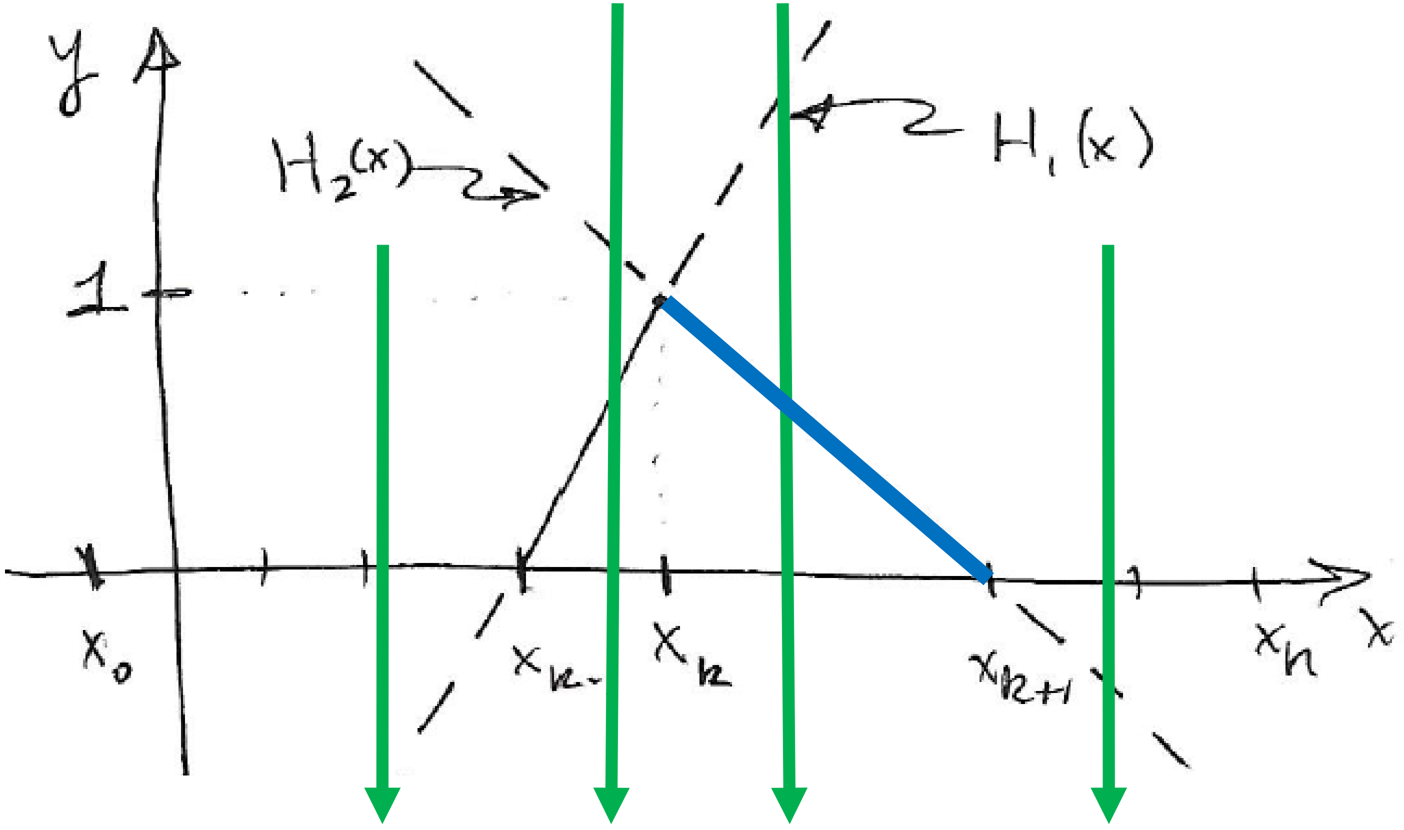
1. Choose 0 to left of previous node x_{k-1}.
2. Linear increase corresponding to left of node x_k
3. Linear decrease corresponding to right of node x_k
4. 0 to right of succeeding node x_{k+1}



Piecewise linear interpolation

- There are four possible choices according to each node k

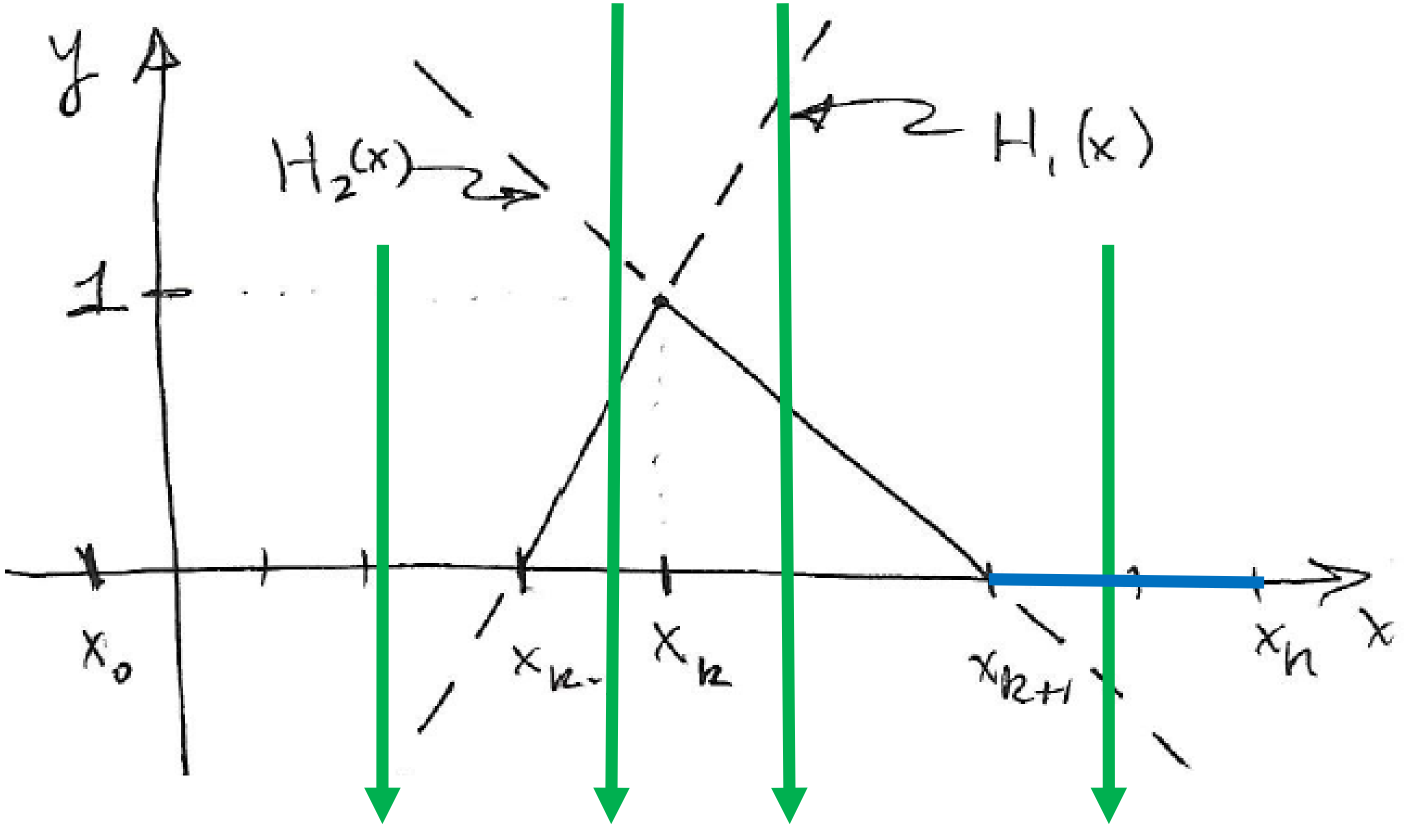
1. Choose 0 to left of previous node x_{k-1}.
2. Linear increase corresponding to left of node x_k
3. Linear decrease corresponding to right of node x_k
4. 0 to right of succeeding node x_{k+1}



Piecewise linear interpolation

- There are four possible choices according to each node k

1. Choose 0 to left of previous node x_{k-1}.
2. Linear increase corresponding to left of node x_k
3. Linear decrease corresponding to right of node x_k
4. 0 to right of succeeding node x_{k+1}



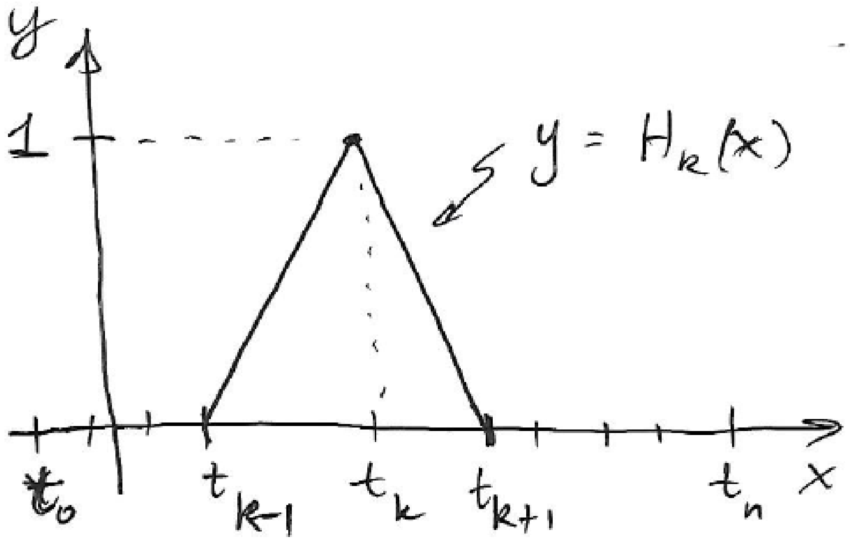
Piecewise linear interpolation

- We want to build the PL interpolant from

$$p(x) = \sum_{k=0}^{n} c_k H_k(x),$$

- But, the hat functions satisy the "cardinal conditions" which returns 1 at a node of interest, and 0 at every other node:

$$H_k(t_j) = \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{otherwise,} \end{cases}$$



- This makes the algebra trivial…

Piecewise linear interpolation

- We can easily recover the data by requiring

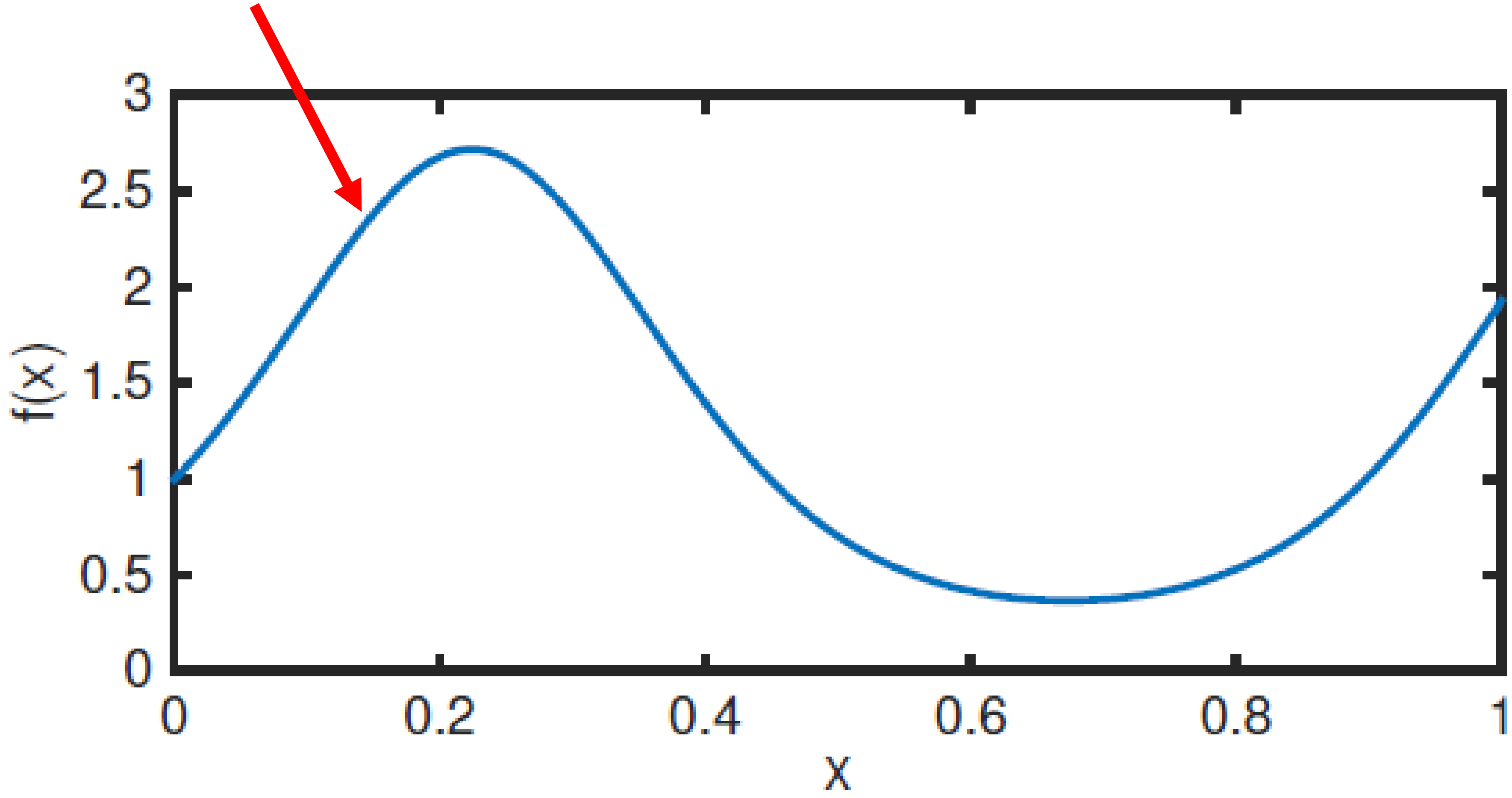
$$p(t_i) = c_i H_i(x) \quad \Rightarrow \quad p(x) = \sum_{k=0}^{n} y_k H_k(x).$$

- This makes it easy to do this kind of interpolation once you can make the hat functions

```
function P = plinterp(x,t,y)
% PLINTERP Piecewise linear interpolation.
% Input:
%    x   evaluation points for the interpolant (vector)
%    t   interpolation nodes (vector, length n+1)
%    y   interpolation values (vector, length n+1)
% Output:
%    P   values of the piecewise linear interpolant (vector)

n = length(t)-1;
P = zeros(size(xi));
for k = 0:n
  P = P + y(k+1)*hatfun(x,t,k);
end
```

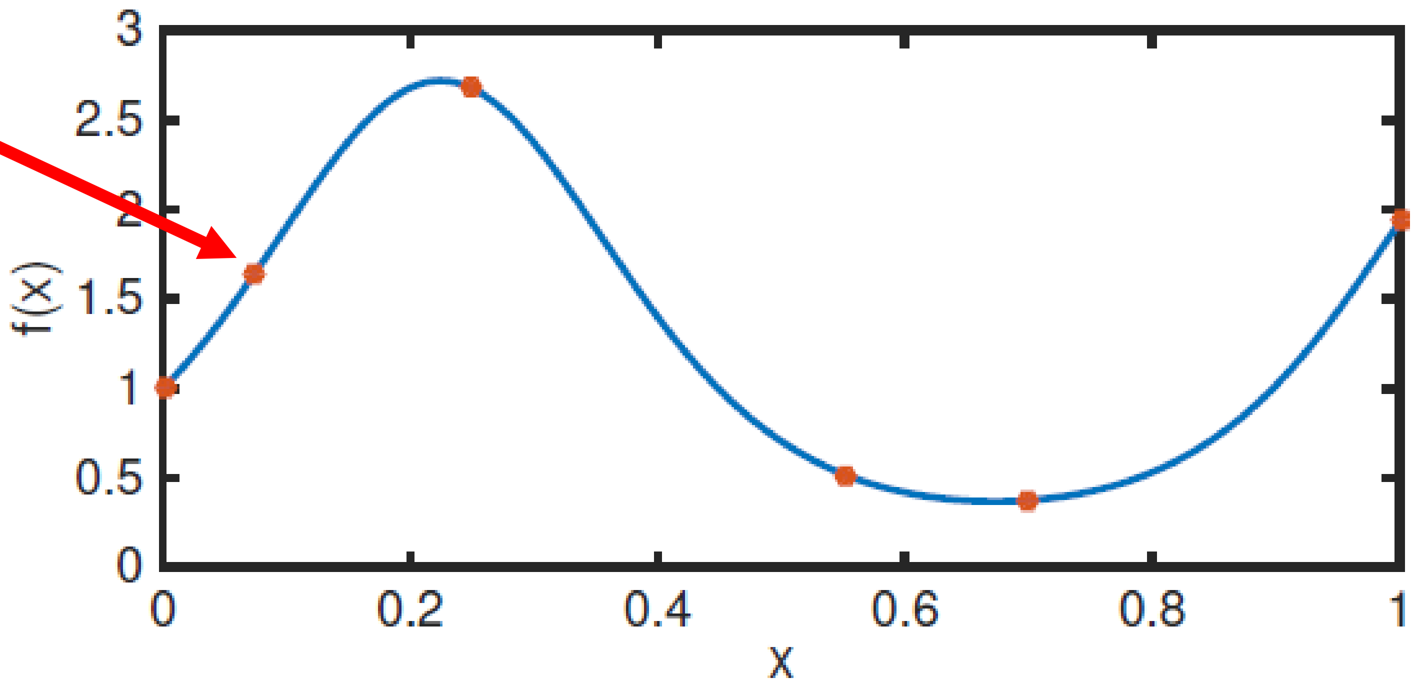
PL interpolation example

```
f = @(x) exp(sin(7*x));   % example function
fplot(f,[0,1])
xlabel('x'), ylabel('f(x)')

t = [0 0.075 0
y = f(t);
hold on
plot(t,y,'o')

p = @(x) plint
fplot(p,[0 1])
```



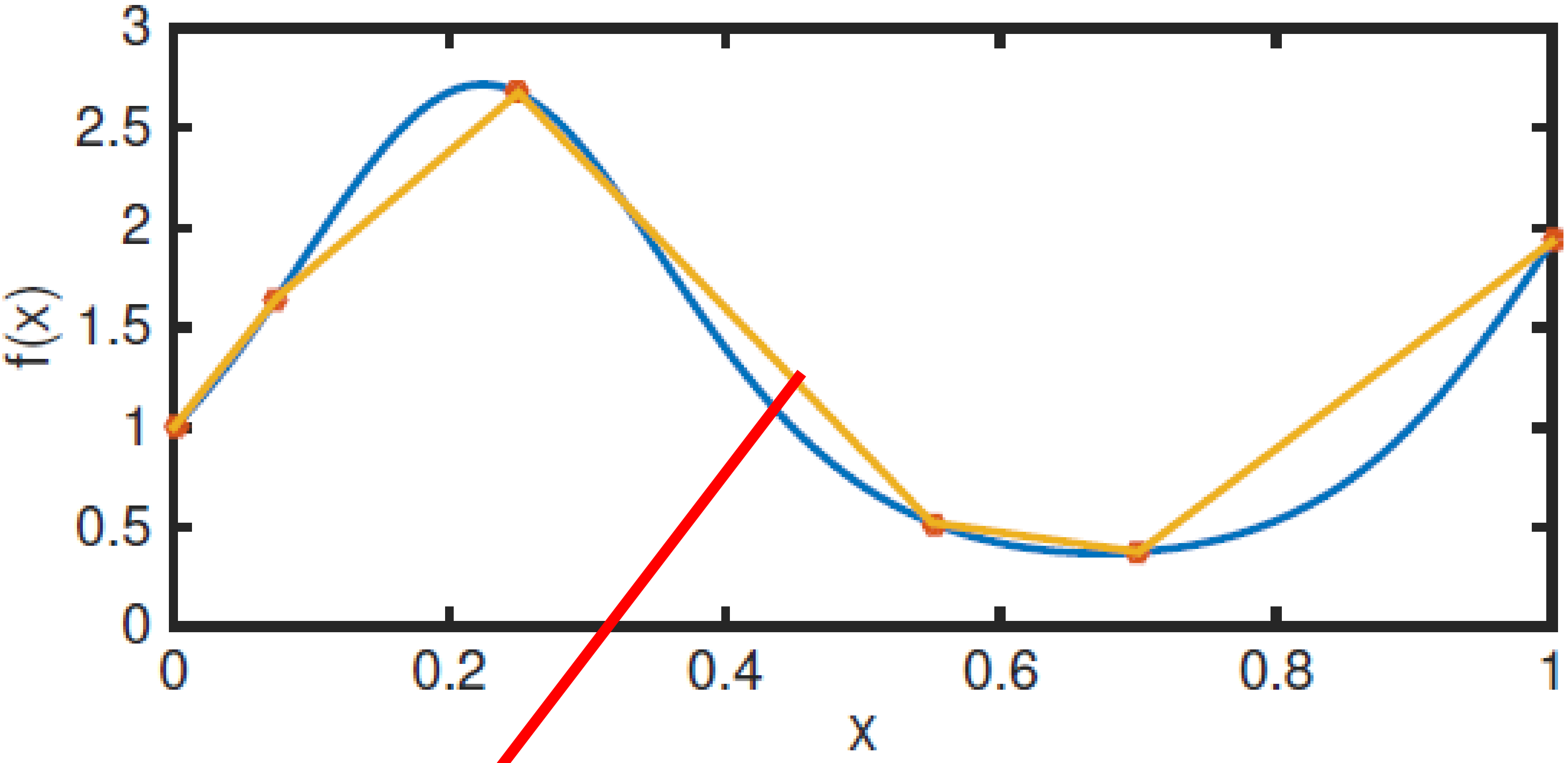
PL interpolation example

```
f = @(x) exp(sin(7*x));   % example function
fplot(f,[0,1])
xlabel('x'), ylabel('f(x)')

t = [0 0.075 0.25 0.55 0.7 1]';
y = f(t);           % create the data to plot
hold on
plot(t,y,'o')

p = @(x) plinterp(x
fplot(p,[0 1])
```



PL interpolation example

```
f = @(x) e
fplot(f,[0
xlabel('x'

t = [0 0.0
y = f(t);
hold on
plot(t,y,'
```




```
p = @(x) plinterp(x,t,y);
fplot(p,[0 1])          % plot the interpolant
```

PL Interpolation

- The conditioning is perfect; the condition number is unity in the ∞-norm
- Modifying the data: $\{(t_k, y_k)\}$ to $\{(t_k, \tilde{y}_k)\}$
- The difference between the interpolant is then

$$\left[\sum_{k=0}^{n} \tilde{y}_k H_k(x)\right] - \left[\sum_{k=0}^{n} y_k H_k(x)\right] = \sum_{k=0}^{n} (\tilde{y}_k - y_k) H_k(x).$$

- The biggest error will be $\|\tilde{\mathbf{y}} - \mathbf{y}\|_\infty.$
- This leads to the first bullet: the difference between the interpolants is at most the difference in the data

PL Interpolation

- How far off will the interpolant be from a function that gave the data?
- Thrm: Suppose that $a = t_0 < t_1 < \cdots < t_n = b$ are given; f and its first two derivatives are continuous; $y_k = f(t_k)$ for $k = 0,1,\ldots,n$ and $p(x)$ is the PL interpolant. Then,

$$\left\| f(x) - p(x) \right\|_\infty = \max_{x\in[a,b]} |f(x) - p(x)| \le Mh^2$$

- Here $M = \left\| f'' \right\|_\infty$ *and* $h = \max_{k=0,\ldots,n-1} |t_{k+1} - t_k|.$
- Now if the spacing between every node is h, that is $h = t_{k+1} - t_k$, then the $h = (b-a)/n$ and $t_k = a + kh$, and the error decreases quadratically as n increases or h decreases: $O(h^2)$ accuracy

PL interpolation convergence

```
f = @(x) exp(sin(7*x));
x = linspace(0,1,10001)';
err_ = [];
n_ = 2.^(3:10)';

for n = n_'
    t = linspace(0,1,n+1)';
    err = max(abs( f(x) - plinterp(x,t,f(t)) ));
    err_ = [ err_; err ];
end
loglog( n_, err_, '.-' )
hold on, loglog( n_, 0.1*(n_/n_(1)).^(-2), '--' )
xlabel('n'), ylabel('||f-p||_\infty')
title('Convergence of PL interpolation')
legend('error','2nd order')
```

Example function

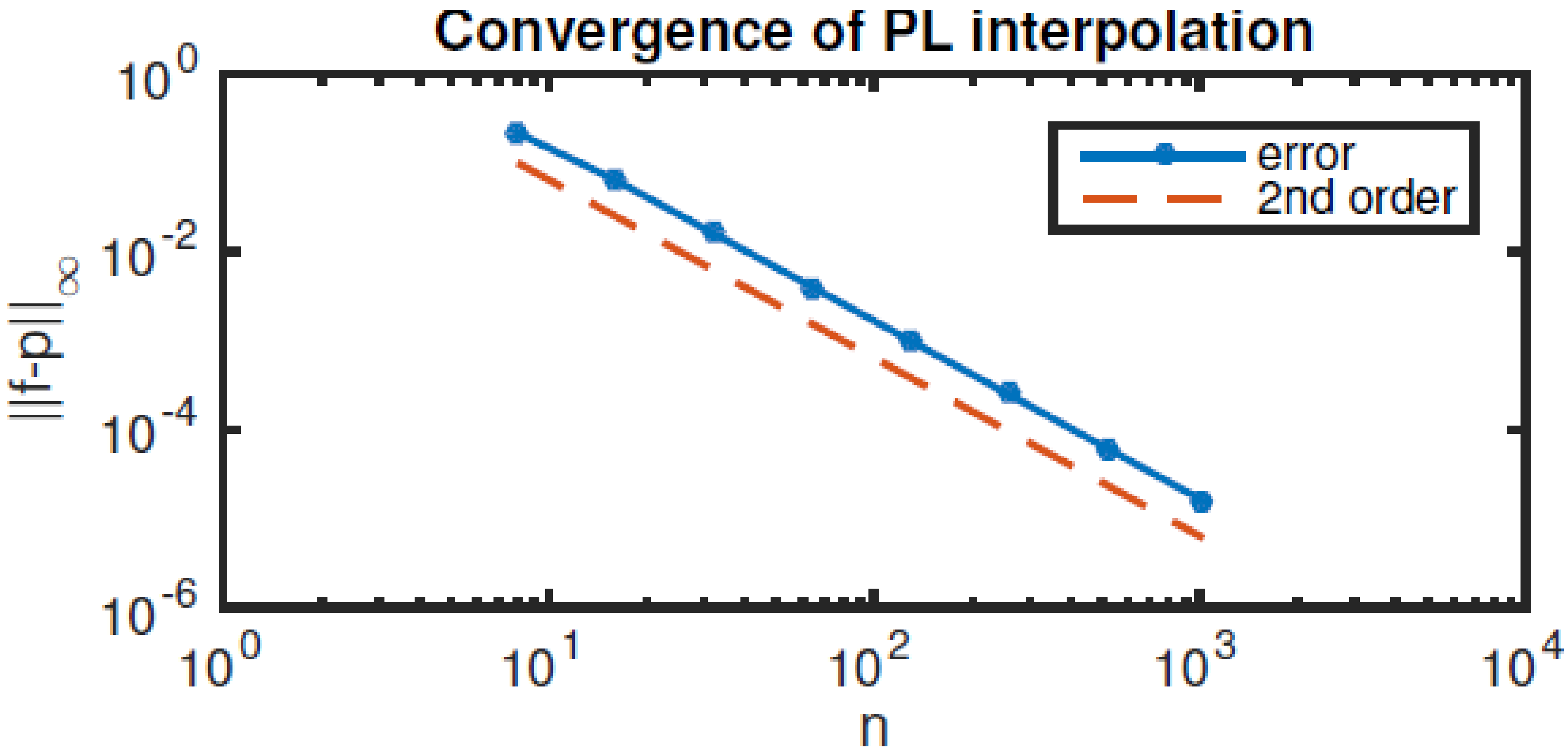
Comparison sequence

Compute error in lots of node sets

Plot error and comparison sequence

PL interpolation convergence

- Prediction of $O(h^2)$ accuracy works well!



Interpolation – cubic splines

- We created a piecewise linear function $p(x)$ of the continuous variable x that passed through, or recovers, given data
- The error was $O(h^2)$ but the derivatives of $p(x)$ weren’t smooth
- How to make a smooth and accurate interpolant?
- We will use the strategy to make small intervals and relatively low degree (n) polynomials: cubics

Interpolation – cubic splines

- Consider the data to be $n+1$ distinct points (nodes) with $(t_0, y_0), (t_1, y_1), \ldots, (t_n, y_n)$ with $t_0 < t_1 < \cdots < t_n$
- Note that the nodes $t_k, k = 0,1,\ldots,n$ still need to be distinct
- Call the interpolant $S(x)$
- This time,

$$S(x) = \bigcup_{k=1}^{n} S_k(x)$$

- For $k = 1,2,\ldots,n$,

$$S_k(x) = a_k + b_k(x - t_{k-1}) + c_k(x - t_{k-1})^2 + d_k(x - t_{k-1})^3$$

- The numbering is associated with the node at the right end of each interval

Interpolation – cubic splines

- How to connect them up?
- Over the whole interval,

$$S(x) = \bigcup_{k=1}^{n} S_k(x)$$

- Let the length of each interval be $h_k = t_k - t_{k-1}$
- We require S$(t_k) = y_k, i = 0,1, \dots, n \Longrightarrow S(x)$ passes through the data
- This has to happen at both ends of each interval
- We require that $S(x)$ be continuous, but also that $S'(x)$ and $S''(x)$ also be continuous at all of the interior nodes

Interpolation – cubic splines

- For $k = 1,2,\ldots,n$,
  $$S_k(x) = a_k + b_k(x - t_{k-1}) + c_k(x - t_{k-1})^2 + d_k(x - t_{k-1})^3$$
- Let the length of each interval be $h_k = t_k - t_{k-1}$, $k = 1,2,\ldots,n$
- Start with $S_1(x)$
- At the left end, $x = t_0$, and $S_1(t_0) = a_1 + 0 + 0 + 0 = y_0$
- At the first node, $S_1(t_1) = a_1 + b_1 h_1 + c_1 h_1^2 + d_1 h_1^3$
- For $S_2(x)$ at $x = t_1$, $S_2(t_1) = a_2 + 0 + 0 + 0 = y_1$
- At right end $x = t_2$, $S_2(t_2) = a_2 + b_2 h_2 + c_2 h_2^2 + d_2 h_2^3$
- And we can carry on for the rest of the $k = 3,\ldots,n$ …

Interpolation – cubic splines

- For $k = 1,2, \dots, n$,
  $$S_k(x) \; = \; a_k + b_k(x - t_{k-1}) + c_k(x - t_{k-1})^2 + d_k(x - t_{k-1})^3$$
- Let the length of each interval be $h_k = t_k - t_{k-1}, k = 1,2, \dots, n$
- So, for the n subintervals, there is one left endpt and one right endpt
- Left endpts: $a_k = y_{k-1}, k = 1,2, \dots, n$
- Right endpts: $a_k + b_k h_k + c_k h_k^2 + d_k h_k^3 = y_k, k = 1,2, \dots, n$
- Now we have $2n$ equations
- There are $4n$ coefficients, so we need more conditions…

Interpolation – cubic splines

- We enforced that $S(x)$ be continuous, but we also need that $S'(x)$ and $S''(x)$ also be continuous at all of the interior nodes
- For $k = 1,2, \dots, n$,
$$S'_k(x) = b_k + 2c_k(x - t_{k-1}) + 3d_k(x - t_{k-1})^2$$
- We need the slopes from each side of each interior node to be equal: $S'_k(t_k) = S'_{k+1}(t_k)$, $k = 1,2, \dots, n-1$
- This becomes $b_k + 2c_k h_k + 3d_k h_k^2 = b_{k+1}, k = 1,2, \dots, n-1$
- This is another $n - 1$ equations

Interpolation – cubic splines

- We enforced that $S(x)$ and $S'(x)$ be continuous, but we also need $S''(x)$ also be continuous at all of the interior nodes
- For $k = 1,2, \dots, n,$
$$S''_k(x) = 2c_k + 6d_k(x - t_{k-1}) + 3d_k(x - t_{k-1})^2$$
- We need the slopes from each side of each interior node to be equal: $S''_k(t_k) = S''_{k+1}(t_k)\ , k = 1,2, \dots, n - 1$
- This becomes $2c_k + 6d_k h_k = 2c_{k+1}, k = 1,2, \dots, n - 1$
- This is another $n - 1$ equations
- We are now up to 4n-2 equations, and need two more

Interpolation – cubic splines

- We enforced that $S(x), S'(x)$ and $S''(x)$ be continuous, but we need 2 more eqns
- There are several options; two common ones follow
- "Natural splines": For $k = 0, n$, use
$$S''_k(t_k) = 0$$
- "Not-a-knot splines": For $k = 1, n-1$, use
$$S'''_k(t_k) = S'''_{k+1}(t_k)$$
- Natural splines are better for some theory, but we will use Not-a-knot spines because they typically work better.
- Note that these NAK conditions effectively combine the two splines at either end.

Interpolation – cubic splines

- Our equations are then
- $a_k = y_{k-1}, k = 1,2,\dots,n$
- $a_k + b_k h_k + c_k h_k^2 + d_k h_k^3 = y\_k, k = 1,2,\dots,n$
- $b_k + 2c_k h_k + 3d_k h_k^2 - b_{k+1} = 0, k = 1,2,\dots,n-1$
- $c_k + 3d_k h_k - c_{k+1} = 0, k = 1,2,\dots,n-1$
- $d_1 = d_2,\ d_{n-1} = d_n$
- Now we have our $4n$ equations
- We need a matrix system
- First, theory

Interpolation – cubic splines

- Let f have at least four continuous derivatives on $x \in [a, b]$, $t_0 = a, t_n = b$
- Use not-a-knot cubic spline $S(x)$
- Let the length of each interval be $h_k = t_k - t_{k-1}, k = 1,2, \ldots, n$
- Then, $\|S - f\|_\infty = O(h^4)$, where $h = \max\{h_1, \ldots, h_n\}$

Interpolation – cubic splines

- We need some definitions
- Let $\boldsymbol{a} = [a_1\ a_2\ \ldots\ a_n]^T$ (column vector)
- Define $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$ similarly
- We seek the unknowns $\boldsymbol{z} = [\boldsymbol{a};\ \boldsymbol{b};\ \boldsymbol{c};\ \boldsymbol{d}]$ (column vector)
- Let ***H*** = diag($[h_1\ h_2\ \ldots\ h_n]$); $\boldsymbol{I}, \boldsymbol{0}$ are $n \times n$ matrices
- For the *n* equations, $a_k = y_{k-1}$ we can write

$$[\boldsymbol{I}\ \boldsymbol{0}\ \boldsymbol{0}\ \boldsymbol{0}]\boldsymbol{z} = [y_0 \ldots y_{n-1}]^T$$

- For the n equations, $a_k + b_k h_k + c_k h_k^2 + d_k h_k^3 = y_k$ we can write

$$[\boldsymbol{I}\ \boldsymbol{H}\ \boldsymbol{H}^2\ \boldsymbol{H}^3]\boldsymbol{z} = [y_1 \ldots y_n]^T$$

- These are the first *2n* rows of the system for $\boldsymbol{z}$

Interpolation – cubic splines

- For the $n-1$ first derivative eqns $b_k + 2c_k h_k + 3d_k h_k^2 - b_{k+1} = 0$,
$$\boldsymbol{E}[\boldsymbol{0}\ \ \boldsymbol{I}-\boldsymbol{J}\ \ 2\boldsymbol{H}\ 3\boldsymbol{H}^2]\boldsymbol{z} = \boldsymbol{0}$$
- $\boldsymbol{0}$ is a $n \times 1$ vector here
- Here

$$\mathrm{E} = \begin{bmatrix} 1 & 0 & & & \\ & 1 & 0 & & \\ & & \ddots & \ddots & \\ & & & 1 & 0 \end{bmatrix} \qquad \mathrm{J} = \begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & \ddots & \ddots & \\ & & & 0 & 1 \\ & & & & 0 \end{bmatrix}$$

- This is the next $n-1$ rows

Interpolation – cubic splines

- For the $n-1$ second derivative eqns $c_k + 3d_k h_k - c_{k+1} = 0$,
$$\boldsymbol{E}[\mathbf{0}\ \ \mathbf{0}\ \ \boldsymbol{I}-\boldsymbol{J}\ \ 3\boldsymbol{H}\ ]\boldsymbol{z}\ = \mathbf{0}$$
- We need two row appropriate row vectors to apply the last two equations.
- Putting all the equations together into a single matrix system is called assembling the equations
- This is implemented in `spinterp.m`

Cubic splines function

- Build up parts of system for all coefficients in A's
- RHS is v's
- Last two lines are two rows for NAK conditions

```
% Building blocks
I = eye(n);  E = I(1:n-1,:);
J = diag(ones(n-1,1),1);
H = diag(h);

% Left endpoint interpolation
AL = [ I, 0*I, 0*I, 0*I ];
vL = y(1:n);

% Right endpoint interpolation
AR = [ I, H, H^2, H^3 ];
vR = y(2:n+1);

% Continuity of first derivative
A1 = E*[ 0*I, I-J, 2*H, 3*H^2 ];
v1 = zeros(n-1,1);

% Continuity of second derivative
A2 = E*[ 0*I, 0*I, I-J, 3*H ];
v2 = zeros(n-1,1);

% Not-a-knot conditions
nakL = [ zeros(1,3*n), [1,-1, zeros(1,n-2)] ];
nakR = [ zeros(1,3*n), [zeros(1,n-2), 1,-1] ];
```

Cubic splines function

- Assemble the A's into a single matrix
- Same for RHS with v's
- Solve
- Break up coefficients and evaluate with polyval

```
% Assemble and solve the full system
A = [ AL; AR; A1; A2; nakL; nakR ];
v = [ vL; vR; v1; v2; 0 ;0 ];
z = A\v;

% Break the coefficients into separate vectors.
rows = 1:n;
a = z(rows);
b = z(n+rows);  c = z(2*n+rows);  d = z(3*n+rows);
S = @evalspline;

% Evaluate the individual cubic pieces.
  function p = evalspline(x)
    p = zeros(size(x));
    for k = 1:n
        index = (x>=t(k)) & (x<=t(k+1));
        p(index) = polyval( [d(k),c(k),b(k),a(k)], x(index)-t(k) );
    end
  end
```

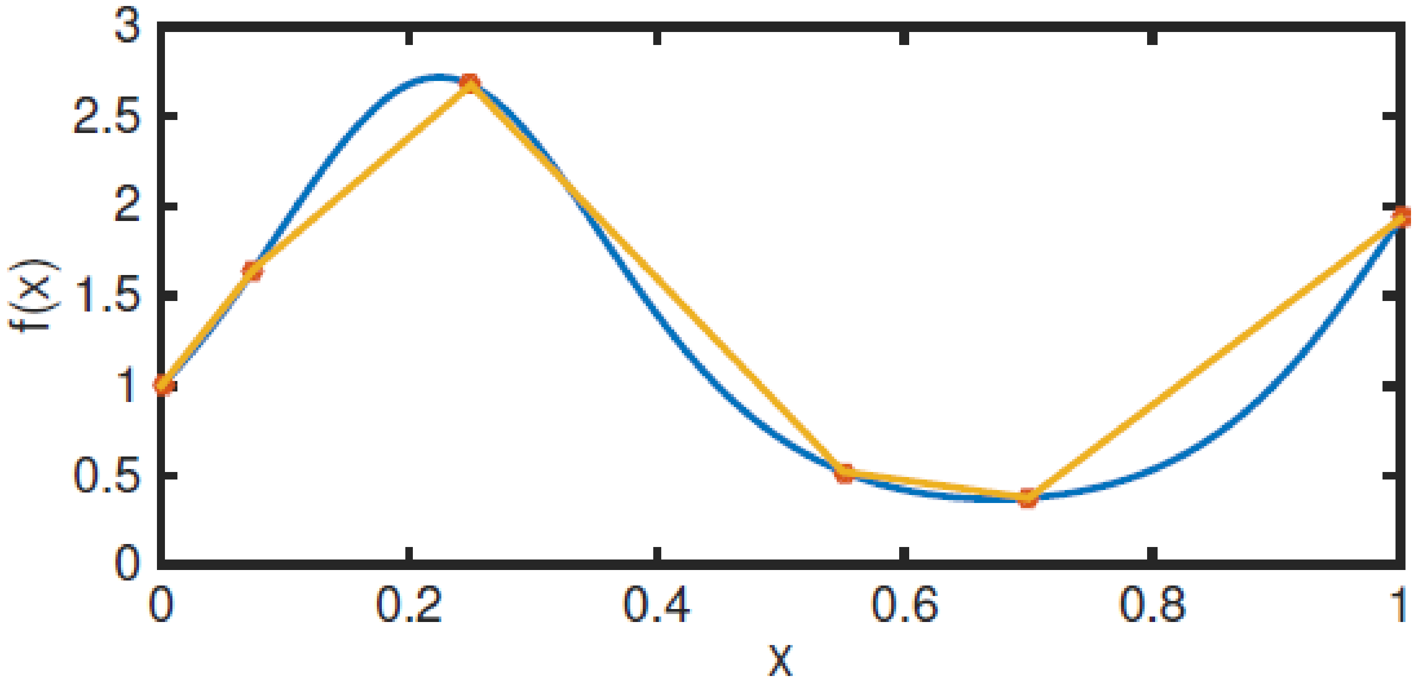
Cubic splines -- example

- Return to f(x)=exp(sin(7x))
- Use 6 nodes again, t = [0, 0.075, 0.25, 0.55, 0.7, 1]
- Do PL interpolation again for comparison
- Then cubic spline and superimpose

Cubic splines example

PL first for comparison

```
f = @(x) exp(sin(7*x));   % example function
fplot(f,[0,1])
xlabel('x'), ylabel('f(x)')

t = [0 0.075 0.25 0.55 0.7 1]';
y = f(t);           % create
hold on
plot(t,y,'o')

p = @(x) plinterp(x,t,y);
fplot(p,[0 1])         % plo
```

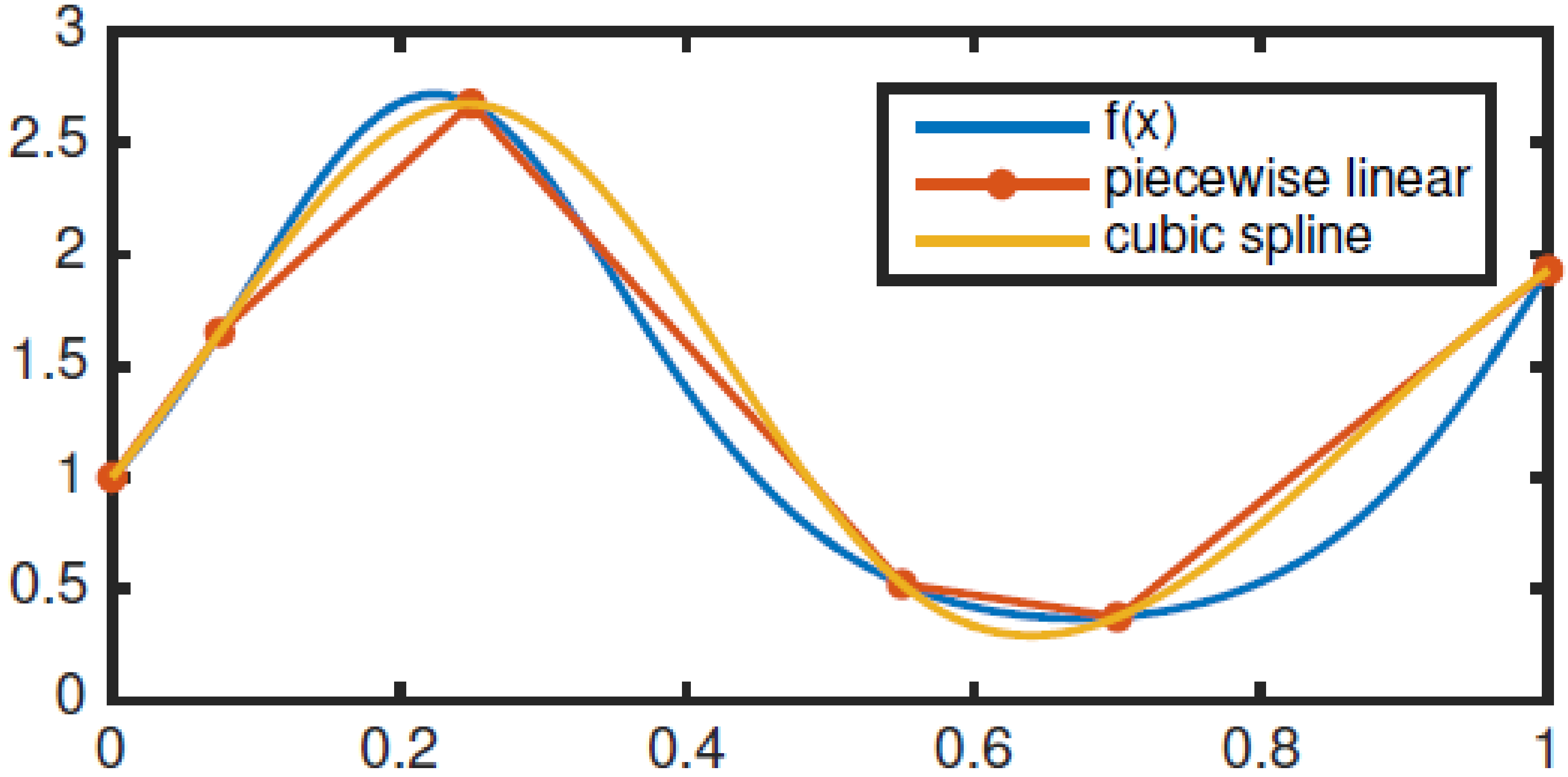


Cubic splines example

Now cubic spline

- Better, but not super
- Try to refine a little

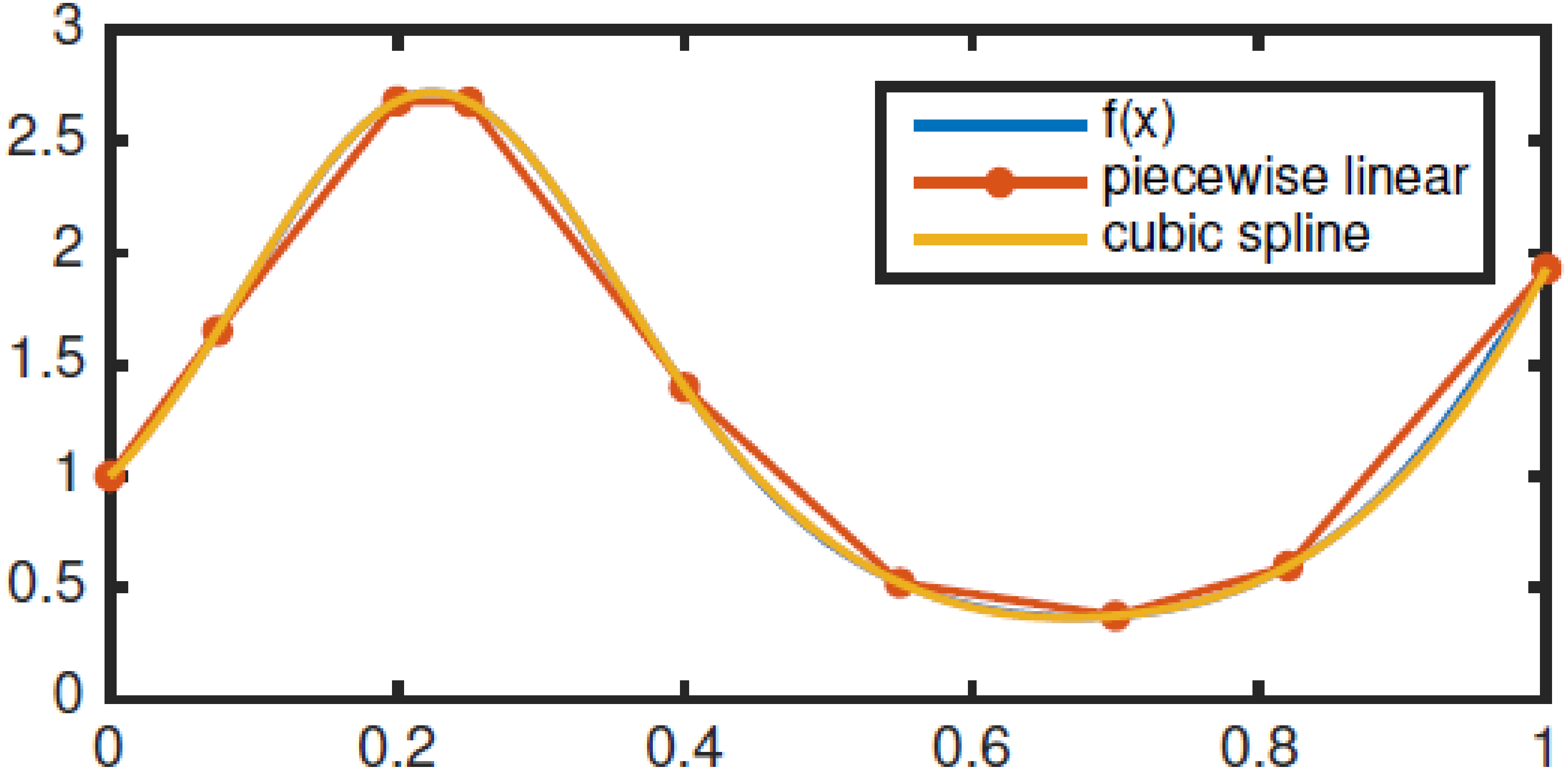
```
S = spinterp(t,y);
fplot(S,[0,1])
legend('f(x)','piecewise linear','cubic spline')
```



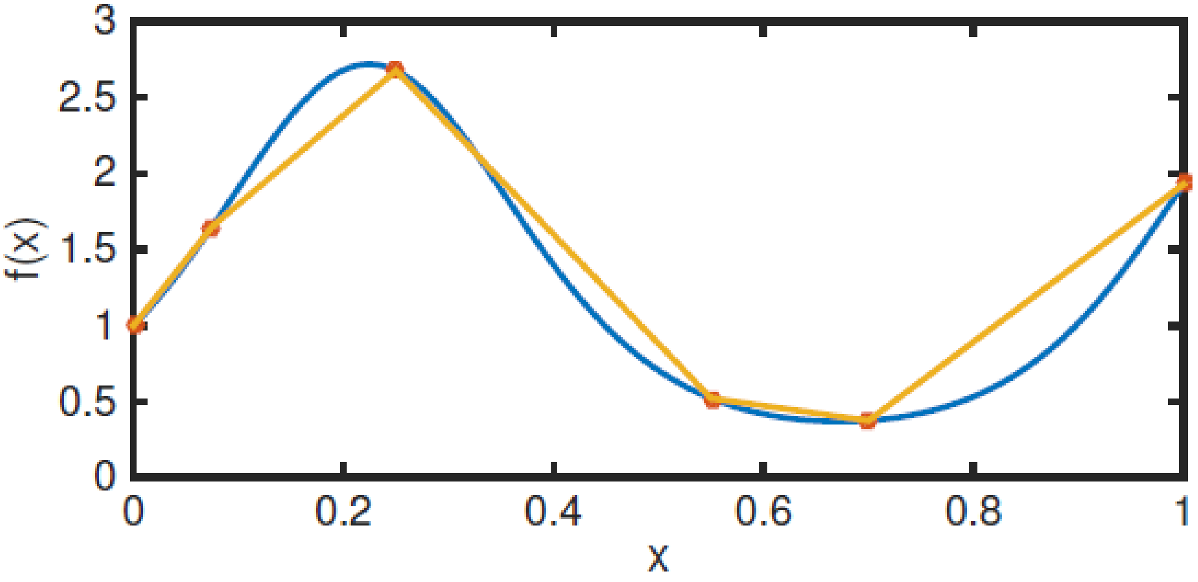
Cubic splines example

Now with more points

```
t = [0, 0.075, 0.2, 0.25, 0.4, 0.55, 0.7, 0.82, 1];
y = f(t);
clf, fplot(f,[0,1])
hold on, plot(t,y,'.-')
S = spinterp(t,y);
fplot(S,[0,1])
legend('f(x)','piecewise linear','cubic spline')
```



Differentiation



Differentiation: finite differences

- Consider approximating derivatives of given function *f(x)*
- We want to do this at discrete points like the nodes
- Assume even spacing, with $a = t_0, b = t_n, h = (b - a)/n$, and $t_i = a + ih, \quad i = 0,1, \dots , n$
- As we have discussed previously, the definition of a derivative is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

- One can approximate the derivative $f'(x)$ with small but finite h in a lot of ways

Differentiation: finite differences

- A general form for approximating the derivatives is

$$f'(t_i) \approx \frac{1}{h} \sum_{k=-p}^{q} a_k f(t_i + kh),$$

- This approximates the derivative at grid point (node) $x = t_i$
- We need to specify p, q which is how many neighboring point to use
- Also need the weights a_k, which need one for each grid point in the range of k
- For convenience, we can make shift the independent variable with $s = x - t_i$, so that we are approximating $\tilde{f}'(s)$ at $s = 0$

Differentiation: finite differences

- A general form for approximating the derivatives is

$$f'(t_i) \approx \frac{1}{h} \sum_{k=-p}^{q} a_k f(t_i + kh),$$

- With $s = x - t_i$, so that we approximate $\tilde{f}'(s)$ at $s = 0$
- The neighboring points are then at kh
- The formula becomes (no chain rule terms needed)

$$f'(t_i) = \tilde{f}'(0) \approx \frac{1}{h} \sum_{k=i-p}^{i+q} a_k \tilde{f}(kh).$$

Differentiation: finite differences

- We don't have to evaluate at nodes; we can use arbitrary x

$$f'(x) \approx \frac{1}{h} \sum_{k=-p}^{q} a_k f(x+kh).$$

- Examples using two grid points: two point formulae
- Forward: $p = 0, q = 1, a_0 = -1, a_1 = 1$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

Differentiation: finite differences

- We don't have to evaluate at nodes; we can use arbitrary x

$$f'(x) \approx \frac{1}{h} \sum_{k=-p}^{q} a_k f(x + kh).$$

- Examples using two grid points: two point formulae
- Forward: $p = 0, q = 1, a_0 = -1, a_1 = 1$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

- Backward: $p = 1, q = 0, a_{-1} = -1, a_0 = 1$

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Finite differences: accuracy

- How accurate are these formulas?
- Define the truncation error

$$\tau_f(h) = \left| f'(0) - \frac{1}{h} \sum_{k=i-p}^{i+q} a_k f(kh) \right|$$

- To evaluate what the truncation error is, Taylor expand each value of $f(kh)$ about 0 ($k = 0$)
- There is cancellation of many terms, and the largest remaining term is the order of the error

$$f'(t_i) = \tilde{f}'(0) \approx \frac{1}{h} \sum_{k=i-p}^{i+q} a_k \tilde{f}(kh).$$

Finite differences: accuracy

- Taylor expanding about 0 and substituting gives

$$\begin{aligned}
\tau_f(h) &= \left| f'(0) - \frac{f(h) - f(0)}{h} \right| \\
&= \left| f'(0) - h^{-1} \left[\left(f(0) + hf'(0) + \tfrac{1}{2}h^2 f''(0) + \cdots \right) - f(0) \right] \right| \\
&= -\frac{1}{2} h f''(0) + O(h^2).
\end{aligned}$$

- The biggest term in the error decreases proportional to h: $O(h)$ error or first order error
- Can we do better?

Finite differences: accuracy

- To get a better approximation, use more points
- To do this, first use three points and interpolate them with a quadratic

$$P(x) = \frac{x(x-h)}{2h^2} f(-h) - \frac{x^2 - h^2}{h^2} f(0) + \frac{x(x+h)}{2h^2} f(h)$$

- Differentiating and setting x=0 gives the following:

$$f'(0) \approx P'(0) = \frac{-1}{2h} f(-h) + 0 f(0) + \frac{1}{2h} f(h)$$

- More neatly,

$$f'(0) \approx \frac{f(h) - f(-h)}{2h}$$

Finite differences: accuracy

- Centered difference formula (3-pt formula)

$$f'(0) \approx \frac{f(h) - f(-h)}{2h}$$

- How about the error? Taylor expand about $x = 0$ again

$$\begin{aligned}
\tau_f(h) &= \left| f'(0) - \frac{f(h) - f(-h)}{2h} \right| \\
&= \left| f'(0) - (2h)^{-1} \left[\left(f(0) + hf'(0) + \tfrac{1}{2}h^2 f''(0) + O(h^3) \right) - \left(f(0) - hf'(0) + \tfrac{1}{2}h^2 f''(0) + O(h^3) \right) \right] \right| \\
&= (2h)^{-1} \cdot O(h^3) = O(h^2).
\end{aligned}$$

- The error is now $O(h^2)$ (the h^3 terms don't cancel)

Exploring accuracy

- What difference does this make in the error?
- Consider $f(x) = \sin(e^{x+1})$
- Exact derivative is $f'(x) = \cos(e^{x+1})$
- Test them with two formulas at $x = 0$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \qquad f'(0) \approx \frac{f(h) - f(-h)}{2h}$$

- Try it in Matlab

Finite differences: accuracy test

- Setting up the functions;

```
% define functions, spacings, arrays
f = @(x) sin( exp(x+1) );
exact = cos( exp(1) )*exp(1) ;
h = 2.^(-1:-1:-16);   % spacing is halved each time
err1 = 0*h; err2 = 0*h;
```

- h changes by factor of 2 each step
- Easy to see order in that case

Finite differences: accuracy test

- Calculate the errors

```
% calculate error
for k = 1:length(h)
    h_ = h(k);
    FD1 = (f(h_) - f (0) ) / h_;
    err1 (k) = abs( exact - FD1 );
    FD2 = (f(h_) - f(-h_)) / (2*h_);
    err2 (k) = abs( exact - FD2 );
end
```

Finite differences: accuracy test

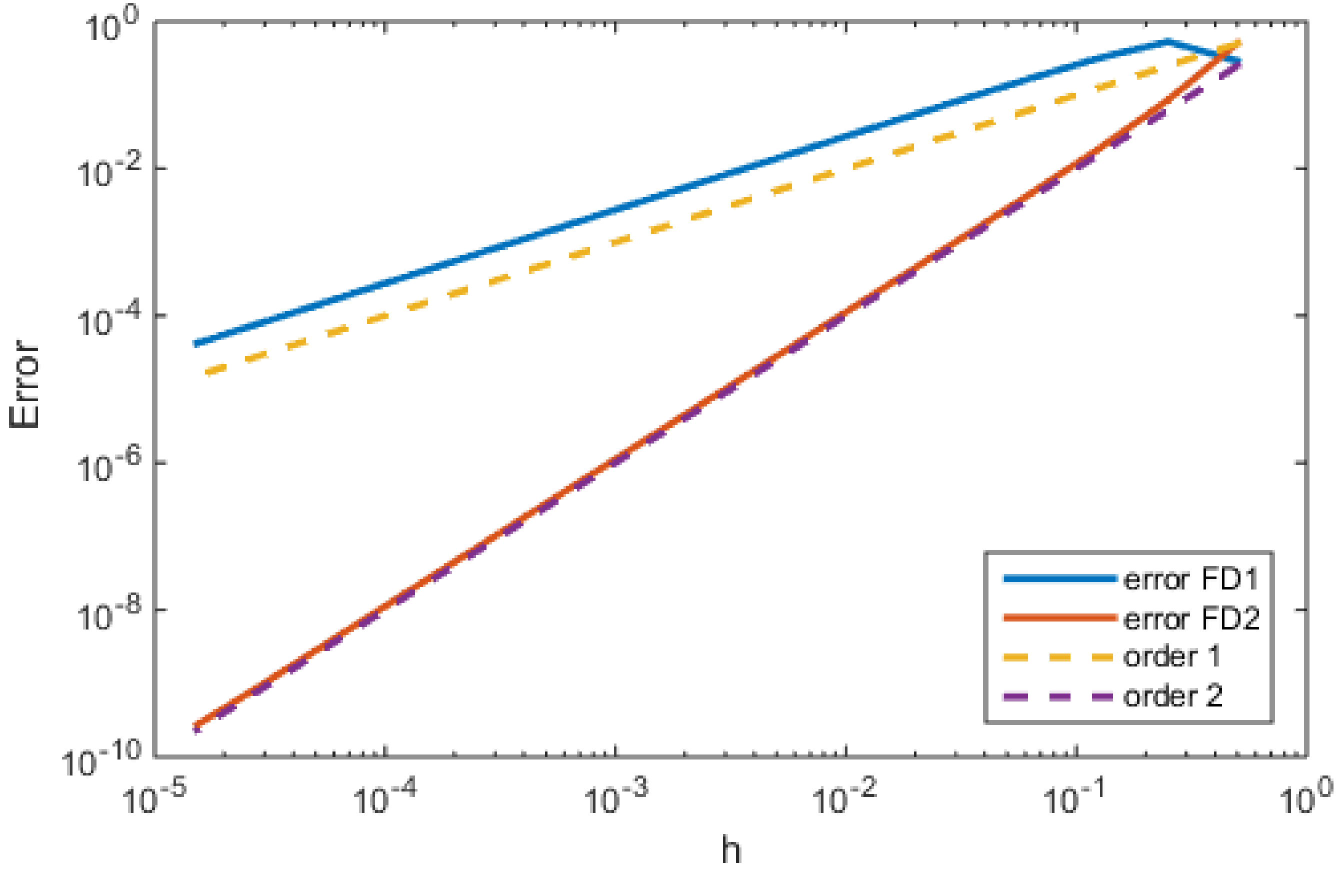
- Calculate the errors and ratios

```
% calculate error
for k = 1:length(h)
    h_ = h(k);
    FD1 = (f(h_) - f (0) ) / h_;
    err1 (k) = abs( exact - FD1 );
    FD2 = (f(h_) - f(-h_)) / (2*h_);
    err2 (k) = abs( exact - FD2 );
end


% compute ratios and plot
factorFD1 = err1(1:end-1)'./err1(2: end)';
factorFD2 = err2(1:end-1)'./err2(2: end)';
hh = h(2:end)';
table(hh,factorFD1,factorFD2)
```

hh	factorFD1	factorFD2
2.5000e-01	5.4588e-01	5.9126e+00
1.2500e-01	1.6761e+00	4.6355e+00
6.2500e-02	1.9022e+00	4.1699e+00
3.1250e-02	1.9638e+00	4.0432e+00
1.5625e-02	1.9847e+00	4.0108e+00
7.8125e-03	1.9930e+00	4.0027e+00
3.9063e-03	1.9967e+00	4.0007e+00
1.9531e-03	1.9984e+00	4.0002e+00
9.7656e-04	1.9992e+00	4.0000e+00
4.8828e-04	1.9996e+00	4.0000e+00
2.4414e-04	1.9998e+00	4.0000e+00
1.2207e-04	1.9999e+00	4.0001e+00
6.1035e-05	2.0000e+00	4.0008e+00
3.0518e-05	2.0000e+00	3.9905e+00
1.5259e-05	2.0000e+00	3.8383e+00

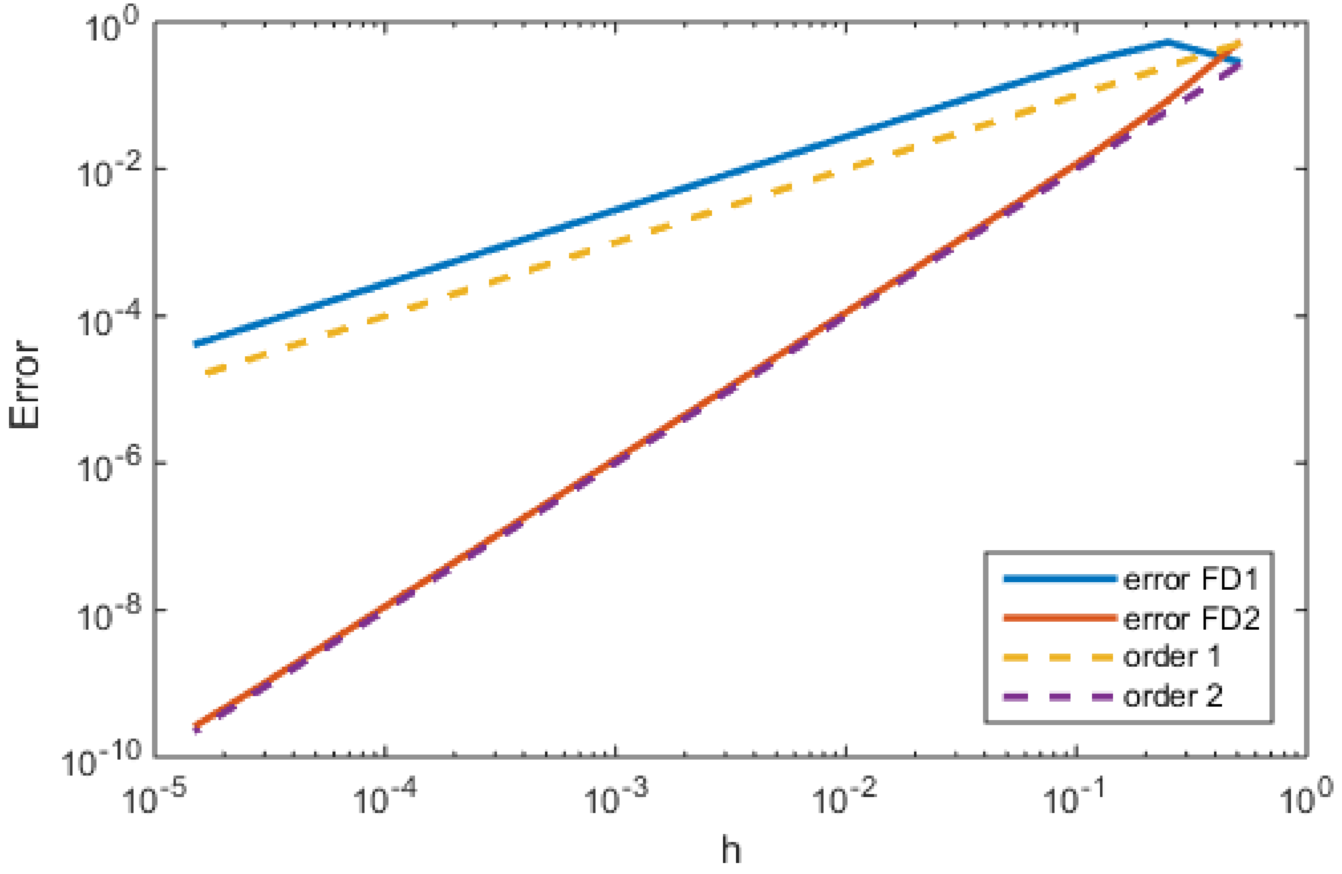
Finite differences: accuracy test

- Plot the errors using loglog
- Slope of $O(h)$ is 1
- Slot of $O(h^2)$ is 2
- 2nd order accuracy significantly better



Finite differences: accuracy test

- For exploration: What happens for h smaller than the values used here?
- Can we increase the order of accuracy further?
- Use non-centered formulas?



Finite differences: more accuracy, one-sided

- The general form:

$$f'(x) \approx \frac{1}{h} \sum_{k=-p}^{q} a_k f(x + kh)$$

- If p=0, forward difference
- If q=0, backward difference
- Table at right is for finite difference formulas
- A transformation of $h \rightarrow -h$ will switch the forward difference to the backward difference

Order of accuracy	Node location 0	h	$2h$	$3h$	$4h$
1	-1	1			
2	$-\frac{3}{2}$	2	$-\frac{1}{2}$		
3	$-\frac{11}{6}$	3	$-\frac{3}{2}$	$\frac{1}{3}$	
4	$-\frac{25}{12}$	4	-3	$\frac{4}{3}$	$-\frac{1}{4}$

Finite differences: Higher order derivatives

- We can start with quadratic polynomial

$$P(x) = \frac{x(x-h)}{2h^2}f(-h) - \frac{x^2-h^2}{h^2}f(0) + \frac{x(x+h)}{2h^2}f(h)$$

- Differentiating twice results in the formula

$$f''(0) \approx \frac{f(-h) - 2f(0) + f(h)}{h^2}$$

- Error is $O(h^2)$
- This is true for centered formula, but not for one-sided
- Can derive using Taylor expansion or Lagrange interpolating polynomial

Higher order derivatives, one sided

- Using a quadratic interpolating polynomial

$$f''(0) = \frac{f(0) - 2f(h) + f(2h)}{h^2} + O(h)$$

- But using a cubic interpolating polynomial gives

$$f''(0) = \frac{2f(0) - 5f(h) + 4f(2h) - f(3h)}{h^2} + O(h^2)$$

- Better accuracy for latter but more points needed
- Additional concerns can arise for differential equations, e.g.

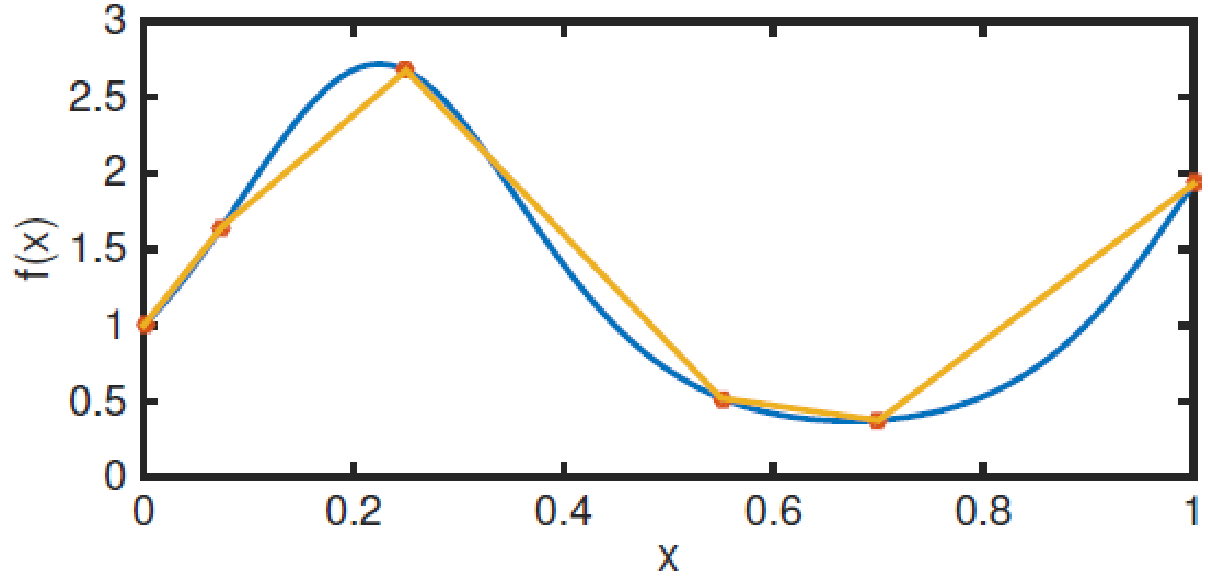
General finite differences

- General approach beyond this text
- Function 5.3 (fdweights.m) calculates derivative approximations for the requested order of the derivative and accuracy for node locations that need not be uniformly spaced.
- It uses Fornberg’s method.

Finite difference error

- Try out script with wider range of h
- That is, decrease h to very small values, even to eps
- What happens as h decreases? Why?

Differentiation matrices



Derivative matrices

- Say we have a list of function values:
- These are located at grid or mesh points

$$t_i = a + ih, \qquad i = 0, \ldots, n.$$

$$\mathbf{f} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ \vdots \\ f(t_{n-1}) \\ f(t_n) \end{bmatrix}$$

- We can calculate derivatives at each grid point in one operation if we premultiply by the correct matrix
- This is a very handy operation

Derivative matrices

- We want to compute a vector $\boldsymbol{g}$ where $g_i \approx f'(t_i)$
- Try this with the forward difference formula, with $x = t_i, i = 0,1, \dots, n$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

- This works great for $i = 0,1, \dots, n-1$ but we can do the same thing at $i = n$
- There, just use a backward formula, which ends up just the same as for $i = n - 1$

$$\mathbf{f} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ \vdots \\ f(t_{n-1}) \\ f(t_n) \end{bmatrix}$$

$$g_i = \frac{f_{i+1} - f_i}{h}$$

$$g_n = \frac{f_n - f_{n-1}}{h}$$

Finite differences: accuracy test

- Putting all the rows together gives:

$$\begin{bmatrix} f'(t_0) \\ f'(t_1) \\ \vdots \\ f'(t_{n-1}) \\ f'(t_n) \end{bmatrix} \approx \frac{1}{h} \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & -1 & 1 \end{bmatrix} \begin{bmatrix} f(t_0) \\ f(t_1) \\ \vdots \\ f(t_{n-1}) \\ f(t_n) \end{bmatrix}, \qquad \text{or} \qquad \mathbf{f}' = \mathbf{D}_x \mathbf{f}.$$

- Then, we get a first order accurate approximation at all grid points from premultiplying by differentiation matrix $\boldsymbol{D}_x$

Finite differences: accuracy test

- What if we wanted $O(h^2)$ accuracy?

$$\mathbf{D}_x = \frac{1}{h}\begin{bmatrix} -1 & 1 & & & & & \\ -\frac{1}{2} & 0 & \frac{1}{2} & & & & \\ & -\frac{1}{2} & 0 & \frac{1}{2} & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & -\frac{1}{2} & 0 & \frac{1}{2} \\ & & & & & -1 & 1 \end{bmatrix}$$

- This matrix gives second order accurate approximations at interior points
- Ends are still only first order because we can't apply centered formula at either end

Finite differences: accuracy test

- What if we wanted $O(h^2)$ accuracy at the ends too?
- Use one-sided three point formulas at each end

$$
\mathrm{D}_x = \frac{1}{h}\begin{bmatrix}
-\frac{3}{2} & 2 & -\frac{1}{2} & & & & \\
-\frac{1}{2} & 0 & \frac{1}{2} & & & & \\
 & -\frac{1}{2} & 0 & \frac{1}{2} & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & & -\frac{1}{2} & 0 & \frac{1}{2} \\
 & & & & \frac{1}{2} & -2 & \frac{3}{2}
\end{bmatrix}
$$

- This matrix gives second order accurate approximations at all grid points

2^{nd} derivative matrix

- For second derivative and $O(h^2)$ accuracy at all points:
- Use one-sided four point formulas at each end

$$\begin{bmatrix} f''(t_0) \\ f''(t_1) \\ f''(t_2) \\ \vdots \\ f''(t_{n-1}) \\ f''(t_n) \end{bmatrix} \approx \frac{1}{h^2} \begin{bmatrix} 2 & -5 & 4 & -1 & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & 1 & -2 & 1 \\ & & & -1 & 4 & -5 & 2 \end{bmatrix} \begin{bmatrix} f(t_0) \\ f(t_1) \\ f(t_2) \\ \vdots \\ f(t_{n-1}) \\ f(t_n) \end{bmatrix} = \mathbf{D}_{xx}\mathbf{f}.$$

- This matrix gives second order accurate approximations at all grid points

Derivative matrices

- In both of the cases we saw increasing the accuracy led to more nonzero diagonals
- If we increase the order of accuracy more, this trend continues
- Function diffmats.m from text will build the second order accurate first and second derivative matrices (most commonly used)

```
function [t,Dx,Dxx] = diffmats(a,b,n)
%DIFFMATS Differentiation matrices.
```