Chapter 5

Piecewise Interpolation and Calculus

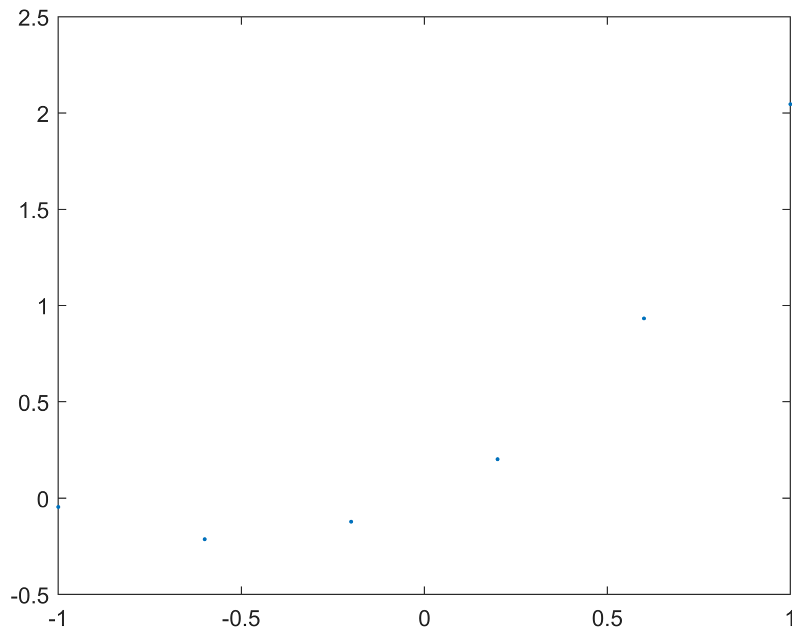
*This is a MATLAB Live Editor version of the **Fundamentals of Numerical Computation** Jupyter Notebooks found at https://tobydriscoll.net/project/fnc/.*

Polynomials

Example 5.1.1

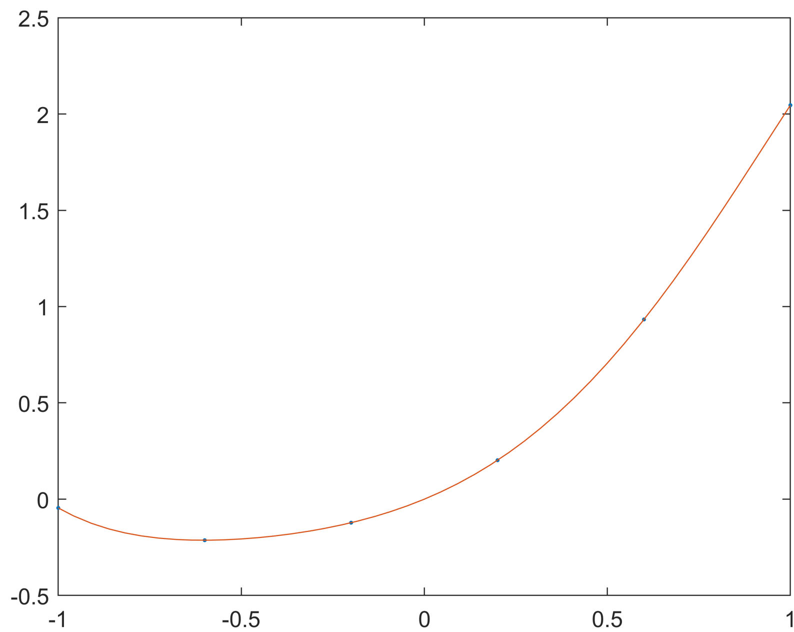
Here are some points that we could consider to be observations of an unknown function on $[-1,1]$.

```
clear
n = 5;
t = linspace(-1,1,n+1)';  y = t.^2 + t + 0.05*sin(20*t);
plot(t,y,'.')
```



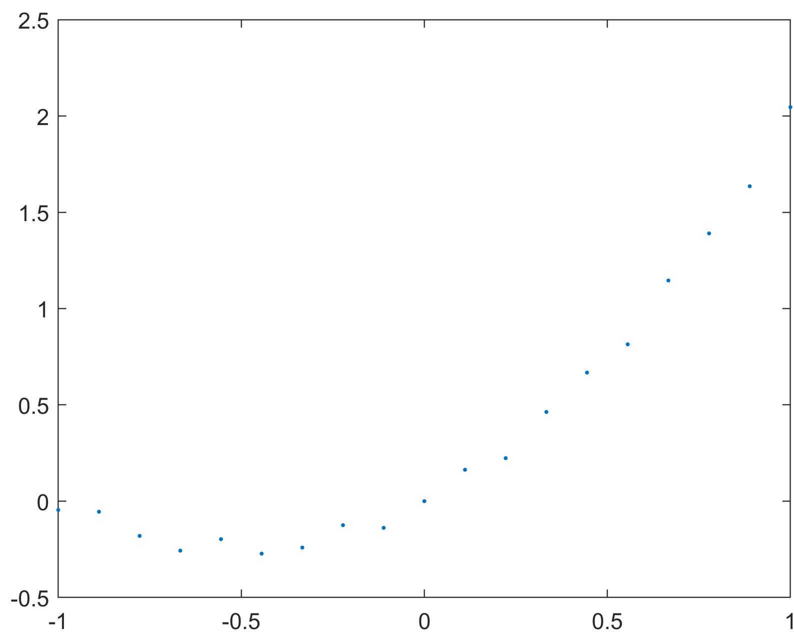
The polynomial interpolant, as computed using `polyfit`, looks very sensible. It's the kind of function you'd take home to meet your parents. The interpolant plot is created using `fplot`.

```
c = polyfit(t,y,n);     % polynomial coefficients
p = @(x) polyval(c,x);
hold on, fplot(p,[-1 1])
```



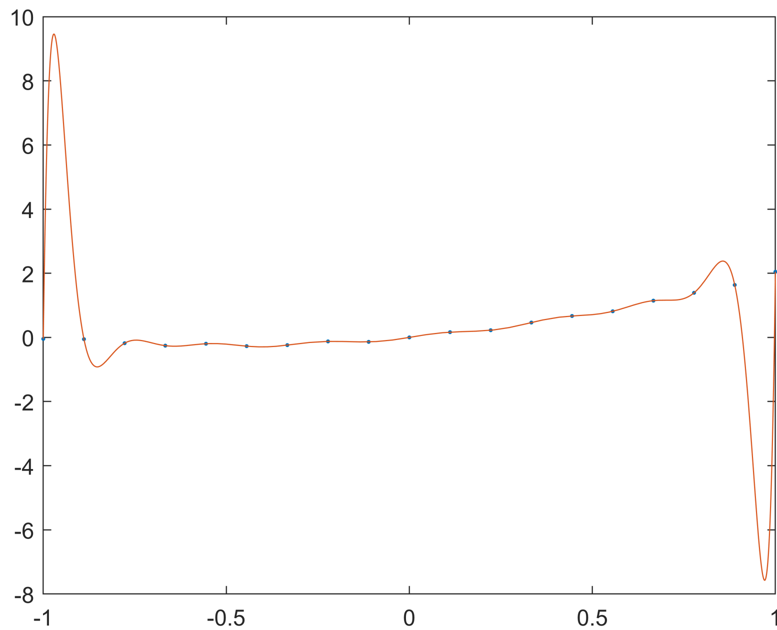
But now consider a different set of points generated in almost exactly the same way.

```
n = 18;
t = linspace(-1,1,n+1)';  y = t.^2 + t + 0.05*sin(20*t);
clf, plot(t,y,'.')
```



The points themselves are unremarkable. But take a look at what happens to the polynomial interpolant.

```
c = polyfit(t,y,n);     % polynomial coefficients
p = @(x) polyval(c,x);
hold on, fplot(p,[-1 1])
```

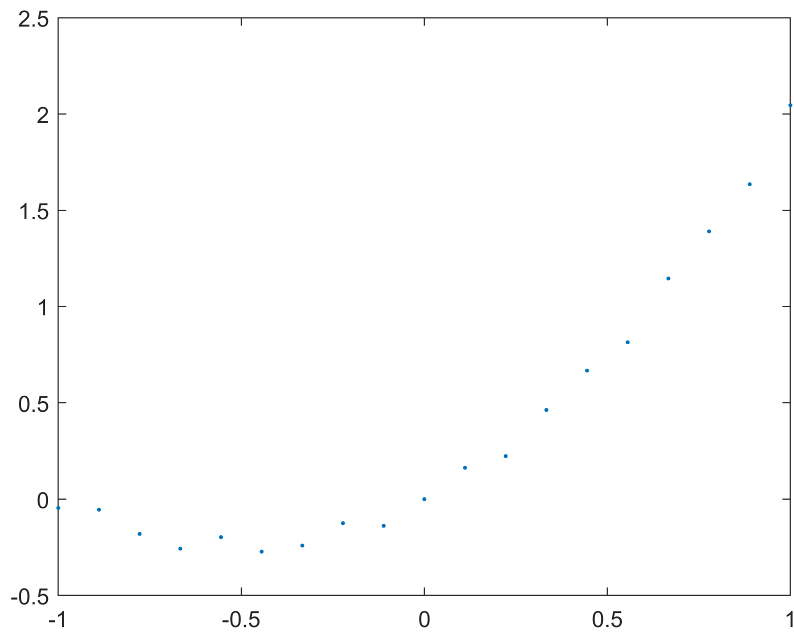


Surely there must be functions that are more intuitively representative of those points!

Piecewise Polynomials

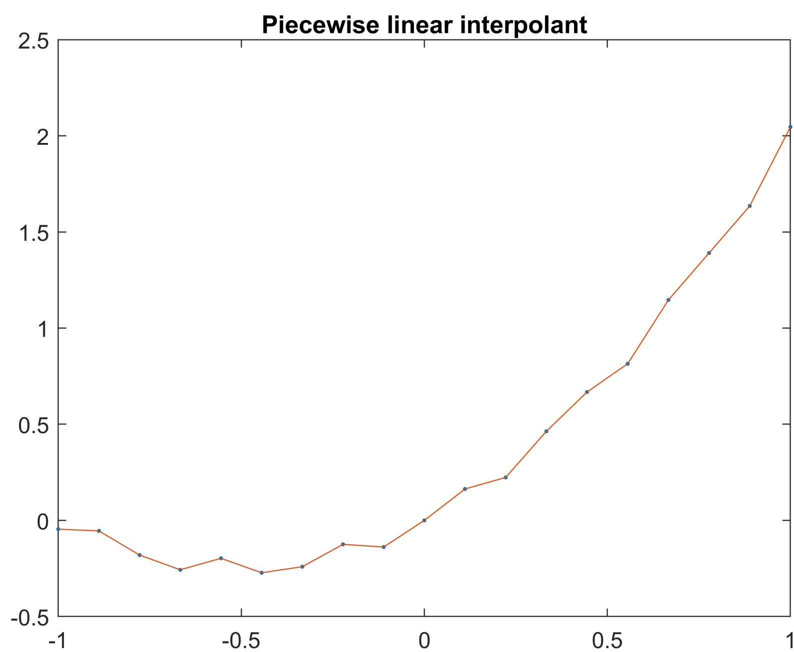
Example 5.1.3

```
n = 18;
t = linspace(-1,1,n+1)';
y = t.^2 + t + 0.05*sin(20*t);
clf, plot(t,y,'.')
```



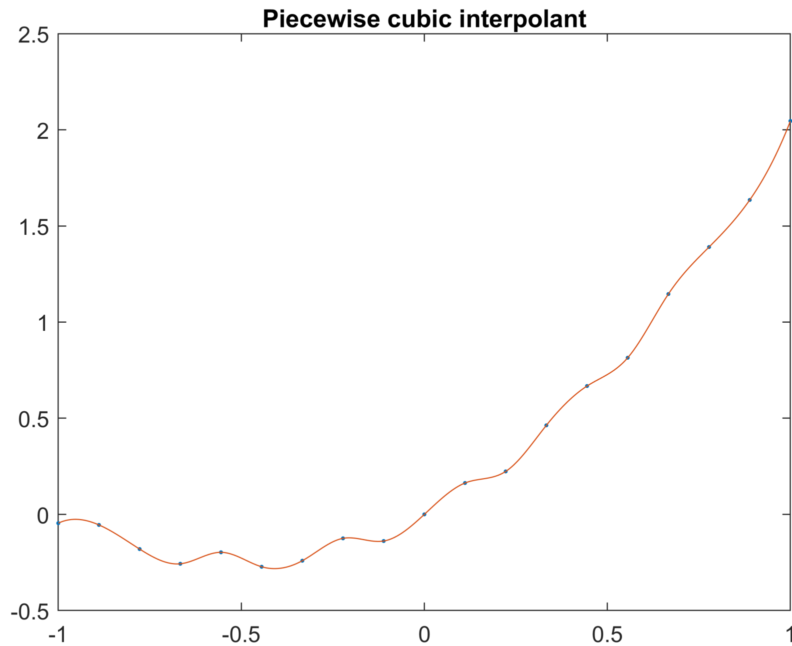
By default `interp1` gives us an interpolating function that is linear between each pair of consecutive nodes.

```
x = linspace(-1,1,400)';
hold on, plot(x,interp1(t,y,x))
title('Piecewise linear interpolant')    % ignore this line
```



We may instead request a smoother interpolant that is piecewise cubic.

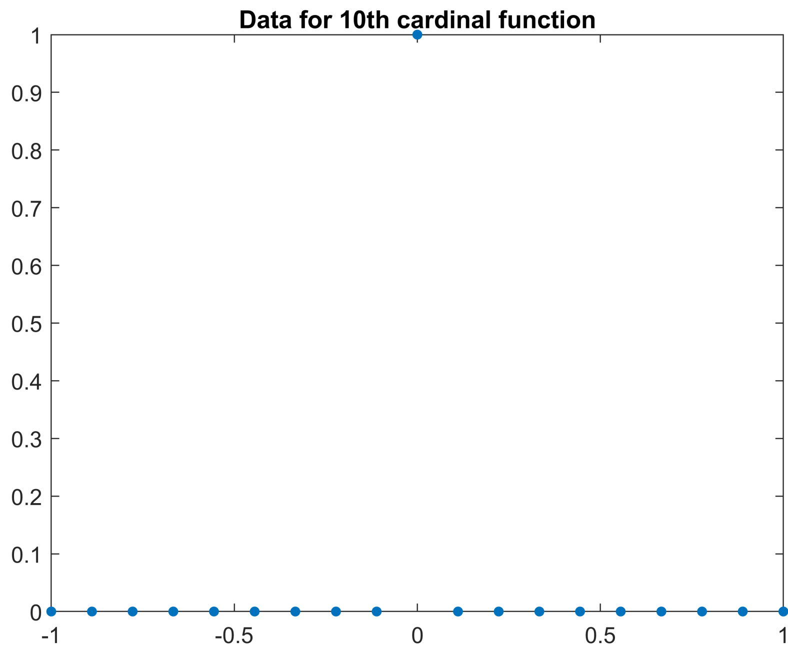
```
cla
plot(t,y,'.')
plot(x,interp1(t,y,x,'spline'))
title('Piecewise cubic interpolant')   % ignore this line
```



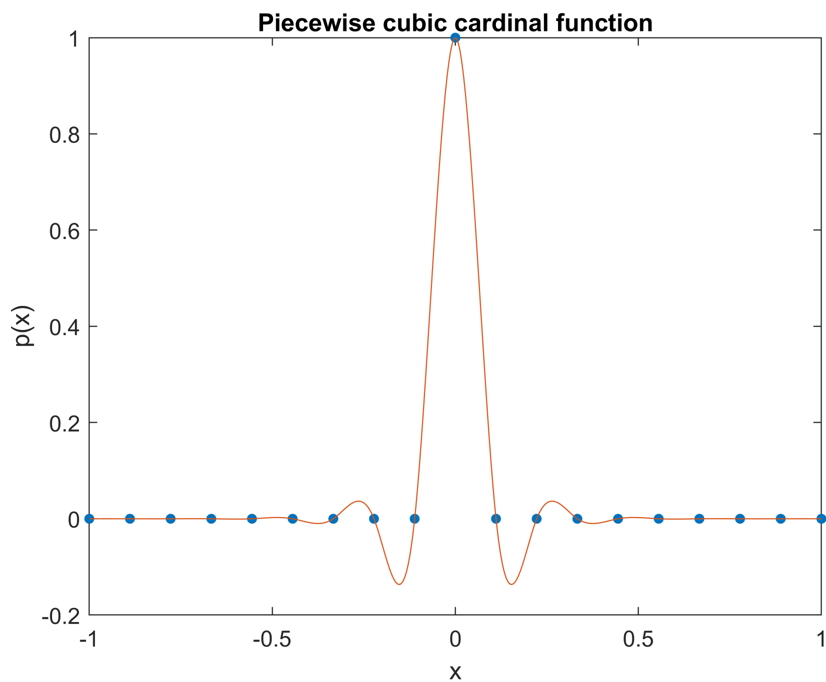
Conditioning of Interpolation

Example 5.1.4

```
figure
n = 18;
t = linspace(-1,1,n+1)';
y = [zeros(9,1);1;zeros(n-9,1)];  % 10th cardinal function
plot(t,y,'.','MarkerSize',15)
title('Data for 10th cardinal function')
```

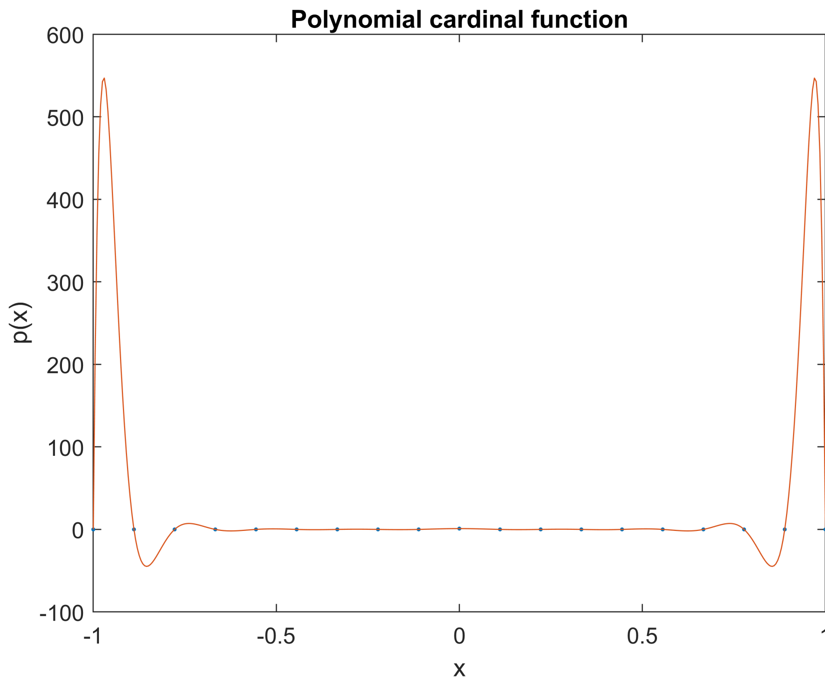



```
hold on
x = linspace(-1,1,400)';
plot(x,interp1(t,y,x,'spline'))
title('Piecewise cubic cardinal function')   % ignore this line
xlabel('x'), ylabel('p(x)')   % ignore this line
hold off
```



The piecewise cubic cardinal function is nowhere greater than one in absolute value. This happens to be true for all the cardinal functions, ensuring a good condition number for the interpolation. But the story for global polynomials is very different.

```
clf, plot(t,y,'.')    % ignore this line
c = polyfit(t,y,n);
hold on,
plot(x,polyval(c,x))
title('Polynomial cardinal function')     % ignore this line
xlabel('x'), ylabel('p(x)')   % ignore this line
hold off
```



From the figure we can see that the condition number for polynomial interpolation on these nodes is at least 500.

Piecewise Linear Interpolation

Hat Functions

Example 5.2.1

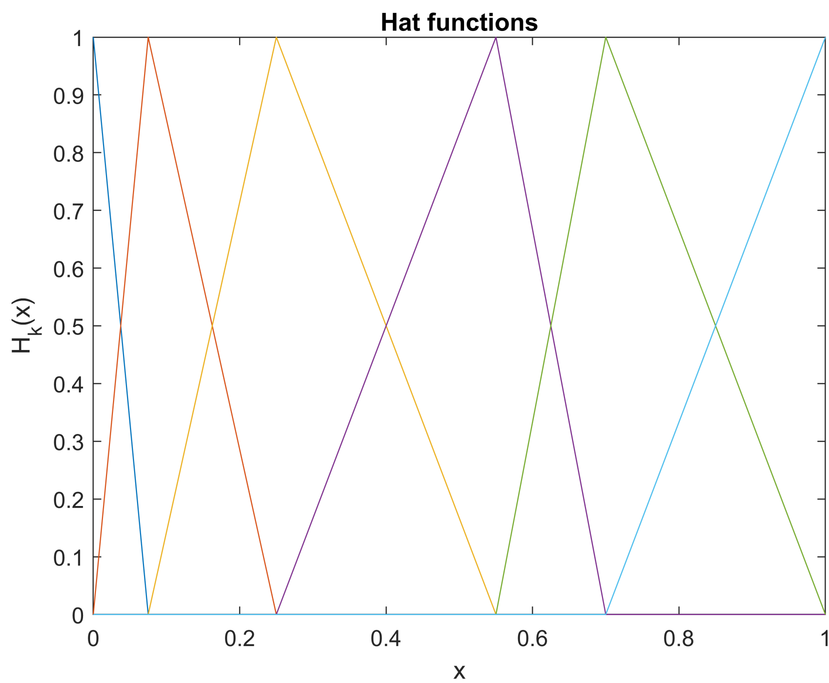
Let's define a set of 6 nodes (i.e., $n = 5$ in our formulas).

```
t = [0 0.075 0.25 0.55 0.7 1]';
```

We plot the hat functions $H_0, \ldots, H_5$.

```
for k = 0:5
    fplot(@(x) hatfun(x,t,k),[0 1])
    hold on
```

```
end
xlabel('x'), ylabel('H_k(x)'), title('Hat functions')
hold off
```

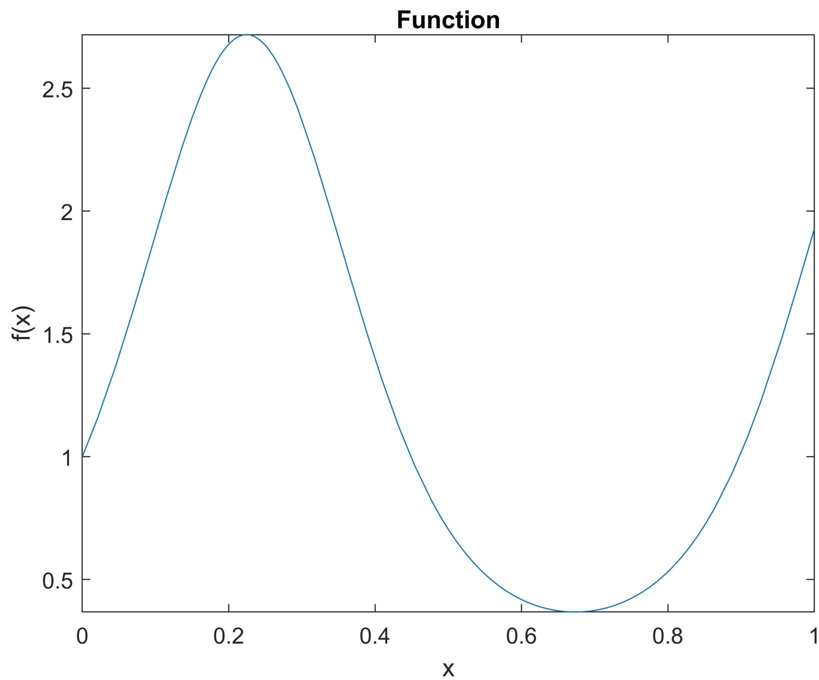


Cardinality Conditions

Example 5.2.2

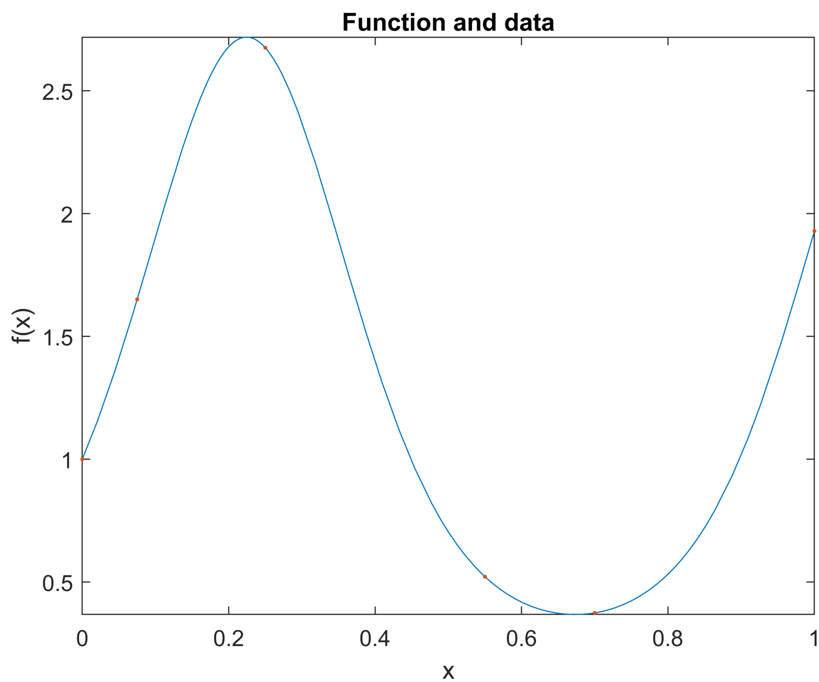
We generate a piecewise linear interpolant of $f(x) = e^{\sin 7x}$.

```
f = @(x) exp(sin(7*x));
fplot(f,[0 1])
xlabel('x'), ylabel('f(x)')    % ignore this line
title('Function')    % ignore this line
```



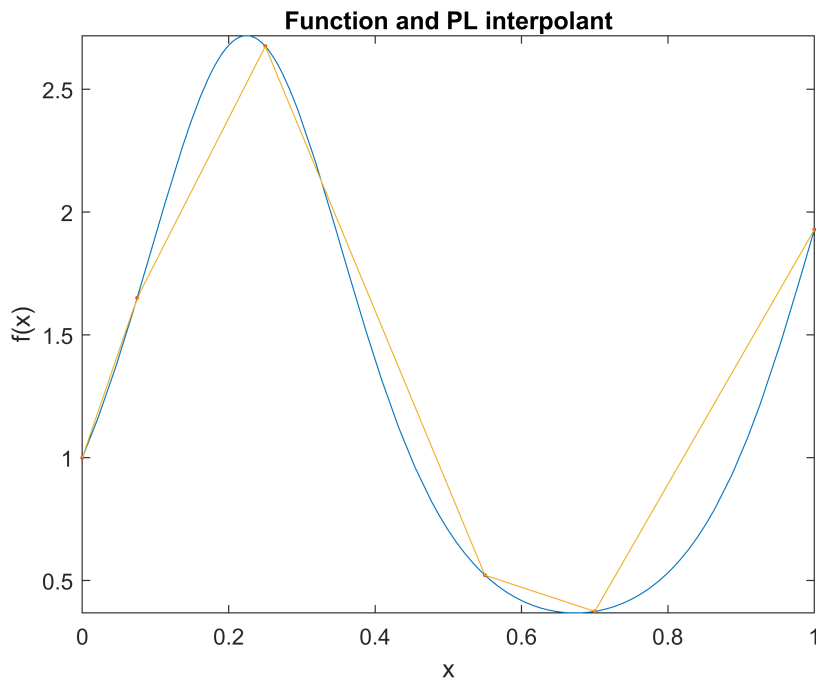
First we sample the function to create the data.

```
t = [0 0.075 0.25 0.55 0.7 1];    % nodes
y = f(t);
hold on, plot(t,y,'.')
title('Function and data')    % ignore this line
```



Now we create a callable function that will evaluate the piecewise linear interpolant at any x.

```
p = plinterp(t,y);
fplot(p,[0 1])
title('Function and PL interpolant')    % ignore this line
```



Example 5.2.3

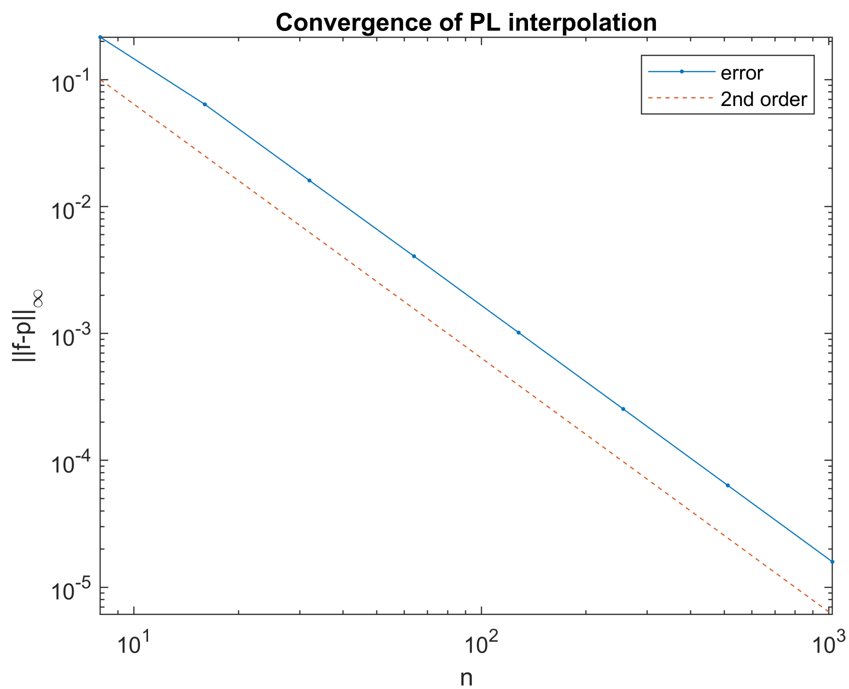
We measure the convergence rate for piecewise linear interpolation of $e^{\sin 7x}$.

```
f = @(x) exp(sin(7*x));
x = linspace(0,1,10001)';  % sample the difference at many points
n_ = 2.^(3:10)';
err_ = 0*n_;

for i = 1:length(n_)
    n = n_(i);
    t = linspace(0,1,n+1)';   % interpolation nodes
    p = plinterp(t,f(t));
    err = max(abs( f(x) - p(x) ));
    err_(i) = err;
end
```

Since we expect convergence that is $O(h^2) = O(n^{-2})$, we use a log-log graph of error and expect a straight line of slope -2.

```
figure
loglog( n_, err_, '.-' )
hold on
loglog( n_, 0.1*(n_/n_(1)).^(-2), '--' )
xlabel('n'), ylabel('||f-p||_\infty')     % ignore this line
title('Convergence of PL interpolation')     % ignore this line
```

```
legend('error','2nd order')     % ignore this line
axis tight    % ignore this line
hold off
```

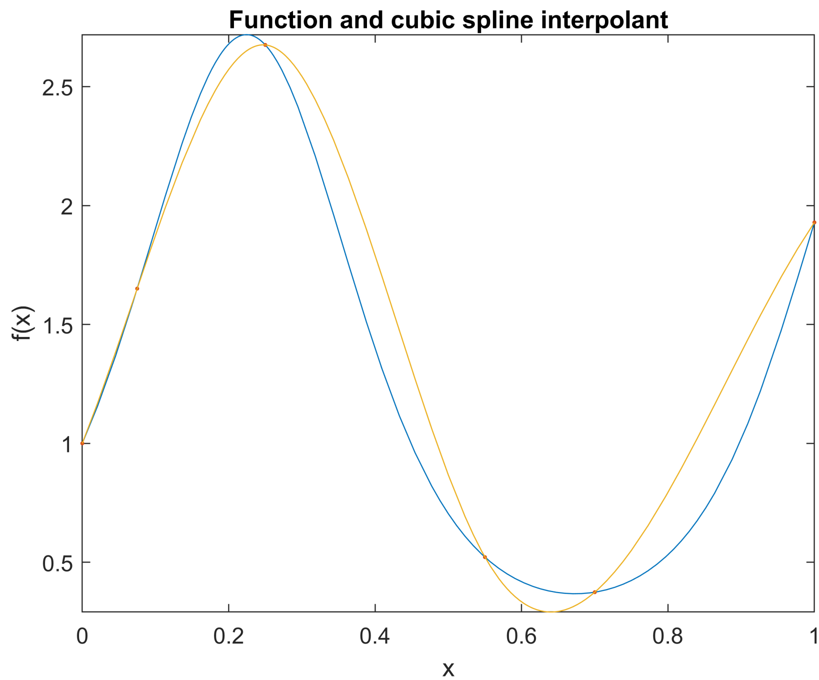


Cubic Splines

Example 5.3.1

For illustration, here is a spline interpolant using just a few nodes.

```
f = @(x) exp(sin(7*x));
fplot(f,[0,1])
t = [0, 0.075, 0.25, 0.55, 0.7, 1]';  % nodes
y = f(t);                            % values at nodes
hold on,
plot(t,y,'.')
S = spinterp(t,y);
fplot(S,[0,1])
xlabel('x'), ylabel('f(x)')   % ignore this line
title('Function and cubic spline interpolant')   % ignore this line
hold off
```

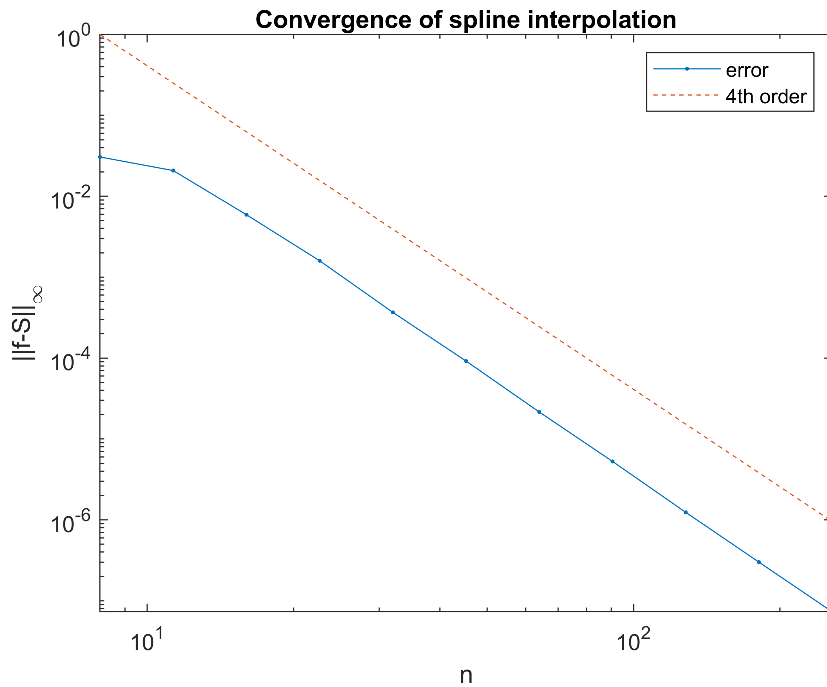


Now we look at the convergence rate as the number of nodes increases.

```
x = linspace(0,1,10001)';  % sample the difference at many points
n_ = 2.^(3:0.5:8)';
err_ = 0*n_;

for i = 1:length(n_)
    n = n_(i);
    t = linspace(0,1,n+1)';   % interpolation nodes
    S = spinterp(t,f(t));
    err = norm( f(x) - S(x), inf );
    err_(i) = err;
end
```

Since we expect convergence that is $O(h^4) = O(n^{-4})$, we use a log-log graph of error and expect a straight line of slope -4.

```
clf, loglog( n_, err_, '.-' )
hold on,
loglog( n_, (n_/n_(1)).^(-4), '--' )
xlabel('n'), ylabel('||f-S||_\infty'), axis tight   % ignore this line
title('Convergence of spline interpolation')   % ignore this line
legend('error','4th order')   % ignore this line
hold off
```



Finite Differences

Example 5.4.2

We try to estimate the derivative of $\cos(x^2)$ at $x = 0.5$ using five nodes.

```
t = [ 0.35 0.5 0.57 0.6 0.75 ]';   % nodes
f = @(x) cos(x.^2);
dfdx = @(x) -2*x.*sin(x.^2);
exact_value = dfdx(0.5)
```

```
exact_value =
  -2.4740e-01
```

We have to shift the nodes so that the point of estimation for the derivative is at $x = 0$.

```
w = fdweights(t-0.5,1);
fd_value = w'*f(t)
```

```
fd_value =
  -2.4731e-01
```

We can reproduce the weights in the finite difference tables by using equally spaced nodes with $h = 1$. For example, here are two one-sided formulas.

```
format rat
fdweights(0:2,1)
```

```
ans =
      -3/2             2             -1/2
```

```
fdweights(-3:0,1)
```

```
ans =
      -1/3           3/2           -3            11/6
```

Convergence of finite differences

Example 5.5.2

Let's observe the convergence of the forward-difference formula applied to the function $\sin(e^{x+1})$ at $x = 0$.

```
f = @(x) sin( exp(x+1) );
FD1 = [ (f(0.1)-f(0))   /0.1
        (f(0.05)-f(0))  /0.05
        (f(0.025)-f(0)) /0.025 ]
```

```
FD1 =
   -6319/2308
    -776/297
   -1999/785
```

It's not clear that the sequence is converging. As predicted, however, the errors are cut approximately by a factor of 2 when h is divided by 2.

```
exact_value = cos(exp(1))*exp(1);
err = exact_value - FD1
```

```
err =
     593/2285
     137/1019
     224/3287
```

Asymptotically as $h \to 0$, the error is proportional to h.

Higher-order accuracy

Example 5.5.4

```
f = @(x) sin( exp(x+1) );
exact_value = cos(exp(1))*exp(1);
```

We'll run both formulas in parallel for a sequence of h values.

```
h = 4.^(-1:-1:-8)';
FD1 = 0*h;  FD2 = 0*h;
for k = 1:length(h)
    FD1(k) = (f(h(k)) - f(0)) / h(k);
    FD2(k) = (f(h(k)) - f(-h(k))) / (2*h(k));
end
```

In each case h is decreased by a factor of 4, so that the error is reduced by a factor of 4 in the first-order method and 16 in the second-order method.

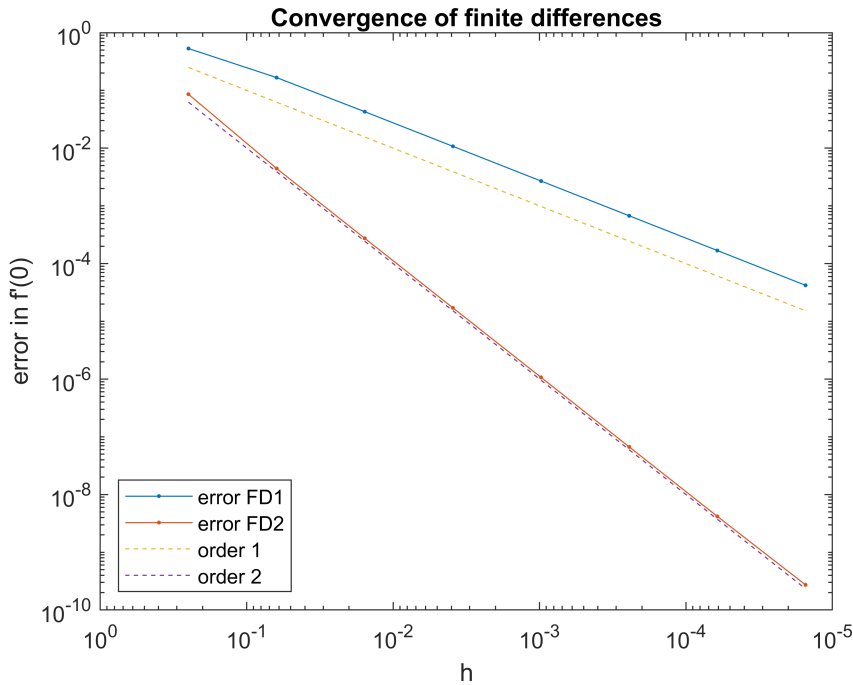
```
error_FD1 = exact_value-FD1;
error_FD2 = exact_value-FD2;
table(h,error_FD1,error_FD2)
```

ans = 8×3 table

	h	error_FD1	error_FD2
1	2.5000e-01	5.3167e-01	-8.5897e-02
2	6.2500e-02	1.6675e-01	-4.4438e-03
3	1.5625e-02	4.2784e-02	-2.7403e-04
4	3.9062e-03	1.0751e-02	-1.7112e-05
5	9.7656e-04	2.6911e-03	-1.0695e-06
6	2.4414e-04	6.7298e-04	-6.6841e-08
7	6.1035e-05	1.6826e-04	-4.1767e-09
8	1.5259e-05	4.2065e-05	-2.7269e-10

A graphical comparison can be clearer. On a log-log scale, the error should (roughly) be a straight line whose slope is the order of accuracy. However, it's conventional in convergence plots, to show h *decreasing* from left to right, which negates the slopes.

```
loglog(h,[abs(error_FD1),abs(error_FD2)],'.-')
hold on,
loglog(h,[h,h.^2],'--')      % perfect 1st and 2nd order
set(gca,'xdir','reverse')
xlabel('h'), ylabel('error in f''(0)')     % ignore this line
legend('error FD1','error FD2','order 1','order 2','location','southwest') % ignore this line
title('Convergence of finite differences')   % ignore this line
hold off
```

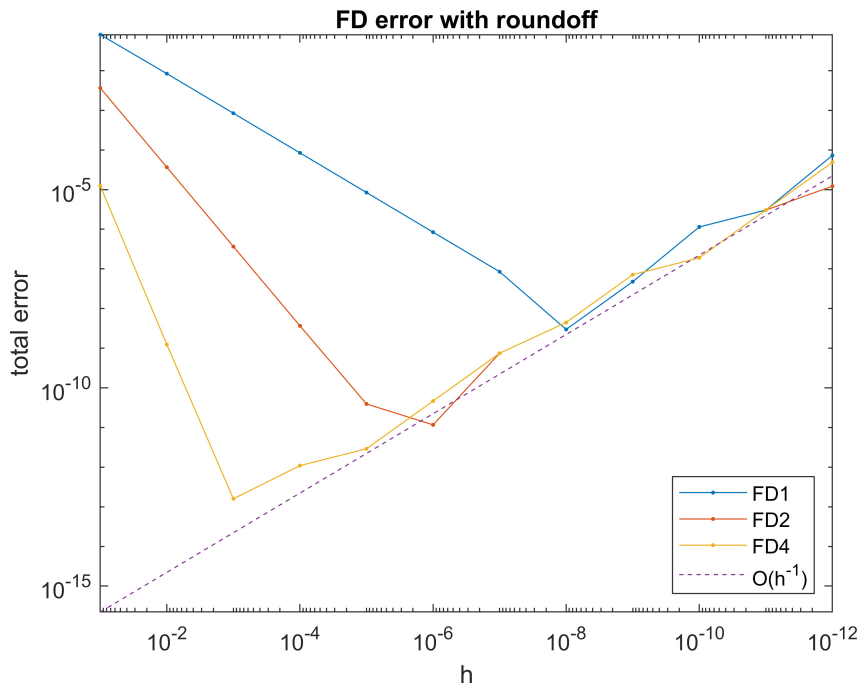


Finite Difference Conditioning

Example 5.5.5

Let $f(x) = e^{-1.3x}$. We apply finite difference formulas of first, second, and fourth order to estimate $f'(0) = -1.3$.

```
h = 10.^(-1:-1:-12)';
f = @(x) exp(-1.3*x);
for j = 1:length(h)
    nodes = h(j)*(-2:2)';
    vals = f(nodes)/h(j);
    fd1(j) = [   0    0 -1    1     0] * vals;
    fd2(j) = [   0 -1/2  0  1/2     0] * vals;
    fd4(j) = [1/12 -2/3  0  2/3 -1/12] * vals;
end
loglog(h,abs(fd1+1.3),'.-'), hold on
loglog(h,abs(fd2+1.3),'.-')
loglog(h,abs(fd4+1.3),'.-')
loglog(h,0.1*eps./h,'--')
set(gca,'xdir','reverse')
legend('FD1','FD2','FD4','O(h^{-1})','location','southeast')  % ignore this line
xlabel('h'), ylabel('total error')     % ignore this line
title('FD error with roundoff'), axis tight    % ignore this line
hold off
```



Again the graph is made so that h decreases from left to right. The errors are dominated at first by truncation error, which decreases most rapidly for the 4th order formula. However, increasing roundoff error eventually equals and then dominates the truncation error as h continues to decrease. As the order of accuracy increases, the crossover point moves to the left (greater efficiency) and down (greater accuracy).

Numerical Integration

Example 5.6.1

The antiderivative of e^x is, of course, itself. That makes evaluation of $\int_0^1 e^x\,dx$ by the Fundamental Theorem trivial.

```
format long, I = exp(1)-1
```

```
I =
   1.718281828459046
```

MATLAB has a built-in `integral` that estimates the value numerically without finding the antiderivative first. As you can see here, it's often just as accurate.

```
integral(@(x) exp(x),0,1)
```

```
ans =
   1.718281828459045
```

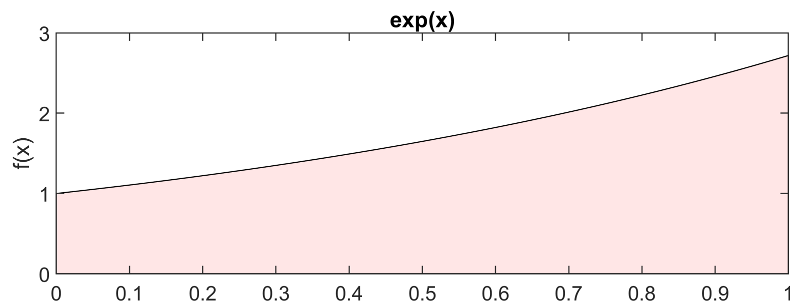
The numerical approach is far more robust. For example, $e^{\sin x}$ has no useful antiderivative. But numerically it's no more difficult.

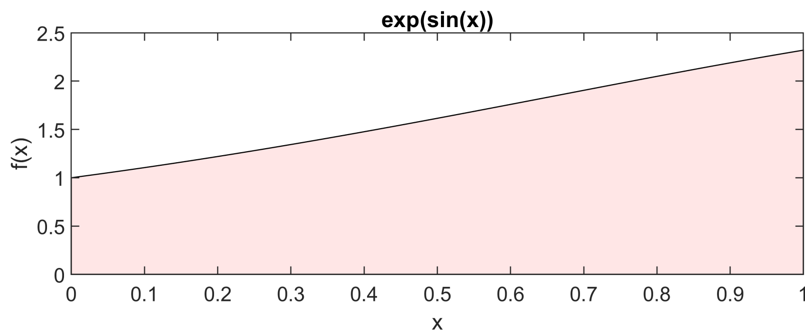
```
integral(@(x) exp(sin(x)),0,1)
```

```
ans =
   1.631869608418051
```

When you look at the graphs of these functions, what's remarkable is that one of these areas is the most basic calculus while the other is almost impenetrable analytically. From a numerical standpoint, they are practically the same problem.

```
x = linspace(0,1,201)';
subplot(2,1,1), fill([x;1;0],[exp(x);0;0],[1,0.9,0.9])
title('exp(x)')  % ignore this line
ylabel('f(x)')    % ignore this line
subplot(2,1,2), fill([x;1;0],[exp(sin(x));0;0],[1,0.9,0.9])
title('exp(sin(x))')  % ignore this line
xlabel('x'), ylabel('f(x)')    % ignore this line
```





Trapezoid Rule

Example 5.6.2

We approximate the integral of the function $f(x) = e^{\sin 7x}$ over the interval $[0,2]$.

```
f = @(x) exp(sin(7*x));
a = 0;  b = 2;
```

In lieu of the exact value, we will use the built-in `integral` function to find an accurate result.

```
I = integral(f,a,b,'abstol',1e-14,'reltol',1e-14);
fprintf('Integral = %.15f\n',I)
```

```
Integral = 2.663219782761539
```

Here is the error at $n = 40$.

```
T = trapezoid(f,a,b,40);
err = I - T
```

```
err =
    9.168471592522209e-04
```

In order to check the order of accuracy, we double n a few times and observe how the error decreases.

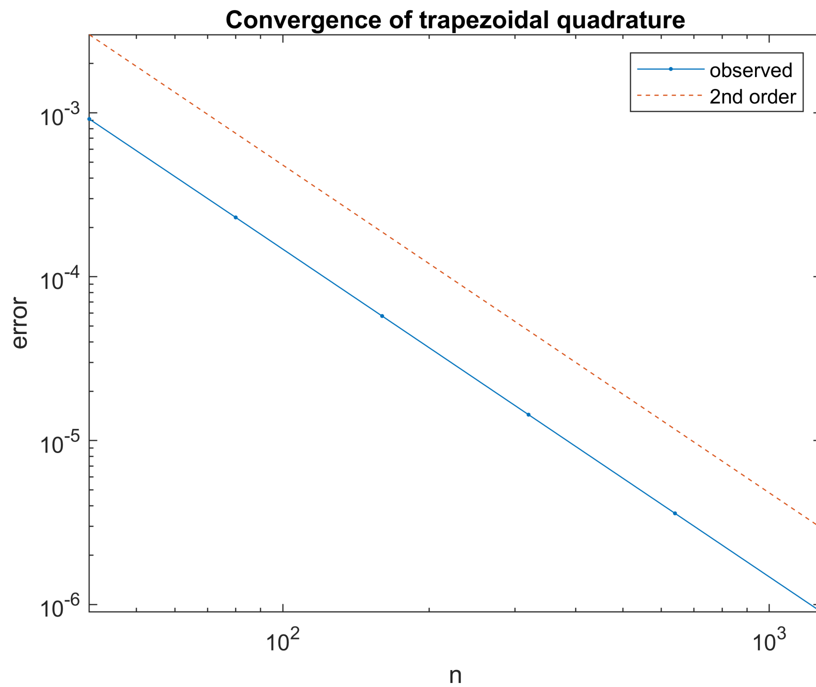
```
n = 40*2.^(0:5)';
err = zeros(size(n));
for k = 1:length(n)
    T = trapezoid(f,a,b,n(k));
    err(k) = I - T;
end

table(n,err)
```

ans = 6×2 table

	n	err
1	40	9.1685e-04
2	80	2.3006e-04
3	160	5.7568e-05
4	320	1.4395e-05
5	640	3.5990e-06
6	1280	8.9975e-07

Each doubling of n cuts the error by a factor of about 4, which is consistent with second-order convergence. Another check: the slope on a log-log graph should be -2.

```
figure
loglog(n,abs(err),'.-')
hold on, loglog(n,3e-3*(n/n(1)).^(-2),'--')
xlabel('n'), ylabel('error'), axis tight         % ignore this line
title('Convergence of trapezoidal quadrature')  % ignore this line
legend('observed','2nd order')                 % ignore this line
hold off
```



Extrapolation

Example 5.6.3

We estimate $\int_0^2 x^2 e^{-2x}\,dx$ using extrapolation.

```
f = @(x) x.^2.*exp(-2*x);
a = 0;   b = 2;   format short e
I = integral(f,a,b,'abstol',1e-14,'reltol',1e-14);
```

We start with the trapezoid formula on $n = N$ nodes.

```
N = 20;       % the coarsest formula
n = N;   h = (b-a)/n;
t = h*(0:n)';    y = f(t);
```

We can now apply weights to get the estimate $T_f(N)$.

```
T = h*( sum(y(2:N)) + y(1)/2 + y(n+1)/2 );
err_2nd = I - T
```

```
err_2nd =
   6.2724e-05
```

Now we double to $n = 2N$, but we only need to evaluate f at every other interior node.

```
n = 2*n;   h = h/2;   t = h*(0:n)';
```

```
T(2) = T(1)/2 + h*sum( f(t(2:2:n)) );
err_2nd = I - T
```

```
err_2nd = 1×2
   6.2724e-05   1.5368e-05
```

As expected for a second-order estimate, the error went down by a factor of about 4. We can repeat the same code to double n again.

```
n = 2*n;  h = h/2;  t = h*(0:n)';
T(3) = T(2)/2 + h*sum( f(t(2:2:n)) );
err_2nd = I - T
```

```
err_2nd = 1×3
   6.2724e-05   1.5368e-05   3.8223e-06
```

Let us now do the first level of extrapolation to get results from Simpson's formula. We combine the elements `T(i)` and `T(i+1)` the same way for $i = 1$ and $i = 2$.

```
S = (4*T(2:3) - T(1:2)) / 3;
err_4th = I - S
```

```
err_4th = 1×2
  -4.1755e-07  -2.6175e-08
```

With the two Simpson values $S_f(N)$ and $S_f(2N)$ in hand, we can do one more level of extrapolation to get a 6th-order accurate result.

```
R = (16*S(2) - S(1)) / 15;
err_6th = I - R
```

```
err_6th =
  -8.2748e-11
```

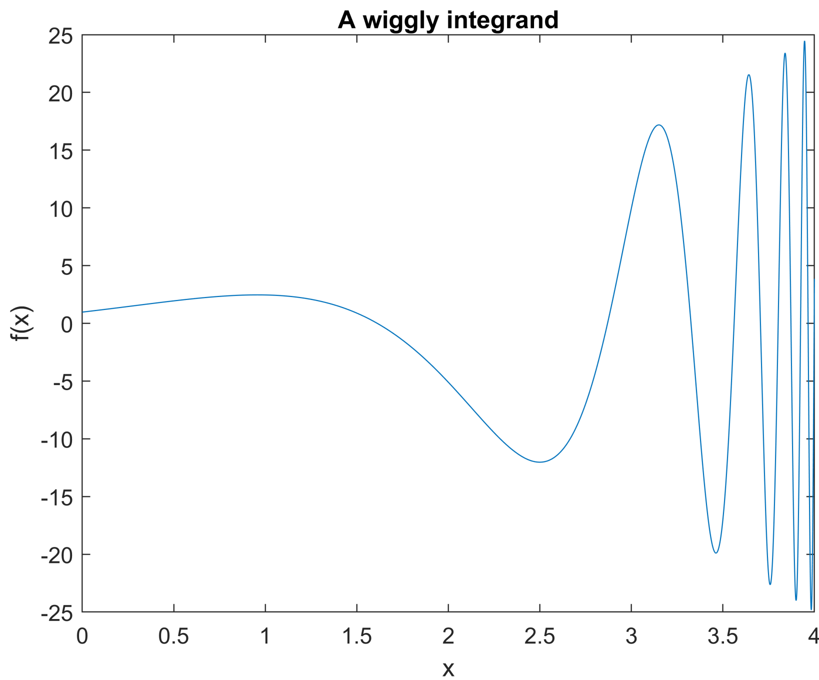
If we consider the computational time to be dominated by evaluations of f, then we have obtained a result with twice as many accurate digits as the best trapezoid result, at virtually no extra cost.

Adaptive

Example 5.7.1

This function gets increasingly oscillatory near the right endpoint.

```
f = @(x) (x+1).^2.*cos((2*x+1)./(x-4.3));
fplot(f,[0 4],2000)
xlabel('x'), ylabel('f(x)')    % ignore this line
title('A wiggly integrand')
set(gca(),'XScale','linear','YScale','linear')
axis([0,4,-25,25])
```



Accordingly, the trapezoid rule is more accurate on the left half of the interval than on the right half.

```
n_ = 50*2.^(0:3)';
T_ = [];
for i = 1:length(n_)
    n = n_(i);
    T_(i,1) = trapezoid(f,0,2,n);
    T_(i,2) = trapezoid(f,2,4,n);
end

left_val = integral(f,0,2,'abstol',1e-14,'reltol',1e-14);
right_val = integral(f,2,4,'abstol',1e-14,'reltol',1e-14);
table(n_,T_(:,1)-left_val,T_(:,2)-right_val,...
    'variablenames',{'n','left_error','right_error'})
```

ans = 4×3 table

	n	left_error	right_error
1	50	-2.4911e-03	5.0423e-01
2	100	-6.2271e-04	9.6004e-02
3	200	-1.5568e-04	2.2547e-02
4	400	-3.8919e-05	5.5542e-03

Both the picture and the numbers suggest that more nodes should be used on the right half of the interval than on the left half.

Example 5.7.2

```
clear
f = @(x) (x+1).^2.*cos((2*x+1)./(x-4.3)); % changed to myfun
I = integral(f,0,4,'abstol',1e-14,'reltol',1e-14)  % 'exact' value
```

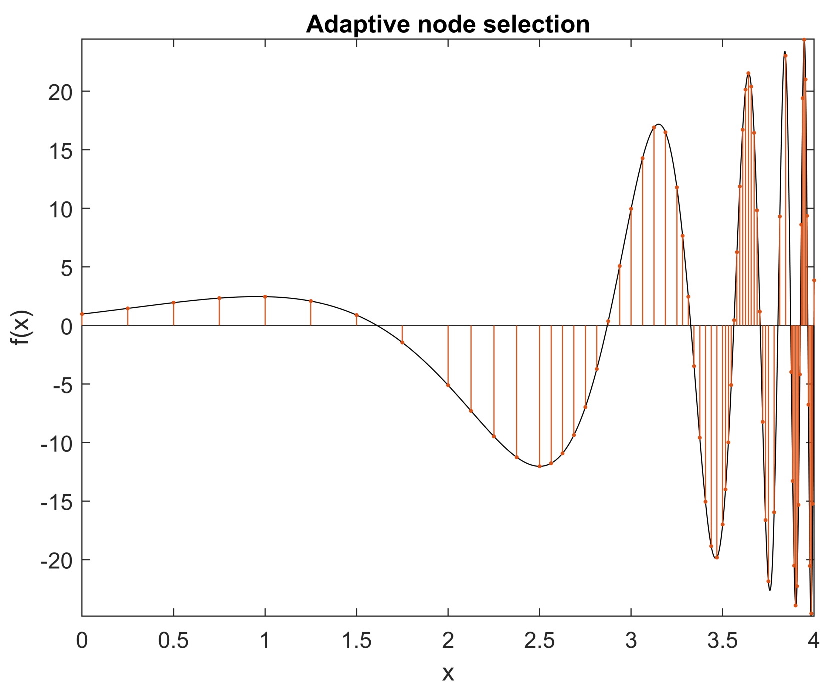
```
I =
  -2.8255e+00
```

We perform the integration and show the nodes selected underneath the curve.

```
[Q,t] = intadapt(f,0,4,.001);
fplot(f,[0 4],2000,'k'),
hold on
stem(t,f(t),'.-')
num_nodes = length(t)
```

```
num_nodes =
    69
```

```
title('Adaptive node selection')    % ignore this line
xlabel('x'), ylabel('f(x)')    % ignore this line
hold off
```



The error turns out to be a bit more than we requested. It's only an estimate, not a guarantee.

```
err = I - Q
```

```
err =
  -2.2003e-02
```

Let's see how the number of integrand evaluations and the error vary with the requested tolerance.

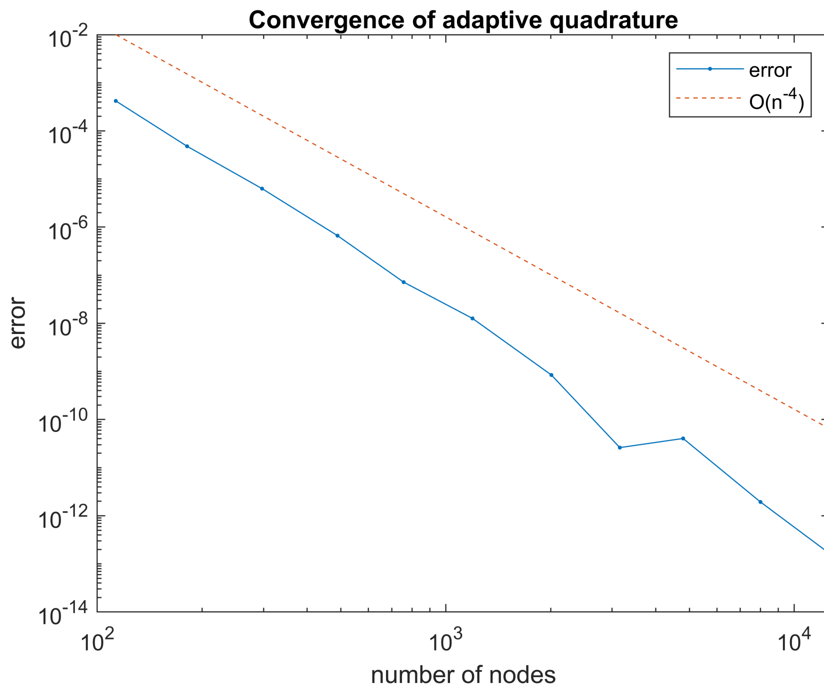
```
tol_ = 10.^(-4:-1:-14)';
err_ = 0*tol_;
num_ = 0*tol_;

for i = 1:length(tol_)
    [Q,t] = intadapt(f,0,4,tol_(i));
    err_(i) = I - Q;
    num_(i) = length(t);
end

table(tol_,err_,num_,'variablenames',{'tol','error','f_evals'})
```

ans = 11×3 table

	tol	error	f_evals
1	1.0000e-04	-4.1947e-04	113
2	1.0000e-05	4.7898e-05	181
3	1.0000e-06	6.3144e-06	297
4	1.0000e-07	-6.6392e-07	489
5	1.0000e-08	7.1808e-08	757
6	1.0000e-09	1.2652e-08	1193
7	1.0000e-10	-8.4412e-10	2009
8	1.0000e-11	2.6129e-11	3157
9	1.0000e-12	4.0449e-11	4797
10	1.0000e-13	-1.9349e-12	7997
11	1.0000e-14	1.6520e-13	12609

As you can see, even though the errors are not less than the estimates, the two columns decrease in tandem. If we consider now the convergence not in h (which is poorly defined) but in the number of nodes actually chosen, we come close to the fourth order accuracy of the underlying Simpson scheme.

```
clf,  loglog(num_,abs(err_),'.-')
hold on
loglog(num_,0.01*(num_/num_(1)).^(-4),'--')
title('Convergence of adaptive quadrature')   % ignore this line
xlabel('number of nodes'),  ylabel('error')   % ignore this line
legend('error','O(n^{-4})')   % ignore this line
%xlim(num_([1 end]))   % ignore this line
hold off
```



Below are the functions used in this chapter.

```
function H = hatfun(x,t,k)
% HATFUN   Hat function/piecewise linear basis function.
% Input:
%   x      evaluation points (vector)
%   t      interpolation nodes (vector, length n+1)
%   k      node index (integer, in 0,...,n)
% Output:
%   H      values of the kth hat function

n = length(t)-1;
k = k+1;  % adjust for starting with index=1

% Fictitious nodes to deal with first, last funcs.
t = [ 2*t(1)-t(2); t(:); 2*t(n+1)-t(n) ];
k = k+1;  % adjust index for the fictitious first node

H1 = (x-t(k-1))/(t(k)-t(k-1));   % upward slope
H2 = (t(k+1)-x)/(t(k+1)-t(k));   % downward slope

H = min(H1,H2);
H = max(0,H);
end

function p = plinterp(t,y)
% PLINTERP   Piecewise linear interpolation.
% Input:
%   t     interpolation nodes (vector, length n+1)
%   y     interpolation values (vector, length n+1)
```

```
% Output:
%   p     piecewise linear interpolant (function)

n = length(t)-1;
p = @evaluate;

    % This function evaluates p when called.
    function f = evaluate(x)
        f = 0;
        for k = 0:n
            f = f + y(k+1)*hatfun(x,t,k);
        end
    end

end

function S = spinterp(t,y)
% SPINTERP   Cubic not-a-knot spline interpolation.
% Input:
%   t     interpolation nodes (vector, length n+1)
%   y     interpolation values (vector, length n+1)
% Output:
%   S     not-a-knot cubic spline (function)

t = t(:);  y = y(:);  % ensure column vectors
n = length(t)-1;
h = diff(t);          % differences of all adjacent pairs

% Preliminary definitions.
Z = zeros(n);
I = eye(n);  E = I(1:n-1,:);
J = I - diag(ones(n-1,1),1);
H = diag(h);

% Left endpoint interpolation:
AL = [ I, Z, Z, Z ];
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*[ Z, J, 2*H, 3*H^2 ];
v1 = zeros(n-1,1);

% Continuity of second derivative:
A2 = E*[ Z, Z, 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] ];
```

```
% 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 = @evaulate;

    % This function evaluates the spline when called with a value for x.
    function f = evaulate(x)
        f = zeros(size(x));
        for k = 1:n       % iterate over the pieces
            % Evalaute this piece's cubic at the points inside it.
            index = (x>=t(k)) & (x<=t(k+1));
            f(index) = polyval( [d(k),c(k),b(k),a(k)], x(index)-t(k) );
        end
    end

end

function w = fdweights(t,m)
%FDWEIGHTS   Fornberg's algorithm for finite difference weights.
% Input:
%   t    nodes (vector, length r+1)
%   m    order of derivative sought at x=0 (integer scalar)
% Output:
%   w    weights for the approximation to the jth derivative (vector)

% This is a compact implementation, not an efficient one.

r = length(t)-1;
w = zeros(size(t));
for k = 0:r
  w(k+1) = weight(t,m,r,k);
end
end

function c = weight(t,m,r,k)
% Implement a recursion for the weights.
% Input:
%   t   nodes (vector)
%   m   order of derivative sought
%   r   number of nodes to use from t (<= length(t))
%   k   index of node whose weight is found
% Output:
%   c   finite difference weight

if (m<0) || (m>r)        % undefined coeffs must be zero
  c = 0;
elseif (m==0) && (r==0)  % base case of one-point interpolation
  c = 1;
```

```
else                        % generic recursion
  if k<r
    c = (t(r+1)*weight(t,m,r-1,k) - ...
        m*weight(t,m-1,r-1,k))/(t(r+1)-t(k+1));
  else
    beta = prod(t(r)-t(1:r-1)) / prod(t(r+1)-t(1:r));
    c = beta*(m*weight(t,m-1,r-1,r-1) - t(r)*weight(t,m,r-1,r-1));
  end
end
end

function [T,t,y] = trapezoid(f,a,b,n)
%TRAPEZOID   Trapezoid formula for numerical integration.
% Input:
%   f     integrand (function)
%   a,b   interval of integration (scalars)
%   n     number of interval divisions
% Output:
%   T     approximation to the integral of f over (a,b)
%   t     vector of nodes used
%   y     vector of function values at nodes

h = (b-a)/n;
t = a + h*(0:n)';
y = f(t);
T = h * ( sum(y(2:n)) + 0.5*(y(1) + y(n+1)) );
end

function y=myfun(x)
y=(x+1).^2.*cos((2*x+1)./(x-4.3));
end

function [Q,t] = intadapt(f,a,b,tol)
%INTADAPT   Adaptive integration with error estimation.
% Input:
%   f     integrand (function)
%   a,b   interval of integration (scalars)
%   tol   acceptable error
% Output:
%   Q     approximation to integral(f,a,b)
%   t     vector of nodes used

m = (b+a)/2;
[Q,t] = do_integral(a,f(a),b,f(b),m,f(m),tol);
end

% Use error estimation and recursive bisection.
function [Q,t] = do_integral(a,fa,b,fb,m,fm,tol)

% These are the two new nodes and their f-values.
xl = (a+m)/2;  fl = myfun(xl);
xr = (m+b)/2;  fr =myfun(xr);
t = [a;xl;m;xr;b];              % all 5 nodes at this level
```

```
% Compute the trapezoid values iteratively.
h = (b-a);
T(1) = h*(fa+fb)/2;
T(2) = T(1)/2 + (h/2)*fm;
T(3) = T(2)/2 + (h/4)*(fl+fr);

S = (4*T(2:3)-T(1:2)) / 3;      % Simpson values
E = (S(2)-S(1)) / 15;           % error estimate

if abs(E) < tol*(1+abs(S(2)))   % acceptable error?
    Q = S(2);                   % yes--done
else
    % Error is too large--bisect and recurse.
    [QL,tL] = do_integral(a,fa,m,fm,xl,fl,tol);
    [QR,tR] = do_integral(m,fm,b,fb,xr,fr,tol);
    Q = QL + QR;
    t = [tL;tR(2:end)];         % merge the nodes w/o duplicate
end
end
```