piecewise linear interpolation

The piecewise linear (or, broken line) interpolant to f at $x_1 < \cdots < x_n$ is, by definition, the function

$$L(z) := L_i(z) := f(x_i) + f[x_i, x_{i+1}](z - x_i), \quad x_i \le z \le x_{i+1},$$

with

$$L(z) := \begin{cases} L_1(z), & z < x_1; \\ L_{n-1}(z), & z > x_n. \end{cases}$$

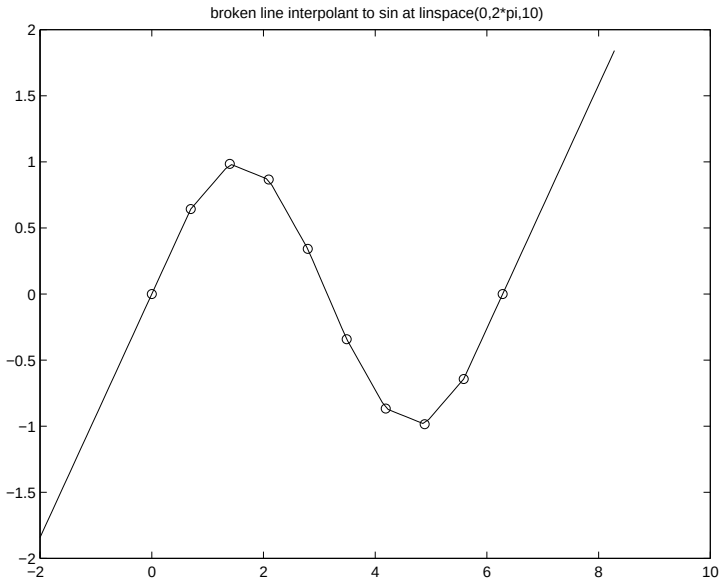
If `y = f(x)` provides the values $\mathtt{y}(i) = f(x_i)$ of f at the x_i's, then

```
pl = mkpp(x,[diff(y)./diff(x) y(1:n-1)]);
```

provides, in `pl`, a description of the broken line interpolant that can be used in `ppval` to provide values of the broken line interpolant, as in the following script:

```
% script: myshowpl
n = 10;
x=linspace(0,2*pi,n);
y = sin(x);
pl = mkpp(x,[diff(y)./diff(x) y(1:n-1)]);
plot(x,y,'o')
xx = linspace(-2,2*pi+2,100);
hold on, plot(xx,ppval(pl,xx)), hold off
title('broken line interpolant to sin at linspace(0,2*pi,10)')
```

which gives a fine figure.



piecewise polynomials in matlab: the mkpp command

Assuming that `x` has the entries $x_1 < x_2 < \cdots < x_{\ell+1}$, and the matrix `c` has the entries c_{ij}, $i = 1{:}\ell$, $j = 1{:}k$, the command

```
pp = mkpp(x,c);
```

returns a structure, `pp`, that can be used in the command `ppval(pp,xx)` to produce the value(s) at `xx` of the piecewise polynomial function f, given by the rule

$$f(x) = c_{i1}(x - x_i)^{k-1} + c_{i2}(x - x_i)^{k-2} + \cdots + c_{ik}, \qquad x_i \le x < x_{i+1}; i = 1{:}\ell.$$

Note that $c(i,:)$ provides the polynomial coefficients, from highest to lowest, for the ith polynomial piece, but these are multiplied by *shifted* powers, for greater stability. The points x_i are called the breakpoints since one switches from one polynomial piece to the next one as one goes across such a point.

What if x does not lie in the interval $[x_1 \,..\, x_{\ell+1}]$? If $x < x_1$, then `ppval` will simply use the first polynomial piece, i.e.,

$$f(x) = c_{11}(x - x_1)^{k-1} + c_{12}(x - x_1)^{k-2} + \cdots + c_{1k}, \qquad x < x_1.$$

If $x > x_{\ell+1}$, `ppval` will use the last polynomial piece.

With this in mind, we see that the statement

```
pl = mkpp(x,[diff(y)./diff(x) y(1:n-1)]);
```

used above makes up the function

$$L(x) := \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i, \qquad i = 1{:}\texttt{length(x)} - 1,$$

exactly as wanted.

piecewise polynomials in matlab: the ppval command

`matlab`'s command `ppval` is (now) fully vectorized, thanks to an approach quite different from the one our textbook takes to the evaluation of broken lines (and other pp functions). Here's the issue. To evaluate L at z, we must first determine the i for which $x_i \le z \le x_{i+1}$. The book proposes to do this by binary search (alias bisection), as realized in the function `Locate`. However, such a search is not easy to vectorize, i.e., to carry out simultaneously for all the entries of a *vector* `z`. `matlab`'s `ppval` is based on the following observation.

Suppose we had in hand the vector `I` with the property that `x(I(k)) <= z(k) <= x(I(k)+1)` for all `k`. Then we could compute $L(\texttt{z})$ in the *one* statement

```
v = b(I).*(z-x(I)) + a(I);
```

assuming only that all the vectors occurring here, i.e., `b`, `z`, `x`, and `a`, have the same orientation, so the *pointwise* vector multiplication works correctly.

We can obtain that vector `I` by making use of `matlab`'s fast sorting command, `sort`, which, with the command `[s,index] = sort(t)`, provides the sequence `index` with the property that `t(index)` equals the sorted version, `s`, of the entries of `t`. (Do a `help sort`.) In effect, `index` tells us where each entry of `t` ends up when we sort the entries of `t`, from smallest to largest, i.e., from left to right if we think of them as points on the real line. This is exactly the kind of information we'd like to know for the combined sequence `[x z]`.

For example, if `x=1:3` and `z=0:4`, then `[ignore,index] = sort([x z]);` gives us

```
>> index = 4     1     5     2     6     3     7     8
```

Since `x` comes first in `[x z]` and has 3 entries, we know that, in this particular index sequence, the numbers 1, 2, 3 refer to `x(1)`, `x(2)`, `x(3)`, respectively, while the numbers greater than 3=`length(x)` refer to entries of `z`. In particular, the `matlab` command

```
>> j = find(index>length(x))
j = 1     3     5     7     8
```

provides the positions in the sorted sequence of the entries of `z`.

Now assume (as is the case in our example) that the entries of `z` are ordered. Then the position `j(i)` of `z(i)` in the sorted sequence `ignore` differs from `i` by the number of entries from `x` to the left of it in that sequence. We check this for our example:

```
>> find(index>length(x)) - (1:length(z))
ans = 0     1     2     3     3
```

Indeed, e.g. for `z(2)`, that difference is `3-2`, corresponding to the fact that there is exactly one of the entries of `x` to the left of it. But this says that `x(j(i)-i)` is the largest entry of `x` to the left of `z(i)`, hence `j(i)-i` is the number `I(i)` we are seeking, – except when this number is `0`, as it is in our example for the first entry. This indicates that the corresponding `z(i)` lies entirely to the left even of the first entry of `x` and, in that case, we defined the value of L to be the value of its first linear piece, hence want the interval index to be 1 in this case. One way to ensure this is to make sure that all the entries of `I` are at least 1, using `max` as follows:

```
>> I = max(find(index>length(x)) - (1:length(z)) , 1 )
I = 1     1     2     3     3
```

Have a look at `matlab`'s `ppval`, – and note that their construction of `I` looks slightly more formidable

```
I = max([ find(index>length(x)) - (1:length(z)); ones(1,length(z))] )
```

Note that `matlab`'s `ppval` is even capable of evaluating the given pp function at all the entries of a *matrix*, returning the values in a matrix of the same size.

Here is a script for the comparison of the book's evaluation, i.e., the command `pwleval`, with `matlab`'s command, `ppval`:

```
% script for timing two versions of pp evaluation:
disp(' Length(x)    pwleval(x)      ppval(x)')
disp('                 Time          Time  ')
disp('-----------------------------------------------------')
n = 50;
x = [0 sort(rand(1,n-2)) 1];
y = sin(10*pi*x);
a = y(1:n-1); b = diff(y)./diff(x);
pl = mkpp(x,[b a]);
if ~exist('tosort'), tosort=0; end
for L=50:200:950
   z = rand(1,L); if tosort, z = sort(z); end;
   tic
   yy = pwleval(a,b,x,z);
   t1 = toc;
   tic
   yy = ppval(pl,z);
   t2 = toc;
   disp(sprintf('%6.0f  %13.2f  %13.2f  ',L,t1,t2))
end
```

Here is the output of a run, for which `tosort` was not defined, hence the sequence `z` of evaluation sites is not at all ordered. This is the worst situation for the book's `Locate`, as it is designed to take advantage of the ordering in a typical `z`:

```
 Length(x)    pwleval(x)      ppval(x)
                 Time          Time
--------------------------------------
    50           0.28           0.00
   250           1.04           0.05
   450           1.81           0.11
   650           2.69           0.16
   850           3.57           0.17
```

For a sorted `z`, the book's `pwleval` does much better, but still, `ppval` also benefits from such sorting and is still the champion:

```
 Length(x)    pwleval(x)      ppval(x)
                 Time          Time
--------------------------------------
    50           0.11           0.05
   250           0.44           0.05
   450           0.71           0.05
   650           0.99           0.11
   850           1.32           0.11
```


error analysis for piecewise linear interpolation

On $[x_i \,..\, x_{i+1}]$,

$$f(z) = L_i(z) + f[x_i, x_{i+1}, z](z - x_i)(z - x_{i+1})$$

while

$$\max_z |z - x_i||z - x_{i+1}| = (\Delta x_i)^2/4$$

$$\max_z |f[x_i, x_{i+1}, z]| \leq \max_z |f''(z)|/2$$

with

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

the standard notation for the **forward difference**.

For equally spaced breakpoints: `x = linspace(alpha,beta,n)`, get $\Delta x_i = h := (\beta - \alpha)/(n-1)$ for all i, hence $\max_z |f(z) - L(z)| \leq \max_z |f''(z)|/8h^2$. To force this error *bound* to be less than a given tolerance τ, assuming that we know some M_2 with

$$\max_z |f''(z)| \leq M_2,$$

we would demand that

$$h^2 M_2/8 \leq \tau,$$

or, solving for h,

$$h \leq \sqrt{8\tau/M_2}.$$

Since $h = (\beta - \alpha)/(n-1)$, this says that we want

$$(\beta - \alpha)\sqrt{M_2/(8\tau)} \leq n - 1,$$

hence, finally, with $\lceil r \rceil$ the smallest integer $\geq r$ (this is the so-called **ceiling function**),

$$n = 1 + \lceil (\beta - \alpha)\sqrt{M_2/(8\tau)}\, \rceil$$

is the smallest n we can get away with here.

However, by going to a uniform mesh, we have possibly thrown away a chance to be efficient in use of memory. Our error bound for the interval $[x_i, x_{i+1}]$ gave us

$$|f(z) - L_i(z)| \leq \max_{x_i < \eta < x_{i+1}} |f''(\eta)|/8\ (\Delta x_i)^2.$$

So, to achieve a given tolerance τ, we should be able to use relatively large Δx_i in places where $|f''|$ is small, as is the case in the interval $[1 \,..\, 3]$ for the `humps` function shown in Figure 3.2 of the book. This leads to the powerful idea of choosing the sequence x **adaptively**, adjusting the size of Δx_i to the size of $|f''|$ rather than working with one constant M_2, i.e., working with the worst possible assumption.

For this, one would have to know the size of $|f''|$. However, it is often sufficient to work with some crude assumption, like the following: *a good estimate for the size of the error in the interval* $[x_i \,..\, x_{i+1}]$ *is the error* $|f(m_i) - L(m_i)|$ *at the midpoint,* $m_i := (x_i + x_{i+1})/2$. Of course, such an estimate can easily be totally wrong, as it would be for $f = \sin$ and $x_i = 0, x_{i+1} = 2\pi$. But, in many cases, it works fine, as it does in the book's example in which f is `matlab`'s function `humps` and the interval is [0..3].

T/F: If f'' is of one sign, then the error in the interval $[x_i \,..\, x_{i+1}]$ is at most $2|f(m_i) - L(m_i)|$.

Appreciate the power inherent in `matlab`'s ability to run functions *recurrently*, as is evidenced by the simplicity of the book's `pwladapt`.

