10dec98

Loop-free `MATLAB` (as of 26aug00)

Since `MATLAB` is an interpretive language, one wants to make use of matrix algebra to avoid loops wherever possible. Here is a list of simple examples, to illustrate how this can be done, with the aid of such `MATLAB` commands as
(1) *pointwise* matrix operations, such as `a.*b`, `a./b`, `a.^b`;
(2) the construction of a matrix from some rows and columns of a given matrix `a` by the command `a(b,c)` in which also `b` and `c` are matrices; also its special case `a(b)`;
(3) the resetting of part of a matrix `a` by the command `a(b,c)= ...` in which also `b` and `c` are matrices; also its special case `a(b) = ... .`
(4) `repmat`;
(5) `reshape`;
(6) `find`;
(7) the reduction operators `sum`, `prod`, `cumsum`, `cumprod`, `min`, `max`.

As a first example, here is **Horner's scheme** for the evaluation of the polynomial $p(x) := a(1)x^{n-1} + a(2)x^{n-2} + \cdots + a(n)$ at the point x:

```
pval = a(1);
for i=2:length(a)
   pval = x*pval + a(i);
end
```

To evaluate this polynomial at *all* the points in an *array* `x`, we could, of course, simply run the above script inside a second loop over all the entries of `x`, getting the *array* `pval` of the same size as `x`, and containing the corresponding values of the polynomial. But it is more efficient to make use of `MATLAB`'s matrix algebra to calculate in one go all the entries of the array `pval` (much as `sin(x)` gives an array of the same size as `x` containing the sine of the entries of `x`).

```
pval = zeros(size(x));
for i=1:length(a)
   pval = x.*pval + a(i);
end
```

Note the use of the *pointwise* matrix multiply, in `x.*pval`.

What if we wanted to evaluate several polynomials of the same degree at several points? Suppose, to be concrete, that `a` is an array of size $k \times n$, with `a(:,j)` the coefficient vector for the jth of n polynomials, and that we are to evaluate the `jx(i)`-th polynomial at the point `x(i)`, all `i`. This is accomplished by the following script in which it is assumed that `x` is a 1-row matrix.

```
pval = a(1,jx);
for i=2:k
   pval = x.*pval + a(i,jx);
end
```

The expression `a(i,jx)` is a first instance here of a very powerful `MATLAB` construction, for generating a matrix from the rows and columns of a given matrix. In full generality, if `a` is of size (m, n) and `b` and `c` are vectors with entries in the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$, respectively, then

$$\mathtt{d} = \mathtt{a}(\mathtt{b}, \mathtt{c}) \quad \Longleftrightarrow \quad \mathtt{d}(\mathtt{i}, \mathtt{j}) = \mathtt{a}(\mathtt{b}(\mathtt{i}), \mathtt{c}(\mathtt{j})), \quad \textit{all}\ \mathtt{i}, \mathtt{j}.$$

I.e., the resulting `d` has `length(b)` rows and `length(c)` columns, and satisfies d(i,j) = a(b(i),c(j)). (If `b` and/or `c` are not just vectors but arrays, the equivalent vector is used.) In the particular example `a(i,jx)` above, `i` is just a scalar, but `jx` is, by assumption, a vector of the same length as the 1-row matrix `x`. Therefore, `a(i,jx)` is a 1-row matrix of the same length as `jx` and `x`, and has `a(i,jx(j))` as its `j`-th entry.

What if we wanted to evaluate *all* those n polynomials at *all* the entries of the given vector `x`? Here is how one might generate the matrix `pval` for which `pval(i,j)` contains the value at `x(i)` of the `j`-th polynomial. This time, I assume again that `x` is a 1-column matrix, something I can always achieve by the statement

```
x = x(:);
```

which tells `MATLAB` to treat `x` the way it handles it internally anyway, namely as the 1-column matrix obtained by concatenating all the columns of `x` consecutively. But this time, I won't even take for granted that I somehow know the number of rows in `a`.

```
[k,n] = size(a); nx = length(x);
jx = repmat(1:n, nx,1);
xx = reshape(x(:,ones(1,n)),1,nx*n);

pval = a(1,jx);
for i=2:length(a(:,1))
   pval = xx.*pval + a(i,jx);
end

pval = reshape(pval,nx,n);
```

This script makes use of the earlier one. It simply replicates the point sequence n times and generates a corresponding index vector `jx`. Note the appearance of another `MATLAB` command useful in this connection, the `reshape` command. Recall that `MATLAB` treats an array internally as the 1-column matrix obtained by concatenating the consecutive columns of the array. In addition, `MATLAB` stores the size, i.e., the number of rows and columns, of the array. So, all the command `reshape(pval,nx,n)` does is to reset the size of `pval` to `nx` rows and `n` columns. In particular, `xx` is constructed by reshaping, into one long 1-row matrix, the matrix that has all its n columns equal to `x`; hence, `xx` is simply one long vector, of the form

```
[x(1),...,x(nx), x(1), ..., x(nx), x(1),..., ...,x(nx)]
```

Further, `jx` is the matrix that has all its `nx` rows equal to `[1,2,...,n]`, hence, when it is used in `a(i,jx)`, it acts as one long vector, of the form

```
[1,...,1,2,...,2,3,..., ..., n]
```

Correspondingly, after the loop is done, `pval` is the long vector

```
[p1(x(1)),...p1(x(nx)),p2(x(1)),...,p2(x(nx)),p3(x(1)),..., ...,pn(x(nx))]
```

and therefore, the final `pval = reshape(pval,nx,n)` changes `pval` into the array, of size `nx`-by-`n`, with `pval(i,j)` equal to the value of the `j`-th polynomial at `x(i)`.

What about that loop that appears in all these scripts? It, too, can be avoided, but that may not be worth the computational cost. Consider again the script

```
pval = zeros(size(x));
for i=1:length(a)
   pval = x.*pval + a(i);
end
```

for the evaluation of the polynomial $p(x) = a(1)x^{k-1} + \cdots + a(k)$ at all the entries of `x`. The same result can be obtained with `MATLAB`'s `polyval` command:

```
pval = polyval(a,x);
```

Unfortunately, a look into `polyval` (obtained by the command `type polyval`) shows that all we have done is to hide the loop, in `polyval`. Here is one way to avoid an explicit loop. This script assumes that `x` is a 1-column matrix and that `a` is a vector of length `k`:

```
pval = (x(:,ones(1,k)).^repmat((k-1):-1:0, length(x),1))*a(:);
```

Note again the use of the `a(b,c)` construct, to generate matrices having all its columns, respectively all its rows, the same. The final `a(:)` is just a precaution, in case `a` happens to be a 1-row matrix. But now note the computational cost here, of computing all those powers, without even using the fact that each column is the pointwise product of the next column.

This latter objection can be overcome, with the aid of the `MATLAB` command `cumprod` (and using the fact that all such vector operators can be told, with the aid of an optional argument, whether to work on rows or columns).

```
a = a(:); x = x(:);
pval = [ones(length(x),1) cumprod(repmat(x,1,k-1),2)]*a(k:-1:1);
```


Even the simultaneous evaluation, of the `jx(i)`-th polynomial at `x(i)`, all `i`, can be done without a(n explicit) loop, with the aid of the `sum` command. Assume again that `a(:,i)` contains the coefficients of the `i`-th polynomial. Then the following script will do the job:

```
nx = length(x); x = reshape(x,1,nx);
pval = sum(a(k:-1:1,jx).*[ones(1,nx);cumprod(repmat(x,k-1,1))]);
```

Timings, for various values of `k` and `nx`, show that this script gives not as fast an execution as the earlier script with the one loop. For, by comparison, the present script does, in effect, an extra `nx*k` floatingpoint multiplies. Also, there is some trouble here, for both reduction operators, `sum` and `cumprod` used here, in case they are being applied to a 1-row matrix (as is the case for `cumprod` when `k` equals 2 and for `sum` when `k` equals 1). For, the reduction operators will ordinarily work on each column of a matrix, – except when the matrix has only one row, in which case they will work on that one row. In Version 5 of `MATLAB`, this can be taken care of by using an optional additional input argument to specify that they should always be working on columns, i.e., on the first dimension of these arrays. So, in `MATLAB v5`, that line should read

```
pval = sum(a(k:-1:1,jx).*[ones(1,nx);cumprod(x(ones(k-1,1),:),1)],1);
```

Because use of vector and matrix arithmetic speeds execution of `MATLAB` scripts, it is good practice to follow `MATLAB`'s example and design any `MATLAB function`, for evaluating some function f at some point x, to accept an *array* `x` as input and, correspondingly, return the array of the same size as `x` that contains the corresponding function values.

As an example, here is the `function` that implements the 'subplus' function, defined by

$$x \mapsto (x)_+ := \begin{cases} x & x > 0; \\ 0 & \text{otherwise.} \end{cases}$$

Using the fact that the reduction operator `max` can also work on two matrices of the same size, we get the following one-liner:

```
function y = subplus(x)
y = max(x,zeros(size(x)))
```

A more challenging example is the `MATLAB` command `ppval` which evaluates a *piecewise* polynomial function. Such a function is of the form

$$f(x) = p_i(x), \qquad \xi_i \le x < \xi_{i+1},$$

with (p_i) some sequence of polynomials, of degree $\le k$ say, and (ξ_i) a strictly increasing point sequence (the *break(point)s*). In `MATLAB` (see `mkpp`, `unmkpp`), the coefficients of the `i`-th piece can be found in the `i`-th *row* of the coefficient array `coefs`, something to watch out for as we try to make use of the scripts given above. Further, the coefficients are actually with respect to *shifted* powers, i.e.,

$$p_i(x) = \mathsf{a}(i,1)(x-\xi_i)^{k-1} + \mathsf{a}(i,2)(x-\xi_i)^{k-2} + \cdots + \mathsf{a}(i,k),$$

something else to watch out for. But the major challenge is to generate, in a loop-free way, the index vector jx so that

$$\xi_{\mathtt{jx}(i)} \le x(i) < \xi_{\mathtt{jx}(i)+1}.$$

Here is a script for this which assumes that the (ordered) break sequence xi is a 1-row matrix, but makes no assumption about the array x, and uses MATLAB's sort command:

```
[z, x2z] = sort(x(:).');
[ignored,index] = sort([xi z]);
jx = zeros(size(x));
jx(x2z) = find(index>length(xi))-[1:length(z)];
```

This is mysterious enough to deserve some explanation. First, recall that the command [s,index] = sort(t) provides the sequence s that contains the entries of t, but in nondecreasing order, as well as the sequence index with the property that t(index) equals s. (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. So, the first line of the script provides the 1-row matrix z that contains the entries of x in increasing order.

The crucial information is generated in the second statement, in index. This might be best explained by trying an example. Suppose that xi=1:3 and z=0:4. Then [ignore,index] = sort([xi z]); gives us

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

Since xi comes first in [xi z] and has 3 entries, we know that, in this index sequence, the numbers 1, 2, 3 refer to xi(1), xi(2), xi(3), respectively, while the numbers greater than 3=length(xi) refer to entries of z; e.g., 5 refers to z(5-3), i.e., to the second entry of z. With this in mind, we see that index tells us exactly, and in one blow, what is the right interval for *every* entry of z. For, the sequence tells us that

```
z(4-3) <= xi(1) <= z(5-3) <= xi(2) <= z(6-3) <= xi(3) <= z(7-3) <= z(8-3)
```

Consequently,

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

is exactly the sequence jx we want for this example, – except for that initial 0 which tells us that there are 0 entries of xi to the left of z(1), i.e., z(1) < xi(1). In that case, MATLAB defines the value at z(1) to be that of the first polynomial piece, hence we want jx(1) to be 1 in this case. One way to ensure this is to make sure that all the entries of jx are at least 1, using max as follows:

```
jx = max(find(index>length(xi)) - (1:length(z)), ones(1,length(z)) )
jx =
     1     1     2     3     3
```

To finish the story, note that `find(index>length(xi)) - (1:length(z))` gives us the correct index vector for `z`, while we want the correct index vector for `x`. Since we know that `z` is the same as `x(x2z)`, we know that the index that is good for `z(i)` is the one that is good for `x(x2z(i))`. In other words, we want `jx(x2z)` to equal `find(index>length(xi)) - (1:length(z))`, and that is what the final statements actually ensure. To be sure, `jx` is initialized as an array of the same size as the original `x`. However, whatever the actual size of `jx`, in the statement `jx(x2z)`, `jx` is treated as the equivalent 1-column matrix.

Putting it all together, here is a bare-bones version of what `MATLAB`'s `ppval` should be (and now is, starting with `v5.2`) (and in which the needed reordering is postponed until after the function values have been computed):

```
function v=ppval(pp,x)
%PPVAL  Evaluate piecewise polynomial.
%   V = PPVAL(PP,X) returns the value at the points XX of the
%   piecewise polynomial contained in PP, as constructed by SPLINE
%   (or MKPP).

% get breaks and coefficients ...
[xi,c,l,k]=unmkpp(pp);

% for each entry of x , find the break interval in which it lies ...
[z,x2z] = sort(x(:).');
[ignored,index] = sort([xi(1:l) z]);
jx = max(find(index>l)-(1:length(z)),ones(size(z)));

% now go to local coordinates ...
z = z - xi(jx);
% ... and apply nested multiplication:
   u = c(jx,1).';
   for i=2:k
      u = z.*u + c(jx,i).';
   end

v = zeros(size(x)); v(x2z) = u(:);
```

Since it is much cheaper to check whether sorting of the input array `x` is needed than to sort even an already increasing `x`, and `x` is likely to be increasing, a more complete version of `ppval` should include such an initial test. Also, `ppval` should work for vector-valued pp's. In any case, the above version of `ppval` is much faster than the one currently (22jun97) in `MATLAB`.

A related example concerns the differentiation of a pp function, something not explicitly provided by `MATLAB`. Suppose `pp` contains a piecewise polynomial function in the `MATLAB`-accepted format. Then the following script will put the first derivative of this pp function into dpp:

```
[xi,c,l,k,d] = unmkpp(pp);
if k==1
   dpp = mkpp(xi([1 l+1]),zeros(d,1),d);
else
   dpp = mkpp(xi,repmat(k-1:-1:1,d*l,1).*c(:,1:k-1),d);
end
```

Note the use of repmat to generate a matrix with all rows the same, and the use of pointwise multiplication to multiply, in one go, all the polynomial coefficients with the appropriate integers.

Here is a totally different application of loop-free MATLAB, namely the plotting of a direction field for an ODE. Without going into details, this involves the plotting of a potentially very large number of planar segments, all in the same linestyle. Assuming the endpoints of the j-th segment stored in segments(:,j), the following loop will do the plotting:

```
hold on
for j=1:length(segments(1,:))
   plot(segments([1 3],j), segments([2 4],j),'-w','linewidth',.5)
end
hold off
```

It is also possible to do all this with the *one* command

```
   plot(segments([1 3],:), segments([2 4],:),'-w','linewidth',.5)
```

But it is even faster (when many segments are to be plotted) to use the fact that, when plotting a sequence of points, MATLAB reacts to a NaN by 'lifting the pen', i.e., by plotting nothing from the previous (ordinary) point to the next (ordinary) point, if any. This means that we we can string together all the segments, being sure to put a NaN point in between segments. The following script will do that:

```
ns = length(segments(1,:));
nan = NaN;
x = [segments([1 3],:);nan(1,ones(1,ns))];
y = [segments([2 4],:);nan(1,ones(1,ns))];
plot(x(:), y(:),'-w','linewidth',.5)
```

Note that now there is no need to use hold. Also note the use of nan and of the construct a(b,c) to generate a constant vector without multiplication, i.e., not by using the more standard NaN*ones(1,ns). Starting with MATLAB 5, such *replication* can also be accomplished directly (and less mysteriously) with the aid of the command

```
repmat(A,m,n);
```

which constructs, from the matrix A of size [r,c], the matrix of size [m*r,n*c] containing m*n copies of A organized in m rows and n columns (and which was used consistently earlier in this updated version of these notes).

to be continued