matrices in `matlab`

Every variable in `matlab` is a matrix, even constants such as `3` (which is a 1-by-1 matrix). The size of a matrix `a` is provided by the command `size(a)` which returns a two-entry row matrix containing the number of rows and of columns of `a` (in that order). If the matrix `a` appears at a place where a vector is expected, `matlab` will treat it as the equivalent vector `a(:)` which is the column vector obtained by concatenating the columns of `a` in order. E.g.,

```
>> a = [1 2 3;4 5 6]
a =
     1     2     3
     4     5     6
>> a(:)'
ans =
     1     4     2     5     3     6
```

For example, in constructing a new matrix from a given one, as in `a(b,c)`, both `b` and `c` are permitted to be matrices rather than vectors, but will be treated as if you had written there their equivalent column vector, i.e., `a(b(:),c(:))`. For example, continuing with the above `a`,

```
>> b = [1 2;3 1];
>> a(:,b)
ans =
     1     3     2     1
     4     6     5     4
```

This is most interesting in the assignment statement `a(b)=c`, which changes the matrix `a` in such a way that, *treating* `a`*,* `b`*,* `c` *as the equivalent vectors*, `a(b(i))==c(i)`, `i=1:length(b(:))`. E.g., continuing with the above `a` but a slightly different `b`,

```
>> b = [1 2;1 6]; c = [7 8 9 10];
>> a(b)= c
a =
     8     2     3
     9     5    10
```

There are various things to notice here:

(i) The entries of `b` are now used as indices into the *vector* `a(:)` rather than as row or column indices for the *matrix* `a`. In particular, the entries of `b` here may be anything from `1:length(a(:))`.

(ii) The way the entries of `c` are turned into entries of `a(:)` depends on the order of the indices in `b(:)`. E.g., although you might read matrices row by row, `matlab` reads a matrix column by column, hence `a(b)=c` above really reads `a([1 1 2 6]) = [7 8 9 10]`, causing `a(1,1)` to be set to `7` and then to `8`.

(iii) If an index appears more than once in `b`, its latest appearance in `b(:)` determines the final value of the corresponding entry of `a`.

As a silly example, suppose you wanted to zero out the diagonal entries of the n-by-n matrix `a`. Then the following would accomplish this:

```
a(1:n+1:n*n) = 0;
```

If, instead, you wanted to zero out the entries in the first subdiagonal, use `a(2:n+1:n^2) = 0;`

matrix multiplication

The product AB of matrices A and B is defined for any matrices A and B with $\#cols(A) = \#rows(B)$, in which case $\#rows(AB) = \#rows(A)$ and $\#cols(AB) = \#cols(B)$, and its (i,j)-entry is given by

$$(AB)(i,j) := A(i,:) * B(:,j) = \sum_k A(i,k)B(k,j),$$

i.e., as the scalar product or dot product or inner product of the ith row $A(i,:)$ of A with the jth column $B(:,j)$ of B. This is the **dot-product** or **inner product** view of matrix multiplication, as reflected in `function MatMatDot`, in the book.

There are many ways to view a matrix. For example, one may think of a matrix in terms of its rows or in terms of its columns, and both are equally useful and legitimate points of view. For this reason, some languages (like `Fortran` or `matlab`) assume that matrices are stored in columns and other languages (like `C`) assume that matrices are stored in rows. Efficient programming will adapt to this and select the correspondingly best way to view and carry out matrix multiplication from the several available ones.

Let A and B be matrices compatible in the sense that the matrix product, AB, is defined, i.e., assume that $\#cols(A) = \#rows(B)$.

column view: Each column of AB is the linear combination of the columns of A with weights as specified by the corresponding column of B. This view is reflected in the `function MatMatSax` in the book. In symbols:

$$(AB)(:,j) = \sum_k A(:,k)B(k,j).$$

Therefore, if M is compatible with both B and C, then $[B,C]$ makes sense, and

$$M[B,C] = [MB,MC].$$

For vectors x and y (of compatible lengths), Ax is the linear combination of the columns of A using the entries of x as weights, i.e.,

$$Ax = \sum_j A(:,j)x(j),$$

while, for each j, $(y*B)(j)$ is the dot product of y with the jth column of B.

row view: Each row of AB is the linear combination of the rows of B with weights as specified by the corresponding row of A. This view is not reflected in the book. In symbols:

$$(AB)(i,:) = \sum_k A(i,k)B(k,:).$$

Therefore, if both A and C are compatible with N, then $[A;C]$ makes sense, and

$$[A;C]N = [AN;CN].$$

For vectors x and y (of compatible lengths), $y*B$ is the linear combination of the rows of B using the entries of y as weights, i.e.,

$$y*B = \sum_i y(i)B(i,:),$$

while, for each i, $(Ax)(i)$ is the dot product of the ith row of A with x.

Finally, there is the **outer product view**, reflected in the book's `MatMatOuter`, namely

$$AB = A(:,1)*B(1,:) + A(:,2)*B(2,:) + A(:,3)*B(3,:) + \cdots.$$

Each of the summands here is, by definition, the **outer product** of two vectors. While the 'inner product' or 'scalar product' or 'dot product' of two vectors (such as $A(i,:)*B(:,j)$ for compatible matrices A and B) is defined only when the two vectors have the same length, and results in a scalar (well, in `matlab`, that means a matrix of size `[1,1]`), the outer product is defined for any two vectors x and y regardless of their lengths, and results in a matrix, of size `[length(x),length(y)]`. However, in `matlab`, the first vector will have to be given by a column matrix and the second by a row matrix.

The outer product is very useful in matrix theory as well as in matrix calculations. In matrix *theory*, the most efficient and illuminating definition of the *rank* of a matrix A is as the minimum number of summands in a sum of outer products which adds up to A. In matrix *calculations*, **elementary** matrices, i.e., matrices of the form `eye(n) + A(:,1)*B(1,:)`, are the main tool for transforming a given matrix problem into a more tractable one.

Which of these four different views is to be used in the construction of the matrix product AB depends very much on just how matrices are stored and, possibly, on specifics of the programming language being used. In `matlab`, we simply say `A*B` and trust that the version of `matlab` we use will have been designed properly for the kind of machine on which we run it (as is confirmed by the timing run reported on page 182 of the book). However, it is important to keep in mind the above four different ways of viewing the product AB since, for special matrices, one of the four may be obviously superior to all the others, and because each of the four will, at times, best reflect the theoretical purpose of a particular matrix multiplication.

The most important property of the matrix product is its **associativity**: *If A, B, C are compatible matrices in the sense that both AB and BC are defined, then*

$$(AB)C = A(BC).$$

matrix norms

One measures the distance between vectors by the *norm* of their difference. In particular, the size of length of a vector x in this sense is its norm, $\|x\|$, i.e., its distance from the zero vector. One also measures the distance between two matrices by the norm of their difference, with the norm of a matrix A its corresponding distance $\|A\|$ from the zero matrix.

One often refers to the norm of a matrix or a vector as measuring the 'size' or 'length' of that matrix or vector and does *NOT* mean `matlab`'s `size` or `length`.

The standard or Euclidean norm of an n-vector is the number

$$\|x\| = \|x\|_2 := \sqrt{\sum_{j=1}^{n} |x_j|^2},$$

and this is a fine tool except for the following fact. The mathematically most satisfactory definition of the norm $\|A\|$ of a matrix is as the smallest number for which the inequality

$$\|A * x\| \le \|A\| \|x\|$$

is true for all compatible vectors x. The difficulty with the Euclidean norm of a vector is that the corresponding norm $\|A\|_2$ is not that easy to compute exactly.

By contrast, if the vector norm is one of

$$\|x\|_1 := \sum_i |x_i|, \qquad \|x\|_\infty := \max_i |x_i|,$$

then the corresponding matrix norms are easy to compute. They are

$$\|A\|_1 := \max_j \|A(:,j)\|_1, \qquad \|A\|_\infty := \max_i \|A(i,:)\|_1.$$