Matlab – logical vectors and matrices

Logical matrix (or vector) is a result of a comparison or other logical operation whose arguments are matrices (or vectors). In Matlab, the following comparison operators are available:

```
<    >    <=    >=    ==    ~=
```

the following logical matrix (vector) element-wise operators:

```
&    |    ~    xor(A,B)
```

and scalar logical operators:

```
&&    ||
```

The arguments of scalar logical operators must be logical scalars. The scalar logical operators are optimized in a sense that a logical (scalar) expression is evaluated (from left to right) until the result is obvious.

Let

```
x = [1 4 2 5], y = [3 1 -2 5]
```

and

```
A = [1 2 3; 4 5 6; 7 8 9], B = magic(3)
```

Then, the call

```
x < y
```

will compare the elements of the two vectors resulting with

```
ans =
  1×4 logical array
   1   0   0   0
```

Similarly, after comparing the two above matrices,

```
A < B
```

we shall get the following result:

```
ans =
  3×3 logical array
   1   0   1
   0   0   1
   0   1   0
```

The 0 above represents the logical value false, and 1 the logical value true. One may write, e.g.:

```
true
```

to get

```
ans =
  logical
   1
```

Since we can compare vectors and matrices, a natural question arises, how the conditional statement `if x < y ...` (where `x` and `y` are vectors/matrices) is interpreted by Matlab? In such a case, to decide the further execution of the `if` statement (the `if` statement requires a scalar logic value), the `all(...)` function will be automatically called in case of vectors, and `all(...,'all')` in case of matrices (see also below or call `help all`). However, is it a very good practice to use `all()` or `any()`, depending on our intentions, function implicitly, e.g., `if all(x < y) ...`

The **all** and **any** functions are equivalent to multiargument logical operators **"and"** and **"or"**. These functions, similarly to, e.g., the **sum** function, work column-wise unless we ask otherwise (see the **help** for **all** and **any**). The **all** and **any** functions can, of course, be mixed together

```
all(any(A < 2*B))
```

One of the most useful features of logical matrices (or vectors) is that we can use them for indexing other matrices (vectors). The values true point the elements (positions) to be considered, while the values false – the elements (positions) to be ignored. Let us assume that we want to change all even elements of a matrix **B** to the opposite values. We can do this with one simple instruction:

```
B(mod(B,2) == 0) = -B(mod(B,2) == 0);
```

However, for larger matrices, it would be more efficient to call:

```
w = (mod(B,2) == 0);
B(w) = -B(w);
```

The brackets "around" the comparison condition in the first of the two above instructions are not required. They were added for better readability.

When we assign values to only limited number of elements of a matrix (vector), there is rule that the number of elements that are set must be equal to the number of elements that appear on the right hand side of the "=" instruction unless we assign a single scalar value. In this case, this value is assigned to all selected elements of a matrix (vector).

For example, to set all negative elements of a matrix (vector) to zero, one may write:

```
B(B < 0) = 0;
```

The logical values false (logical 0) and true (logical 1) can be used as 0 and 1 in all arithmetic expressions. On the other hand, the (real) numbers 0 and 1 cannot be used for selective matrices indexing. For example, the following instruction is not correct:

```
x([1 0 1 0]) = 0;
```

However, we may do as follows:

```
x(logical([1 0 1 0])) = 0; % or: x([true false true false]) = 0;
```

An example exercise

Now, we shall create a function that finds the number of negative elements of a matrix. First we shall write a C language like, scalar, dual-loop solution:

```
function w = countneg(A)
% Counts the negative elements of A

s = size(A);
w = 0;
for i = 1:s(1)
    for j = 1:s(2)
        if A(i,j) < 0
            w = w + 1;
        end
    end
end
```

The above solution is pretty long and not very fast (especially that we check the matrix elements row-wise). It would be better to change the search order:

```
for j = 1:s(2)
    for i = 1:s(1)
```

However, a more efficient and simpler way to do this is as follows:

```
function w = countneg(A)
% Counts the negative elements of A

w = sum(A < 0,'all'); % or: w = sum(sum(A < 0));
```