Logical Operations and Round-off Errors

黃世強 (Sai-Keung Wong)

National Chiao Tung University, Taiwan

Relational Operators

The purpose of a relational operator

- Perform element-by-element comparisons between two arrays.
- Return a logical array of the same size.
  - An element is set to logical 1 (true) if the relation is true.
  - Otherwise, the element is set to logical 0(false).

Logical data type

- Use in relational conditions or to test state.
- Can have one of two values: true or false.
- Also useful in array indexing.
- Two-dimensional arrays can be sparse.

```
x = boolean([1 3 0 2 -1 0 2])
x =
  1×7 logical array
   1   1   0   1   1   0   1
```

Logical data type

```
x = boolean([1 3 0 2 -1 0 2])
x =
  1×7 logical array
   1   1   0   1   1   0   1

a(x) = 7
a = ?
```

Logical data type

```
x = boolean([1 3 0 2 -1 0 2])
x =
  1×7 logical array
   1   1   0   1   1   0   1

a(x) = 7
a =
    7     7     0     7     7     0     7
```

Example

```
clear
a([1 3 0 2 -1 0 2]) = 7
Subscript indices must either be real positive integers
or logicals.
```

```
a([1 3 7 2 1 5 2]) = 7          % the numbers are linear indices
```

```
a =
```

7	7	7	0	7	0	7	
1	2	3	4	5	6	7	linear index

Logical data type

```
x = boolean([1 3 0 2 -1 0 2; 3 2 1 7 0 4 0])
x =
  2×7 logical array


   1   1   0   1   1   0   1
   1   1   1   1   0   1   0
```

Logical data type

```
x = boolean([1 3 0 2 -1 0 2; 3 2 1 7 0 4 0])
x =
  2×7 logical array
   1   1   0   1   1   0   1
   1   1   1   1   0   1   0
a(x) = 7
a = ?
```

Logical data type

```
x = boolean([1 3 0 2 -1 0 2; 3 2 1 7 0 4 0])
x =
  2×7 logical array
   1   1   0   1   1   0   1
   1   1   1   1   0   1   0
a(x) = 7
a =
   7   7   0   7   7   0   7
   7   7   7   7   0   7   0
```

Do we obtain
a 2x7 matrix?

Logical data type

```
clear
x = boolean([1 3 0 2 -1 0 2; 3 2 1 7 0 4 0])
x =
  2×7 logical array
   1   1   0   1   1   0   1
   1   1   1   1   0   1   0
a(x) = 7
a =
  Columns 1 through 13
    7     7     7     7     0     7     7     7     7     0     0    7    7
```

a is a row vector of size 14.

Fill in a column-major order.

When you do this exercise, make sure that you clear x and a first.

Logical data type

```
clear
x = boolean([1 3 0 2; 0 -1 0 2; 3 2 1 7;  0 4 0 0])
x =
  4×4 logical array
   1   1   0   1
   0   1   0   1
   1   1   1   1
   0   1   0   0
a(x) = 7
a =
7     0     7     0     7     7     7     7     0     0     7   0     7     7     7
```

Fill in a column-major order.

When you do this exercise, make sure that you clear x and a first.

Logical data type

```
x = boolean([1 1 1 0 0 0])
x =
  1×6 logical array
   1   1   1   0   0   0


>> a(x) = 7
a =
     7     7     7
```

```
x = boolean([1 0 1 0 0 0; ...
    1 1 0 0 1 0 ])
x =
  2×6 logical array
   1   0   1   0   0   0
   1   1   0   0   1   0
a(x) = 7
a =
7  7  0  7  7  0  0  0  0 7
```

Logical data type

```
clear
x = boolean([1 0 1 0 0 0; ...
   1 1 0 0 1 0 ])
x =
  2×6 logical array
   1   0   1   0   0   0
   1   1   0   0   1   0
a(x) = 7
a =
7  7  0  7  7  0  0  0  0 7
```

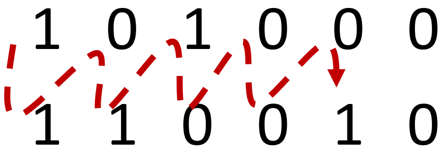
Fill in a column-major order

If a is not initialized, a is created as a row vector. 7 is filled to a if the position value is true. The process is stopped if all the remaining values are false.

```
clear
x = boolean([1 0 1 0 0 0; ...
   1 1 0 0 1 0 ])
x =
  2×6 logical array
   1   0   1   0   0   0
   1   1   0   0   1   0
a = zeros(size(x));
>> a(x) = 7
a =
     7     0     7     0     0     0
     7     7     0     0     7     0
```

Logical data type

```
clear
x = boolean([1 0 1 0 0 0; ...
   1 1 0 0 1 0 ])
x =
  2×6 logical array
```



```
   1   0   1   0   0   0
   1   1   0   0   1   0
a(x) = 7
a =
7  7  0  7  7  0  0  0  0 7
```

Fill in a column-major order

If a is not initialized, a is created as a row vector. 7 is filled to a if the position value is true. The process is stopped if all the remaining values are false.

```
clear
x = boolean([1 0 1 0 0 0; ...
   1 1 0 0 1 0 ])
x =
  2×6 logical array
   1   0   1   0   0   0
   1   1   0   0   1   0
a = zeros(size(x));
>> a(x) = 7
a =
     7     0     7     0     0     0
     7     7     0     0     7     0
```

Relational Operators for Logical Operations

Operator	Description
<	Less than
>	Greater than
==	Equal to (logical equality)
~	Not
<=	Less than or equal to
>=	Greater than or equal to
~=	Not equal to

Relational Operators for Logical Operations

Example	Answer	Data type
2 < 3		
[3 4] > [2 1]		
[1 2 3 4]== [4 3 3 4]		
~[2 -1 2 1 0]		
[2 3; 4 5] <= [2 1; 7 5]		
[1 2 5]' >= [1 3 4]'		
[2 4 5 1] ~= [1 5 3 1]		

Relational Operators for Logical Operations

Example	Answer	Data type
2 < 3	1	
[3 4] > [2 1]	1 1	
[1 2 3 4]== [4 3 3 4]	0 0 1 1	
~[2 -1 2 1 0]	0 0 0 0 1	
[2 3; 4 5] <= [2 1; 7 5]	1 0 1 1	
[1 2 5]' >= [1 3 4]'	1 0 1	
[2 4 5 1] ~= [1 5 3 1]	1 1 1 0	

Relational Operators for Logical Operations

Example	Answer	Data type
2 < 3	1	logical
[3 4] > [2 1]	1 1	1×2 logical array
[1 2 3 4]== [4 3 3 4]	0 0 1 1	1×4 logical array
~[2 -1 2 1 0]	0 0 0 0 1	1×5 logical array
[2 3; 4 5] <= [2 1; 7 5]	1 0 1 1	2×2 logical array
[1 2 5]' >= [1 3 4]'	1 0 1	3×1 logical array
[2 4 5 1] ~= [1 5 3 1]	1 1 1 0	1×4 logical array

Logical (or Boolean) Operators

- A logical expression produces a logical result.

Logical Operator	**Description**
&	Logical and
|	Logical or
~	Logical not

Logical (or Boolean) Operators

- A logical expression produces a logical result.

Expression	Answer	Data type
1 | false		
0 & true		
1 & (0 == 0)		
~(-1 & (2 == 2))		
[1 2 0] & [0 1 1]		
(~[1 2 -1]) | [0 1 -2]		
[1 2 -1] | [0 1 0]		

Logical (or Boolean) Operators

- A logical expression produces a logical result.

Expression	Answer	Data type
1 | false	1	logical
false | false	0	logical
true & true	1	logical
0 & true	0	logical
1 & (0 == 0)	1	logical
~(-1 & (2 == 2))	0	logical
[1 2 0] & [0 1 1]	0 1 0	1×3 logical array
(~[1 2 -1]) | [0 1 -2]	0 1 1	1×3 logical array
[1 2 -1] | [0 1 0]	1 1 1	1×3 logical array

Operation priority

1. **Arithmetic**
2. **Relational Operators**
3. **All ~ (Not) operators**
4. **All & (And) operators evaluated from left to right**
5. **All | (Or) operators evaluated from left to right**

```
A) ~0 - 2

B) 7 -4 + ~0 -2

C) 7 -4 + ~1 -1 & 5 -4

D) 1 + 5 > 7  -4 + ~1 -1 & 5 -4 | 6-6
```

Operation priority

1. **Arithmetic**
2. **Relational Operators**
3. **All ~ (Not) operators**
4. **All & (And) operators evaluated from left to right**
5. **All | (Or) operators evaluated from left to right**

A) ~0 – 2

B) B) 7 -4 + ~0 -2

C) 7 -4 + ~1 -1 & 5 -4

D) 1 + 5 > 7 -4 + ~1 -1 & 5 -4 | 6-6

A: -1

B: 2

C: true

D:true

Operation priority

1. **Arithmetic**
2. **Relational Operators**
3. **All ~ (Not) operators**
4. **All & (And) operators evaluated from left to right**
5. **All | (Or) operators evaluated from left to right**

A) ~0 - ~2

B) (7 |4) + ~0 -2

C) 7 -4 + ~1 -1 & 5 -5

D) 1 + 5 > 7 -4 + ~1 -1 & 5 -4 && 6-6

A: 1

B: 0

C: false

D: false

Logical Functions

function	Description
find	I = find(X) returns **the linear indices** corresponding to the nonzero entries of the array X.
sparse	S = sparse(X) converts a sparse or full matrix to sparse form by squeezing out any zero elements.
ind2sub	ind2sub is used to determine the equivalent subscript values corresponding to a given single index into an array.
nonzeros	nonzeros(S) is a full column vector of the nonzero elements of S. This gives the s, but not the i and j, from [i,j,s] = find(S).

Logical Functions

Example	answer	Description
a = 0 1 2 3 0 1 (find(a<=1)) '	1 3 4 6	
a = [0 1 2; ... 3 0 1]; sparse(a)	(2,1) 3 (1,2) 1 (1,3) 2 (2,3) 1	2x3 sparse array
a = 0 1 2 3 0 1 [i j] = ind2sub(size(a), 3) [A B] = ind2sub(size(a), [1 2 3 4])	i = 1 % row index j = 2 % column index A = 1 2 1 2 B = 1 1 2 2	Subscripts from linear index
a = 0 1 2 3 0 1 nonzeros(a) '	3 1 2 1	

Logical Functions

function	Description
ischar	char?
isinf	Inf?
isnan	Not a number?
isempty	Is [ ] ?

Round-off errors in numerical computations

```
x = sin(pi);
y = x;
x == y
ans =
  logical
   1
```

```
x = sin(pi);
y = 0;
x == y
ans =
  logical
   0
```

```
x = sin(pi);
y = 0;
x ~= y
ans =
  logical
   1
```

Round-off errors in numerical computations

```
x = 0.7 -0.3
x = 0.400000000000000


x == 0.4
ans =
  logical
   0


x - 0.4
= -5.551115123125783e-17
```

Round-off errors in numerical computations

$x^2 – x – 1.3 = 0.$

```
x1 = (1+sqrt(1+4*1.3))/2
x1 =
   1.744989959798873


x1^2 - x1 - 1.3
ans =
    -2.220446049250313e-16
```

Round-off errors in numerical computations

- When we check for correctness, we need to do an upper test.

c’ = f(x_0)

Assume that the actual value is c_0

How accurate do we need to be?

We check

abs(c’-c_0) <= e,

where e is a small number which depends on the complexity of the computation of f(x_0).

We can use |c’|*eps if |c’| is non-zero. However, if there is a term in f(x_0) is much larger than other terms, then probably it does not work.

Round-off errors in numerical computations

x^2 -3.754006020081200e-04x-3.3517068341857e-08 =0

```
a = 1;
b = -3.754006020081200e-04;
c = 3.3517068341857e-08;


x1 = (-b+sqrt(b^2-4*a*c))/2
x1 = 4.498995979879988e-04
>> x1^2 + b*x1 + c
ans =
    1.985233470127266e-23
```

```
>> eps*x1

ans =

    9.989777849117561e-20
```

Round-off errors in numerical computations

x^2 -3.754006020081200e-04x-3.3517068341857e-08 =0

```
a = 1;
b = -3.754006020081200e-04;
c = 3.3517068341857e-08;


x1 = (-b+sqrt(b^2-4*a*c))/2
x1 = 4.498995979879988e-04
>> x1^2 + b*x1 + c
ans =
    1.985233470127266e-23
```

```
>> eps*x1

ans =

    9.989777849117561e-20
```

Round-off errors in numerical computations

```
a = 1; b = -3.754006020081200e-04; c = 3.3517068341857e-08;
x1 = (-b+sqrt(b^2-4*a*c))/2
x = x1;
y = abs(x^2 + b*x + c)
if ( abs(y) <= abs(x*eps))
    fprintf('correct. Absolute error is %f\n', y);
else
    fprintf('Incorrect. Absolute error is %f\n', y);
end
==
x1 =    2.2910e-04
y =     6.6174e-24
correct. Absolute error is 0.000000
```

Round-off errors in numerical computations

```
a = 1; b = -3.754006020081200e-04; c = 3.3517068341857e-08;
x1 = (-b+sqrt(b^2-4*a*c))/2; x = x1; y = abs(x^2 + b*x + c)
err =  max( abs( [x^2, b*x, c] ) )*eps;
if ( abs(y) <= err)
    fprintf('correct. Absolute error is %f\n', y);
else
    fprintf('Incorrect. Absolute error is %f\n', y);
end
==
x1 =     2.291048255775620e-04
y =     6.617444900424221e-24
err =     1.909718815191355e-23
correct. Absolute error is 0.000000
```

Round-off errors in numerical computations: Large numbers

```
format long
a = 1; b = -37548; c = 1;
x1 = (-b+sqrt(b^2-4*a*c))/2;
x2 = (-b-sqrt(b^2-4*a*c))/2;
x = [x1 x2]
err = abs(x.^2 + b.*x + c)
e = max(abs([x.^2, b.*x, c]))*eps
```

```
x =
   1.0e+04 *
   3.754799997336742
0.000000002663258


err =
   1.0e-08 *
    0   0.255738741294920


e =
     3.130500976222805e-07
```

Round-off errors in numerical computations: Large numbers

```
format long
a = 1; b = -3754800; c = 1;
x1 = (-b+sqrt(b^2-4*a*c))/2;
x2 = (-b-sqrt(b^2-4*a*c))/2;
x = [x1 x2]
err = abs(x.^2 + b.*x + c)
e = max(abs([x.^2, b.*x, c]))*eps
```

```
x =
   1.0e+06 *
   3.754799999999733
0.000000000000266


err =
   1.0e-03 *
    0   0.121980905461783


e =
   0.003130500978443
```

Round-off errors in numerical computations: Large numbers

```
format long
a = 1; b = -375480000; c = 1;
x1 = (-b+sqrt(b^2-4*a*c))/2;
x2 = (-b-sqrt(b^2-4*a*c))/2;
x = [x1 x2]
err = abs(x.^2 + b.*x + c)
e = max(abs([x.^2, b.*x, c]))*eps
```

```
x =
   375480000          0

err =


     1     1


e =
    31.305009784432514
```

Round-off errors in numerical computations: Large numbers

```
format long
a = 1; b = -37548000000; c = 1;
x1 = (-b+sqrt(b^2-4*a*c))/2;
x2 = (-b-sqrt(b^2-4*a*c))/2;
x = [x1 x2]
err = abs(x.^2 + b.*x + c)
e = max(abs([x.^2, b.*x, c]))*eps
```

```
x =
   1.0e+10 *
   3.754800000000000                    0


err =
     1     1

e =
     3.130500978443251e+05
```