CHAPTER 5

Logical vectors

THE OBJECTIVES OF THIS CHAPTER ARE TO ENABLE YOU TO:

- Understand logical operators more fully
  And to introduce you to:
- Logical vectors and how to use them effectively in a number of applications
- Logical functions

CONTENTS

This chapter introduces a most powerful and elegant feature of MATLAB, viz., the *logical vectors*. The topic is so useful and, hence, important that it deserves a chapter of its own.

Try these exercises on the command line:

1. Enter the following statements:

```
r = 1;
r <= 0.5   % no semi-colon
```

   If you correctly left out the semi-colon after the second statement you will have noticed that it returned the value 0.
2. Now enter the expression `r >= 0.5` (again, no semi-colon). It should return the value 1. Note that we already saw in Chapter 2 that a logical expression in MATLAB involving only scalars returns a value of 0 if it is FALSE, and 1 if it is TRUE.
3. Enter the following statements:

```
r = 1:5;
r <= 3
```


http://dx.doi.org/10.1016/B978-0-12-394398-9.00005-8

Now the logical expression `r <= 3` (where `r` is a vector) returns a *vector*:

```
1  1  1  0  0
```

Can you see how to interpret this result? For each element of `r` for which `r <= 3` is true, 1 is returned; otherwise 0 is returned. Now enter `r == 4`. Can you see why `0 0 0 1 0` is returned?

When a vector is involved in a logical expression, the comparison is carried out *element by element* (as in an arithmetic operation). If the comparison is true for a particular element of the vector, the resulting vector, which is called a *logical vector*, has a 1 in the corresponding position; otherwise it has a 0. The same applies to logical expressions involving matrices.

You can also compare vectors with vectors in logical expressions. Enter the following statements:

```
a = 1:5;
b = [0 2 3 5 6];
a == b     % no semi-colon!
```

The logical expression `a == b` returns the logical vector

```
0   1   1   0   0
```

because it is evaluated element by element, i.e., `a(1)` is compared with `b(1)`, `a(2)` with `b(2)`, and so on.

5.1 EXAMPLES

5.1.1 Discontinuous graphs

One very useful application of logical vectors is in plotting discontinuities. The following script plots the graph, shown in Figure 5.1, defined by:

$$y(x) = \begin{cases} \sin(x) & (\sin(x) > 0), \\ 0 & (\sin(x) \leqslant 0), \end{cases}$$

over the range 0 to 3π:

```
x = 0 : pi/20 : 3 * pi;
y = sin(x);
y = y .* (y > 0);   % set negative values of sin(x) to zero
plot(x, y)
```

The expression `y > 0` returns a logical vector with 1s where $\sin(x)$ is positive, and 0s otherwise. Element-by-element multiplication by `y` with `.*` then picks out the positive elements of `y`.

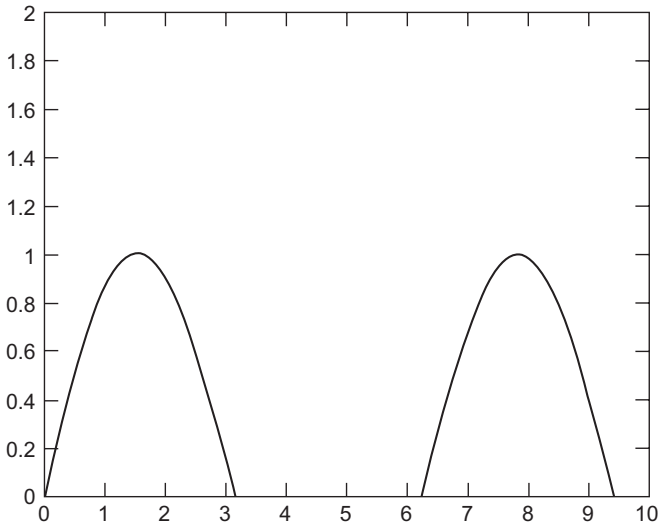

FIGURE 5.1 A discontinuous graph using logical vectors.

5.1.2 Avoiding division by zero

Suppose you want to plot the graph of $\sin(x)/x$ over the range -4π to 4π. The most convenient way to set up a vector of the x co-ordinates is:

```
x = -4*pi: pi/20 : 4*pi;
```

But then when you try:

```
y = sin(x)./x;
```

you get the `Divide by zero` warning, because one of the elements of `x` is exactly zero. A neat way around this problem is to use a logical vector to replace the zero with `eps`. This MATLAB function returns the difference between 1.0 and the next largest number which can be represented in MATLAB, i.e., approximately `2.2e-16`. Here is how to do it:

```
x = x + (x == 0)*eps;
```

The expression `x == 0` returns a logical vector with a single 1 for the element of x which is zero, and so eps is added only to that element. The following script plots the graph correctly—without a missing segment at $x=0$ (see Figure 5.2).

```
x = -4*pi : pi/20 : 4*pi;
x = x + (x == 0)*eps;       % adjust x = 0 to x = eps
y = sin(x) ./ x;
plot(x, y)
```

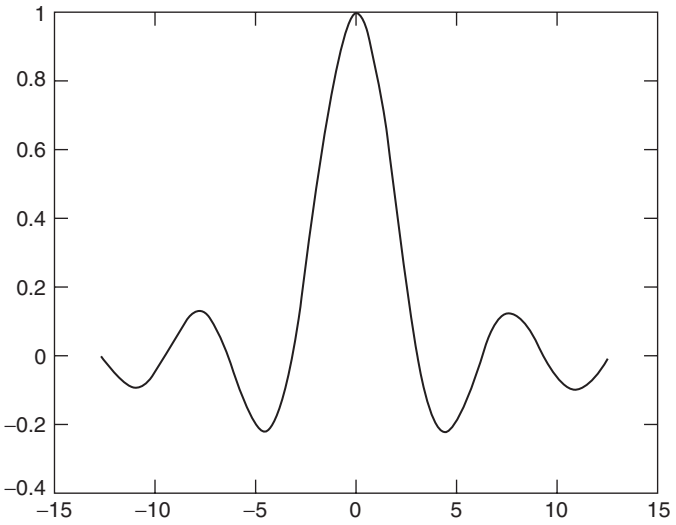

FIGURE 5.2 sin(*x*) / *x*.

When `x` has the value `eps`, the value of `sin(eps)/eps` has the correct limiting value of 1 (check it), instead of `NaN` (Not-a-Number) resulting from a division by zero.

5.1.3 Avoiding infinity

The following script attempts to plot $\tan(x)$ over the range $-3\pi/2$ to $3\pi/2$. If you are not too hot at trig graphs, perhaps you should sketch the graph roughly with pen and paper before you run the script!

```
x = -3/2*pi : pi/100 : 3/2*pi;
y = tan(x);
plot(x, y)
```

The MATLAB plot—the left graph in Figure 5.3—should look nothing like your sketch. The problem is that $\tan(x)$ approaches $\pm\infty$ at odd multiples of $\pi/2$. The scale on the MATLAB plot is therefore very large (about 10^{15}), making it impossible to see the structure of the graph anywhere else.

If you add the statement:

```
y = y.* (abs (y) < 1e10); % remove the big ones
```

just before the `plot` statement you'll get a much nicer graph, as shown in Figure 5.3b. The expression `abs(y) < 1e10` returns a logical vector which is zero only at the asymptotes. The graph thus goes through zero at these points,

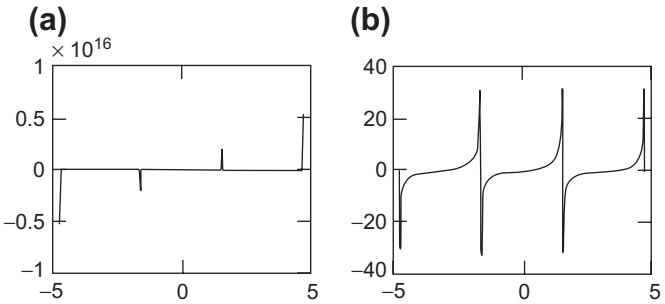

FIGURE 5.3 Variations on tan(*x*): (a) Unrestricted vertical coordinate. (b) Restricted vertical coordinate.

which incidentally draws nearly vertical asymptotes for you. These "asymptotes" become more vertical as the increment in `x` becomes smaller.

5.1.4 Counting random numbers

The function `rand` returns a (pseudo-)random number in the interval [0, 1); `rand(1, n)` returns a row vector of `n` such numbers. Work out the following problem on the command line:

1. Set up a vector `r` with seven random elements (leave out the semi-colon so that you can see its elements):

   ```
   r = rand (1,7)   % no semi-colon
   ```

   Check that the logical expression `r < 0.5` gives the correct logical vector.
2. Using the function `sum` on the logical expression `r < 0.5` will effectively count how many elements of `r` are less than 0.5. Try it, and check your answer against the values displayed for `r`:

   ```
   sum( r < 0.5 )
   ```

3. Now use a similar statement to count how many elements of `r` are greater than or equal to 0.5 (the two answers should add up to 7, shouldn't they?).
4. Since `rand` generates *uniformly distributed* random numbers, you would expect the number of random numbers which are less than 0.5 to get closer and closer to half the total number as more and more are generated.

   Generate a vector of a few thousand random numbers (suppress display with a semi-colon this time) and use a logical vector to count how many are less than 0.5.

   Repeat a few times with a new set of random numbers each time. Because the numbers are random, you should never get quite the same answer each time.

Without logical vectors this problem is a little more involved. Here is the program:

```
tic                     % start
a = 0;                  % number >= 0.5
b = 0;                  % number < 0.5

for n = 1:5000
  r = rand;             % generate one number per loop
  if r >= 0.5
    a = a + 1;
  else
    b = b + 1;
  end;
end;

t = toc;                % finish
disp( ['less than 0.5: ' num2str(a)] )
disp( ['time: ' num2str(t)] )
```

It also takes a lot longer. Compare times for the two methods on your computer.

5.1.5 Rolling dice

When a fair dice is rolled, the number uppermost is equally likely to be any integer from 1 to 6. So if `rand` is a random number in the range [0, 1), `6 * rand` will be in the range [0, 6), and `6 * rand + 1` will be in the range [1, 7), i.e., between 1 and 6.9999. Discarding the decimal part of this expression with `floor` gives an integer in the required range. Try the following exercises:

1. Generate a vector `d` of 20 random integers in the range 1–6:

   ```
   d = floor(6 * rand(1, 20)) + 1
   ```

2. Count the number of "sixes" thrown by summing the elements of the logical vector `d == 6`.
3. Verify your result by displaying `d`.
4. Estimate the probability of throwing a six by dividing the number of sixes thrown by 20. Using random numbers like this to mimic a real situation based on chance is called *simulation*.
5. Repeat with more random numbers in the vector `d`. The more you have, the closer the proportion of sixes gets to the theoretical expected value of 0.1667, i.e., 1/6.
6. Can you see why it would be incorrect to use `round` instead of `floor`? The problem is that `round` rounds in both directions, whereas `floor` rounds everything down.

5.2 LOGICAL OPERATORS

We saw briefly in Chapter 2 that logical expressions can be constructed not only from the six relational operators, but also from the three *logical operators* shown in Table 5.1. Table 5.2 shows the effects of these operators on the general logical expressions *lex1* and *lex2*.

The OR operator `(|)` is technically an *inclusive* OR, because it is true when either or both of its operands are true. MATLAB also has an *exclusive* OR function `xor(a, b)` which is 1 (true) only when either *but not both of* `a` and `b` is 1 (Table 5.2).

MATLAB also has a number of functions that perform bitwise logical operations. See Help on `ops`.

The precedence levels of the logical operators, among others, are shown in Table 5.3. As usual, precedences may be overridden with brackets, e.g.,

Table 5.1 Logical Operators

Operator	Meaning
~	NOT
&	AND
|	OR

Table 5.2 Truth Table (T = True; F = False)

lex1	lex2	~lex1	lex1 & lex2	lex1|lex2	xor(lex1, lex2)
F	F	T	F	F	F
F	T	T	F	T	T
T	F	F	F	T	T
T	T	F	T	T	F

Table 5.3 Operator Precedence (See **Help** on `operator precedence`)

Precedence	Operators
1.	( )
2.	^ .^ ' .' (pure transpose)
3.	+ (unary plus) - (unary minus) ~ (NOT)
4.	*/ \ .* ./ .\
5.	+ (addition) - (subtraction)
6.	:
7.	> < >= <= == ~=
8.	& (AND)
9.	| (OR)

```
~ 0 & 0
```

returns 0 (false), whereas

```
~ (0 & 0)
```

returns 1 (true). Some more examples:

```
(b * (b == 4) * a * c) & (a ~= 0)
(final >= 60) & (final < 70)
(a ~= 0) | (b ~= 0) | (c != 0)
~((a == 0) & (b == 0) & (c == 0))
```

It is never wrong to use brackets to make the logic clearer, even if they are syntactically unnecessary. Incidentally, the last two expressions above are logically equivalent, and are false only when `a` = `b` = `c` = 0. It makes you think, doesn't it?

5.2.1 Operator precedence

You may accidentally enter an expression like:

```
2 > 1 & 0
```

(try it) and be surprised because MATLAB (a) accepts it, and (b) returns a value of 0 (false). It is surprising because:

a. `2 > 1 & 0` doesn't appear to make sense. If you have got this far you deserve to be let into a secret. MATLAB is based on the notorious language C, which allows you to mix different types of operators in this way (Pascal, for example, would never allow such flexibility!).

b. We instinctively feel that `&` should have the higher precedence. `1 & 0` evaluates to 0, so `2 > 0` should evaluate to 1 instead of 0. The explanation is due partly to the resolution of surprise (a). MATLAB groups its operators in a rather curious and non-intuitive way. The complete operator precedence is given in Table 5.3 (reproduced for ease of reference in Appendix B). (Recall that the transpose operator "performs a *complex conjugate* transpose on complex data;" the dot-transpose operator "performs a 'pure' transpose without taking the complex conjugate.") Brackets always have the highest precedence.

5.2.2 Danger

I have seen quite a few students incorrectly convert the mathematical inequality $0<r<1$, say, into the MATLAB expression:

```
0 < r < 1
```

Once again, the first time I saw this I was surprised that MATLAB did not report an error. Again, the answer is that MATLAB doesn't really mind how you mix

up operators in an expression. It simply churns through the expression according to its rules (which may not be what you expect).

Suppose r has the value 0.5. Mathematically, the inequality is true for this value of r since it lies in the required range. However, the expression `0 < r < 1` is evaluated as 0. This is because the left hand operation (`0 < 0.5`) is first evaluated to 1 (true), followed by `1 < 1` which is false.

Inequalities like this should rather be coded as:

```
(0 < r) & (r < 1)
```

The brackets are not strictly necessary (see Table 5.3) but they certainly help to clarify the logic.

5.2.3 Logical operators and vectors

The logical operators can also operate on vectors (of the same size), returning logical vectors, e.g.,

```
~(~[1 2 0 -4 0])
```

replaces all the non-zeros by 1s, and leaves the 0s untouched. Try it.

The script in Section 5.1 that avoids division by zero has the critical statement:

```
x = x + (x == 0)*eps;
```

This is equivalent to:

```
x = x + (~x)*eps;
```

Try it, and make sure you understand how it works.

Exercise

Work out the results of the following expressions before checking them at the command line:

```
a = [-1 0 3];
b = [0 3 1];
~a
a & b
a | b
xor(a, b)
a > 0 & b > 0
a > 0 | b > 0
~ a > 0
a + (~ b)
a > ~ b
~ a > b
~ (a > b)
```

5.3 SUBSCRIPTING WITH LOGICAL VECTORS

We saw briefly in Chapter 2 that elements of a vector may be referenced with subscripts, and that the subscripts themselves may be vectors, e.g.,

```
a = [-2 0 1 5 9];
a([5 1 3])
```

returns,

```
9  −2  1
```

i.e., the fifth, first, and third elements of `a`. In general, if `x` and `v` are vectors, where `v` has *n* elements, then `x(v)` means:

```
[x(v(1)), x(v(2)), ..., x(v(n))]
```

Now with `a` as defined above, see what:

```
a(logical([0 1 0 1 0]))
```

returns. The function `logical(v)` returns a logical vector, with elements which are 1 or 0 according as the elements of `v` are non-zero or 0.

A summary of the rules for the use of a logical vector as a subscript are as follows:

- A logical vector `v` may be a subscript of another vector `x`.
- Only the elements of `x` corresponding to 1s in `v` are returned.
- `x` and `v` must be the same size.

Thus, the statement above returns:

```
 0  5
```

i.e., the second and fourth elements of `a`, corresponding to the 1s in `logical([0 1 0 1 0])`. What will:

```
a(logical([1 1 1 0 0]))
```

return? And what about `a(logical([0 0 0 0 0]))`?

Logical vector subscripts provide an elegant way of removing certain elements from a vector, e.g.,

```
a = a(a > 0)
```

removes all the non-positive elements from `a`, because `a > 0` returns the logical vector `[0 0 1 1 1]`. We can verify incidentally that the expression `a > 0` is a logical vector, because the statement:

```
islogical(a > 0)
```

returns 1. However, the *numeric* vector `[0 0 1 1 1]` is not a logical vector; the statement:

```
islogical([0 0 1 1 1])
```

returns 0.

5.4 LOGICAL FUNCTIONS

MATLAB has a number of useful *logical functions* that operate on scalars, vectors and matrices. Examples are given in the following list (where `x` is a vector unless otherwise stated). See Help on `logical functions`. (The functions are defined slightly differently for matrix arguments—see Chapter 6 or Help.)

`any(x)` returns the scalar 1 (true) if *any* element of `x` is non-zero (true).

`all(x)` returns the scalar 1 if *all* the elements of `x` are non-zero.

`exist('a')` returns 1 if `a` is a workspace variable. For other possible return values see `help`. Note that `a` must be enclosed in apostrophes.

`find(x)` returns a vector containing the subscripts of the *non-zero* (true) elements of `x`, so for example,

```
a = a( find(a) )
```

removes all the zero elements from `a`! Try it. Another use of `find` is in finding the subscripts of the largest (or smallest) elements in a vector, when there is more than one. Enter the following:

```
x = [8 1 −4 8 6];
find(x >= max(x))
```

This returns the vector `[1 4]`, which are the subscripts of the largest element (8). It works because the logical expression `x >= max(x)` returns a logical vector with 1s only at the positions of the largest elements.

`isempty(x)` returns 1 if `x` is an empty array and 0 otherwise. An empty array has a size of 0-by-0.

`isinf(x)` returns 1s for the elements of `x` which are `+Inf` or `-Inf`, and 0s otherwise.

`isnan(x)` returns 1s where the elements of `x` are `NaN` and 0s otherwise. This function may be used to remove `NaN`s from a set of data. This situation could arise while you are collecting statistics; missing or unavailable values can be temporarily represented by `NaN`s. However, if you do any calculations involving the `NaN`s, they propagate through intermediate calculations to the final result. To avoid this, the `NaN`s in a vector may be removed with a statement like:

```
x(isnan(x)) = [ ]
```

MATLAB has a number of other logical functions starting with the characters `is`. Search for `is*` in the Help for the complete list.

5.4.1 Using `any` and `all`

Because `any` and `all` with vector arguments return *scalar*s, they are particularly useful in `if` statements. For example:

```
if all(a >= 1)
   do something
end
```

means "if all the elements of the vector `a` are greater than or equal to 1, then *do something*."

Recall from Chapter 2 that a vector condition in an `if` statement is true only if *all* its elements are non-zero. So if you want to execute *statement* below when two vectors `a` and `b` are equal (i.e., the same) you could say:

```
if a == b
   statement
end
```

since `if` considers the logical vector returned by `a == b` true only if every element is a 1.

If, on the other hand, you want to execute *statement* specifically when the vectors `a` and `b` are *not* equal, the temptation is to say:

```
if a ~= b      % wrong wrong wrong!!!
   statement
end
```

However this will *not* work, since *statement* will only execute if *each* of the corresponding elements of `a` and `b` differ. This is where `any` comes in:

```
if any(a ~= b)      % right right right!!!
   statement
end
```

This does what is required since `any(a~= b)` returns the scalar 1 if *any* element of `a` differs from the corresponding element of `b`.

5.5 LOGICAL VECTORS INSTEAD OF `elseif` LADDERS

Those of us who grew up on more conventional programming languages in the last century may find it difficult to think in terms of using logical vectors when solving general problems. It's a nice challenge whenever writing a program to ask yourself whether you can possibly use logical vectors. They are almost

always faster than other methods, although often not as clear to read later. You must decide when it's important for you to use logical vectors. But it's a very good programming exercise to force yourself to use them whenever possible! The following example illustrates these points by first solving a problem conventionally, and then with logical vectors.

It has been said that there are two unpleasant and unavoidable facts of life: death and income tax. A very simplified version of how income tax is calculated could be based on the following table:

Taxable income	Tax payable
$10,000 or less	10% of taxable income
Between $10,000 and $20,000	$1000 + 20% of amount by which taxable income exceeds $10,000
More than $20,000	$3000 + 50% of amount by which taxable income exceeds $20,000

The tax payable on a taxable income of $30,000, for example, is:

$3000 + 50% of ($30,000 − $20,000), i.e., $8000.

We would like to calculate the income tax on the following taxable incomes (in dollars): 5000, 10,000, 15,000, 30,000, and 50,000.

The conventional way to program this problem is to set up a vector with the taxable incomes as elements and to use a `loop` with an `elseif` ladder to process each element, as follows:

```
% Income tax the old-fashioned way

inc = [5000 10000 15000 30000 50000];

for ti = inc

  if ti < 10000
    tax = 0.1 * ti;
  elseif ti < 20000
    tax = 1000 + 0.2 * (ti - 10000);
  else
    tax = 3000 + 0.5 * (ti - 20000);
  end;

  disp( [ti tax] )
end;
```

Here is the output, suitably edited (note that the amount of tax paid changes continuously between tax "brackets"—each category of tax is called a bracket):

```
Taxable income    Income tax

       5000.00        500.00
      10000.00       1000.00
      15000.00       2000.00
      30000.00       8000.00
      50000.00      18000.00
```

Now here is the logical way:

```
% Income tax the logical way

inc = [5000 10000 15000 30000 50000];

tax = 0.1 * inc .* (inc <= 10000);
tax = tax + (inc > 10000 & inc <= 20000) ...
                            .* (0.2 * (inc-10000) + 1000);
tax = tax + (inc > 20000) .* (0.5 * (inc-20000) + 3000);

disp( [inc' tax'] );
```

To understand how it works, it may help to enter the statements on the command line. Start by entering the vector `inc` as given. Now enter:

```
inc <= 10000
```

which should give the logical vector `[1 1 0 0 0]`. Next enter:

```
inc .* (inc <= 10000)
```

which should give the vector `[5000 10000 0 0 0]`. This has successfully selected only the incomes in the first bracket. The tax for these incomes is then calculated with:

```
tax = 0.1 * inc .* (inc <= 10000)
```

which returns `[500 1000 0 0 0]`. Now for the second tax bracket. Enter the expression:

```
inc > 10000 & inc <= 20000
```

which returns the logical vector `[0 0 1 0 0]`, since there is only one income in this bracket. Now enter:

```
0.2 * (inc-10000) + 1000
```

This returns `[0 1000 2000 5000 9000]`. Only the third entry is correct. Multiplying this vector by the logical vector just obtained blots out the other entries, giving `[0 0 2000 0 0]`. The result can be safely added to the vector `tax` since it will not affect the first two entries already there.

I am sure you will agree that the logical vector solution is more interesting than the conventional one!

SUMMARY

- When a relational and/or logical operator operates on a vector expression, the operation is carried out element by element. The result is a logical vector consisting of 0s (FALSE) and 1s (TRUE).
- A vector may be subscripted with a logical vector of the same size. Only the elements corresponding to the 1s in the logical vector are returned.
- When one of the logical operators (~ & |) operates on an expression any non-zero value in an operand is regarded as TRUE; zero is regarded as FALSE. A logical vector is returned.
- Arithmetic, relational, and logical operators may appear in the same expression. Great care must be taken in observing the correct operator precedence in such situations.
- Vectors in a logical expression must all be the same size.
- If a logical expression is a vector or a matrix, it is considered true in an "`if`" statement only if all its elements are non-zero.
- The logical functions `any` and `all` return scalars when taking vector arguments, and are consequently useful in "`if`" statements.
- Logical vectors may often be used instead of the more conventional `elseif` ladder. This provides faster more elegant code, but requires more ingenuity and the code may be less clear to read later on.

CHAPTER EXERCISES

5.1 Determine the values of the following expressions yourself before checking your answers using MATLAB. You may need to consult Table 5.3:

(a) `1 & -1`
(b) `13 & ~ (-6)`
(c) `0 < -2|0`
(d) `~[1 0 2] * 3`
(e) `0 <= 0.2 <= 0.4`
(f) `5 > 4 > 3`
(g) `2 > 3 & 1`

5.2 Given that `a = [1 0 2]` and `b = [0 2 2]` determine the values of the following expressions. Check your answers with MATLAB:

(a) `a~= b`

(b) `a < b`

(c) `a < b < a`

(d) `a < b < b`

(e) `(a |~a)`

(f) `b & (~b)`

(g) `a(~(~b))`

(h) `a = b == a` (determine final value of `a`)

5.3 Write some MATLAB statements on the command line which use logical vectors to count how many elements of a vector `x` are negative, zero, or positive. Check that they work, e.g., with the vector:

```
[-4 0 5 -3 0 3 7 -1 6]
```

5.4 The Receiver of Revenue (Internal Revenue Service) decides to change the tax table used in Section 5.5 slightly by introducing an extra tax bracket and changing the tax-rate in the third bracket, as follows:

Taxable income	Tax payable
$10,000 or less	10% of taxable income
Between $10,000 and $20,000	$1000 + 20% of amount by which taxable income exceeds $10,000
Between $20,000 and $40,000	$3000 + 30% of amount by which taxable income exceeds $20,000
More than $40,000	$9000 + 50% of amount by which taxable income exceeds $40,000

Amend the logical vector script to handle this table, and test it on the following list of incomes (dollars): 5000, 10,000, 15,000, 22,000, 30,000, 38,000, and 50,000.

5.5 A certain company offers seven annual salary levels (dollars): 12,000, 15,000, 18,000, 24,000, 35,000, 50,000, and 70,000. The number of employees paid at each level are, respectively: 3000, 2500, 1500, 1000, 400, 100, and 25. Write some statements at the command line to find the following:

(a) The average salary level. Use mean. (Answer: 32,000)

(b) The number of employees above and below this average salary level. Use logical vectors to find which salary levels are above and below the average level. Multiply these logical vectors element by element with the employee vector, and `sum` the result. (Answer: 525 above, 8000 below)

(c) The *average salary earned* by an individual in the company (i.e., the total annual salary bill divided by the total number of employees). (Answer: 17,038.12).

5.6 Write some statements on the command line to remove the largest element(s) from a vector. Try it out on `x = [1 2 5 0 5]`. The idea is to end up with `[1 2 0]` in `x`. Use `find` and the empty vector `[ ]`.

5.7 The electricity accounts of residents in a very small rural community are calculated as follows:

- if 500 units or less are used the cost is 2 cents per unit;
- if more than 500, but not more than 1000 units are used, the cost is $10 for the first 500 units, and then 5 cents for every unit in excess of 500;
- if more than 1000 units are used, the cost is $35 for the first 1000 units plus 10 cents for every unit in excess of 1000;
- in addition, a basic service fee of $5 is charged, no matter how much electricity is used.

The five residents use the following amounts (units) of electricity in a certain month: 200, 500, 700, 1000, and 1500. Write a program which uses logical vectors to calculate how much they must pay. Display the results in two columns: one for the electricity used in each case, and one for amount owed. (Answers: $9, $15, $25, $40, $90)