Program Design and Algorithm Development

Objectives

- Program design
- User-defined MATLAB functions
- Logical vectors

Program Design (1)

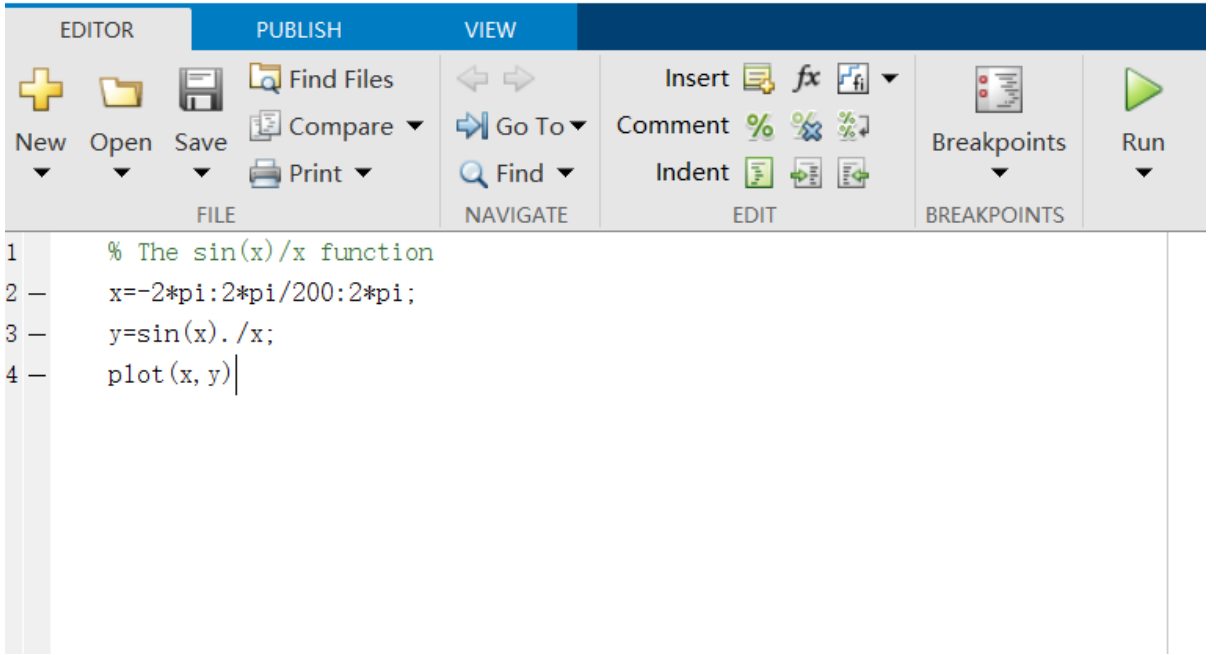
There are numerous toolboxes available through MathWorks on a variety of engineering and scientific topics. A great example is the Heat Transfer Toolbox, which provides reference standards, environmental models, and thermos fluid coefficients which can be imported for advanced thermal engineering designs.

Users are also able to design their own toolbox to be included among those already available with MATLAB, such as Simulink, Symbolic Math, and Control Systems. This is a big advantage of MATLAB. It allows you to customize your working environment to meet your own needs. It is not only the “mathematics handbook” of today’s students, engineer, and scientist, it is also a useful environment in which to develop software tools that go beyond any handbook to help you to solve relatively complicated mathematical problems.

To design a successful program you need to understand a problem thoroughly and break it down into its most fundamental logical stages. In other words, you have to develop a systematic procedure or algorithm for solving it.

Program Design (2)

The goals in designing a software tool are that it works, it can easily be read and understood, and, hence, it can be systematically modified when required. For programs to work well they must satisfy the requirements associated with the problem or class of problems they are intended to solve. The specifications (i.e., the detailed description of purpose, or function, inputs, method of processing, outputs, and any other special requirements) must be known to design an effective algorithm or computer program, which must work completely and correctly. That is, all options should be usable without error within the limits of the specifications



Program Design (3)

A well written code, when it works, is much more easily evaluated in the testing phase of the design process. If changes are necessary to correct sign mistakes and the like, they can be easily implemented. One thing to keep in mind when you add comments to describe the process programmed is this: Add enough comments and references so that a year from the time you write the program you know exactly what was done and for what purpose. Note that the first few comment lines in a script file are displayed in the Command Window when you type `help` followed by the name of your file (file naming is also an art).

Program Design (4)

The design process is outlined as follows:

Step 1: *Problem analysis.* The context of the proposed investigation must be established to provide the proper motivation for the design of a computer program. The designer must fully recognize the need and must develop an understanding of the nature of the problem to be solved.

Step 2: *Problem statement.* Develop a detailed statement of the mathematical problem to be solved with a computer program.

Step 3: *Processing scheme.* Define the inputs required and the outputs to be produced by the program.

Step 4: *Algorithm.* Design the step-by-step procedure in a *top-down* process that decomposes the overall problem into subordinate problems. The subtasks to solve the latter are refined by designing an itemized list of steps to be programmed. This list of tasks is the *structure plan* and is written in *pseudo-code* (i.e., a combination of English, mathematics, and anticipated MATLAB commands). The goal is a plan that is understandable and easily translated into a computer language.

Program Design (5)

Step 5: *Program algorithm.* Translate or convert the algorithm into a computer language (e.g., MATLAB) and debug the syntax errors until the tool executes successfully.

Step 6: *Evaluation.* Test all of the options and conduct a validation study of the program. For example, compare results with other programs that do similar tasks, compare with experimental data if appropriate, and compare with theoretical predictions based on theoretical methodology related to the problems to be solved. The objective is to determine that the subtasks and the overall program are correct and accurate. The additional debugging in this step is to find and correct *logical* errors (e.g., mistyping of expressions by putting a plus sign where a minus sign was supposed to be) and *runtime* errors that may occur after the program successfully executes (e.g., cases where division by zero unintentionally occurs).

Step 7: *Application.* Solve the problems the program was designed to solve. If the program is well designed and useful, it can be saved in your working directory (i.e., in your user-developed toolbox) for future use.

The projectile problem (1)

- Step 1.

In this example we want to calculate the flight of a projectile (e.g., a golf ball) launched at a prescribed speed and a prescribed launch angle. We want to determine the trajectory of the flight path and the horizontal distance the projectile (or object) travels before it hits the ground. Let us assume zero air resistance and a constant gravitational force acting on the object in the opposite direction of the vertical distance from the ground. The launch angle, θ_o, is defined as the angle measured from the horizontal (ground plane) upward toward the vertical direction, $0 < \theta_o \leqslant \pi/2$, where $\theta_o = 0$ implies a launch in the horizontal direction and $\theta_o = \pi/2$ implies a launch in the vertical direction (i.e., in the opposite direction of gravity). If $g = 9.81\ \mathrm{m/s^2}$ is used as the acceleration of gravity, the launch speed, V_o, must be entered in units of m/s. Thus, if the time, $t > 0$, is the time in seconds (s) from the launch time of $t = 0$, the distance traveled in x (the horizontal direction) and y (the vertical direction) is in meters (m).

We want to determine the time it takes the projectile, from the start of motion, to hit the ground, the horizontal distance traveled, and the shape of the trajectory. In addition, we want to plot the speed of the projectile versus the angular direction of this vector. We need, of course, the theory (or mathematical expressions) that describes the solution to the zero-resistance projectile problem in order to develop an algorithm to obtain solutions to it.

The projectile problem (2)

Step 2. The mathematical formulas that describe the solution to the projectile problem are provided in this step. Given the launch angle and launch speed, the horizontal distance traveled from $x=y=0$, which is the coordinate location of the launcher, is:

$$x_{\max} = 2\frac{V_o^2}{g}\sin\theta_o\cos\theta_o.$$

The time from $t=0$ at launch for the projectile to reach $x_{\max}$ (i.e., its range) is:

$$t_{x_{\max}} = 2\frac{V_o}{g}\sin\theta_o.$$

The object reaches its maximum altitude,

$$y_{\max} = \frac{V_o^2}{2g}\sin^2\theta_o$$

at time,

$$t_{y_{\max}} = \frac{V_o}{g}\sin\theta_o.$$

The horizontal distance traveled when the object reaches the maximum altitude is $x_{y_{\max}} = x_{\max}/2.0$.

The trajectory (or flight path) is described by the following pair of coordinates at a given instant of time between $t=0$ and $t_{x_{\max}}$:

$$\begin{aligned} x &= V_o t\cos\theta_o, \\ y &= V_o t\sin\theta_o - \frac{g}{2}t^2. \end{aligned}$$

Step 3. The required inputs are g, V_0, θ_0, and a finite number of time steps between $t=0$ and the time the object returns to the ground. The outputs are the range and time of flight, the maximum altitude, and the time it is reached, and the shape of the trajectory in graphical form.

Steps 4 and 5. The algorithm and structure plan developed to solve this problem are given next as a MATLAB program, because it is relatively straightforward and the translation to MATLAB is well commented with details of the approach applied to its solution (i.e., the steps of the structure plan are enumerated).

The projectile problem (3)

The evaluated and tested code is as follows:

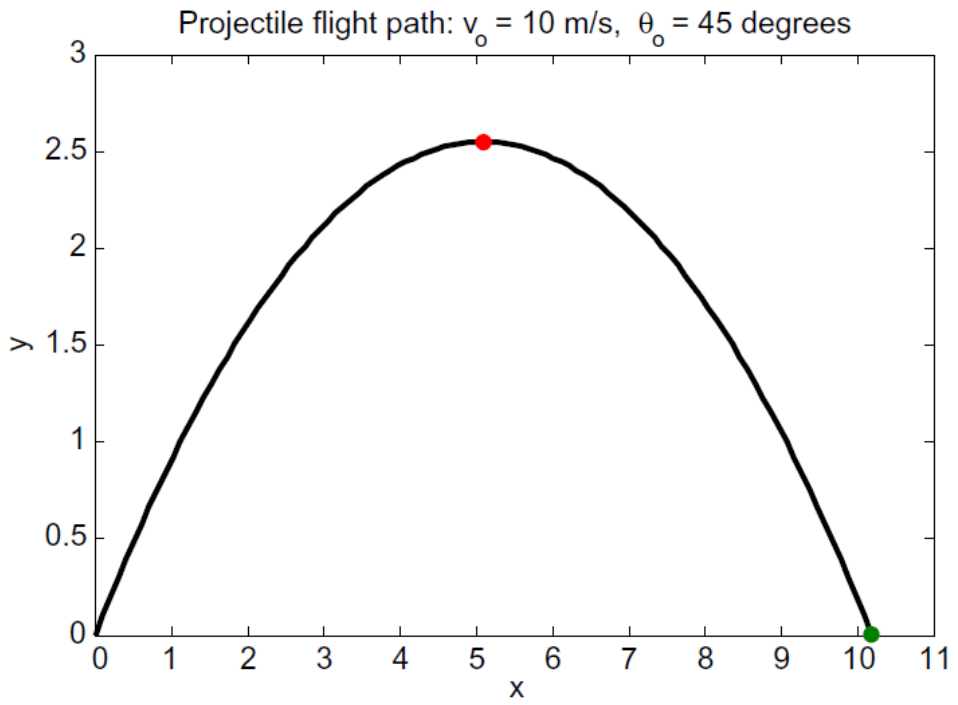
```
%
% 1. Define the input variables.
%
g = 9.81; % Gravity in m/s/s.
vo = input('What is the launch speed in m/s?')
tho = input('What is the launch angle in degrees?')
tho = pi*tho/180; % Conversion of degrees to radians.
%
% 2. Calculate the range and duration of the flight.
%
txmax = (2*vo/g) * sin(tho);
xmax = txmax * vo * cos(tho);
%
% 3. Calculate the sequence of time steps to compute
% trajectory.
%
dt = txmax/100;
t = 0:dt:txmax;
%
% 4. Compute the trajectory.
%
x = (vo * cos(tho)) .* t;
y = (vo * sin(tho)) .* t - (g/2) .* t.^2;
%
% 5. Compute the speed and angular direction of the
% projectile. Note that vx = dx/dt, vy = dy/dt.
%
vx = vo * cos(tho);
vy = vo * sin(tho) - g .* t;
v = sqrt(vx.*vx + vy.*vy);
th = (180/pi) .* atan2(vy,vx);
%
% 6. Compute the time and horizontal distance at the
% maximum altitude.
%
tymax = (vo/g) * sin(tho);
xymax = xmax/2;
ymax = (vo/2) * tymax * sin(tho);
%
% 7. Display in the Command Window and on figures the output
%
disp(['Range in meters = ',num2str(xmax),',' ...
    ' Duration in seconds = ', num2str(txmax)])
```

The projectile problem (4)

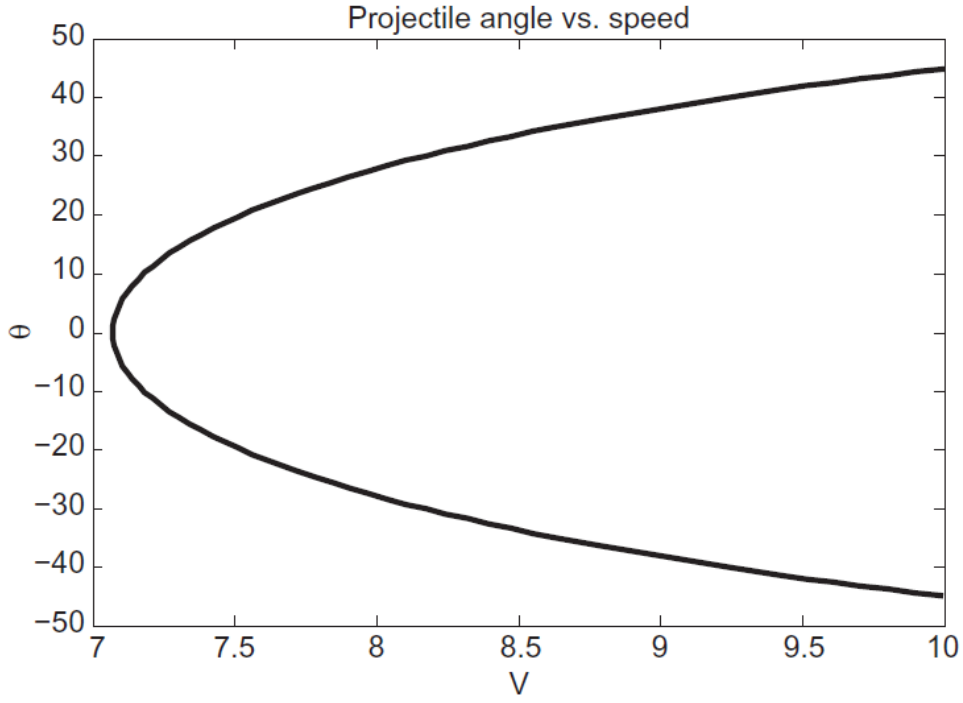
```
disp(' ')
disp(['Maximum altitude in meters = ',num2str(ymax), ...
    ',  Arrival at this altitude in seconds = ', num2str(tymax)])
plot(x,y,'k',xmax,y(size(t)),'o',xmax/2,ymax,'o')
title(['Projectile flight path: v_o = ', num2str(vo),' m/s' ...
    ',  \theta_o = ', num2str(180*tho/pi),' degrees'])
xlabel('x'), ylabel('y')      % Plot of Figure 1.
figure % Creates a new figure.
plot(v,th,'r')
title('Projectile speed vs. angle')
xlabel('V'), ylabel('\theta') % Plot of Figure 2.
%
% 8. Stop.
```

The projectile problem (5)

Steps 6 and 7. The program was evaluated by executing a number of values of the launch angle and launch speed within the required specifications. The angle of 45° was checked to determine that the maximum range occurred at this angle for all specified launch speeds. This is well known for the zero air resistance case in a constant g force field. Executing this code for $V_o = 10$ m/s and $\theta_o = 45°$, the trajectory and the plot of orientation versus speed in were produced.



Projectile trajectory.

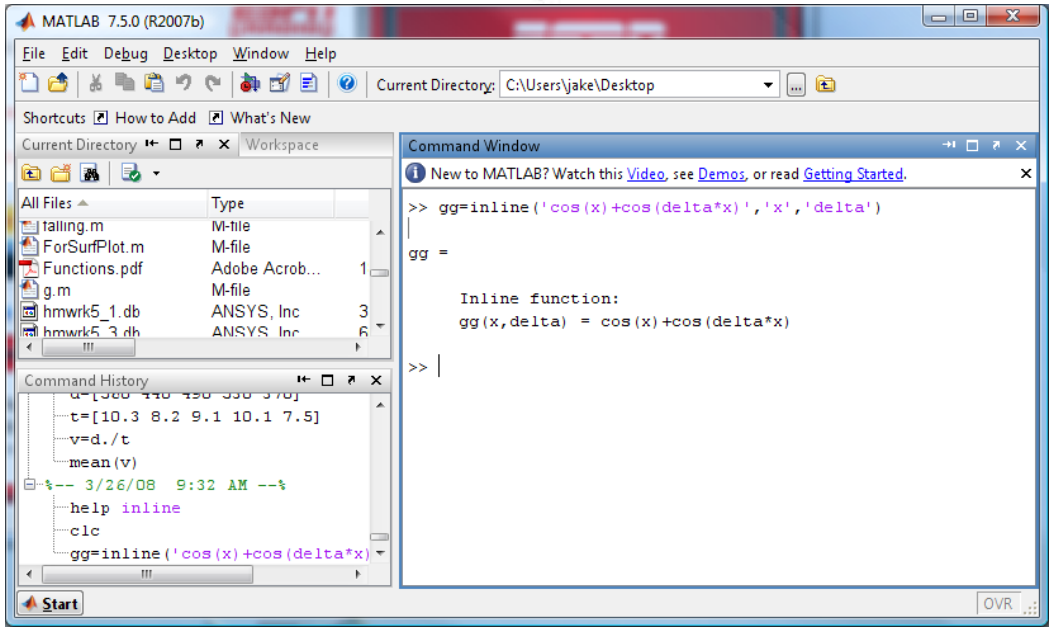


Projectile angle versus speed.

Programming MATLAB Functions

MATLAB enables you to create your own function M-files. A function M-file is similar to a script file in that it also has a `.m` extension. However, a function M-file differs from a script file in that a function M-file communicates with the MATLAB workspace only through specially designated *input* and *output arguments*. Functions are indispensable tools when it comes to breaking a problem down into manageable logical pieces.

Short mathematical functions may be written as one-line *inline objects*.



Inline objects: harmonic oscillators

If two coupled harmonic oscillators, e.g., two masses connected with a spring on a very smooth table, are considered as a single system, the output of the system as a function of time t could be given by something like:

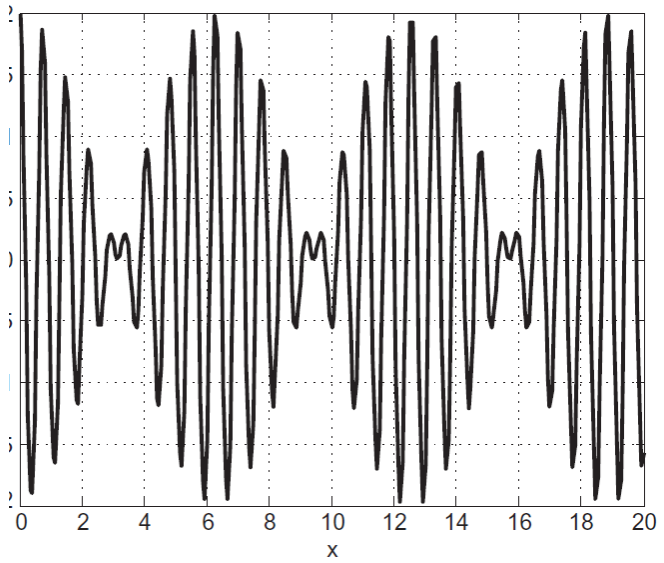
$$h(t) = \cos(8t) + \cos(9t).$$

You can represent $h(t)$ at the command line by creating an *inline object* as follows:

```
h = inline ( 'cos (8*t) + cos (9*t)' );
```

Now write some MATLAB statements in the Command Window which use your function h to draw the graph

```
x = 0 : 20/300 : 20;
plot (x, h (x)), grid
```



cos(8t) + cos(9t).

Inline objects:

```
f1 = inline( 'x.^2 + y.^2', 'x', 'y' );
f1(1, 2)
ans =
     5
```

```
    x = [1 2 3; 1 2 3]; y = [4 5 6; 4 5 6];
    f1(x,y)
    ans =
 17    29    45
 17    29    45
```

The answer is an element-by-element operation that leads to the value of the function for each combination of like-located array elements.

MATLAB function: y=f(x)

function [output arguments]
=function_name(input arguments)

```
function y = f(x)

y =  x.^3 - 0.95*x;
```

```
>> f(2)
ans =
    6.1000
```

MATLAB function: y=f(x)

```
function x = quadratic(a,b,c)
% Equation:
%    a*x^2 + b*x + c = 0
% Input: a,b,c
% Output: x = [x1 x2], the two solutions of
% this eequation.
if a==0 & b==0 & c==0
    disp(' ')
    disp('Solution indeterminate')
elseif a==0 & b==0
    disp(' ')
    disp('There is no solution')
elseif a==0
    disp(' ')
    disp('Only one root: equation is linear')
    disp('x')
    x1 = -c/b;
    x2 = NaN;
elseif b^2 < 4*a*c
    disp(' ')
    disp(' x1, x2 are complex roots')
    disp(' x1 x2')
    x1 = (-b + sqrt(b^2 - 4*a*c))/(2*a);
    x2 = (-b - sqrt(b^2 - 4*a*c))/(2*a);
elseif b^2 == 4*a*c
    x1 = -b/(2*a);
    x2 = x1;
    disp('equal roots')
    disp('x1 x2')
else
    x1 = (-b + sqrt(b^2 - 4*a*c))/(2*a);
    x2 = (-b - sqrt(b^2 - 4*a*c))/(2*a);
    disp('x1 x2')
end
if a==0 & b==0 & c==0
elseif a==0 & b==0
else
    disp([x1 x2]);
end
end
```

MATLAB function: y=f(x)

This function is saved with the file name quadratic.m.

```
>> a = 4; b = 2; c = -2;
>> quadratic(a,b,c)
    x1      x2
    0.5000   -1.0000
>> who
Your variables are:
a  b  c
```

An Alternative Form (Anonymous Functions)

```
x=0:0.01:100;
delta=1.05
gg=@(x, delta) cos(x)+cos(delta*x)
y=gg(x, delta);
plot(x,y)
```

Practice

- Consider the function

$$f(x) = \exp(-a * x) * \sin(x)$$

- Plot using an inline function
- Use 0<x<10 and a=0.25
- Note: syntax can be taken from:

```
gg=inline('cos(x)+cos(delta*x)','x','delta')
gg=@(x, delta) cos(x)+cos(delta*x)
```

Subfunctions

- Subfunctions allow us to put two functions in one file
- The second function will not be available to other functions

```
function example
clear all
r=sumofcubes(20);
fprintf('The sum of the first 20 cubes is %i\n',r)
%
function r=sumofcubes(N)
ans=0;
for i=1:N
   ans=ans+i^3;
end
r=ans;
```

An Example – with Numerics

- Suppose we're looking for a $100k, 30-year mortgage. What interest rate do I need to keep payments below $700 per month?

$$100000 - 700\left[\frac{(1+i)^{360}-1}{i(1+i)^{360}}\right] = 0$$

- Solve for i

Create the function

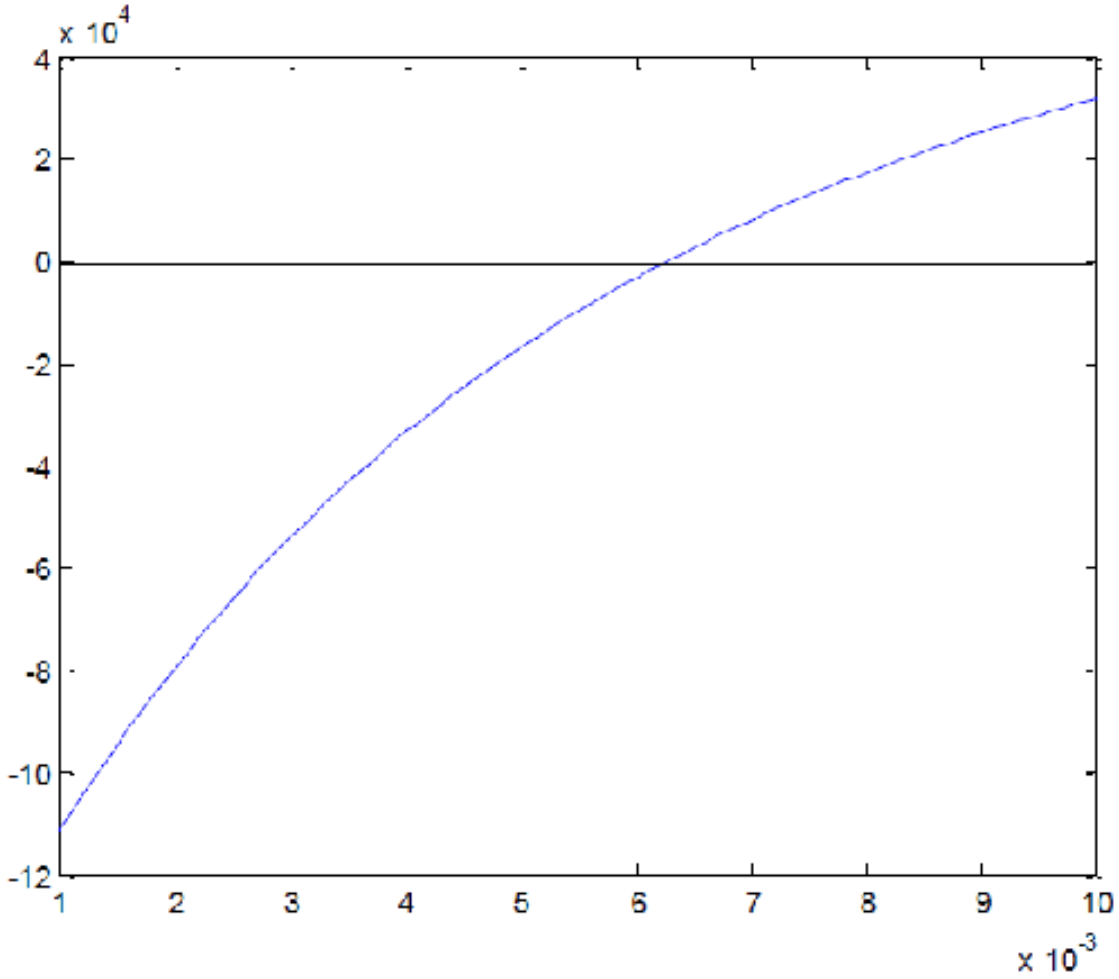
```
function s=f(i)
p=100000;
n=360;
a=700;
s=p-a*((1+i).^n-1)./(i.*(1+i).^n);
```

Plot the Function

- First save file as f.m
- Now enter the following

```
i=0.001:0.0001:0.01;
y=f(i);
plot(i,y)
```

The Plot



Result

- Zero=crossing is around i=0.006
- Annual interest rate is 12*i, or about 7%
- Try more accurate solution

```
12*fzero('f',0.006)
```

- This gives about 7.5%

Approach

- Create user-defined function
- Plot the function
- Find point where function is zero

Practice Problem 1

Consider the following structure plan, where M and N represent MATLAB variables:

1. Set $M = 44$ and $N = 28$
2. While M not equal to N repeat: While $M > N$ repeat: Replace value of M by $M - N$ While $N > M$ repeat: Replace value of N by $N - M$
3. Display M
4. Stop.

(a) Work through the structure plan, sketching the contents of M and N during execution. Give the output.

(b) Repeat (a) for $M = 14$ and $N = 24$.

(c) What general arithmetic procedure does the algorithm carry out (try more values of M and N if necessary)?

Practice Problem 2

Write a script that inputs any two numbers (which may be equal) and displays the larger one with a suitable message or, if they are equal, displays a message to that effect.

Practice Problem 3

Write a script for the general solution to the quadratic equation $ax^2 + bx + c = 0$. Use the structure plan developed before. Your script should be able to handle all possible values of the data a, b, and c. Try it out on the following values:

(a) 1, 1, 1 (complex roots)
(b) 2, 4, 2 (equal roots of −1.0)
(c) 2, 2, −12 (roots of 2.0 and −3.0).

Practice Problem 4

Develop a structure plan for the solution to two simultaneous linear equations (i.e., the equations of two straight lines). Your algorithm must be able to handle all possible situations; that is, lines intersecting, parallel, or coincident. Write a program to implement your algorithm, and test it on some equations for which you know the solutions, such as:

$$x + y = 3,$$
$$2x - y = 3,$$
$$(x = 2, y = 1).$$

Hint: Begin by deriving an algebraic formula for the solution to the system:

$$\begin{aligned} ax + by &= c, \\ dx + ey &= f. \end{aligned}$$

The program should input the coefficients a, b, c, d, e, and f.

Practice Problem 5

We wish to examine the motion of a damped harmonic oscillator. The small amplitude oscillation of a unit mass attached to a spring is given by the formula $y = e^{-(R/2)t} \sin(\omega_1 t)$, where $\omega_1^2 = \omega_o^2 - R^2/4$ is the square of the natural frequency of the oscillation with damping (i.e., with resistance to motion); $\omega_o^2 = k$ is the square of the natural frequency of undamped oscillation; k is the spring constant; and R is the damping coefficient. Consider $k = 1$ and vary R from 0 to 2 in increments of 0.5. Plot y versus t for t from 0 to 10 in increments of 0.1.

Hint: Develop a solution procedure by working backwards through the problem statement. Starting at the end of the problem statement, the solution procedure requires the programmer to assign the input variables

first followed by the execution of the formula for the amplitude and ending with the output in graphical form.

Logic and Relational Operations

• The ability to test conditional statements and make decisions is a fundamental programming concept

– For example: Is A greater than B? Is T equal to Z OR is T less than 5?

• Often used to control program flow

– For example:

• If C equals D and D does not equal 5 do something.
• Did the user enter the right type of data?
• While T is less than 10 add this number together.

• To do this, MATLAB compares two arguments (e.g. “A” and “B”) and returns a “true” or false” value.

Operators

- Arithmetic operators

`+`, `-`, `*`, `/`, `\`, `^`, etc.

- Logical operators

`&`, `|`, `~`, `xor( )`, etc.

- Relational operators

`==`, `~=`, `<`, `>`, `<=`, `>=`, etc.

Relational Operations

- Relational operators perform element-by element comparisons between two scalars or two arrays (must be same size).

- They return a logical array of the same size

- Elements are set to logical 1 (true) where the relation is true

- Elements set to logical 0 (false) where it is not

Relational Operators

<	Less than
>	Greater than
<=	Less than or equal to ($\leq$)
>=	Greater than or equal to ($\geq$)
==	Equal to (logical equality)
~	Not
~=	Not Equal to ($\neq$)

MATLAB is a bit picky about spaces around relational operators
No spaces, a space after the operator, or a space on both sides work
A space before the operator and none after does not work

Logical Data Type

• The ***logical data type*** is a another MATLAB data type; it is either equal to 1 (true) or equal to 0 (false).

input	Output: ans =	Data type
`4 < 5`	1	Logical array
`4 <= 5`	1	Logical array
`4 > 5`	0	Logical array
`4 >= 5`	0	Logical array
`4 ~= 5`	1	Logical array

Relational Operations and Round-off Errors

• The **==** and **~=** commands can produce puzzling results due to round-off error.

• For example:

```
>> a = 0;
>> b = sin(pi);
>> a == b
   ans = 0   % false when you expect
true
```

What happened?

- MATLAB computes sin(p) to within eps, i.e, it computes a number approximately zero

- The == operator properly concludes that 0 and the computed value of sin(p) are different

- The best way to fix this is to check if they are approximately equal (within round off error “eps”)

```
>> abs (a - b) < eps % eps is "relative
accuracy" and equal to ~2*10-16 (very small)
    ans = 1           % true
```

Logical Operators

Logical, or Boolean, operators: a logical operand that produces a logical result.

- Type "help relop" in the command window for complete details

Operator	Operation	
&	**Logical AND**	**true if both are true**
|	**Logical OR**	**true if either is true**
~	**Logical NOT**	
&&	**Short-circuit AND**	**Only tests until false**
||	**Short-circuit OR**	**Only tests until true**
xor	**Exclusive OR**	**true if either, but not both, is true**

Logical Operator Examples

Inputs

```
>> a = true;
>> b = false;
>> c = true;
```

Inputs Results

```
>> a & b      0 % false
>> b & c      0
>> a & c      1 % true
>> ~(a&c)     0
>> a | b      1
>> b | c      1
>> a | c      1
>> c | a      1
>> b | b      0
>> xor(a,c)   0
>> xor(a,b)   1
```

Hierarchy Of Operations

- Logical Operators are evaluated **after** arithmetic and relational operations.

Order of Operations

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

Operator Precedence

1. Parentheses ( ) (Highest precedence)
2. Transpose ('), power (.^),complex conjugate transpose ('), matrix power (^)
3. Unary plus (+), unary minus (−), logical negation (~)
4. Multiplication (.*), right division (./), left division (.\), matrix multiplication (*), matrix right division (/), matrix left division (\)
5. Addition (+), subtraction (−)
6. Colon operator (:)
7. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=),equal to (==), not equal to (~=)
8. Element-wise AND (&)
9. Element-wise OR (|)
10. Short-circuit AND (&&)
11. Short-circuit OR (||) (Lowest precedence)

Combination of Operators

- Operators are often combined
- For example:

```
>> a = 1; b = 2; c = 3; d = 2; e = -3;
>> (a < b) & (b < c)      % 1 (true)
>> (a == b) | (b == d)    % 1 (true)
>> (a > c) | (b < d)      % 0 (false)
>> -5 <= e <= -2          % 0 (false)
***WARNING** doesn't work like you think.
Use (-5 <= e) & (e <= -2) instead since (-5 <= e)
can only be 1 or 0 which is always than -2.
```

- For complex arrangements it is often a good idea to break into a series of simpler tests (or use " ( ) ") and then combine so they are easier to debug

Logical Operator and Arrays (element by element)

When comparing two arrays, MATLAB created a logical array that is **1** where the relational test is **true** and **0** where it is **false**.

```
>> x = [1 2 3 4 5];
>> y = [2 3 3 1 5];
>> x < y % x and y must be same size
    ans = 1 1 0 0 0 %note: same size as x & y
>> x == y % is x equal to y?
    ans = 0 0 1 0 1
>> (x > y) | (x ~= y) %x > y OR x not equal to y?
    ans = 1 1 0 1 0
```

Masking

- A "mask" can be used to extract values from an array that meet specified criteria

```
>> x = [-1 3 -2 -1 4 0]
>> mask1 = x > 0 % array is true "1" where x > 0
    mask = 0   1   0   0   1   0
>> pos_x = x(mask1) % extract only values > 0
    pos_x = 3   4
>> neg_x = x(~mask1) % extract only values <= 0
                     % x(x<=0) does the same thing
    neg_x = -1   -2   -1   0
```

Masking Continued

A "mask" can also be used to change values in an array if they meet specified criteria

```
>> x = [-1 3 -2 -1 4 0]
>> mask = x > 0 % array is true "1" where x > 0
    mask = 0  1  0  0  1  0
>> x(mask) = 5 % set all values in x > 0 to 5
    x = -1  5  -2  -1  5  0
>> x(~mask) = -5 % set all values x <= 0 to -5
                 % x(x<=0)= -5 does same thing
    x = -5  5  -5  -5  5  -5
>> x = 5*(x > 0) + -5*(x <= 0) % single line
soln.
    x = -5  5  -5  -5  5  -5
```

Logical Functions

- Logical functions are MATLAB functions that implement logical operations.
- **`find( )`** returns the index number of the array that meet the given logical statement

```
>> v = [1, 3, 5, 6; 3, 4, 5, 7; 4, 2, 3, 9];
>> [row, col] = find(v == 4)
row = 3 2, col = 1 2 % v(3,1) and v(2,2) are 4
>> element = find(v == 4)
element = 3 5 % 4 is the 3rd and 5th element in v
```

Other Logical Functions

- **`ischar( )`** returns **1** if **`( )`** contains character data
- **`isinf( )`** returns **1** if **`( )`** contains infinity (**`Inf`**)
- **`isnan( )`** returns **1** if **`( )`** contains not a number (**`NaN`**)
- **`isnumeric( )`** returns **1** if **`( )`** contains numeric data
- **`isempty( )`** returns **1** if **`( )`** contains empty (**`x = [ ]`**)

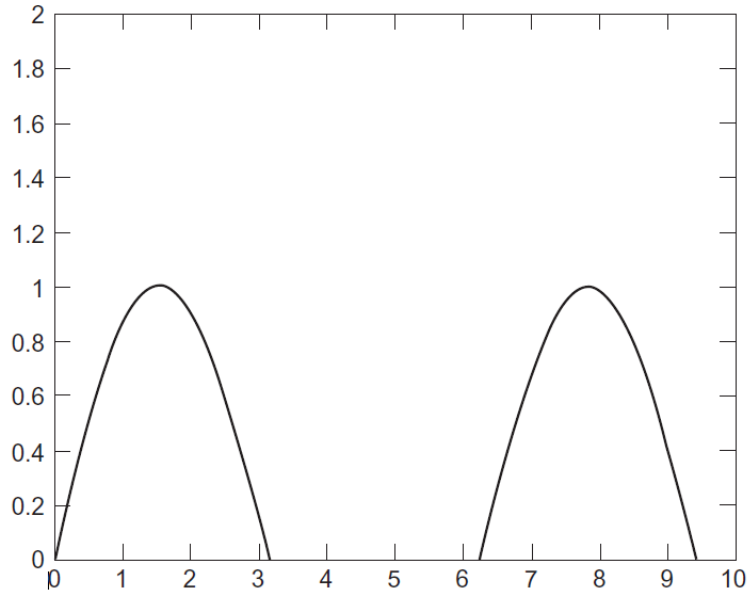
Discontinuous graphs

One very useful application of logical vectors is in plotting discontinuities. The following script plots the graph

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

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)
```



A discontinuous graph using logical vectors.

Avoiding division by zero

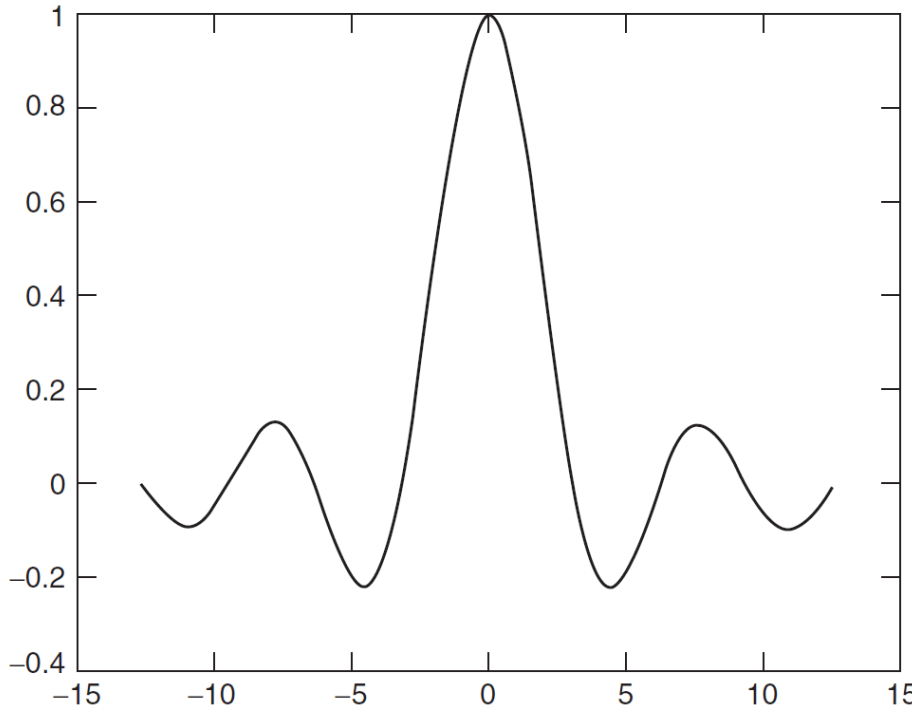
```
x = -4*pi: pi/20 : 4*pi;

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

```
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 $x=0$

```
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)
```



sin(*x*) / *x*.

Avoiding infinity

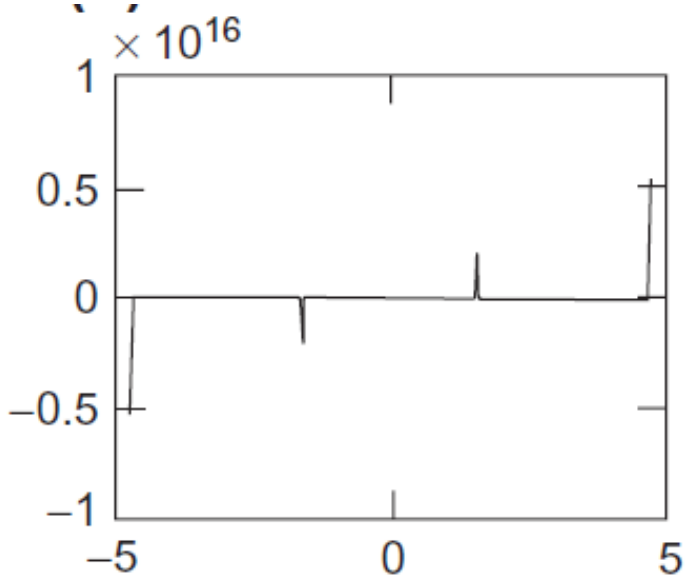
plot $\tan(x)$ over the range $-3\pi/2$ to $3\pi/2$

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

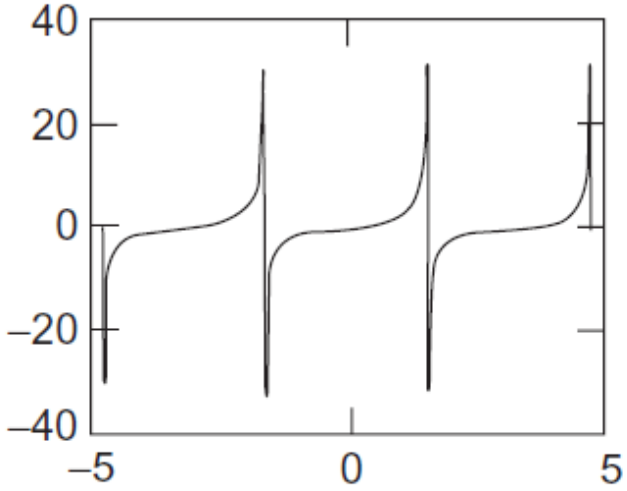
If you add the statement:

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

just before the `plot` statement you'll get a much nicer graph



Unrestricted vertical coordinate



Restricted vertical coordinate

Counting random numbers (1)

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

sum( r < 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.

Counting random numbers (2)

```
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.

Exercise

```
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)
```

Logical functions (1)

`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))
```

Logical functions (2)

`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)) = [ ]
```

Using any and all

```
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*."

```
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`.

Logical vectors instead of elseif ladders

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

```
% 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;
```

```
Taxable income    Income tax

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

Summary

- An algorithm is a systematic logical procedure for solving a problem.
- A structure plan is a representation of an algorithm in pseudo-code.
- A function M-file is a script file designed to handle a particular task that may be activated (invoked) whenever needed.

- 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.

Exercise

- Problem 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`

Exercise

- Problem 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)

Exercise

- Problem 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]
```

Exercise

- Problem 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.

Exercise

- Problem 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).

Exercise

- Problem 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 `[ ]`.