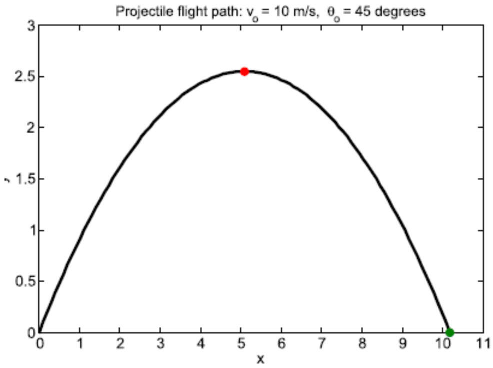
The design process[1] is outlined next. The steps may be listed as follows:

Step 1 *Problem analysis.* The purpose of this problem, and how to implement this problem.

To plot the trajectory of the Fight path. Math: 牛頓運動定律。

Step 2 *Problem statement.* Develop a detailed statement of the mathematical problem to be solved with a computer program. See text p. 87.



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 *pseudocode*

The goal is a plan that is understandable and easily translated into a computer language.

%% 8 steps of the projectile program in p. 89.

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 the result of your program.* Test all of the options and conduct a validation study of the program.

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.

```
%%=========================================================================
% Sec 3,1 The program design procedure
%%=========================================================================


%% (I)  the temperature translation problem.
% 1. input temperature in degree F


% 2. calculate C degree by (F-32)*5/9

% 3. display C degree


%% The proctile problem with zero air resistance
% The proctile problem with zero air resistance
% in a gravitational field with constant g.
% Written by Daniel T. Valentine .. September 2006
% Revised by D. T. Valentine ........... 2012/2016
% An eight-step structure plan applied in MATLAB:
%
% 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;  % time step
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);% Speed
th = (180/pi) .* atan2(vy,vx);% Angular direction
%
% 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 ouput.
%
disp(['Range in meters =',num2str(xmax),',' ...
' Duration in seconds =', num2str(txmax)]);
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 3.4.
figure % Creates a new figure.
plot(v,th,'r')
title('Projectile speed vs. angle')
xlabel('V'), ylabel('\theta') % Plot of Figure 3.5.
%
% 8. Stop.

%%=====================================================================
% Sec 3.2 Matlab function p. 92
%%=====================================================================

h = inline( 'cos(8*t) + cos(9*t)');
x = 0 : 20/300 : 20;
plot(x, h(x)), grid

h = inline( 'x.^2+y.^2','x','y');
tho=-pi:pi/10:pi;
x=cos(tho);y=sin(tho);
figure;polar(tho,h(x,y))
```

```

 % input the number of the student
clear all;close all;
N=input('   number of student:   ');
score=zeros(2,N);
% input the name and score of the student evaluate the average score
for i=1:N
    str1= input('student name:','s');
    eval(['name',int2str(i),'=str1;']);
%     if (i==1)
%         name=str1;
%     else
%         name=char(name,str1); % Create a character array.
%     end
score(1,i)=input('math score:   ');
score(2,i)=input('english score:   ');
avg(i)=(score(1,i)+score(2,i))/2;% avg(i) = sum(score(:,i))/2;
end

% output value
for i=1:N
    eval(['str1=name',int2str(i),';']);
 fprintf('the average score of %s is %3.2f \n',str1,avg(i));
end
save score_data N score


%% Exercise to write the average as a in-line function
close all; clear all;
load score_data % input N score



%%======================================================================
% p. 93 matlab function
%%======================================================================
% Use the following to write a matlab *.m script to evaluate the balance
%  function average = func1(vector)
%  average = sum(vector)/length(vector);

%% define two functions  save in stat2,m
% function [m,s] = stat2(x)
% n = length(x);
% m = avg(x,n);
% s = sqrt(sum((x-m).^2/n));
```

```
% end
%
% function m = avg(x,n)
% m = sum(x)/n;
% end

% values = [12.7, 45.4, 98.9, 26.6, 53.1];
% [ave,stdev] = stat2(values)



clc;clear;close all

money=50;%%本金
newBalance = zeros(1,12);

for k=1:12 %% 月份

   money = money *1.01;%%本金加利息
    newBalance(k)=money;%%每月存款結算
    money=money+50;%%每月定存
end

%%  write this program as the matlab function
% Temperature conversion from C to F
% or F to C as requested by the user
%
Dec = input(' Which way?: 1 => C to F? 0 => F to C: ');
Temp = input(' What is the temperature you want to convert? ');
%
% Note the logical equals sgn (==)
if Dec == 1
    TF = (9/5)*Temp + 32;
    disp(' Temperature in F: ')
    disp(TF)
else
    TC = (5/9)*(Temp-32);
    disp(' Temperature in C: ')
    disp(TC)
end



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


%% Exercise 3.2
m=44; n=28;
while m~=n
    if m > n
        m = m-n;
```

```
    else
        n = n-m;
    end
end
display(m)




t = 0:.1:2*pi;
subplot(2,2,1)
plot(t,sin(t))
subplot(2,2,2)
plot(t,cos(t))
subplot(2,2,3)
plot(t,exp(t))
subplot(2,2,4)
plot(t,1./(1+t.^2))
```

Anonymous Functions

What Are Anonymous Functions?

An anonymous function is a function that is *not* stored in a program file, but is associated with a variable whose data type is `function_handle`. Anonymous functions can accept multiple inputs and return one output. They can contain only a single executable statement.

For example, create a handle to an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;
```

Variable `sqr` is a function handle. The `@` operator creates the handle, and the parentheses `()` immediately after the `@` operator include the function input arguments. This anonymous function accepts a single input `x`, and implicitly returns a single output, an array the same size as `x` that contains the squared values.

Find the square of a particular value (`5`) by passing the value to the function handle, just as you would pass an input argument to a standard function.

```
a = sqr(5)
```

```
a =
   25
```

Many MATLAB® functions accept function handles as inputs so that you can evaluate functions over a range of values. You can create handles either for anonymous functions or for functions in program files. The benefit of using anonymous functions is that you do not have to edit and maintain a file for a function that requires only a brief definition.

For example, find the integral of the `sqr` function from `0` to `1` by passing the function handle to the `integral` function:

```
q = integral(sqr,0,1);
```

You do not need to create a variable in the workspace to store an anonymous function. Instead, you can create a temporary function handle within an expression, such as this call to the `integral` function:

```
q = integral(@(x) x.^2,0,1);
```

Variables in the Expression

Function handles can store not only an expression, but also variables that the expression requires for evaluation.

For example, create a handle to an anonymous function that requires coefficients `a`, `b`, and `c`.

```
a = 1.3;
b = .2;
c = 30;
parabola = @(x) a*x.^2 + b*x + c;
```

Because `a`, `b`, and `c` are available at the time you create `parabola`, the function handle includes those values. The values persist within the function handle even if you clear the variables:

```
clear a b c
x = 1;
y = parabola(x)
```

```
y =
   31.5000
```

To supply different values for the coefficients, you must create a new function handle:

```
a = -3.9;
b = 52;
c = 0;
```

```
parabola = @(x) a*x.^2 + b*x + c;

x = 1;
y = parabola(1)
```

```
y =
   48.1000
```

You can save function handles and their associated values in a MAT-file and load them in a subsequent MATLAB session using the `save` and `load` functions, such as

```
save myfile.mat parabola
```

Use only explicit variables when constructing anonymous functions. If an anonymous function accesses any variable or nested function that is not explicitly referenced in the argument list or body, MATLAB throws an error when you invoke the function. Implicit variables and function calls are often encountered in the functions such as `eval`, `evalin`, `assignin`, and `load`. Avoid using these functions in the body of anonymous functions.

Multiple Anonymous Functions

The expression in an anonymous function can include another anonymous function. This is useful for passing different parameters to a function that you are evaluating over a range of values. For example, you can solve the equation

$$g(c) = \int_0^1 (x^2 + cx + 1)dx$$

for varying values of `c` by combining two anonymous functions:

```
g = @(c) (integral(@(x) (x.^2 + c*x + 1),0,1));
```

Here is how to derive this statement:

1. Write the integrand as an anonymous function,

   ```
   @(x) (x.^2 + c*x + 1)
   ```

2. Evaluate the function from zero to one by passing the function handle to `integral`,

   ```
   integral(@(x) (x.^2 + c*x + 1),0,1)
   ```

3. Supply the value for `c` by constructing an anonymous function for the entire equation,

   ```
   g = @(c) (integral(@(x) (x.^2 + c*x + 1),0,1));
   ```

The final function allows you to solve the equation for any value of `c`. For example:

```
g(2)
```

```
ans =
   2.3333
```

Functions with No Inputs

If your function does not require any inputs, use empty parentheses when you define and call the anonymous function. For example:

```
t = @() datestr(now);
d = t()
```

```
d =
26-Jan-2012 15:11:47
```

Omitting the parentheses in the assignment statement creates another function handle, and does not execute the function:

```
d = t
```

```
d = 
    @() datestr(now)
```

Functions with Multiple Inputs or Outputs

Anonymous functions require that you explicitly specify the input arguments as you would for a standard function, separating multiple inputs with commas. For example, this function accepts two inputs, x and y:

Open Live Script

```
myfunction = @(x,y) (x^2 + y^2 + x*y);

x = 1;
y = 10;
z = myfunction(x,y)
```

```
z = 111
```

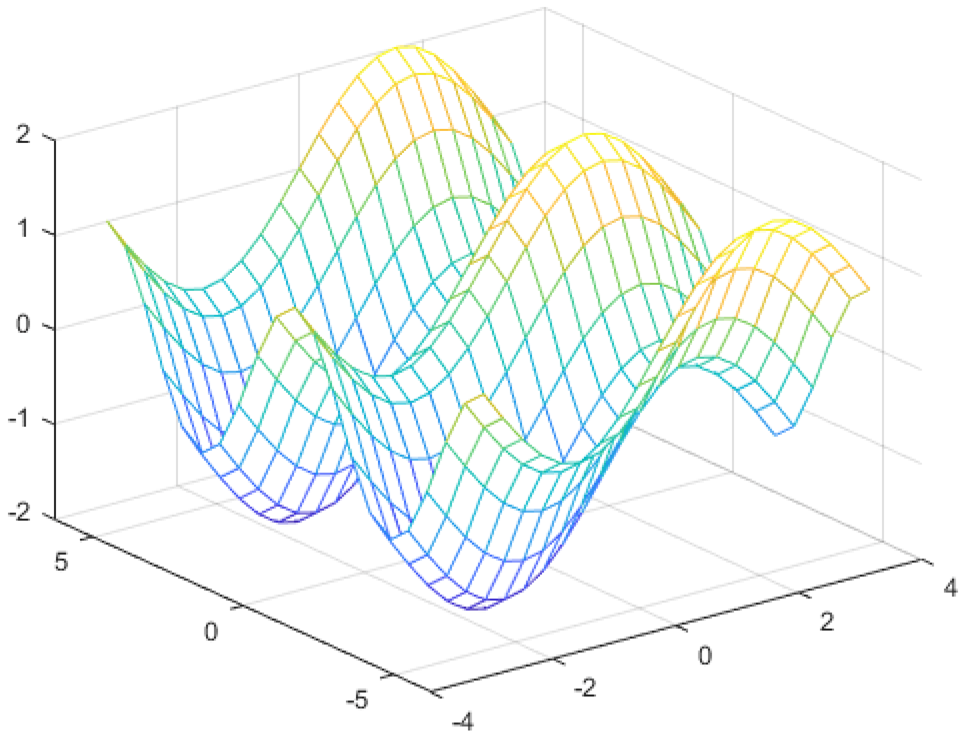
However, an anonymous function returns only one output. If the expression in the function returns multiple outputs, then you can request them when you invoke the function handle.

For example, the `ndgrid` function can return as many outputs as the number of input vectors. This anonymous function that calls `ndgrid` returns only one output (`mygrid`). Invoke `mygrid` to access the outputs returned by the `ndgrid` function.

```
c = 10;
mygrid = @(x,y) ndgrid((-x:x/c:x),(-y:y/c:y));
[x,y] = mygrid(pi,2*pi);
```

You can use the output from `mygrid` to create a mesh or surface plot:

```
z = sin(x) + cos(y);
mesh(x,y,z)
```



Arrays of Anonymous Functions

Although most MATLAB fundamental data types support multidimensional arrays, function handles must be scalars (single elements). However, you can store multiple function handles using a cell array or structure array. The most common approach is to use a cell array, such as

```
f = {@(x)x.^2;
     @(y)y+10;
     @(x,y)x.^2+y+10};
```

When you create the cell array, keep in mind that MATLAB interprets spaces as column separators. Either omit spaces from expressions, as shown in the previous code, or enclose expressions in parentheses, such as

```
f = {@(x) (x.^2);
     @(y) (y + 10);
     @(x,y) (x.^2 + y + 10)};
```

Access the contents of a cell using curly braces. For example, `f{1}` returns the first function handle. To execute the function, pass input values in parentheses after the curly braces:

```
x = 1;
y = 10;

f{1}(x)
f{2}(y)
f{3}(x,y)
```

```
ans =
     1

ans =
    20

ans =
    21
```

Related Topics

- Create Function Handle
- Types of Functions

function

Declare function name, inputs, and outputs

Syntax

```
function [y1,...,yN] = myfun(x1,...,xM)
```

Description

`function [y1,...,yN] = myfun(x1,...,xM)` declares a function named `myfun` that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores. example

You can save your function:

- In a function file which contains only function definitions. The name of the file must match the name of the first function in the file.
- In a script file which contains commands and function definitions. Functions must be at the end of the file. Script files cannot have the same name as a function in the file. Functions are supported in scripts in R2016b or later.

Files can include multiple local functions or nested functions. For readability, use the `end` keyword to indicate the end of each function in a file. The `end` keyword is required when:

- Any function in the file contains a nested function.
- The function is a local function within a function file, and any local function in the file uses the `end` keyword.
- The function is a local function within a script file.

Examples

collapse all

Function with One Output

Define a function in a file named `average.m` that accepts an input vector, calculates the average of the values, and returns a single result.

```
function ave = average(x)
    ave = sum(x(:))/numel(x);
end
```

Call the function from the command line.

```
z = 1:99;
ave = average(z)
```

```
ave =
    50
```

Function with Multiple Outputs

Define a function in a file named `stat.m` that returns the mean and standard deviation of an input vector.

```
function [m,s] = stat(x)
    n = length(x);
```

```
    m = sum(x)/n;
    s = sqrt(sum((x-m).^2/n));
end
```

Call the function from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Function in a Script File

Define a script in a file named `integrationScript.m` that computes the value of the integrand at $2\pi/3$ and computes the area under the curve from 0 to π. Include a local function that defines the integrand, $y = \sin(x)^3$.

Open Script

Note: Including functions in scripts requires MATLAB® R2016b or later.

```
% Compute the value of the integrand at 2*pi/3.
x = 2*pi/3;
y = myIntegrand(x)

% Compute the area under the curve from 0 to pi.
xmin = 0;
xmax = pi;
f = @myIntegrand;
a = integral(f,xmin,xmax)

function y = myIntegrand(x)
    y = sin(x).^3;
end
```

```
y =

    0.6495


a =

    1.3333
```

Multiple Functions in a Function File

Define two functions in a file named `stat2.m`, where the first function calls the second.

```
function [m,s] = stat2(x)
    n = length(x);
    m = avg(x,n);
    s = sqrt(sum((x-m).^2/n));
end
```

```
function m = avg(x,n)
    m = sum(x)/n;
end
```

Function avg is a *local function*. Local functions are only available to other functions within the same file.

Call function `stat2` from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat2(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Function with Argument Validation

Define a function that restricts input to a numeric vector that contains no `Inf` or `NaN` elements. This function uses the `arguments` keyword, which is valid for MATLAB® versions R2019b and later.

```
function [m,s] = stat3(x)
    arguments
        x (1,:) {mustBeNumeric, mustBeFinite}
    end
    n = length(x);
    m = avg(x,n);
    s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
    m = sum(x)/n;
end
```

In the `arguments` code block, `(1,:)` indicates that x must be a vector. The validation functions, `{mustBeNumeric, mustBeFinite}`, restrict the elements in x to numeric values that are not `Inf` or `NaN`. For more information, see Function Argument Validation.

Calling the function with a vector that contains an element that is `NaN` violates the input argument declaration. This violation results in an error being thrown by the `mustBeFinite` validation function.

```
values = [12.7, 45.4, 98.9, NaN, 53.1];
[ave,stdev] = stat3(values)
```

```
Invalid input argument at position 1. Value must be finite.
```

See Also

`arguments` | `nargin` | `nargout` | `pcode` | `return` | `varargin` | `varargout` | `what` | `which`

Topics

Create Functions in Files

Local Functions

Nested Functions

Base and Function Workspaces

Function Precedence Order

Function Argument Validation

Indexing into Function Call Results

Introduced before R2006a