The bisection method in mathematics is a root-finding method that repeatedly bisects an interval and then selects a subinterval in which a root must lie for further processing. (Why?)

- It is often used to obtain an approximate solution.
1. At each step, the algorithm divides the interval in two by computing the midpoint \( c = (a + b)/2 \) of the interval and the value of the function \( f(c) \) at that point.

2. Unless \( c \) is itself a root (which is very unlikely, but possible), there are now two possibilities:
   - either \( f(a) \) and \( f(c) \) have opposite signs and bracket a root,
   - or \( f(c) \) and \( f(b) \) have opposite signs and bracket a root.
3. The method selects the subinterval that is a bracket as a new interval to be used in the next step.

4. In this way the interval that contains a zero of $f$ is reduced in width by $\frac{1}{2}$ at each step.

5. The process is continued until the interval is sufficiently small.\(^1\)

\(^1\)How small? A certain number you give is to be the stop criteria.
Problem Formulation

Input
- Endpoints of the interval \([a, b]\) for any real numbers \(a < b\)
- Minimal interval length \(\epsilon_{int} = |b - a|\)
- Absolute error \(\epsilon_{abs} = |f(x_i) - 0|, i = a, b\)

Output
- the root \(r\)

Note that \(a, b\) will be updated in each step.
clear; clc; format long;
% main: input parameters
disp('Root Finding using Bisection Method in [a,b].')
disp('f = x.ˆ3-x-2'); % target function
a = input('a = ?
');
b = input('b = ?
');
f = @(x) x.ˆ3-x-2; % target function
eps_int = 1e-5;
eps_abs = 1e-9;
%% main: algorithm
if f(a) == 0
    r = a; % lucky a
elseif f(b) == 0
    r = b; % lucky b
else
    while ( (b-a > eps_int) || ...
        (abs(f(a)-0) > eps_abs && abs(f(b)-0) ... > eps_abs ) )
c = (a+b)/2; % middle point
if (f(c) == 0)
    r = c; % lucky c
    break;
elseif (f(a)*f(c) < 0)
    b = c;
elseif (f(b)*f(c) < 0)
    a = c;
else
    error('Failure: f(a)*f(c)>0 and ...
            f(c)*f(b)>0.');
    return; % terminate the program
end
end
r = c % approximate solution which satisfies ... the criterion
end
$c = 1.52137970691547$
Remarks

- Time complexity: $O(\log_2(n))$
  - What is $n$? $n = \frac{b - a}{\text{eps}_\text{int}}$.
  - This means that you need to make a **trade-off** between the numerical precision, that is, the number of digits, and the computation time.

- Be aware that this algorithm works well only with the premise that the behavior in $[a, b]$ is mild.

- Approximate solutions may be significantly influenced by the initial interval $[a, b]$.

- $f(c) \approx 0$ but not equal to exactly 0. (Why?)
Exercise

- Please make your bisection method algorithm into a user-defined function, say, `bisec`.
- So, you can call `bisec` to find a root for a specific function in the command window or other programs.
- Besides, you should extend the function with more input arguments for parameters used in `bisec`. 
Problem Formulation

**Input**

- Target function $f$
- Endpoints of the interval $[a, b]$ for any real numbers $a < b$
- Minimal interval length $\epsilon_{\text{int}} = |b - a|$
- Absolute error $\epsilon_{\text{abs}} = |f(x_i) - 0|$, $x_i = a, b$

**Output**

- The root $r$
function r=bisec(f,a,b,eps_int,eps_abs)

>> bisec(@(x) x.^3-x-2,0,3,1e-9,1e-9)
ans =
  1.5214
Recursive Functions

- **Recursion** is when something is defined in terms of itself.
- A **recursive function** is a function that calls itself.
- Recursion is an alternative form of program control.
- It is essentially repetition **without** a loop.
Example

The **factorial** of a non-negative integer $n$, denoted by $n!$, is the product of all positive integers less than or equal to $n$.\(^2\)

- For example,

\[
4! = 4 \times 3! \\
= 4 \times (3 \times 2!) \\
= 4 \times (3 \times (2 \times 1!)) \\
= 4 \times (3 \times (2 \times (1))) \\
= 4 \times (3 \times (2)) \\
= 4 \times (6) \\
= 24.
\]

- Can you find the pattern?

\(^2\)Note that $0! = 1.$
In general, to solve a problem using recursion, you break it into **subproblems**.

Each subproblem is the same as the original problem but smaller in size.

You can apply the same approach to each subproblem to solve it **recursively**.
Exercise

Write a program which determines return a factorial of \( n \).

```plaintext
function y = recursiveFactorial(n)
    if n == 0 || n == 1 % base condition
        y = 1;
    else
        y = n * recursiveFactorial(n-1);
    end
end
```

- Note that there must be a base condition in recursion.
- So, can you replace a recursive method by a loop?
One more intriguing question is, Can we always turn a recursive method into an iterative one?

Yes, theoretically.\(^3\)

\(^3\)The Church-Turing thesis proves it if the memory serves.
Example: Fibonacci Numbers

- The Fibonacci sequence is a sequence with the following pattern:
  
  \[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots\]

- Alternatively, the Fibonacci sequence is defined by the recurrence relation
  
  \[F_n = F_{n-1} + F_{n-2},\]

  where \(n \geq 3\) and \(F_1 = 0, F_2 = 1\).

- Given a positive integer \(n\), determine \(F_n\).

Input: \(n \geq 1\)
Output: \(F_n\)

\(^4\)See Fibonacci numbers.
It is a **binary recursion**, because there are two calls in the function.

- Time complexity: $O(2^n)$ (Why?)
- Can we implement a **linear** recursion for this problem?
- Also, you can implement it without recursive algorithms. (Try.)
Recursion bears substantial overhead.
So, the recursive algorithm may execute a bit more slowly than the iterative equivalent.\(^5\)
Besides, a deeply recursive method depletes the call stack, which is limited, and causes a exception fast.\(^6\)
In practice, the decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve.

\(^5\)In modern compiler design, recursion elimination is automatic if the recurrence occurs in the last statement in the recursive function, so called tail recursion.
\(^6\)Stack overflow.
Lecture 4

-- Graphics
Engineers use graphing techniques to make the information easier to understand.

With a graph, it is easy to identify trends, pick out highs and lows, and isolate data points that may be measurement or calculation errors.

Graphs can also be used as a quick check to determine whether a computer solution is yielding expected results.

A set of ordered pairs is used to identify points on a two-dimensional graph.

The data points are then connected by lines in common cases.
MATLAB provides the simplest function `plot(x,y,'CLM')` which is an overloaded function, where

- (x,y): data set,
- C: colors,
- L: line types,
- M: data markers.

A graphic window automatically opens when `plot` is called.
### About CLM

<table>
<thead>
<tr>
<th>Data markers†</th>
<th>Line types</th>
<th>Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot (•)</td>
<td>Solid line</td>
<td>Black k</td>
</tr>
<tr>
<td>Asterisk (*)</td>
<td>Dashed line</td>
<td>Blue b</td>
</tr>
<tr>
<td>Cross (×)</td>
<td>Dash-dotted line</td>
<td>Cyan c</td>
</tr>
<tr>
<td>Circle (o)</td>
<td>Dotted line</td>
<td>Green g</td>
</tr>
<tr>
<td>Plus sign (+)</td>
<td></td>
<td>Magenta m</td>
</tr>
<tr>
<td>Square (□)</td>
<td></td>
<td>Red r</td>
</tr>
<tr>
<td>Diamond (◇)</td>
<td></td>
<td>White w</td>
</tr>
<tr>
<td>Five-pointed star (★)</td>
<td></td>
<td>Yellow y</td>
</tr>
</tbody>
</table>

†Other data markers are available. Search for “markers” in MATLAB Help.
Example

```matlab
1 clear all;
2 clc;
3 % main
4 x=0:0.1:2*pi;
5 y=sin(x);
6 figure(1); % create Figure 1
7 plot(x,y,'r-'); % red line
8 grid on; % show grid line
```

- Note that you can simply call `figure` without specifying a number.
Example 2: Multiple Curves

```matlab
1 clear all;
2 clc;
3 % main
4 x=linspace(0,2*pi,100);
5 figure(2); % Create Figure 2
6 plot(x,sin(x),'*',x,cos(x),'o',x,sin(x)+cos(x),'+');
7 grid on;
```

- **plot**(\(x, A\)), where \(A\) is an \(m\)-by-\(n\) matrix and \(x\) is an array of size \(n\), draws a figure for \(m\) lines. (Try.)
Exercise

- Plot $\sin(x)$, $x^2$, $\log_{10}(x)$, and $2^x$ where $x \in [0, 1]$ with 100 points in a single figure.

```matlab
1 clear; clc;
2 3 x = linspace(0,1,100);
4 y = [sin(x);
5 x.^2;
6 log10(x);
7 2.^x];
8 C = 'rgby';
9 M = '.od+';
10 L = {'-', '--','.' ,' ':' , '--'};
11 for i = 1:4
12    plot(x,y(i,:),[C(i) M(i) L{i}]);
13 end
```
Annotating Plots

- **title** adds a title to a plot.
- **xlabel** adds a label to the $x$-axis.
- **ylabel** adds a label to the $y$-axis.
- **legend** allows you to add a legend to your graph.
  - The legend shows a sample of the line and lists the string you have specified.
- **text** allows you to add a text box to the graph.
  - The box is placed at the specified $x$- and $y$- coordinates and contains the string value specified.
- **gtext** is similar to **text**.
  - The box is placed at a location determined interactively by the user by clicking in the figure window.
clear all;
clc;

% main
x = linspace(0,1,100);
y = [sin(x);x.^2;log10(x);2.^x];
color = 'rgbk';
figure(1);hold on; grid on;
for i = 1 : 4
    plot(x,y(i,:),[color(i) 'o:']);
end
title('Demonstration');
xlabel('Time (Year)');
ylabel('This is Y label. (Unit)');
legend({'Curve A', 'x^2', 'Quadtratic', ...
        'Exponential'});
text(0.4,-1.5,'This is text.');
gtext('Support special symbols of LaTex.');
This is Y label. (Unit)

This is text.

Support special symbols of LaTex.

Curve A

\( x^2 \)

Quadtratic

Exponential

Zheng-Liang Lu
Exercise: Simulation on Stock Price

```matlab
1 clear; clc;
2 % main
3 y = zeros(1,1000);
4 y(1) = 100;
5 for i = 2 : length(y)
6     y(i) = y(i - 1) + 10 * randn;
7 end
8 hold on;
9 plot(1 : length(y), y, '.:');
10 title('MATLAB-244');
11 ylabel('Stock Price (TWD)');
12 xlabel('Trading Day (Day)');
13 grid on;
```

In financial theory, the stock price $S$ is modelled by Black-Scholes formula, given by $dS = \mu S dt + \sigma S dW$, where $\mu$ is the annual yield, $\sigma$ is the annual volatility, and $dW$ follows a Brownian motion.