Root finding

Eugeniy E. Mikhailov

The College of William & Mary

Lecture 05

Root finding problem

Generally we want to solve the following canonical problem

$$f(x) = 0$$

Root finding problem

Generally we want to solve the following canonical problem

$$f(x) = 0$$

Example

$$2\sin(x) - 1 = 0$$

Root finding problem

Generally we want to solve the following canonical problem

$$f(x) = 0$$

Example

$$2\sin(x) - 1 = 0$$

Often we have a problem which looks slightly different

$$h(x) = g(x)$$

Root finding problem

Generally we want to solve the following canonical problem

$$f(x) = 0$$

Example

$$2\sin(x) - 1 = 0$$

Often we have a problem which looks slightly different

$$h(x) = g(x)$$

But it is easy to transform to canonical form with

$$f(x) = h(x) - g(x) = 0$$

Root finding problem

Generally we want to solve the following canonical problem

$$f(x) = 0$$

Example

$$2\sin(x) - 1 = 0$$

Often we have a problem which looks slightly different

$$h(x) = g(x)$$

But it is easy to transform to canonical form with

$$f(x) = h(x) - g(x) = 0$$

Example

$$3x^3 + 2 = \sin x \quad \to \quad 3x^3 + 2 - \sin x = 0$$

Trial and error method

One can try to get the solution by just guessing with a hope to hit the solution. This is not highly scientific.

Trial and error method

One can try to get the solution by just guessing with a hope to hit the solution. This is not highly scientific.
However each guess can provide some clues.

Trial and error method

One can try to get the solution by just guessing with a hope to hit the solution. This is not highly scientific.
However each guess can provide some clues.

A general search algorithm is the following

- make a guess i.e. trial
- make intelligent new guess (x_{i+1}) judging from this trial (x_i)
- continue until $|f(x_{i+1})| > \varepsilon_f$ and $|x_{i+1} - x_i| > \varepsilon_x$

Trial and error method

One can try to get the solution by just guessing with a hope to hit the solution. This is not highly scientific.
However each guess can provide some clues.

A general search algorithm is the following

- make a guess i.e. trial
- make intelligent new guess (x_{i+1}) judging from this trial (x_i)
- continue until $|f(x_{i+1})| > \varepsilon_f$ and $|x_{i+1} - x_i| > \varepsilon_x$

Example

Let's play a simple game:

- some one think of any number between 1 and 100
- I will make a guess
- you provide me with either "less" or "more" depending where is my guess with respect to your number

How many guesses do I need?

Trial and error method

One can try to get the solution by just guessing with a hope to hit the solution. This is not highly scientific.
However each guess can provide some clues.

A general search algorithm is the following

- make a guess i.e. trial
- make intelligent new guess (x_{i+1}) judging from this trial (x_i)
- continue until $|f(x_{i+1})| > \varepsilon_f$ and $|x_{i+1} - x_i| > \varepsilon_x$

Example

Let's play a simple game:

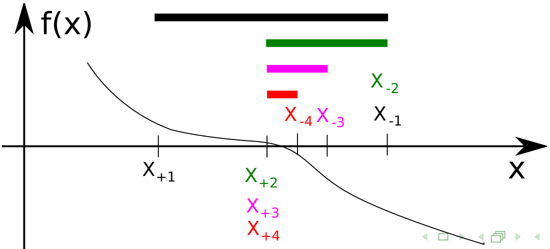
- some one think of any number between 1 and 100
- I will make a guess
- you provide me with either "less" or "more" depending where is my guess with respect to your number

How many guesses do I need? At most 7

Bisection method pseudo code

Works for any continuous function in vicinity of function root

- make initial bracket for search x_+ and x_- such that
  - $f(x_+) > 0$
  - $f(x_-) < 0$
- loop begins
- make new guess value $x_g = (x_+ + x_-)/2$
- if $|f(x_g)| \leq \varepsilon_f$ or $|x_+ - x_g| \leq \varepsilon_x$
  stop we found the solution with desired approximation
- otherwise if $f(x_g) > 0$ then $x_+ = x_g$ else $x_- = x_g$
- continue the loop



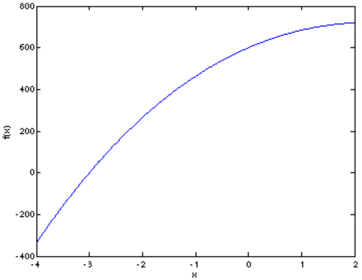
Bisection - simplified matlab implementation

```
function x_sol=bisection(f, xn, xp, eps_f, eps_x)
% solving f(x)=0 with bisection method

  xg=(xp+xn)/2; % initial guess
  fg=f(xg);     % initial function evaluation

  while ( (abs(fg) > eps_f) & (abs(xg-xp)>eps_x) )
    if (fg>0)
      xp=xg;
    else
      xn=xg;
    end
    xg=(xp+xn)/2; % update guess
    fg=f(xg);     % update function evaluation
  end
  x_sol=xg; % solution is ready
end
```

Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

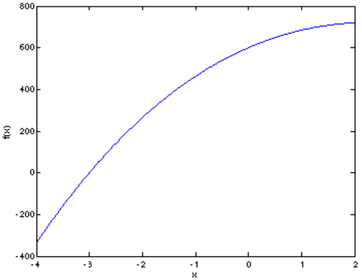
```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```



Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```



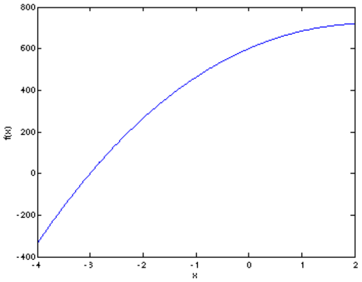
pay attention to the function handle operator @

```
eps_x=1e-8;
eps_f=1e-6;
x0=bisection(...
  @function_to_solve,...
  -4.1, 2, ...
  eps_f, eps_x)
```

Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```



pay attention to the function handle operator @

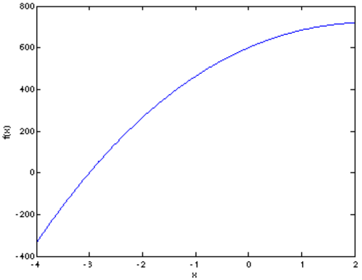
```
eps_x=1e-8;
eps_f=1e-6;
x0=bisection(...
  @function_to_solve,...
  -4.1, 2, ...
  eps_f, eps_x)
```

```
x0 = -3.0000
```

Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```




```
x0 = -3.0000
```

pay attention to the function handle operator @

```
eps_x=1e-8;
eps_f=1e-6;
x0=bisection(...
  @function_to_solve,...
  -4.1, 2, ...
  eps_f, eps_x)
```

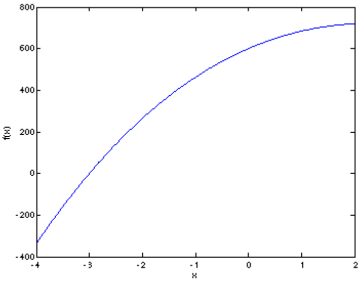
always cross check results

```
>> function_to_solve(x0)
ans = 3.0631e-07
```

Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```




```
x0 = -3.0000
```

pay attention to the function handle operator @

```
eps_x=1e-8;
eps_f=1e-6;
x0=bisection(...
  @function_to_solve,...
  -4.1, 2, ...
  eps_f, eps_x)
```

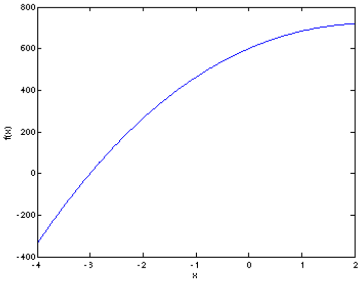
always cross check results

```
>> function_to_solve(x0)
ans = 3.0631e-07
```

Bisection - example of use

Let's define simple test function in the file 'function_to_solve.m'

```
function ret=function_to_solve(x)
  ret=(x-10)*(x-20)*(x+3);
end
```




```
x0 = -3.0000
```

pay attention to the function handle operator @

```
eps_x=1e-8;
eps_f=1e-6;
x0=bisection(...
  @function_to_solve,...
  -4.1, 2, ...
  eps_f, eps_x)
```

always cross check results

```
>> function_to_solve(x0)
ans = 3.0631e-07
```

What is missing in the bisection code?

What is missing in the bisection code?

The simplified bisection code is missing validation of input arguments.

What is missing in the bisection code?

The simplified bisection code is missing validation of input arguments. People make mistakes, typos and all sorts of misuse.

What is missing in the bisection code?

The simplified bisection code is missing validation of input arguments. People make mistakes, typos and all sorts of misuse.

> *"If something can go wrong it will"*
>
> *Muphry's Law*

What is missing in the bisection code?

The simplified bisection code is missing validation of input arguments. People make mistakes, typos and all sorts of misuse.

> *"If something can go wrong it will"*
>
> *Muphry's Law*

Never expect that user will put valid inputs.

What is missing in the bisection code?

The simplified bisection code is missing validation of input arguments. People make mistakes, typos and all sorts of misuse.

> *"If something can go wrong it will"*
>
> *Muphry's Law*

Never expect that user will put valid inputs.

So what should we check for sure

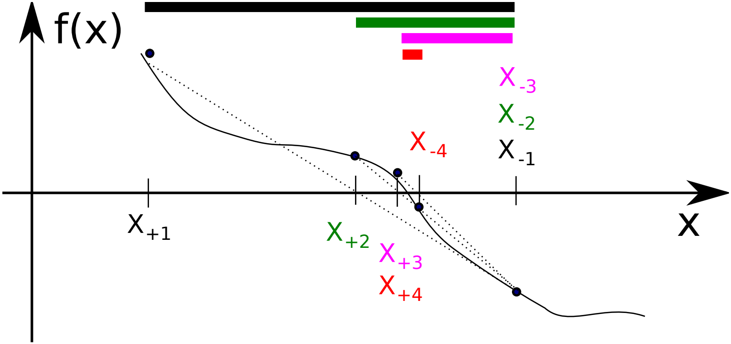
1. $f(xn) < 0$
2. $f(xp) > 0$

It would be handy to return secondary outputs

- with the value of function at the found solution point
- the number of iterations (good for performance tests)

False position (*regula falsi*) method

In this method we naively approximate our function as a line.



False position method - pseudo code

- make initial bracket for search x_+ and x_- such that
  - $f(x_+) > 0$
  - $f(x_-) < 0$
- loop begins
- draw a chord between points $(x_-, f(x_-))$ and $(x_+, f(x_+))$
- make new guess value at the point of the chord intersection with the 'x' axis

$$x_g = \frac{x_- f(x_+) - x_+ f(x_-)}{f(x_+) - f(x_-)}$$

- if $|f(x_g)| \le \varepsilon_f$ or $|x_+ - x_g| \le \varepsilon_x$
  stop we found the solution with desired approximation
- otherwise if $f(x_g) > 0$ then $x_+ = x_g$ else $x_- = x_g$
- continue the loop

Note: it looks like bisection except the way of updating x_g

Solution convergence

We say that algorithm has defined convergence if it is possible to express

$$\lim_{k \to \infty} (x_{k+1} - x_0) = c(x_k - x_0)^m$$

Where x_0 is true root of the equation, c is some constant, and m is the order of convergence.

Solution convergence

We say that algorithm has defined convergence if it is possible to express

$$\lim_{k\to\infty}(x_{k+1} - x_0) = c(x_k - x_0)^m$$

Where x_0 is true root of the equation, c is some constant, and m is the order of convergence.
The best algorithm have quadratic convergence i.e. $m = 2$

Solution convergence

We say that algorithm has defined convergence if it is possible to express

$$\lim_{k\to\infty}(x_{k+1} - x_0) = c(x_k - x_0)^m$$

Where x_0 is true root of the equation, c is some constant, and m is the order of convergence.
The best algorithm have quadratic convergence i.e. $m = 2$

- the bisection algorithm has linear rate of convergence ($m = 1$) and $c = 1/2$
- it is generally impossible to define convergence order for the false position method

Solution convergence

We say that algorithm has defined convergence if it is possible to express

$$\lim_{k\to\infty}(x_{k+1} - x_0) = c(x_k - x_0)^m$$

Where x_0 is true root of the equation, c is some constant, and m is the order of convergence.
The best algorithm have quadratic convergence i.e. $m = 2$

- the bisection algorithm has linear rate of convergence ($m = 1$) and $c = 1/2$
- it is generally impossible to define convergence order for the false position method

Generally the speed of the algorithm is related to its convergence order. How ever other factors may affect the speed.