Scientific Computing
Maastricht Science Program

Week 5

Frans Oliehoek
<frans.oliehoek@maastrichtuniversity.nl>

Conditions: If

- Sometimes you want to do things only is some cases.
- Called '**branching**' and is a very important capability.

```
% longest_side.m
% ---------
% this script determines the longest
% side of a rectangle. It expects 2
% variables 'length_x' and 'length_y'
% to be defined.

% assume y is longest side:
longest_side = length_y;

if length_x > length_y
    longest_side = length_x;
end

disp(longest_side);
```

If...else...

- The previous way of writing is not the most intuitive...
  - the default assumption is awkward
  - use “else”

```
% longest_side_else.m
% ---------
% this script determines the longest
% side of a rectangle. It expects 2
% variables 'length_x' and 'length_y'
% to be defined.

if length_x > length_y
    longest_side = length_x;
else
    longest_side = length_y;
end
disp(longest_side);
```

If...elseif...else...

- More generally, we test multiple conditions

```
if CONDITION1
    …
elseif CONDITION2
    …
elseif CONDITION3
    …
else
    …
end
```

Conditions

- So exactly what are the CONDITIONs?
  - expressions that evaluate to `true' or 'false'
  - 'false' defined as '0'
  - 'true' is any non-zero value

```
truthvalue = 0
if truthvalue
    disp('true')
else
    disp('false')
end
```

- This code can be used to test any truth value expression.

Conditions - 2

- Can make more complex expressions by 'operators'

Relational operators:

- A < B,
- A > B
- A <= B
- A >= B
- A == B
- A ~= B

Logical operators:

- ~A
- A | B,
- A & B

'short-circuit'

- A || B
- A && B

```
octave> ~1
ans = 0
octave> 1 & 0
ans = 0
octave> -1 | 0
ans =  1
octave> 0 | 0
ans = 0
```

Do it again: loops

- Another important capability: repeating instructions.
  - i.e., performing 'loops'.

- Matlab has 2 types of loops:
  - 'for' when you know how often you need to loop in advance.
  - 'while' when you don't, but only have a stopping criteria.

For loop

- For loops: used when you know how often you need to loop.

```
%count to 10
for i = [1:10]
    disp(i)
end

%count down:
start = 10
for i = [start:1]
    disp(i)
end
```

For loop

- For loops: used when you know how often you need to loop.

```
%count to 10
for i = [1:10]
    disp(i)
end

%count down:
start = 10
for i = [start:1]
    disp(i)
end
```

```
octave:12> [1:10]
ans =

    1     2     3     4     5     6     7     8     9    10
```

- (almost) everything in matlab is an array or matrix!

While loop

- Sometimes it is hard to know how often we loop
  → use 'while'

```
% strange count down
n = 14209

i = 1;
while(n > 1)
    disp(i)
    if n % 2 == 0
        n = n / 2
    else
        n = n + 1
    end
    i = i + 1;
end
```

While loop

- Sometimes it is hard to know how often we loop
  → use 'while'

```
% strange count down
n = 14209

i = 1;
while(n > 1)
    disp(i)
    if n % 2 == 0
        n = n / 2
    else
        n = n + 1
    end
    i = i + 1;
end
```

```
n =  14209
     1
n =  14210
     2
n =  7105
     3
n =  7106
     4
n =  3553
     5
n =  3554
     6
n =  1777
     7
n =  1778
     8
n =  889
     9
n =  890
    10
n =  445
    11
n =  446
    12
n =  223
    13
n =  224
    14
n =  112
    15
n =  56
    16
n =  28
    17
n =  14
    18
n =  7
    19
n =  8
    20
n =  4
    21
n =  2
    22
n =  1
```

Reusing code

- A very important concept: code reuse
- All these scripts are nice, but...
  - writing scripts for complex tasks is a lot of work.
  - often there is functionality we want to reuse!

- This is where 'functions' come in...
  - a piece of code that performs a specific task
  - has input and output.

Using Matlab/Octace Functions

- Matlab has many built in functions.
  - We already saw a few: 'mod', 'sqrt'

- Calling a function: FUNCTIONNAME(…, …, ...)
  - 'mod(3,2)'
  - 'pi()' or just 'pi'
  - [m, index] = max([4, 2, 6, 3])

Writing your own Functions

- You can write your own function very simply

```
function output = FunctionName(input1, input2)

…
…
output = …
```

- Need to name the file 'FunctionName.m'

Writing your own Functions

- You can write your own function very simply

```
function output = FunctionName(input1, input2)

…
…
output = …
```

Make sure to assign the output variable!

- Need to name the file 'FunctionName.m'

Writing your own Functions

- You can write your own function very simply

```
function longest = LongestSide(length_x, length_y)

if length_x > length_y
    longest = length_x;
else
    longest = length_y;
end
```

- Need to name the file 'LongestSide.m'
- Capitalization of 'LongestSide' is a convention
  - (no rule)

Writing your own Functions

- You can write your own function very simply

```
function longest = LongestSide(length_x, length_y)

if length_x > length_y
    longest = length_x;
else
    longest = length_y;
end
```

- Need to name the file 'LongestSide.m'
- Capitalization of 'LongestSide' is a convention
  - (no rule)

Writing your own Functions

- You can write your own function very simply

```
function longest = LongestSide(length_x, length_y)

if length_x > length_y
    longest = length_x;
else
    longest = length_y;
end
```

- Need to name the file 'LongestSide.m'
- Capitalization of 'LongestSide' is a convention
  - (no rule)

```
octave:33> LongestSide(3, 5)
ans =  5
```

Writing your own Functions

- Document your functions!

```
function longest = LongestSide(length_x, length_y)
%function longest = LongestSide(length_x, length_y)
%
% this is a special comment block: it is shown when
% calling 'help LongestSide'

if length_x > length_y
    longest = length_x;
else
    longest = length_y;
end
```

- For yourself **and** others.

Recap Programming

- Congrats: Now you know the most important constructs of programming!
- Let's summarize:
  - → Advanced calculator
  - Variables: names for intermediate parts of computation.
  - Arithmetic operators
  - Scripts
  - Branching: if … else …, conditions
  - Loops: for, while
  - Functions
  - ← programming.

A First Bit of Scientific Programming

- Now that you know the most important constructs of programming...

  ...we can implement many of the things you saw!

- This lab: a **scientific** programming problem:

  Solving non-linear equations

What are (non-)linear equations?

- linear equations?

What are (non-)linear equations?

- linear equations

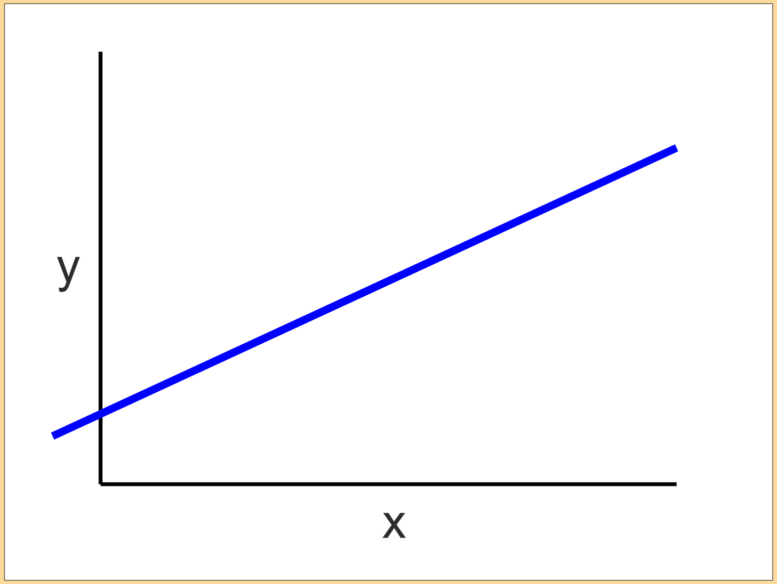
$$3x+7y=4$$

$$x-3y=4+z$$

$$(x-3y)/z=2$$

'General Form'

$$a_0+a_1x_1+a_2x_2+\ldots=0$$



Straight line
(for 2 variables)

What are (non-)linear equations?

- linear equations

$$3x + 7y = 4$$

$$x - 3y = 4 + z$$

$$(x - 3y)/z = 2$$

'General Form'

$$a_0 + a_1 x_1 + a_2 x_2 + \ldots = 0$$

- non-linear equations?

What are (non-)linear equations?

- linear equations

$$3x+7y=4$$

$$x-3y=4+z$$

$$(x-3y)/z=2$$

'General Form'

$$a_0+a_1x_1+a_2x_2+\ldots=0$$

- non-linear equations:

All equations that are **not** linear!

$$x^2=4$$

$$xy=2$$

$$y=\sqrt{x}$$

Finding the 'roots'

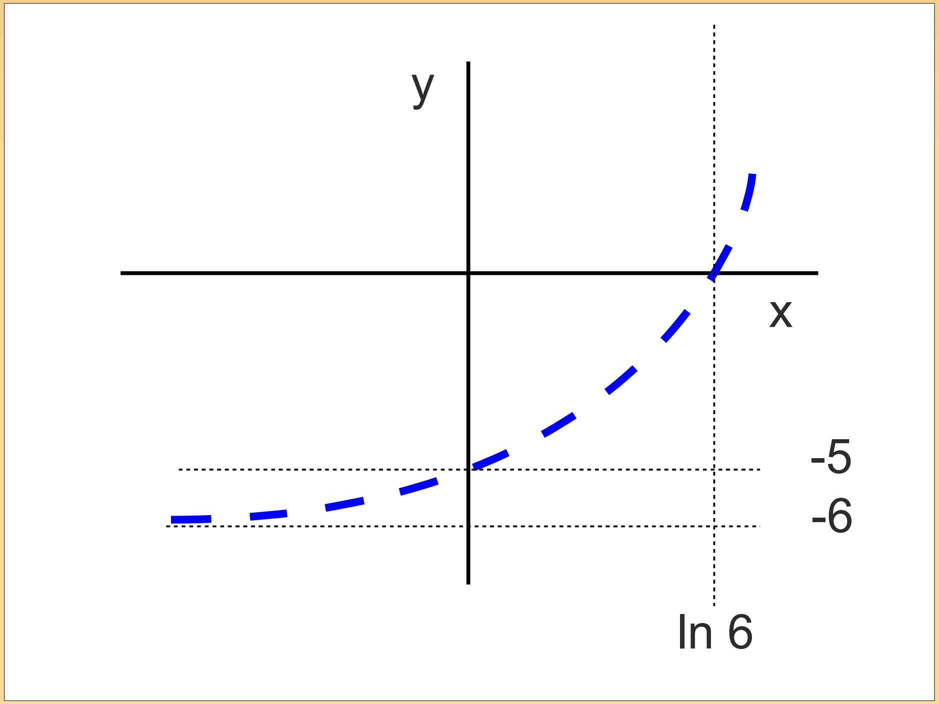
- Many problems can be reformulated as finding the 'roots' or 'zeros' of a function.
- What is ln 6 ?

Finding the 'roots'

- Many problems can be reformulated as finding the 'roots' or 'zeros' of a function.

- What is ln 6 ?

$$e^x = 6$$
$$e^x - 6 = 0$$

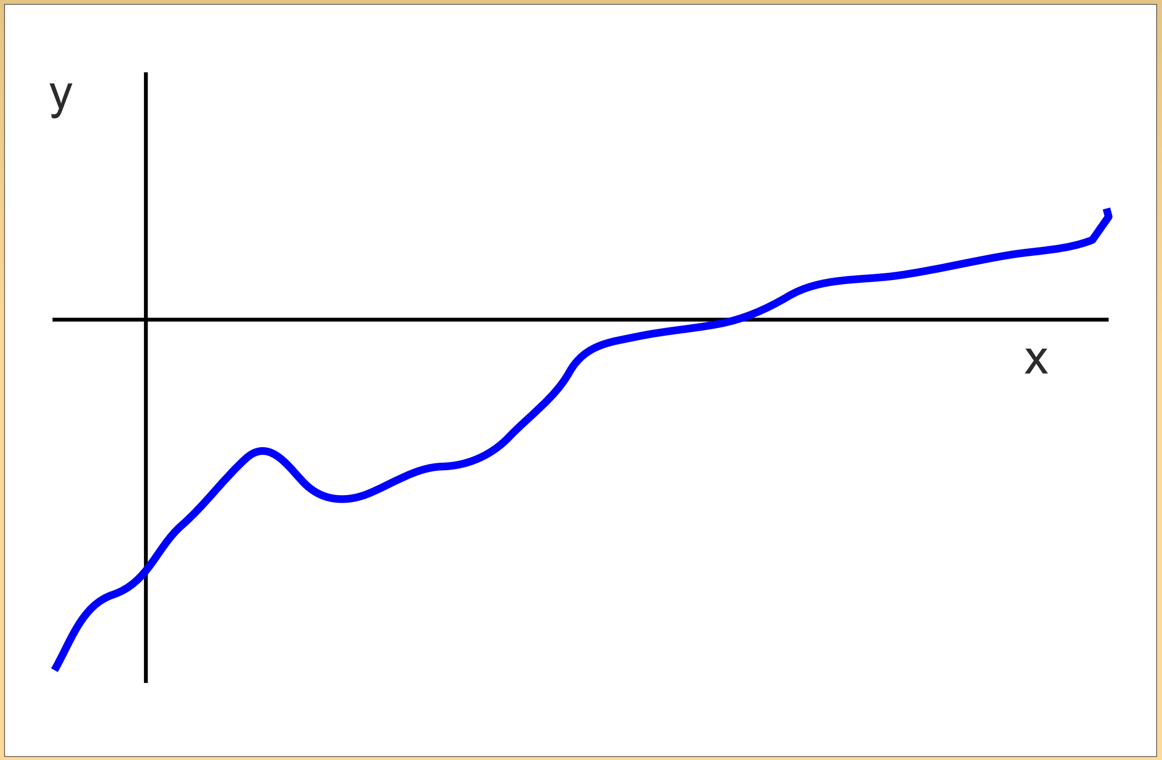


Numerical Algorithms

- To solve this problem we will now discuss our first numerical method, or numerical **algorithm**.
- Roughly:
  - algorithm = cook-book recipe
  - an algorithm can be **implemented** (converted to code in a programming language).

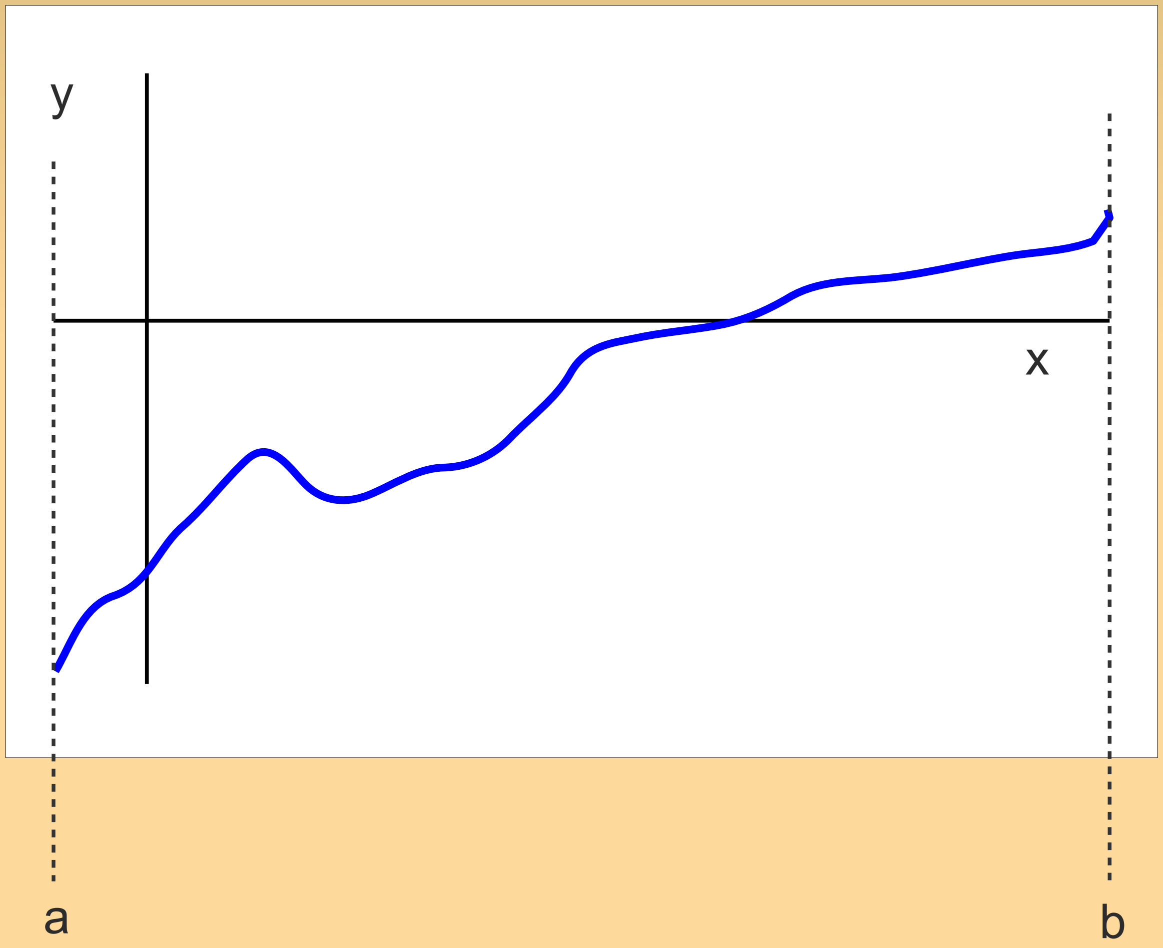
The Bisection Method

- Suppose we want to find the roots of this function?



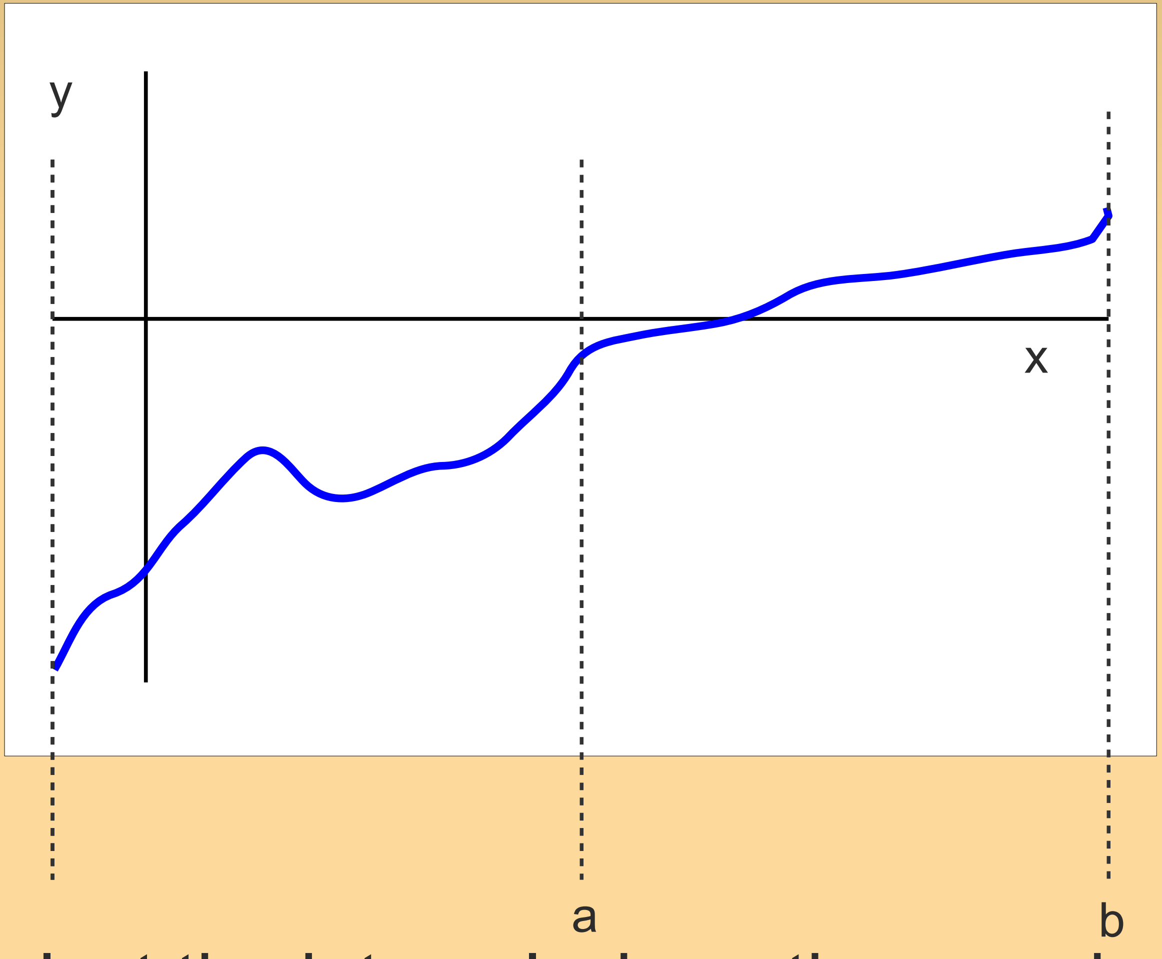
The Bisection Method

- Search the interval [a,b] for the crossing point!



The Bisection Method

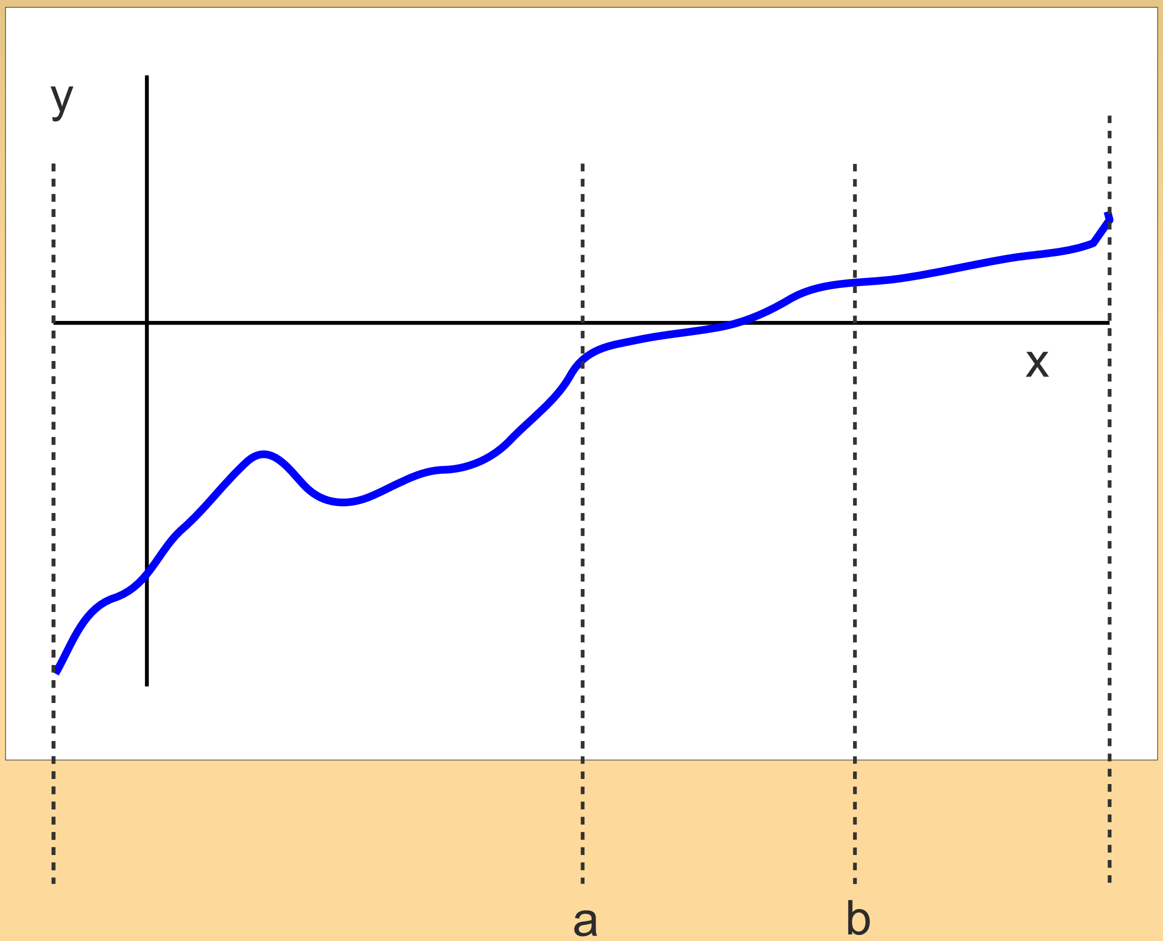
- Halve the interval



- Then select the interval where the crossing occurs

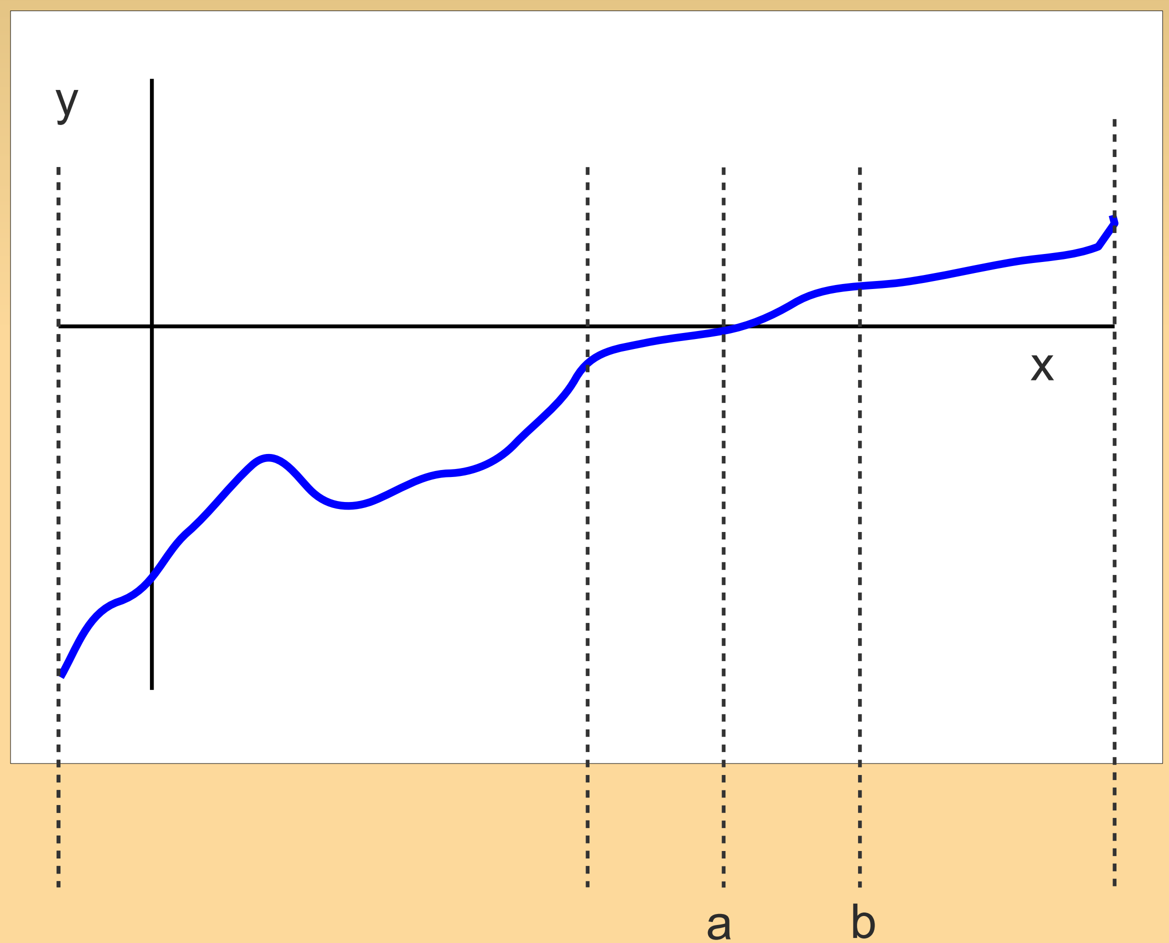
The Bisection Method

- Repeat, until the interval is small enough



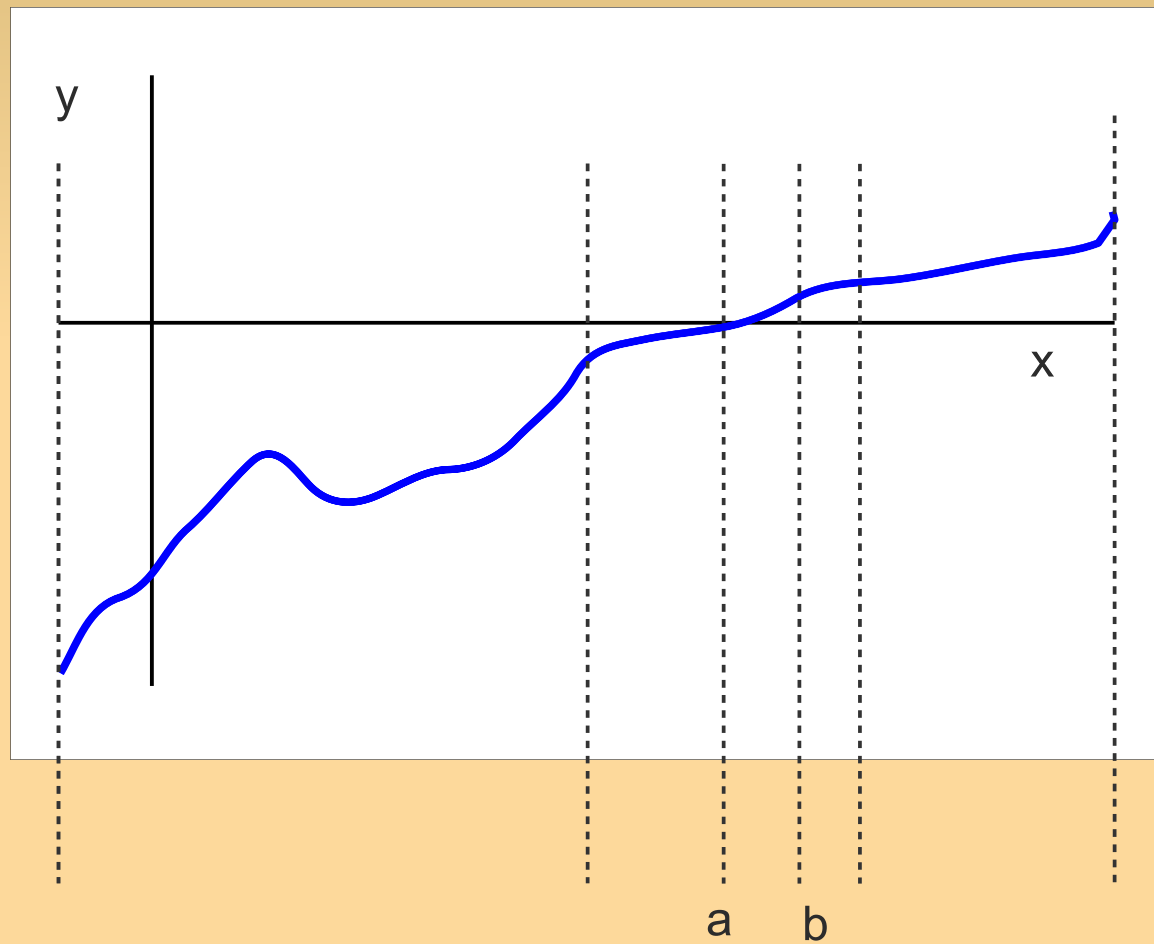
The Bisection Method

- Repeat, until the interval is small enough



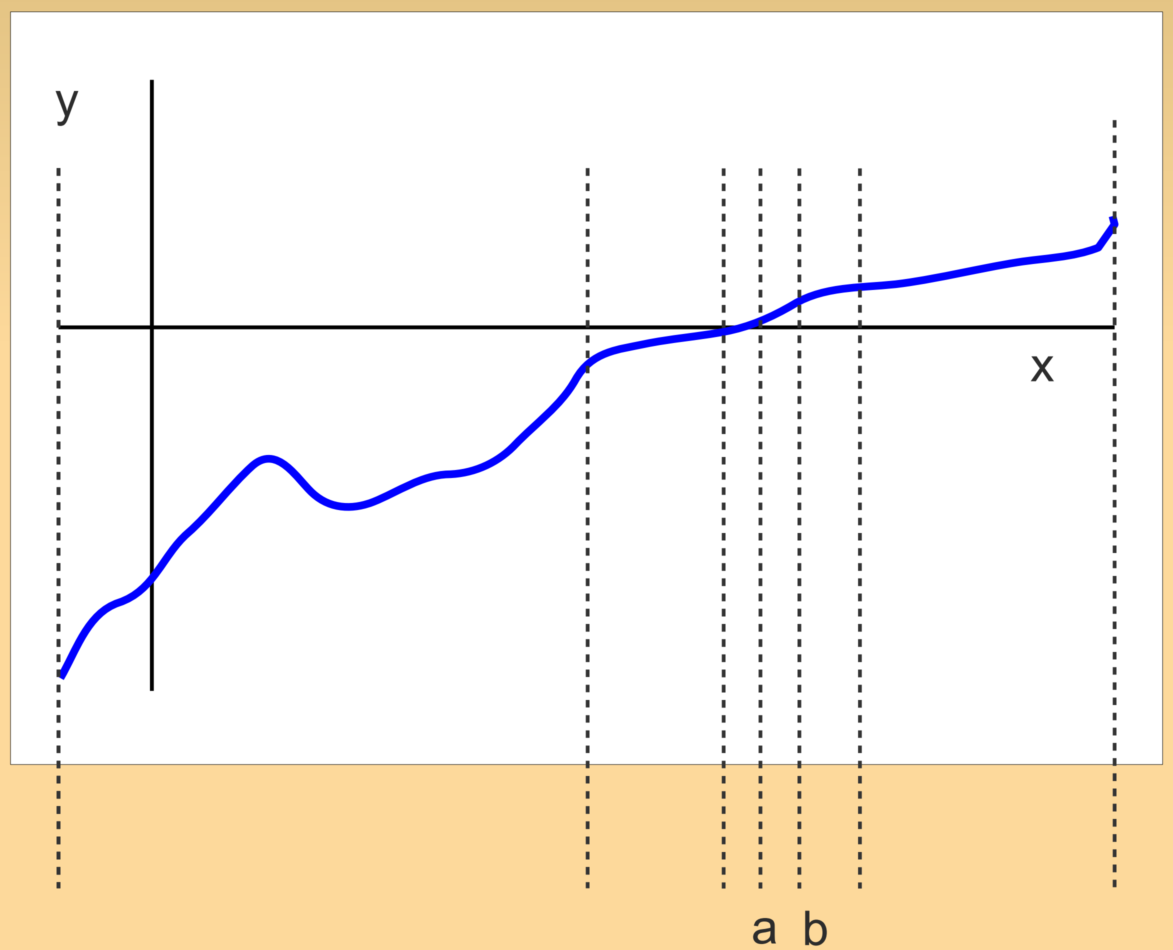
The Bisection Method

- Repeat, until the interval is small enough



The Bisection Method

- Repeat, until the interval is small enough



The Bisection Method

- Conditions to apply the Bisection Method:
  - f is continuous
  - interval [a,b]
    - f(a) is positive and f(b) is negative or vice versa

      → contains an a zero
      ('theorem of zeros of continuous functions')
    - check with $f(a)f(b) < 0$

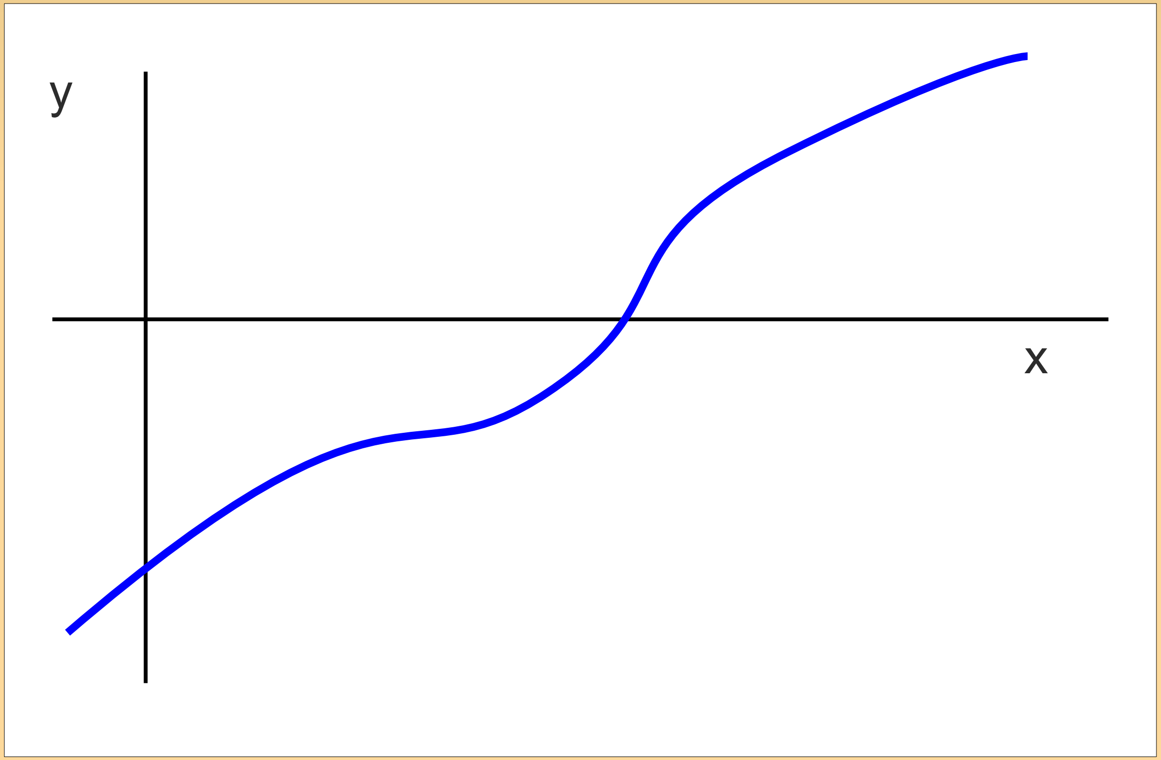
- To find a good initial interval: e.g., plot the function

The Bisection Method

- Pros
  - Simple conceptually
  - Only need information of sign of the function
    - Works in many settings
- Cons
  - Even needs many iterations on a linear function!

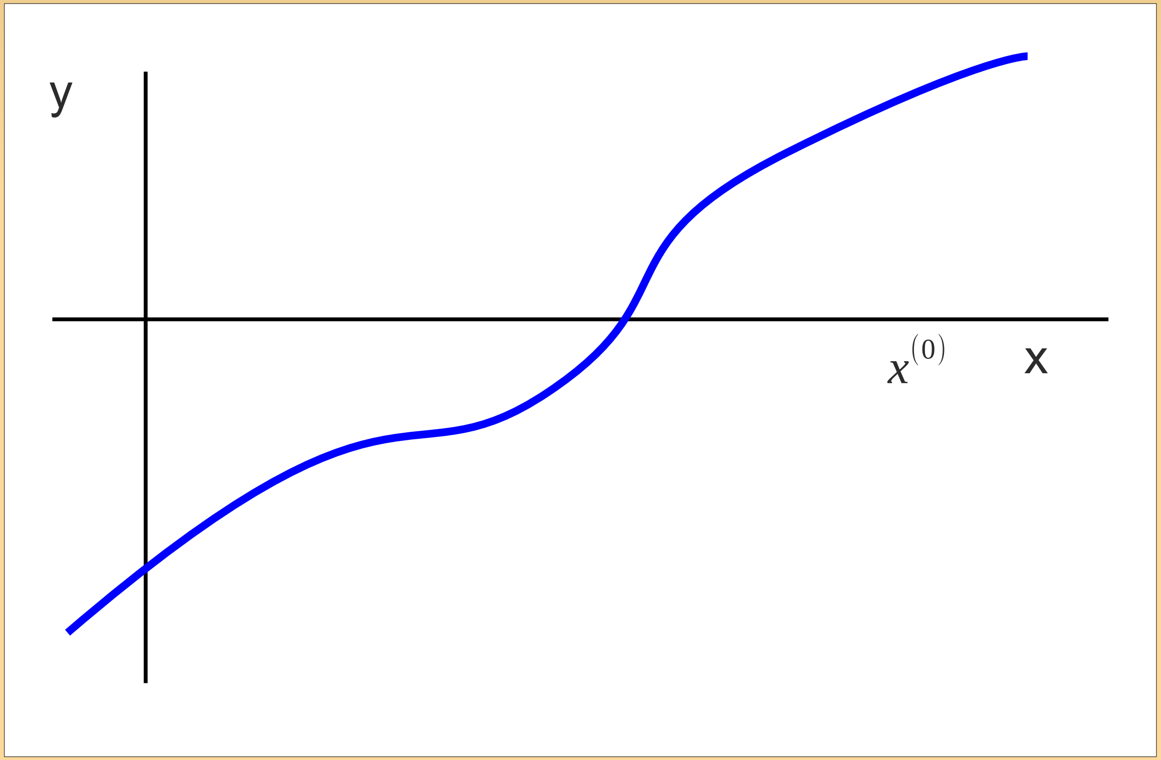
Newton's Method

- Newton's method is a different approach
  - overcomes some problems (but has its own)



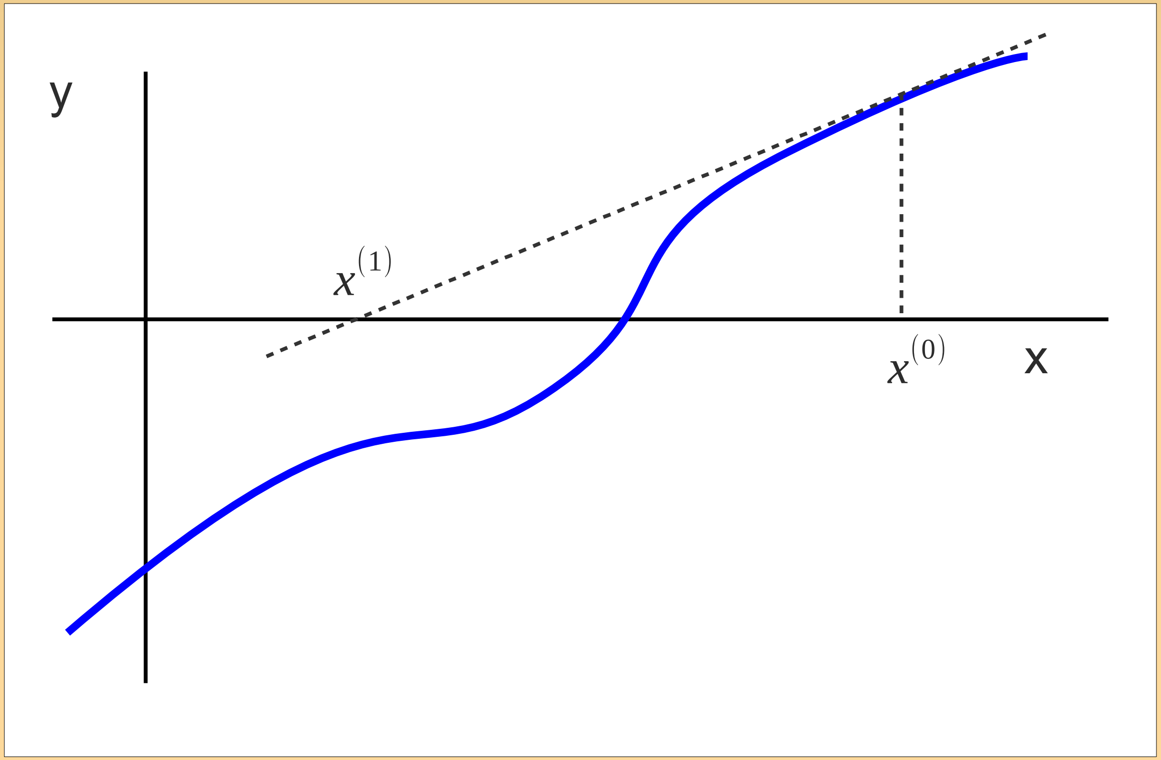
Newton's Method

- Start with an arbitrary point.



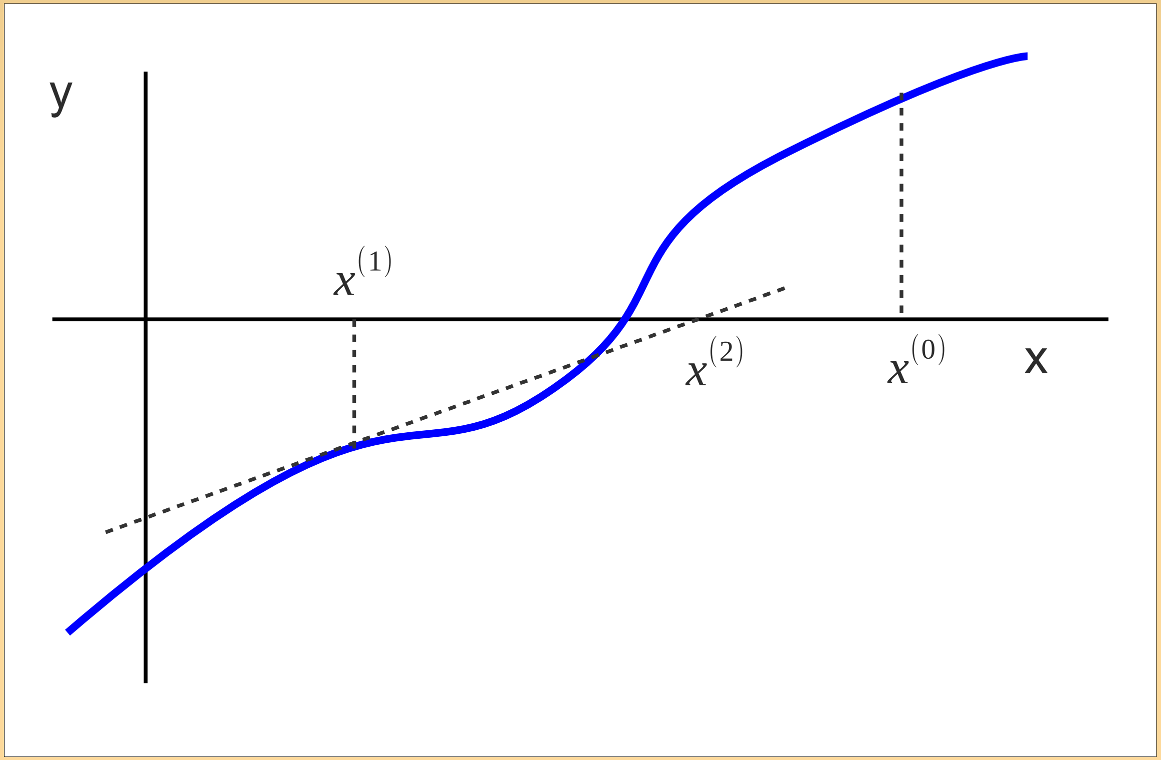
Newton's Method

- Compute next point via the derivative f'



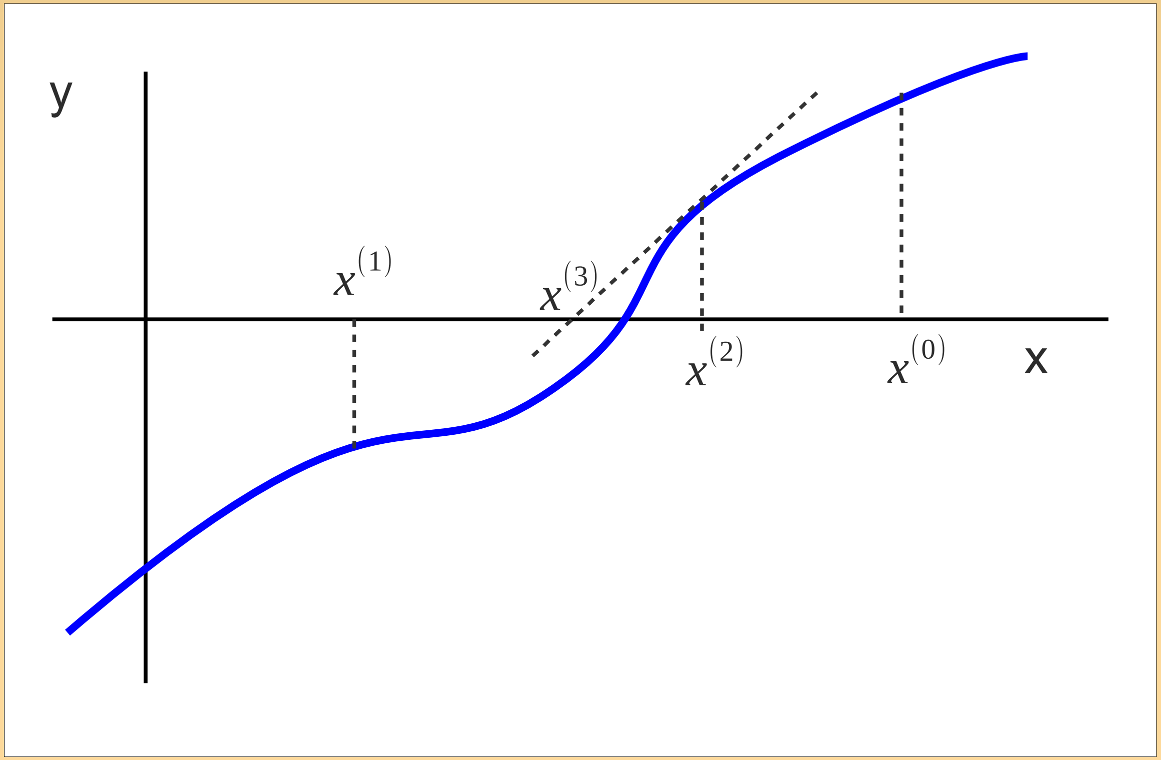
Newton's Method

- etc.



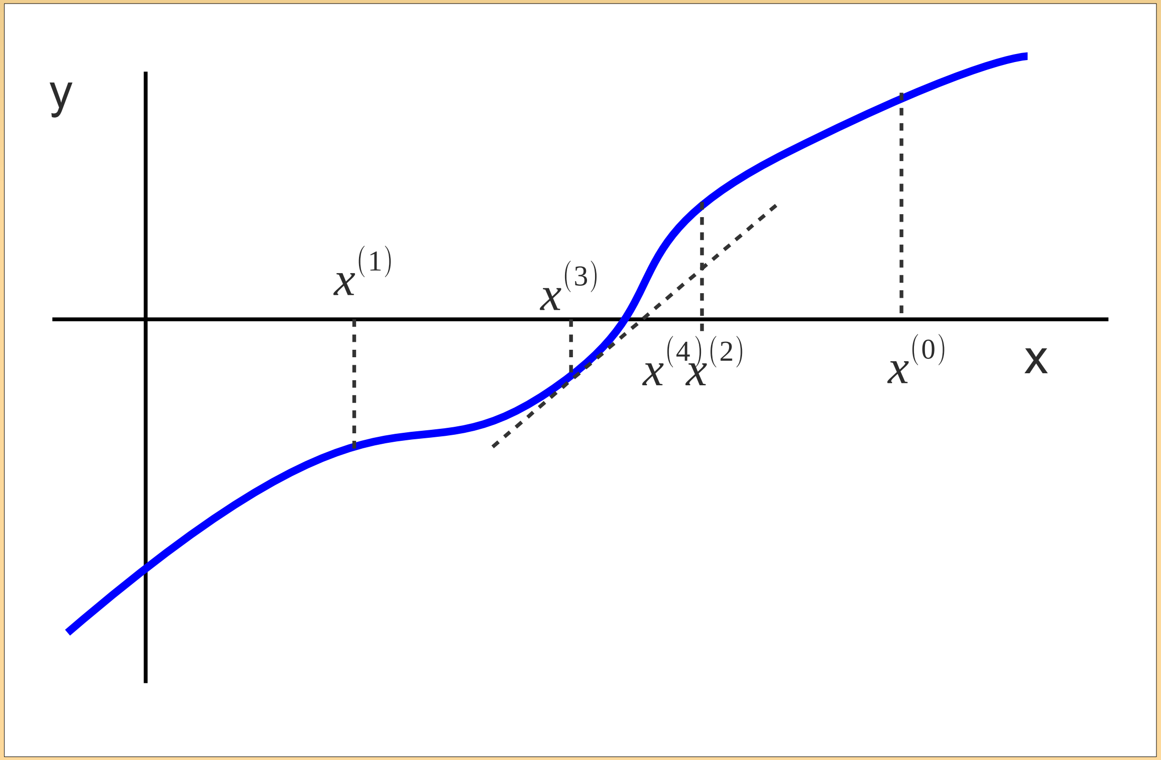
Newton's Method

- etc.



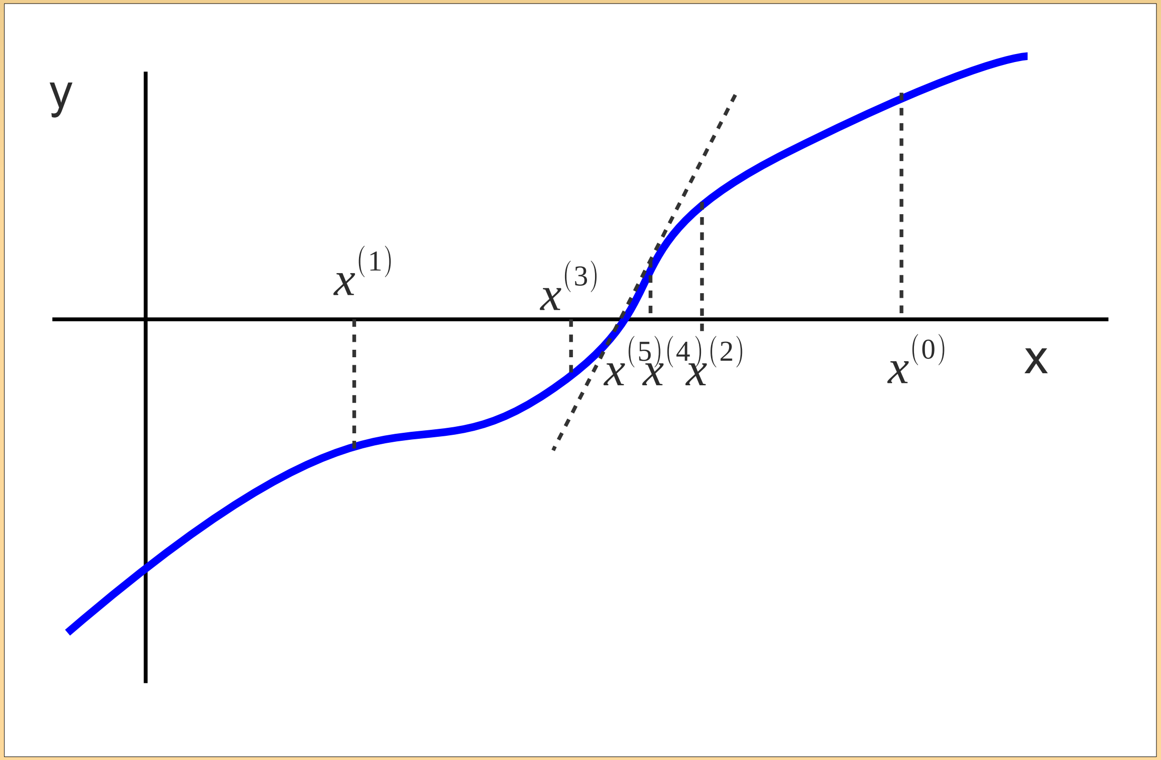
Newton's Method

- etc.



Newton's Method

- until difference with previous point small enough.



Newton's Method

- Algorithm:
  - Start with an arbitrary point $x^{(0)}$
  - Compute the next point $x^{(k+1)} = x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$
  - repeat while $|x^{(k+1)} - x^{(k)}| < \epsilon$

Newton's Method

- Pros
  - From some point on, it is **fast**!
    - converges 'quadratically'
    - error of next error is square of previous one.
- Cons
  - Need more information: function derivative
  - Needs to be initialized sufficiently close to 0
  - Problem when $f'(x^{(k)})=0$