

# sp-14. ニュートン法

(Scheme プログラミング)

URL: <https://www.kkaneko.jp/cc/scheme/index.html>

金子邦彦



# アウトライン

14-1 ニュートン法

14-2 パソコン演習

14-3 課題



# 14-1 ニュートン法

- ニュートン法による非線形方程式の求解
  - $x_0, x_1, x_2 \dots$  の収束の様子を観察し, ニュートン法の理解を深める
  - ニュートン法で, 解が求まらない場合がありえることを理解する

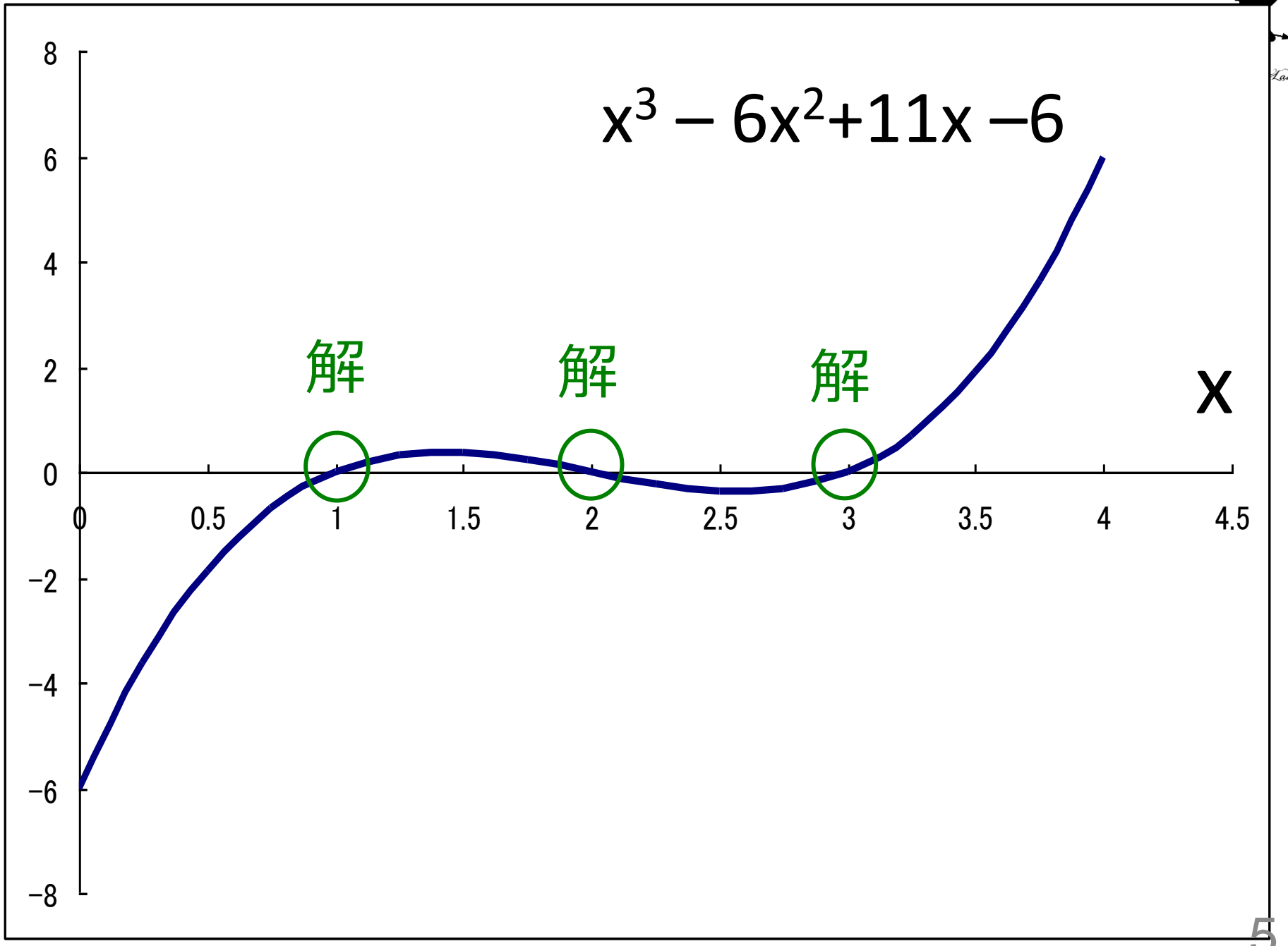
$$x^3 - 6x^2 + 11x - 6$$

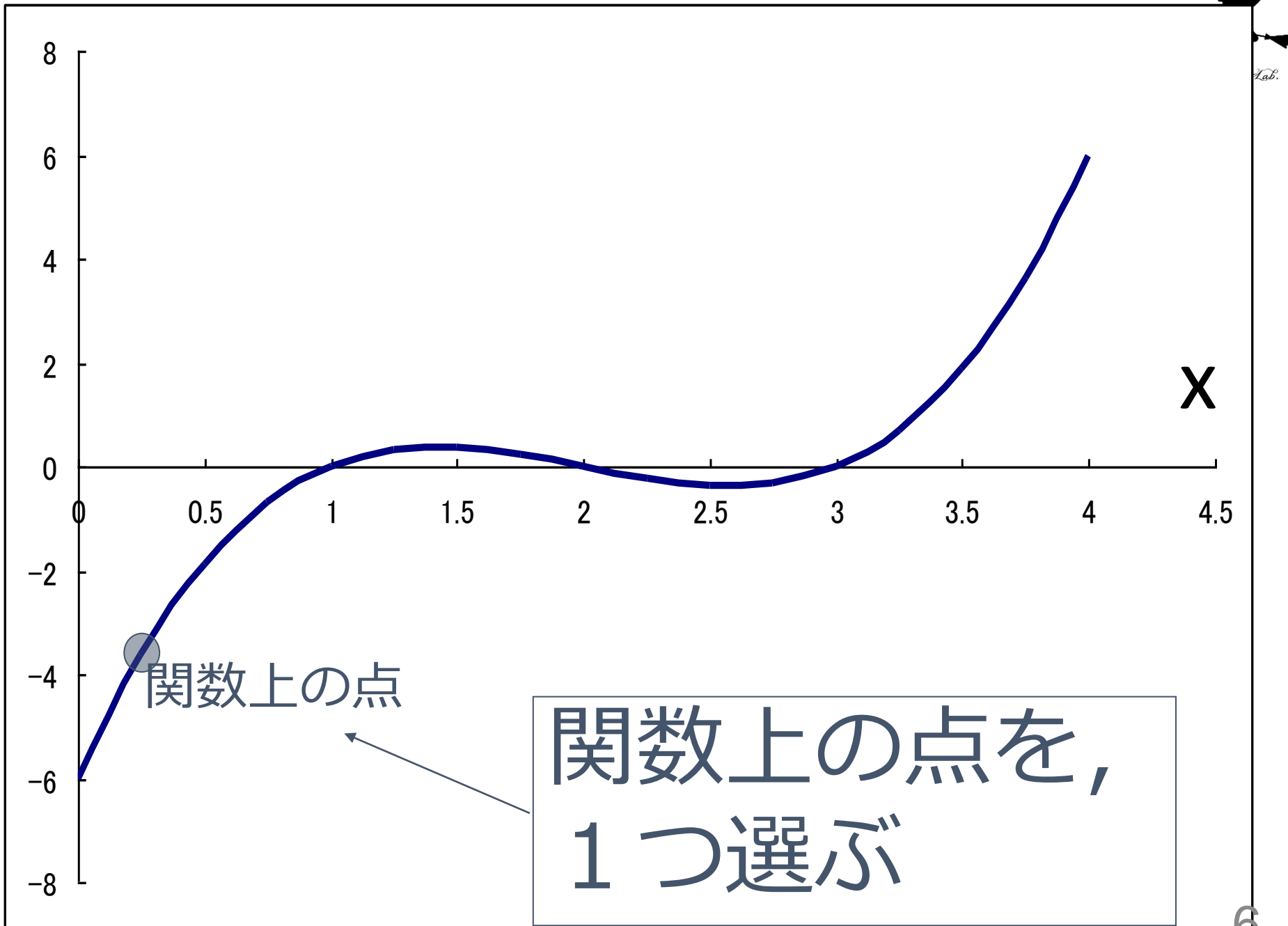
解

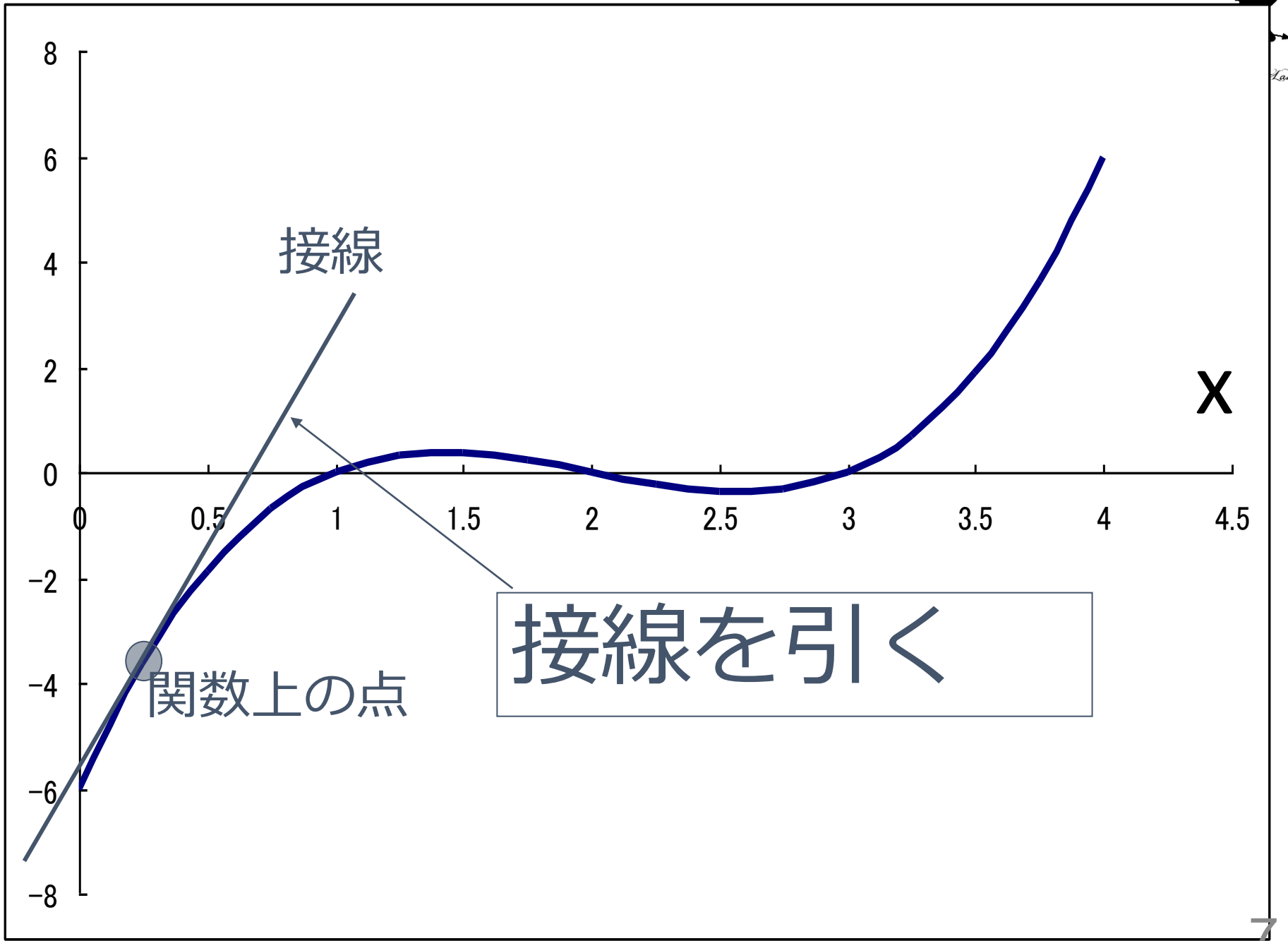
解

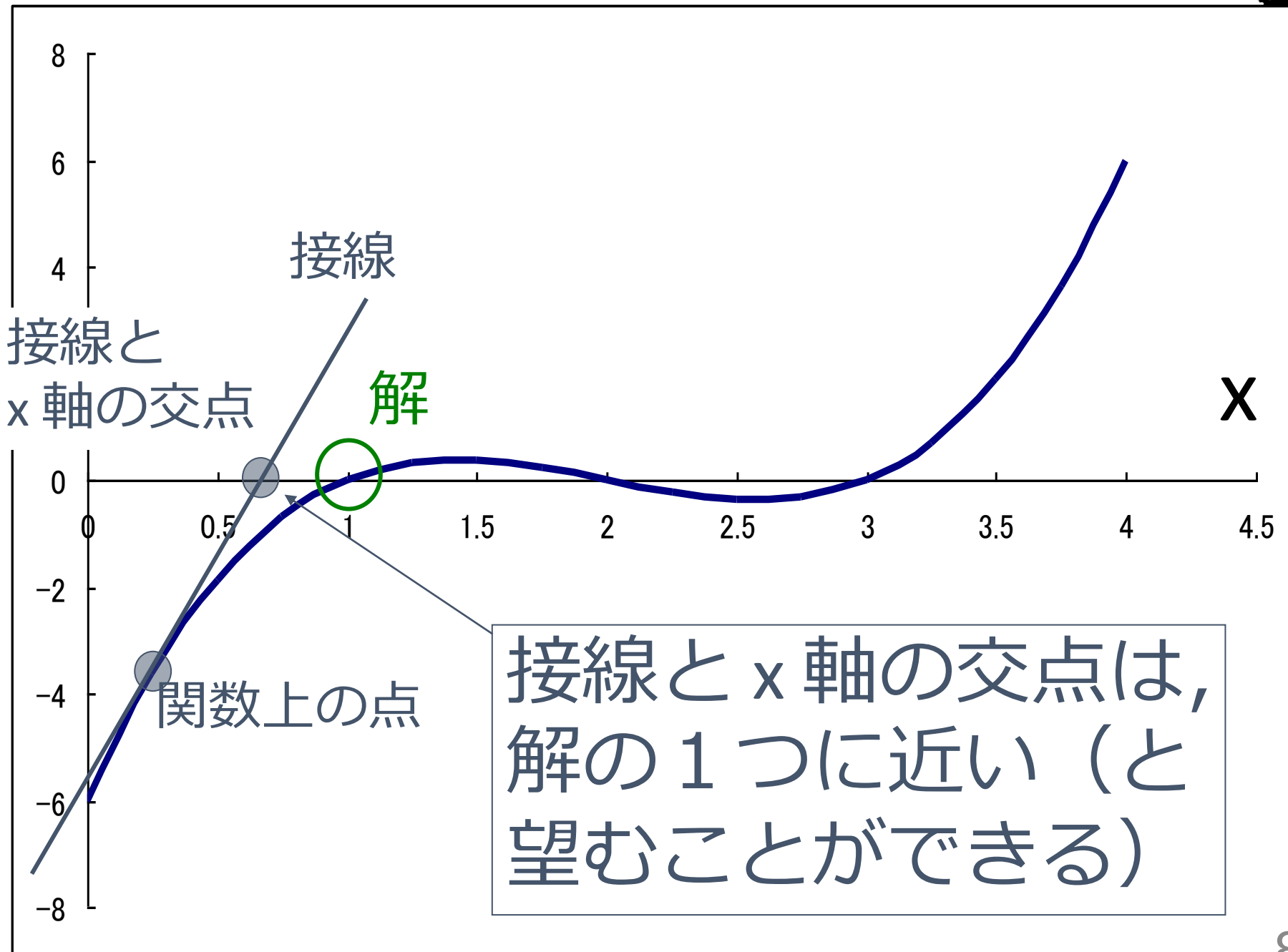
解

x

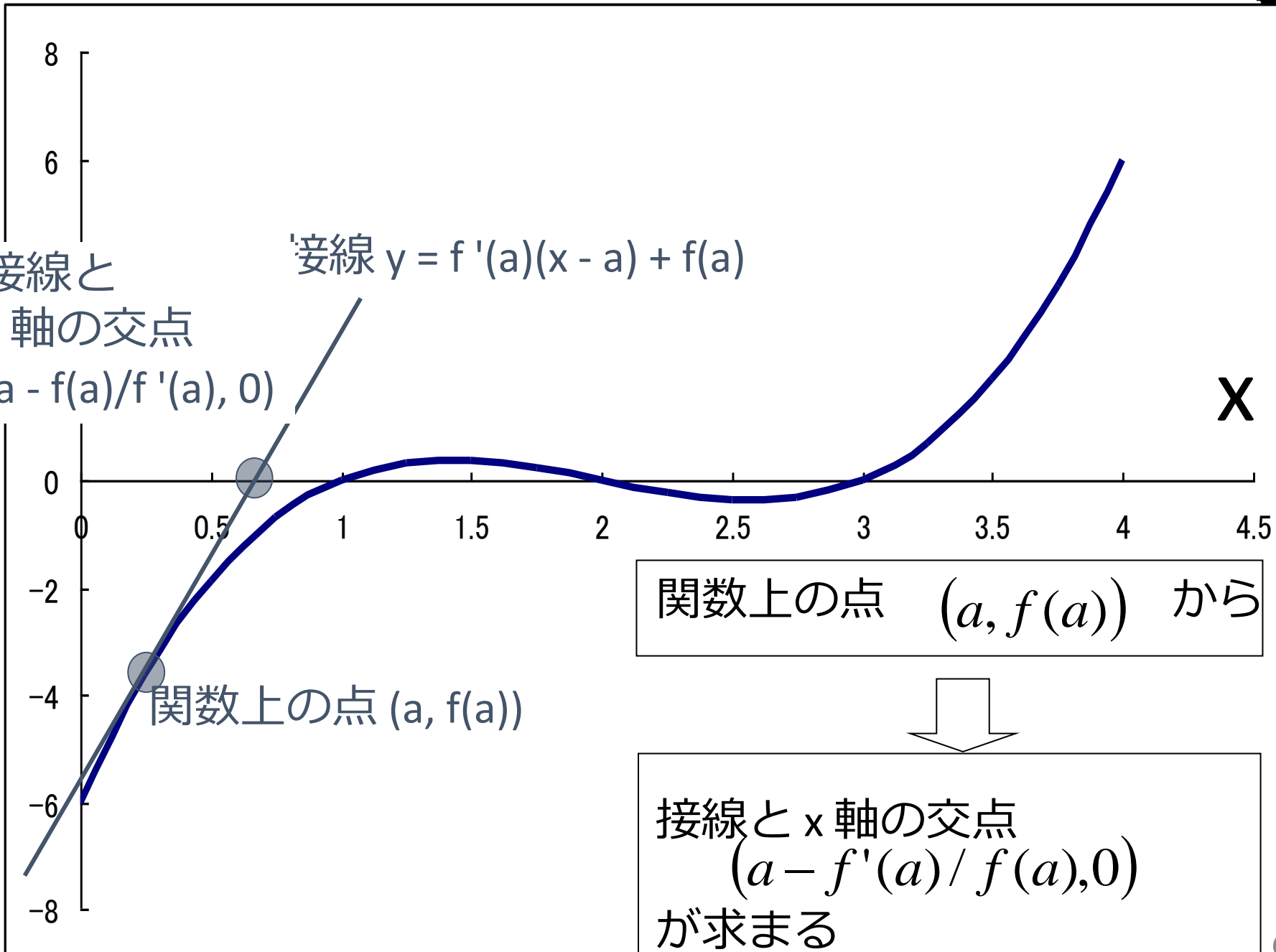












# ニュートン法



- 初期近似値  $x_0$  から出発
- 反復公式

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

を繰り返す

- $x_1, x_2, x_3 \dots$  は, だんだんと解に近づいていく  
(と望むことができる)

# ニュートン法



- 初期近似値  $x_0$  から出発
- 反復公式

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

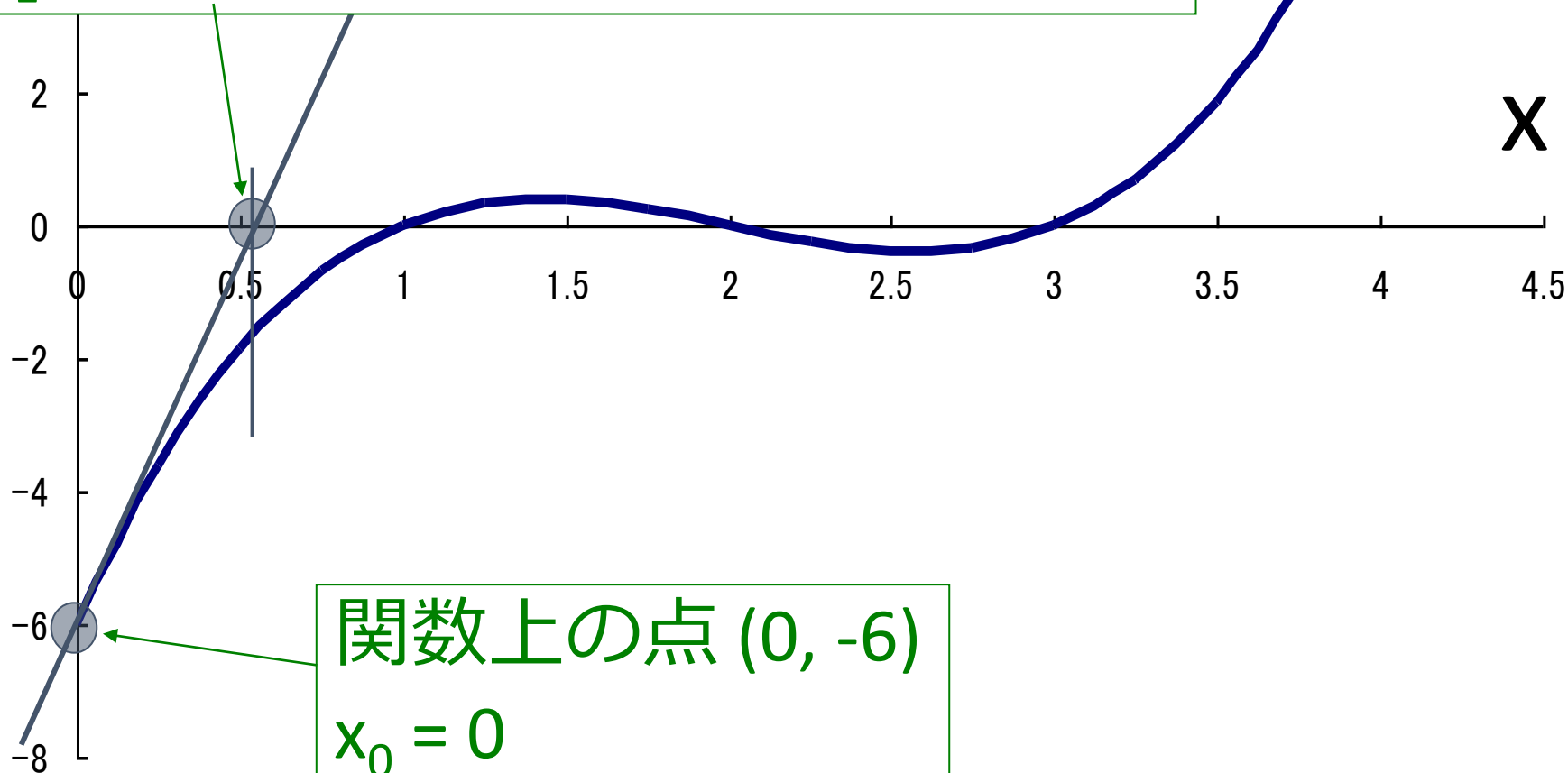
を繰り返す

- $x_1, x_2, x_3 \dots$  は, だんだんと解に近づいていく

これは, 点  $(x_i, f(x_i))$  における接線と,  
x軸 ( $y=0$ ) との交点  $(x_{i+1}, 0)$  を求めている

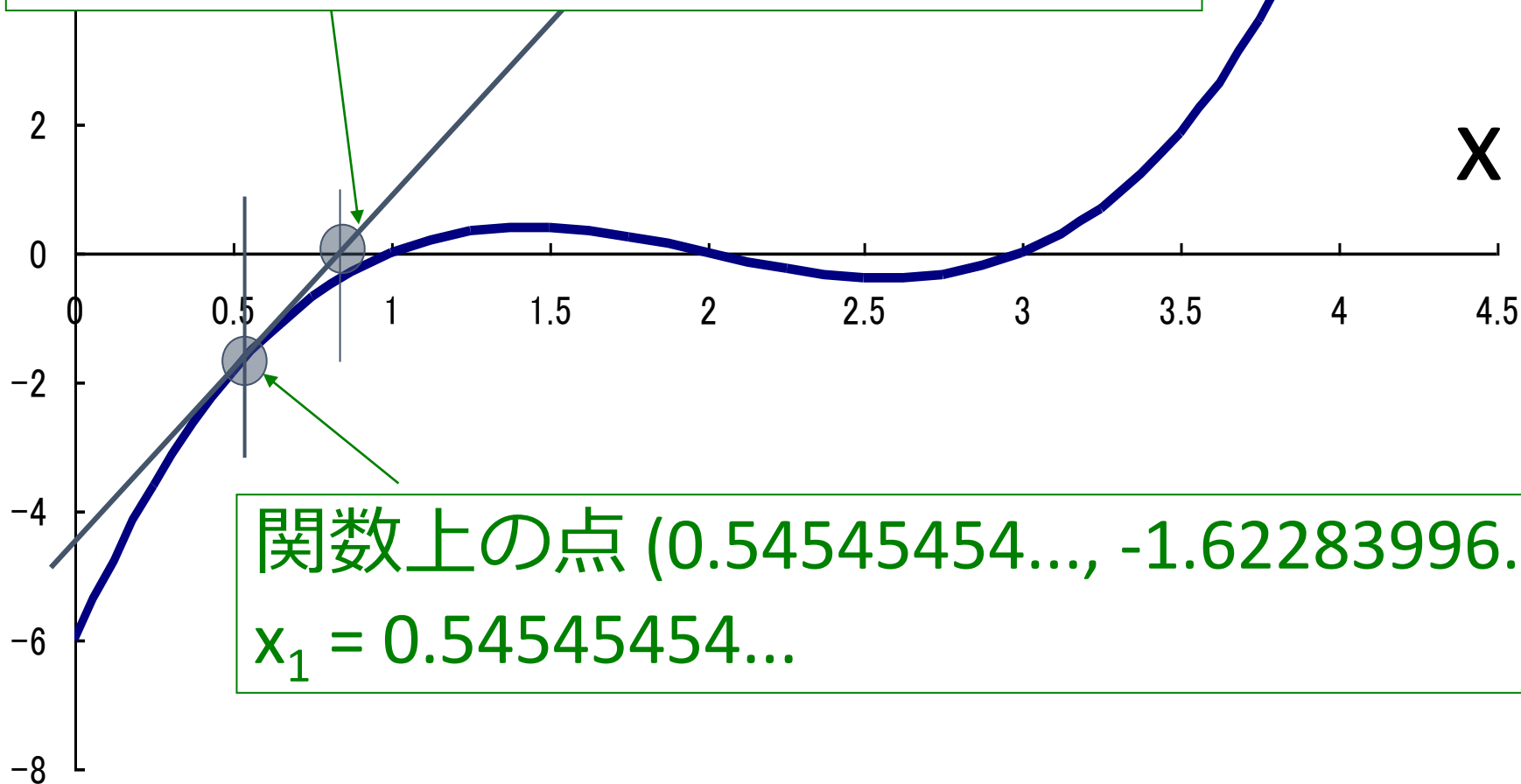
接線と x 軸の交点 (0.54545454..., 0)

$x_1 = 0.54545454\dots$



接線と x 軸の交点 (0.84895321..., 0)

$x_2 = 0.84895321\dots$

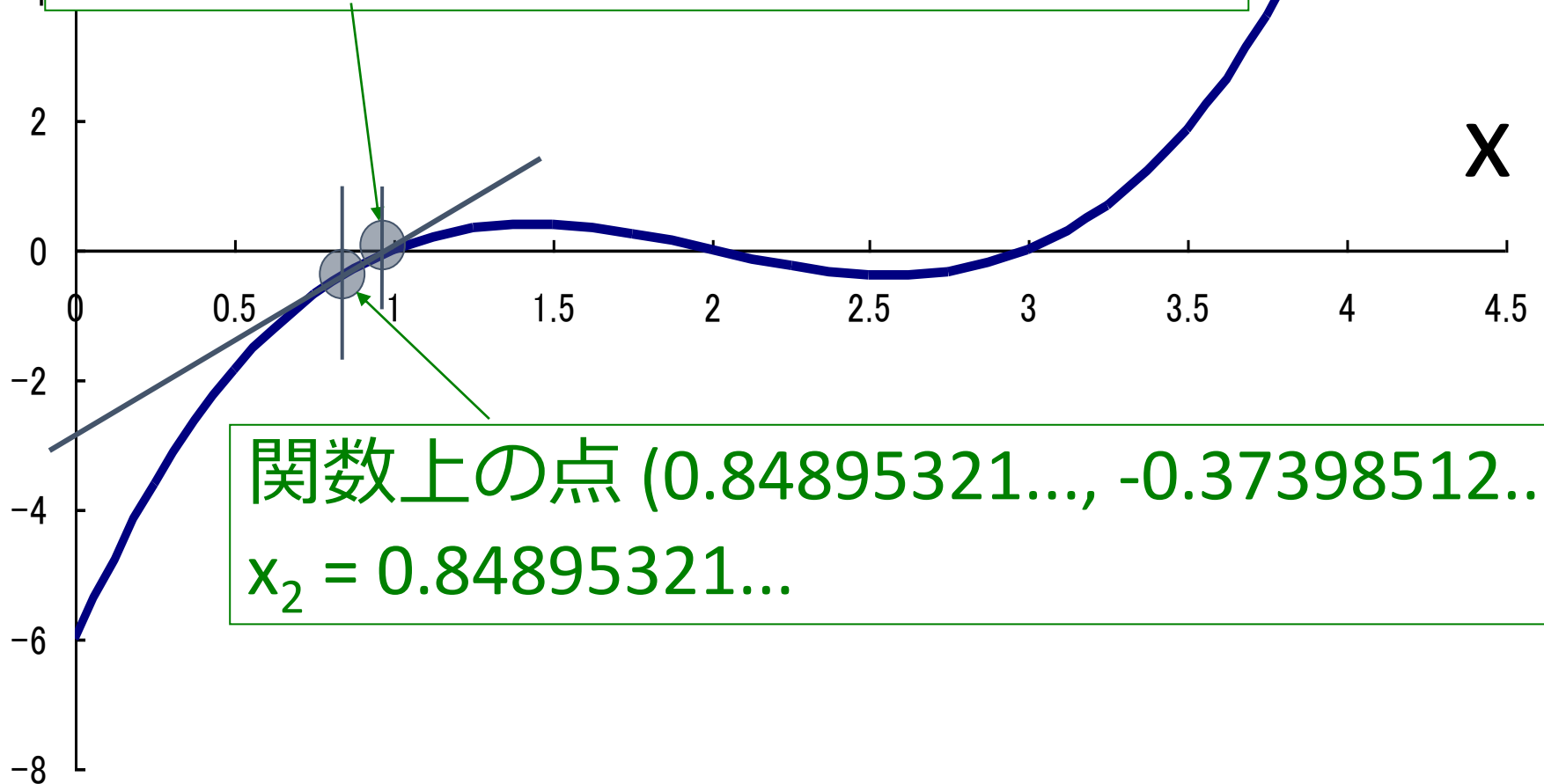


関数上の点 (0.54545454..., -1.62283996...)

$x_1 = 0.54545454\dots$

接線と x 軸の交点 (0.97467407..., 0)

$x_3 = 0.97467407...$

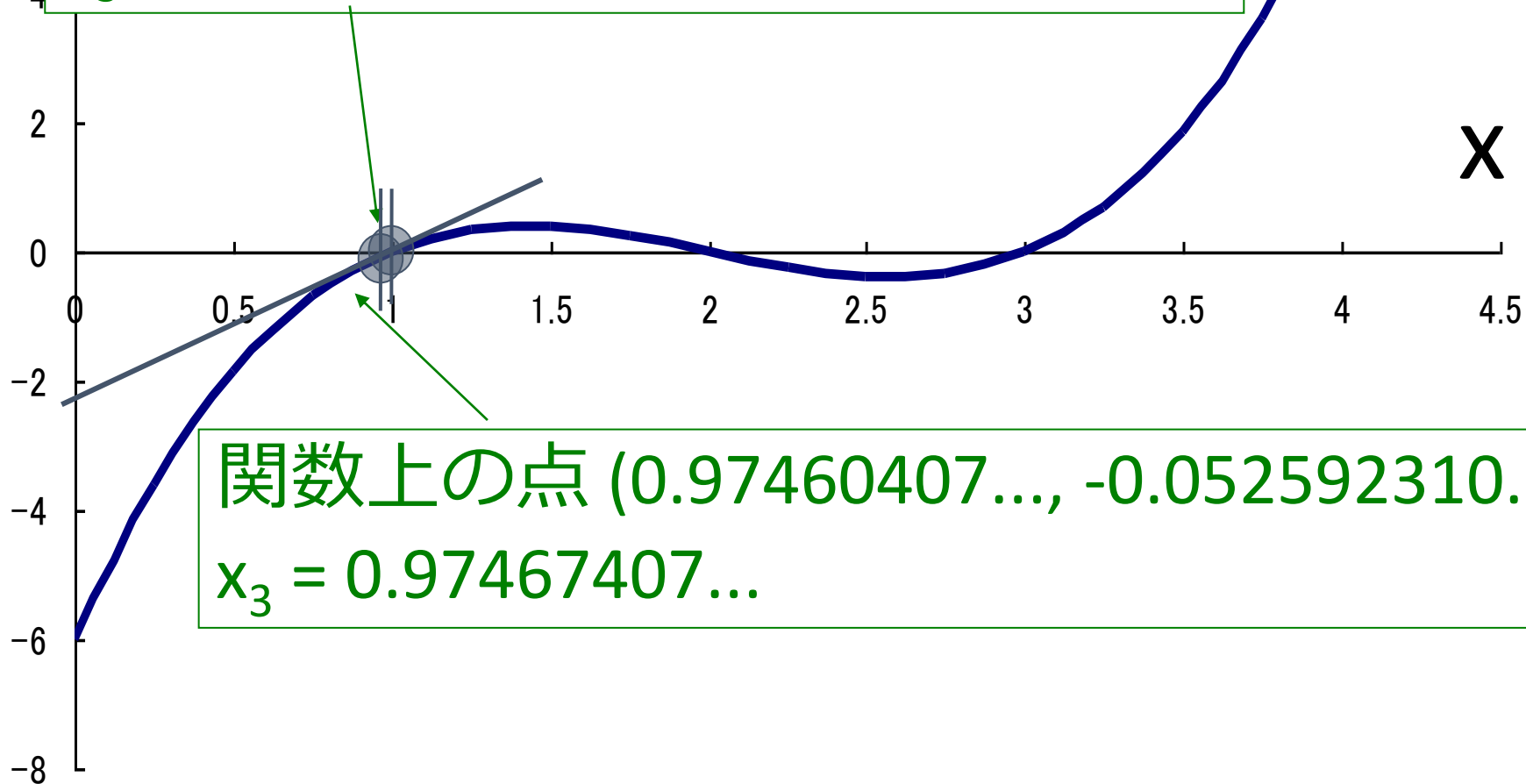


関数上の点 (0.84895321..., -0.37398512...)

$x_2 = 0.84895321...$

接線と x 軸の交点 (0.99909154..., 0)

$x_3 = 0.999909154...$



関数上の点 (0.97460407..., -0.052592310...)

$x_3 = 0.97467407...$

# ニュートン法の例



$$f(x) = x^3 - 6x^2 + 11x - 6, x_0 = 0 \text{ では :}$$

$$x_0 = 0$$

$$f(x_0) = -6, f'(x_0) = 11$$

$$\Rightarrow x_1 = x_0 - f(x_0)/f'(x_0) = 0.54545454...$$

$$x_1 = 0.54545454...$$

$$f(x_1) = -1.62283996..., f'(x_1) = 5.3471074...$$

$$\Rightarrow x_2 = x_1 - f(x_1)/f'(x_1) = 0.84895321...$$

$$x_2 = 0.84895321...$$

$$f(x_2) = 0.37398512..., f'(x_2) = 2.9747261...$$

$$\Rightarrow x_3 = x_2 - f(x_2)/f'(x_2) = 0.97460407...$$



# どうやって計算を終了するか



「反復公式」を繰り返すのだが

⇒ いつ止めたらいいいのか？

- 決定打は無いのだが、今日の授業では次のやり方で  
行ってみる

ある小さな正の数 $\delta$ に対して

$$|f(x_n)| < \delta$$

となった時点で計算を終了し  $x_n$  を解とする

# Scheme での多項式の書き方



- 「+」では, 足したいものを3つ以上並べてもよい

例)  $2x^2 - 3x - 1$   
(+ (\* 2 x x)  
(\* -3 x)  
-1))

# 14-2 パソコン演習

- 資料を見ながら、「例題」を行ってみる
- 各自、「課題」に挑戦する
- 自分のペースで先に進んで構いません

- DrScheme の起動  
プログラム → PLT Scheme → DrScheme
- 今日の演習では「Intermediate Student」  
に設定  
Language  
→ Choose Language  
→ Intermediate Student  
→ Execute ボタン

# 例題 1. ニュートン法による 非線形方程式の解



- 非線型方程式  $f(x) = 0$  をニュートン法で解く関数 `newton` を作り、実行する
  - 方程式:  $f$
  - 初期近似値:  $x_0$

# 「例題 1. ニュートン法による非線形方程式の解」の手順 (1/2)



1. 次を「定義用ウィンドウ」で，実行しなさい
  - 入力した後に，Execute ボタンを押す

```
;; d/dx: (number->number) number number -> number
;; inclination of the tangent
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f (- x h)))
      (* 2 h)))
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
(define (improve f guess)
  (- guess (/ (f guess) (d/dx f guess 0.0001))))
;; newton: (number->number) number number number ->
number
(define (newton f guess delta number)
  (cond
    [(or (is-good? f guess delta)
          (< number 0)) guess]
    [else (newton f (improve f guess) delta (- number 1))]))
(define (f2 x)
  (+ (* x x x) (* -6 x x) (* 11 x) -6))
```

2. その後, 次を「実行用ウィンドウ」で実行しなさい

```
(newton f2 #i0 0.00001 10000)
```

☆ 次は, 課題に進んでください



```
Untitled - DrScheme
File Edit Windows Show Language Scheme Help
[Untitled] [Save] [define...] [Check Syntax] [Step] [Execute] [Break]

;; d/dx: (number->number) number number -> number
;; inclination of the tangent
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f (- x h)))
     (* 2 h)))
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
(define (improve f guess)
  (- guess (/ (f guess) (d/dx f guess 0.0001))))
;; newton: (number->number) number number number -> number
(define (newton f guess delta number)
  (cond
    [(or (is-good? f guess delta)
         (< number 0)) guess]
    [else (newton f (improve f guess) delta (- number 1))]))

>
```



まず、関数 **newton** などを定義している

```
File Edit Windows Show Language Scheme Help
Untitled Save
Check Syntax Step Execute Break

;; d/dx: (number->number) number number -> number
;; inclination of the tangent
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f (- x h)))
      (* 2 h)))
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
(define (improve f guess)
  (- guess (/ (f guess) (d/dx f guess 0.0001))))
;; newton: (number->number) number number number -> number
(define (newton f guess delta number)
  (cond
    [(or (is-good? f guess delta)
         (< number 0)) guess]
    [else (newton f (improve f guess) delta (- number 1))]))
(define (f2 x)
  (+ (* x x x) (* -6 x x) (* 11 x) -6))
```

次に関数 **f2** を定義している

```
(define (f2 x)
  (+ (* x x x) (* -6 x x) (* 11 x) -6))
```

Untitled Save

```
;; d/dx: (num
;; inclinatio
(define (d/dx
  (/ (- (f (+
    (* 2 h))
  (define (is-g
    (< (abs (f
  (define (impr
    (- guess (/
;; newton: (n
(define (newt
  (cond
    [(or (is-
      (< n
    [else (ne
  (define (f2 x)
    (+ (* x x x
```

これは,

(**newton f2 #i0 0.00001 10000**)  
と書いて, guess の値を #i0 に,  
delta の値を 0.0001 に,  
number の値を 10000 に  
f を **f2** に設定しての実行

```
(+ (* x x x (* -6 x x) (* 11 x) -6))
```

```
> (newton f2 #i0 0.00001 10000)
```

```
#i0.9999987646860998
```

実行結果である

「#i0.9999987646860998」  
が表示される

# newton の入力と出力

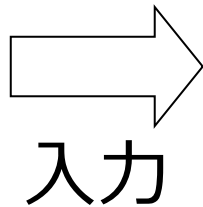


f2

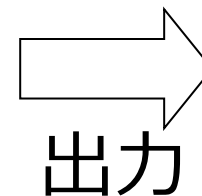
#i1

0.00001

10000



f2 = 0 の近似解  
の1つ



入力は1つの関数と,  
3つの数値

出力は数値

newton は, 関数を入力とするような関数  
(つまり高階関数)

```
;; d/dx: (number->number) number number -> number
```

```
;; inclination of the tangent
```

```
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f (- x h)))
      (* 2 h)))
```

```
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
```

```
(define (improve f guess)
  (- guess (/ (f guess) (d/dx f guess 0.0001))))
```

```
;; newton: (number->number) number number number -> number
```

```
(define (newton f guess delta number)
  (cond
    [(or (is-good? f guess delta)
         (< number 0)) guess]
    [else (newton f (improve f guess) delta (- number 1))]))
```

# ニュートン法の注意点



- 初期近似値の決め方
  - 初期近似値によって，求まる解が変わってくる
- 求まる解は，あくまでも「近似解」
  - 例：この例題では
    - #i0.9999987646860998
    - #i 付きの近似計算で十分
- 虚数解は求まらない

```
;; d/dx: (number->number) number number -> number
;; inclination of the tangent
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f x))
      (* 2 h)))
(define (is-good? f guess)
  (< (abs (f guess)) 0.00001))
(define (improve f guess)
  (- guess (/ (f guess) (d/dx f guess))))
;; newton: (number->number) number number -> number
(define (newton f guess)
  (cond
    [(or (is-good? f guess)
         (< number 0))
     guess]
    [else (newton f (improve f guess))])
(define (f2 x)
  (+ (* x x x) (* -6 x x) (* 11 x) -6))
```

(newton f2 #i0 0.00001 10000)の結果

#i0.9999987646860998

(newton f2 #i10 0.00001 10000)の結果

#i3.0000000168443197

(違う値が得られた)

```
> (newton f2 #i0 0.00001 10000)
#i0.9999987646860998
> (newton f2 #i10 0.00001 10000)
#i3.0000000168443197
>
```

```
;; d/dx: (number->number) number number -> number
;; inclination of the tangent
(define (d/dx f x h)
  (/ (- (f (+ x h)) (f (- x h)))
     (* 2 h)))
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
(define (improve f guess)
  (- guess (/ (f guess)
              (d/dx f guess 1))))
;; newton: (number->number) number number -> number
(define (newton f guess delta)
  (cond
    [(or (is-good? f guess delta)
         (< number 0))
     guess]
    [else (newton f (improve f guess) delta (- number 1))]))
(define (f2 x)
  (+ (* x x x) (* -6 x x) (* 11 x) -6))
```

(newton f2 0 0.00001 10000)の結果  
⇒ 結果は「有理数」で得られる  
(画面には表示しきれない)

```
> (newton f2 0 0.00001 10000)
47128950354837282762235430776585999722889881075479838666168642
67385011641881941708720038136528366012708582237636624964015372
61761855589554288406063302114295123417437515723948012820400442
05197747085364271917392458854046755832571307266118995542549032
41137892137808916844285103026486755013070647659245043008393412
20183944829864183734755612327818933959968085265129396399551612
34973254790719891001059483144346153655903843528926016261762602
12472475421707418789185074958211193223336466697656975664930022
58179157191972559530004986194453310130527614297984515077035712
80435641011573770847983726818365726164705686739005756827794602
85403965733365238571931534952819483688693206506250779427949042
25862187056380159469055689645915342564306247723199170075398372
92349947105435016832307604679303226169111208763534995154817652
62055301472591846410223916224201584099456529357266474862953272
52414172529822983420336745518929877036853207759193631852653042
31347694753749625972730402718935455208966042156722798664833412
```



# 「ニュートン法のプログラム」 の理解のポイント



- 繰り返しの終了条件は2つ
  - 収束の条件を満足した  $|f(x_n)| < \delta$
  - 繰り返しの上限回数を超えた
- 入力は4つ
  - 求めるべき関数: `f`
  - 初期近似値: `guess`
  - 収束条件を決める値: `delta`
  - 繰り返し回数の上限: `number`

# ニュートン法の繰り返し処理



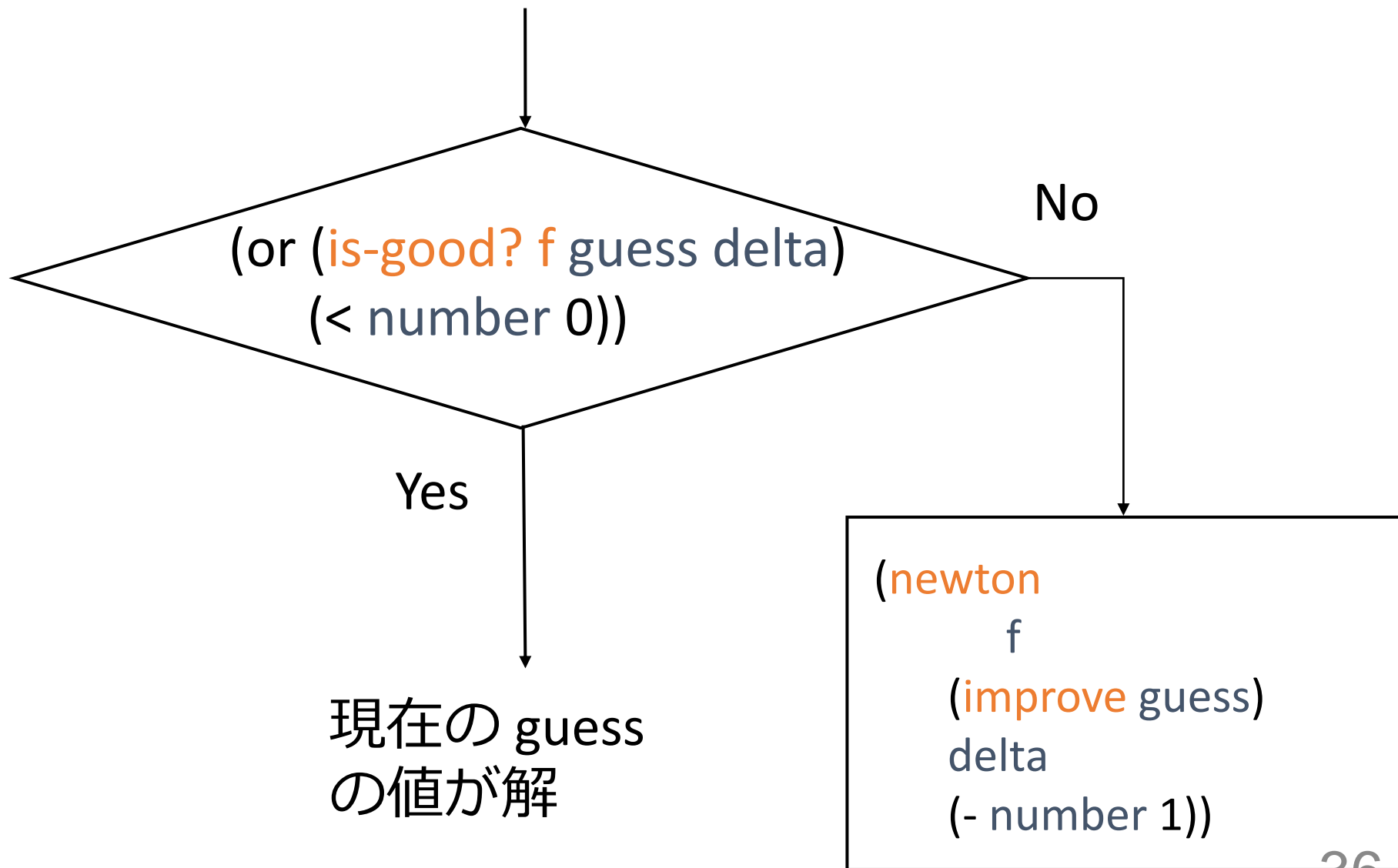
- $x_0$  から始める
- $x_1$  を求める
- $x_2$  を求める
- ...
- 「収束の条件を満足する」か「繰り返しの上限回数を超える」まで続ける

# ニュートン法の繰り返し処理



1. 「収束した」あるいは「繰り返しの上限回数を超えた」ならば：
  - 終了条件
  - 現在の  $x_n$  の値 → 自明な解
2. そうで無ければ：
  - 「接線と  $x$  軸の交点の計算」を行う
    - 求まった交点の  $x$  座標の値は  $x_{n+1}$
  - 結局, 「収束する」か「繰り返しの上限回数を超える」まで, 接線と  $x$  軸の交点の計算を繰り返す

# ニュートン法の繰り返し処理



# (newton f2 #i0 0.00001 10000) から結果が得られる過程

(newton f2 #i0 0.00001 10000) 最初の式

```
= ...  
= (newton (improve f2 #i0) 0.00001 9999)  
= ...  
= (newton #i0.5454545449588037 0.00001 9999)  
= ...  
= (newton (improve f2 #i0.5454545449588037) 0.00001 9998)  
= ...  
= (newton #i0.8489532098093157 0.00001 9998)  
= ...  
= (newton (improve f2 #i0.8489532098093157 ) 0.00001 9997)  
= ...
```

コンピュータ内部での計算

= #i0.9999987646860998 実行結果

# (newton f2 #i0 0.00001 10000) から結果が得られる過程

```
(newton f2 #i0 0.00001 10000)
```

```
= ...
```

```
= (newton (improve f2 #i0) 0.00001 9999)
```

```
= ...
```

これは,

```
(define (newton f guess delta number)
```

```
(cond
```

```
  [(or (is-good? f guess delta)
```

```
       (< number 0)) guess]
```

```
  [else (newton f (improve f guess) delta (- number 1))]))
```

の f を f2 で, guess を #i0 で, delta を 0.00001 で, number を 10000 で置き換えたもの

```
= #i0.9999987646860998
```

- ニュートン法では, 現在の  $x_n$  の誤差 (どれだけ真の値に近いか) は, 正確には分からない

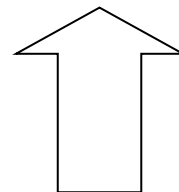
# ニュートン法での収束条件



- `is-good?` が収束条件を判定

```
(define (is-good? f guess delta)
  (< (abs (f guess)) delta))
```

abs は「絶対値」  
を求める



$|f(x_n)| < \delta$  が成立するとき, 出力は true.  
(さもなければ false)



# ニュートン法での繰り返し処理



**number:** カウンタ（繰り返しの残り回数）  
**guess:**  $x_n$  の値（計算の途中結果）

```
number ← number - 1  
guess ← (improve guess)
```

} を繰り返す

# ニュートン法での繰り返し処理



```
(define (newton f guess delta number)
  (cond
    [(or (is-good? f guess delta)
         (< number 0)) guess]
    [else (newton f (improve f guess) delta (- number 1))]))
```

guess ← (improve f guess)

number ← number - 1

# $f(x) = x^2 - 5$ での $x_1, x_2 \dots$ の収束の様子



```
(newton f #i1 0.00001 10000)
= ...
= (newton f #i3.0 0.00001 10000)
= ...
= (newton f #i2.3333333333333335 0.00001 9999)
= ...
= (newton f #i2.238095238095238 0.00001 9998)
= ...
= (newton f #i2.2360688956433634 0.00001 9997)
= ...
= (newton f #i2.236067977499978 0.00001 9996)
= ...
```

方程式  $f(x) = x^2 - 5$   
 $f(x)$ の導関数  $f'(x) = 2x$

$\underbrace{\hspace{10em}}_{\text{guess}} \quad \underbrace{\hspace{2em}}_{\text{delta}} \underbrace{\hspace{2em}}_{\text{number}}$

- 初期近似値の決め方

- 初期近似値の設定の際、あまりに解と掛け離れた値を与えると、収束するのに時間がかかったり、収束しなかったりする

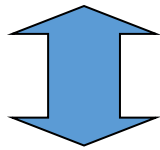
- delta の決め方

- どの程度の精度で計算するかを決定していないと、 $\epsilon$ が決まらない

# ニュートン法の能力と限界



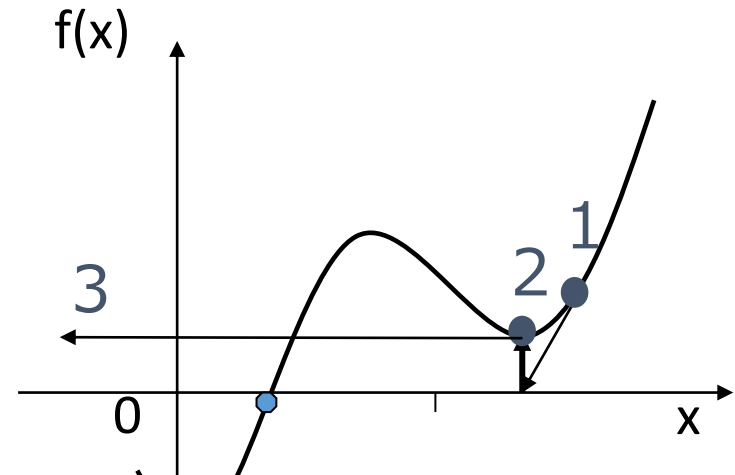
ニュートン法は, 出発点とする十分近い解を見付けることができれば, 収束が早い.



初期近似値の選び方次第では, 収束がしないことがある

**対策**

繰り返しの上限回数 number を設定



関数  $f(x)$  が単調でなくて変曲点を持つ (つまり  $f'(x)$  の符号が変わる) とき例) 上の図では:

2 :  $f'(x) = 0$

3 :  $y$  軸との交点が求まらない (負の無限大に発散)

→ 収束しない

# 14-3 課題

# 課題 1



- 立方根を求めるプログラムを, Newton 法のプログラムを利用して作成せよ
  - $f(x) = x^3 - a = 0$  を解く
  - $a$  が負のときにも正しく負の立方根を求めることができることを確認せよ
  - $\delta$  の値を変えて実行を繰り返し, 得られた解の値の変化も報告しなさい

## 課題 2. ニュートン法での収束



1.  $f(x) = x^2 - 5$  のグラフを書け (手書き)
2. 1. で作成したグラフに, ニュートン法での,  $x_0$ ,  $x_1$ ,  $x_2$ ,  $x_3$  の値を書き加えなさい

但し,

- 関数

方程式  $f(x) = x^2 - 5$

- 入力

初期値  $x_0 = 1$

収束条件を決める値  $\text{delta} = 0.00001$

繰り返し回数の上限  $\text{number} = 10000$

$x_1$ ,  $x_2$ ,  $x_3$  の値は, 実際にニュートン法のプログラムを実行して得ること



# 課題 3



- $(x-1)^4(x-2) = 0$  を newton 法で解け
  - 初期近似値を変えて実行を繰り返し、得られた解の値の変化も報告しなさい

## 演習 4



- Newton 法により  $f(x) = \tan^{-1} x + 0.3x$  を解け
  - 初期近似値の選び方によっては、正しく解を求めることができない。その理由についても考え、グラフを書いて説明しなさい。

# さらに勉強したい人への 補足説明事項

## 区間二分法

# 区間二分法



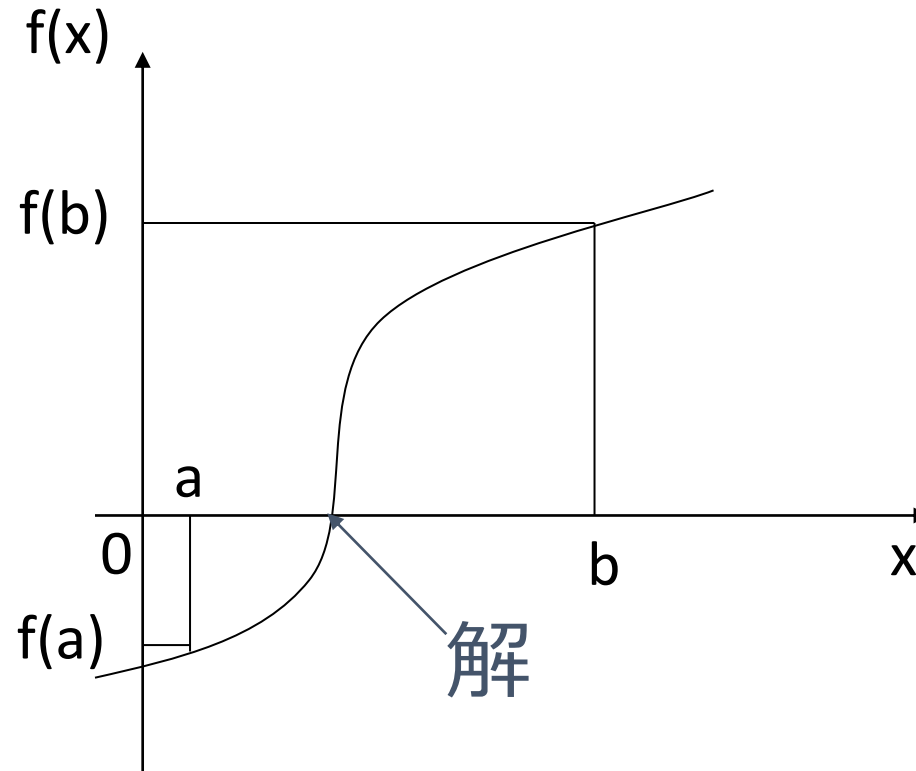
- 非線形方程式の求解の一手法

# 区間二分法 (half-interval method) の考え方



- 連続関数  $f$  の性質

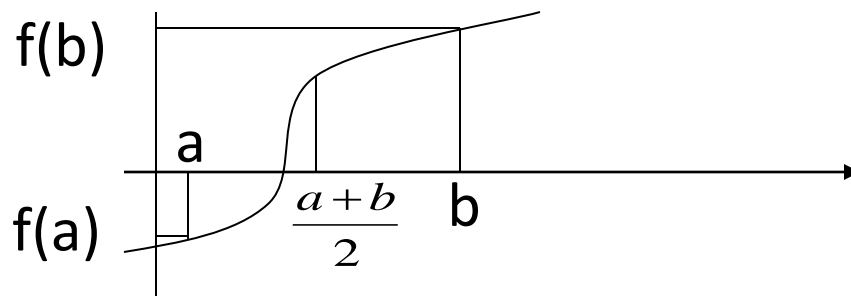
「 $f(a) < 0 < f(b)$ ならば、 $[a, b]$ の間に  $f(x) = 0$  の解がある



# 区間二分法 (half-interval method) の処理手順



- 「区間」を半分ずつ縮小するような処理手順
  1. 2点  $a, b$  を定める
  2. 2点  $a, b$  の中点  $(a+b)/2$  について：  
 $f((a+b)/2) > 0$  ならば → 解は,  $[a, (a+b)/2]$  にある  
 $f((a+b)/2) < 0$  ならば → 解は,  $[(a+b)/2, b]$  にある
  3. 2. を繰り返す. 「区間」の幅が小さくなる → 解の近似値を得られる



## 例題 2. 区間二分法による非線形方程式の解



- $f(x)=x^2 - 2$  を区間二分法で解くプログラム **half-interval** を書く

- $f(x) = x^2 - 2$

解は  $\sqrt{2}$  と  $-\sqrt{2}$

## 「例題 2. 区間二分法による非線形方程式の解」の手順(1/2)



1. 次を「定義用ウィンドウ」で，実行しなさい
  - 入力した後に，Execute ボタンを押す

```
(define (f x)
  (- (* x x) 2))
(define (good-enough? a b)
  (< (- b a) 0.000001))
(define (middle a b)
  (/ (+ a b) 2))
(define (half-interval a b)
  (cond
    [(good-enough? a b) a]
    [else
     (cond
       [(< (f (middle a b)) 0) (half-interval (middle a b) b)]
       [(= (f (middle a b)) 0) (middle a b)]
       [(> (f (middle a b)) 0) (half-interval a (middle a b))]]))])
```



## 「例題 2. 区間二分法による非線形方程式の解」の手順 (2/2)



2. その後, 次を「実行用ウィンドウ」で実行  
しなさい
  - 正しい解が得られている場合と, 得られていない場合があることを確認しなさい

(half-interval 0 2)

(half-interval -2 0)

(half-interval 2 4)

# 「区間二分法による非線形方程式の解」の実行結果

```
Untitled - DrScheme
File Edit Windows Show Language Scheme Help
[Untitled] [Save] [Check Syntax] [Step] [Execute] [Break]
(define ...)
(define (f x)
  (- (* x x) 2))
(define (good-enough? a b)
  (< (- b a) 0.000001))
(define (middle a b)
  (/ (+ a b) 2))
(define (half-interval a b)
  (cond
    [(good-enough? a b) a]
    [else
     (cond
      [(< (f (middle a b)) 0) (half-interval (middle a b) b)]
      [(= (f (middle a b)) 0) (middle a b)]
      [(> (f (middle a b)) 0) (half-interval a (middle a b))]
      )
     ]
  )
)
Welcome to DrScheme, version 103p1.
Language: Intermediate Student.
> (half-interval 0 2)
1.4142131805419921875
> (half-interval -2 0)
-0.00000095367431640625
> (half-interval 2 4)
2
```

> (half-interval 0 2)  
1.4142131805419921875  
→ 正しい

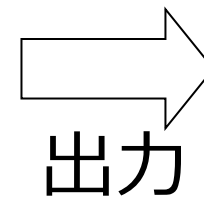
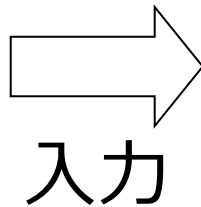
> (half-interval -2 0)  
-0.00000095367431640625  
→ 正しくない

> (half-interval 2 4)  
2  
→ 正しくない

# 入力と出力



a の値: 0  
b の値: 2



1.4142131805419921875

入力は2つの数値

出力は数値

# 区間二分法のプログラム



- 関数

- 方程式  $f(x)$

- 入力

- 初期値  $a, b$

- 出力

- 解

# 区間二分法の処理手順



1. 初期値  $a, b$  の決定

2. 区間を半分に縮小

- $f((a+b)/2)$  の値が

正 :  $b \leftarrow (a+b)/2$

0 :  $(a+b)/2$  が解である

負 :  $a \leftarrow (a+b)/2$

3. 2. を繰り返す

# 区間二分法の処理手順



$f(x) = x^2 - 2$ ,  $a = 0$ ,  $b = 2$  の場合

$a = 0$ ,  $b = 2$

$$f((a+b)/2) = f(1) = -1$$

⇒  $a$  を「1」に

$a = 1$ ,  $b = 2$

$$f((a+b)/2) = f(1.5) = 0.25$$

⇒  $b$  を「1.5」に

$a = 1$ ,  $b = 1.5$

$$f((a+b)/2) = f(1.25) = -0.4375$$

⇒  $a$  を「1.25」に

# 区間二分法の繰り返し処理



- half-interval の内部に half-interval が登場

```
(define (half-interval a b)
  (cond
    [(good-enough? a b) a]
    [else
     (cond
       [(< (f (middle a b)) 0) (half-interval (middle a b) b)]
       [(= (f (middle a b)) 0) (middle a b)]
       [(> (f (middle a b)) 0) (half-interval a (middle a b))]))]))
```

- half-interval の実行が繰り返される

# 区間二分法での収束条件



- `good-enough?` が収束条件を判定

```
(define (good-enough? a b)
  (< (- b a) 0.000001))
```

$b - a < 0.000001$  が成立するとき, 出力は true.  
(さもないければ false)

\* 「0.000001」 は適当に決めた値



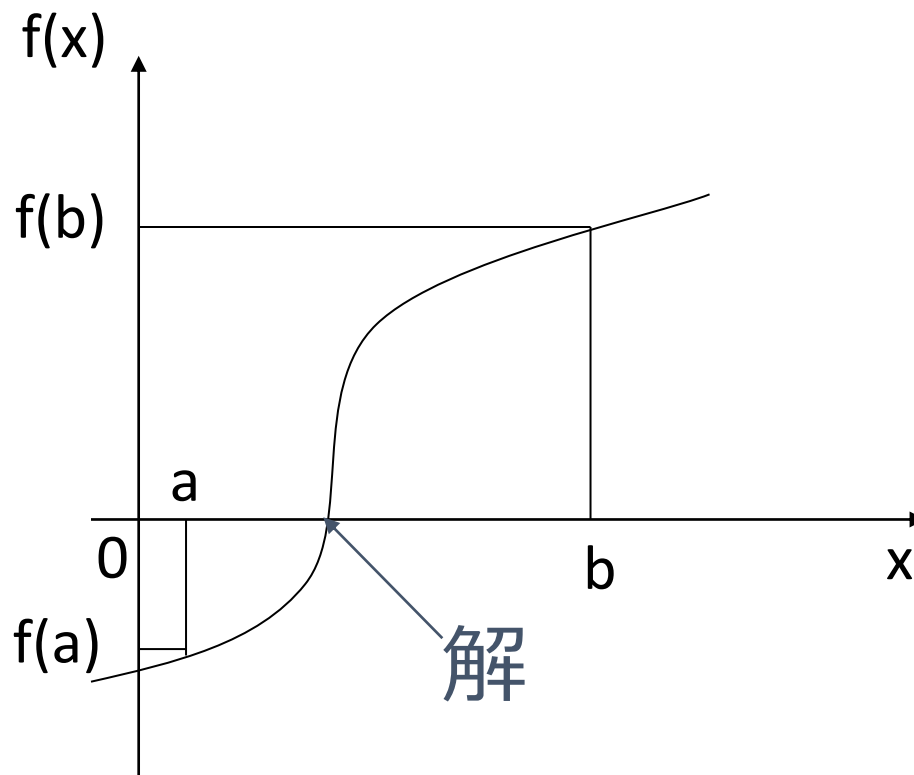
# 「区間二分法のプログラム」 の理解のポイント

- 繰り返しの終了条件は2つ
  - $b - a$  の値が, 「しきい値」を下回った (収束した)
  - 「 $f((a+b)/2) = 0$ 」となった
- 入力は2つ
  - 初期値  $a, b$
- 繰り返し処理: 反復的プロセス

# 区間二分法での注意点



- $f(a) \geq 0, f(b) \leq 0$  でないと, 解が得られない



# 例題 3 . 区間二分法での繰り返し処理



- 例題 2 の「区間二分法」のプログラムについて,  
(half-interval 0 2) から (half-interval (middle 1 3/2) 3/2) までの過程を確認する

(half-interval 0 2)

= ...

= (half-interval (middle 0 2) 2)

= ...

= (half-interval 1 2)

= ...

= (half-interval 1 (middle 1 2))

= ...

= (half-interval 1 3/2)

= ...

= (half-interval (middle 1 3/2) 3/2)

# 「例題 3. 区間二分法での繰り返し処理」の手順

1. 次を「定義用ウィンドウ」で、実行しなさい
  - Intermediate Student で実行すること
  - 入力した後に、Execute ボタンを押す

```
(define (f x)
  (- (* x x) 2))
(define (good-enough? a b)
  (< (- b a) 0.000001))
(define (middle a b)
  (/ (+ a b) 2))
(define (half-interval a b)
  (cond
    [(good-enough? a b) a]
    [else
     (cond
       [(< (f (middle a b)) 0) (half-interval (middle a b) b)]
       [(= (f (middle a b)) 0) (middle a b)]
       [(> (f (middle a b)) 0) (half-interval a (middle a b))])]))
(half-interval 0 2)
```

例題 2 に  
1 行書き加える

2. DrScheme を使って、ステップ実行の様子を確認しなさい (Step ボタン, Next ボタンを使用)
  - 理解しながら進むこと