

Numerical Methods I: root finding

Computers store numbers with a FINITE number of digits

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0).

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0.
- 2) **E_R : round-off error** (error made in the operations between two FP#s). I always have an error related to the last significant digit in the mantissa.

Ex: $1.23 \times 1.27 = 1.578$, but the machine gives me 1.58

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0.
- 2) **E_R : round-off error** (error made in the operations between two FP#s). I always have an error related to the last significant digit in the mantissa.

Ex: $1.23 \times 1.27 = 1.578$, but the machine gives me 1.58

Take N operations, final error is: $E_{\text{final}} \approx \sqrt{N} \times E_R$

i.e., the more one plays with numbers, or the more operations one does, the larger the error will be

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0.
- 2) **E_R : round-off error** (error made in the operations between two FP#s). I always have an error related to the last significant digit in the mantissa.

Ex: $1.23 \times 1.27 = 1.578$, but the machine gives me 1.58

Take N operations, final error is: $E_{\text{final}} \approx \sqrt{N} \times E_R$

i.e., the more one plays with numbers, or the more operations one does, the larger the error will be

- 3) **E_T : truncation error** (completely dependent on the programmer)

$E_T \equiv (\text{numeric solution of a problem}) - (\text{exact solution of the problem})$

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0.
- 2) **E_R : round-off error** (error made in the operations between two FP#s). I always have an error related to the last significant digit in the mantissa.

Ex: $1.23 \times 1.27 = 1.578$, but the machine gives me 1.58

Take N operations, final error is: $E_{\text{final}} \approx \sqrt{N} \times E_R$

i.e., the more one plays with numbers, or the more operations one does, the larger the error will be

- 3) **E_T : truncation error** (completely dependent on the programmer)

$E_T \equiv (\text{numeric solution of a problem}) - (\text{exact solution of the problem})$

$$\text{Ex. : } f(x) = f(x_0) + f'(x_0)\Delta x + f''(x_0)\frac{\Delta x^2}{2} + o(\Delta x^3)$$

Computers store numbers with a FINITE number of digits

There are two basics data types:

- **Integer**: exact representation (within the allowed range, i.e., no approximation) and exact arithmetics (i.e., no division)
- **Floating points numbers (FP#)**: $s \times M \times B^{T-E}$, with s “sign”, M “mantissa” (aka, significand or coefficient), B “base”, T “exponent”, and E “bytes”.

Ex: $1.2345 = 12345 \times 10^{-4}$

12345 = mantissa, 10 = base, -4 = exponent; all integers

Definitions:

- 1) **E_m : machine accuracy** (the smallest floating point number that can be added to 1.0 and produces something different from 1.0).
- 2) **E_R : round-off error** (error made in the operations between two FP#s). I always have an error related to the last significant digit in the mantissa.

Ex: $1.23 \times 1.27 = 1.578$, but the machine gives me 1.58

Take N operations, final error is: $E_{\text{final}} \approx \sqrt{N} \times E_R$

i.e., the more one plays with numbers, or the more operations one does, the larger the error will be

- 3) **E_T : truncation error** (completely dependent on the programmer)

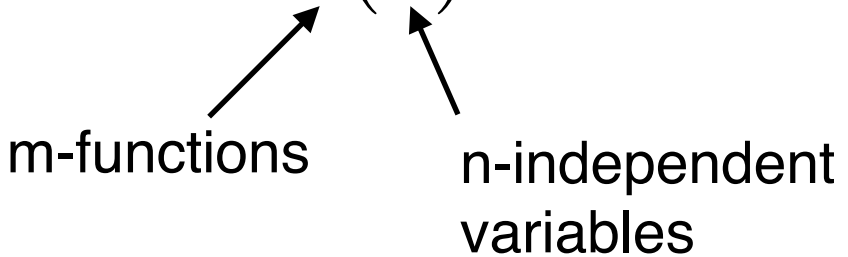
$E_T \equiv (\text{numeric solution of a problem}) - (\text{exact solution of the problem})$

$$\text{Ex. : } f(x) = f(x_0) + f'(x_0)\Delta x + f''(x_0)\frac{\Delta x^2}{2} + o(\Delta x^3) \quad E_T$$

ROOT FINDING: \bar{x} so that $\bar{f}(\bar{x}) = 0$

The diagram illustrates the components of the root finding process. It features the text "ROOT FINDING: \bar{x} so that $\bar{f}(\bar{x}) = 0$ ". Below this, two labels are positioned: "m-functions" on the left and "n-independent variables" on the right. Two arrows originate from these labels and point towards the \bar{f} and \bar{x} terms in the equation above. The arrow from "m-functions" points to the \bar{f} , and the arrow from "n-independent variables" points to the \bar{x} .

ROOT FINDING: \bar{x} so that $\bar{f}(\bar{x}) = 0$



m-functions n-independent
variables

There are no good general methods for solving systems of more than one non-linear equation.

Things are much simpler if $m=n=1$.

Except for linear problems, root finding proceeds by iteration, i.e., we start with a suitable trial value and use an algorithm that will improve it until satisfaction.

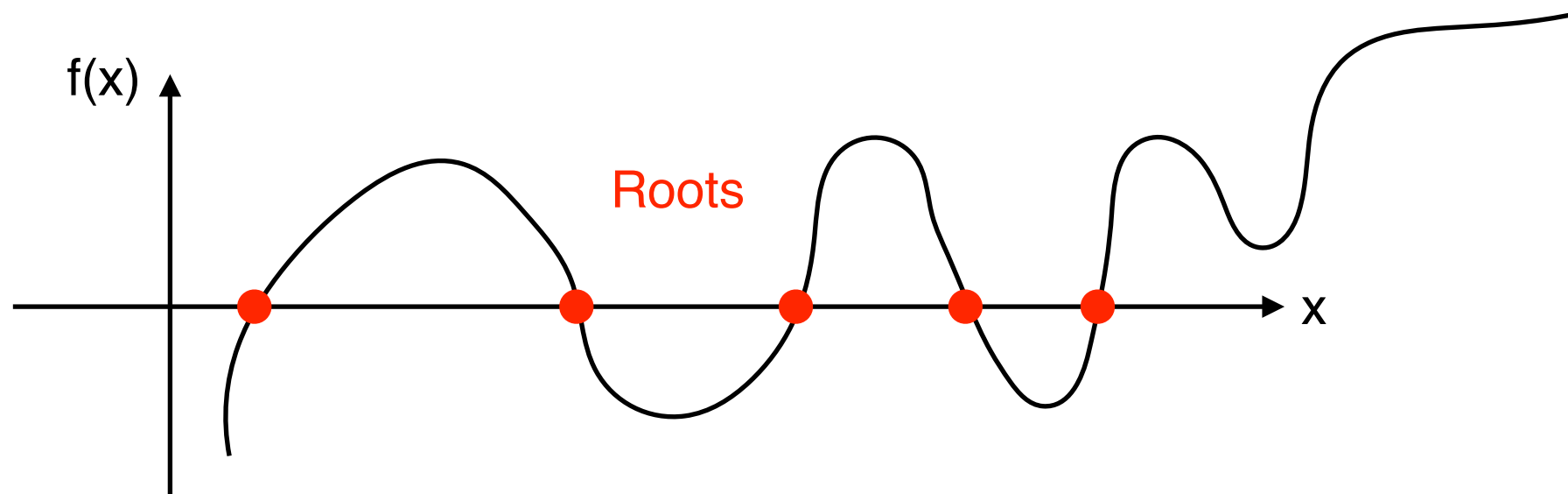
ROOT FINDING: \bar{x} so that $\bar{f}(\bar{x}) = 0$

m-functions n-independent
 variables

There are no good general methods for solving systems of more than one non-linear equation.

Things are much simpler if $m=n=1$.

Except for linear problems, root finding proceeds by iteration, i.e., we start with a suitable trial value and use an algorithm that will improve it until satisfaction.



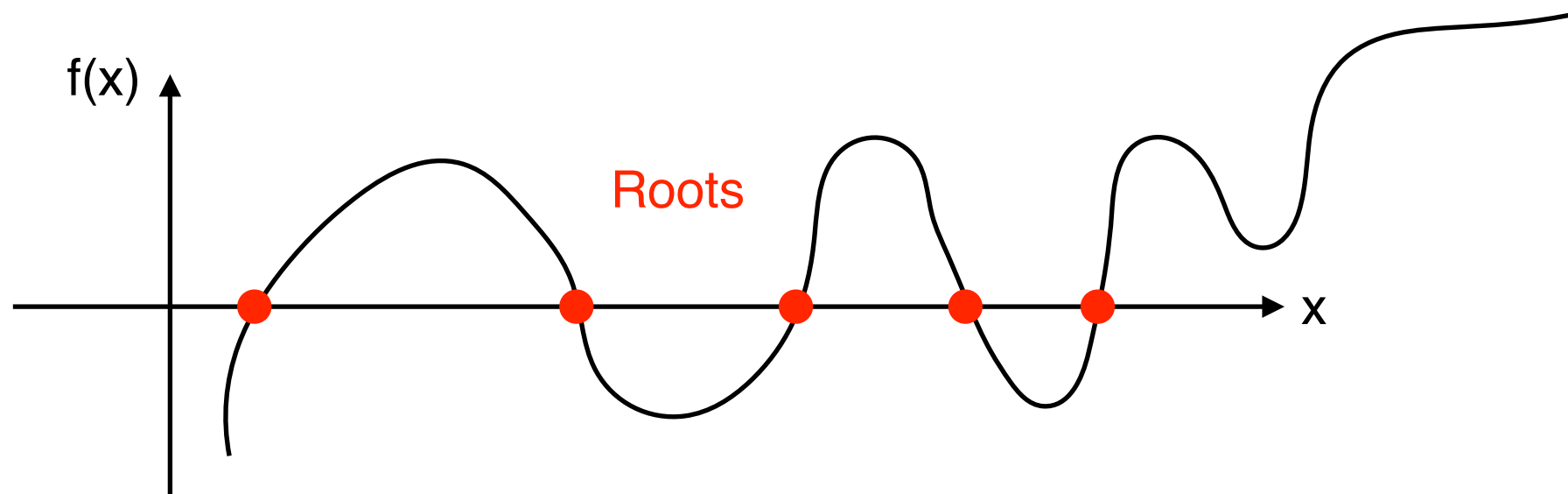
ROOT FINDING: \bar{x} so that $\bar{f}(\bar{x}) = 0$

m-functions n-independent
 variables

There are no good general methods for solving systems of more than one non-linear equation.

Things are much simpler if $m=n=1$.

Except for linear problems, root finding proceeds by iteration, i.e., we start with a suitable trial value and use an algorithm that will improve it until satisfaction.

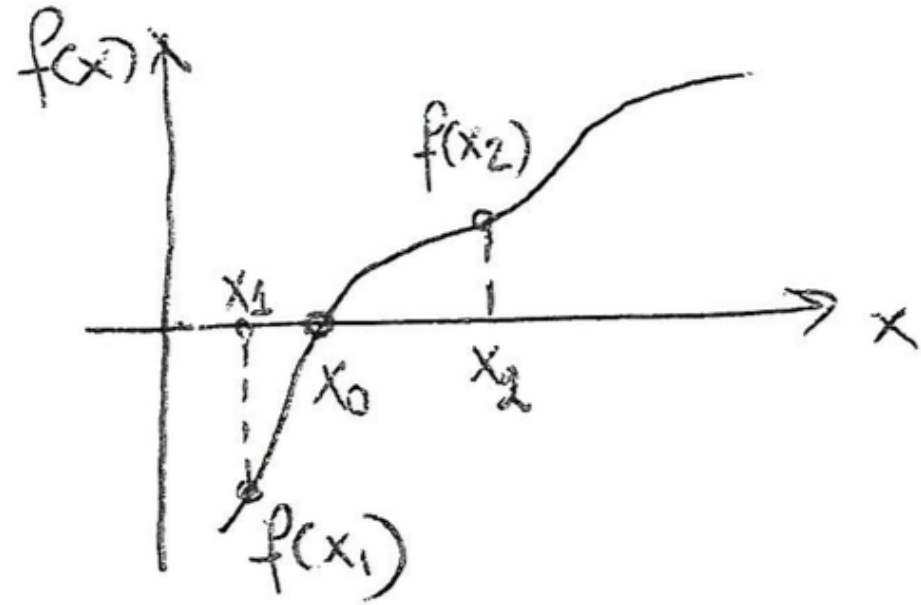


Rule of thumb: first of all, make sure there is a root (plot the function!)

Bracketing:

Let be $f = f(x)$, $x \in \mathbb{R}$

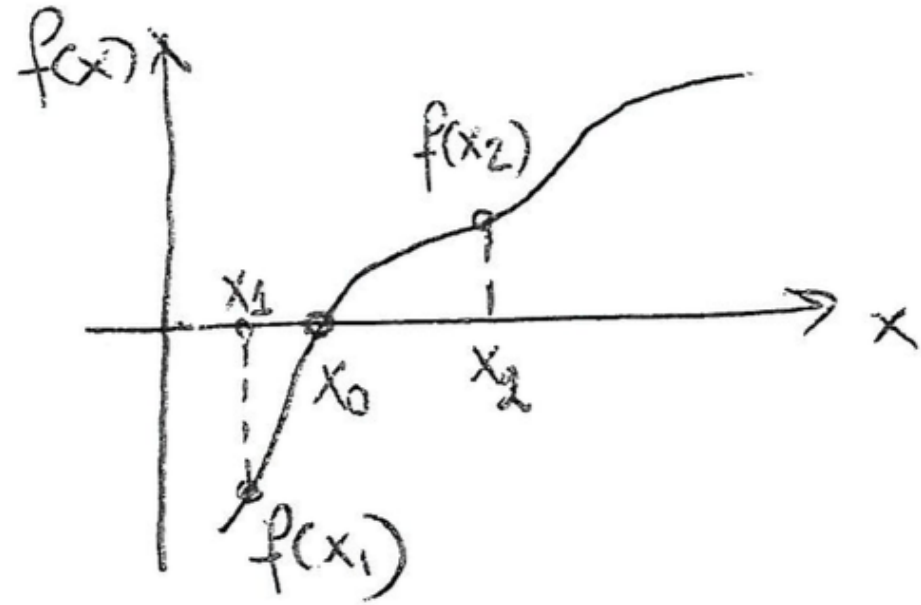
then x_0 is bracketed in $[x_1, x_2]$ if $f(x_1)f(x_2) < 0$



Bracketing:

Let be $f = f(x)$, $x \in \mathbb{R}$

then x_0 is bracketed in $[x_1, x_2]$ if $f(x_1)f(x_2) < 0$

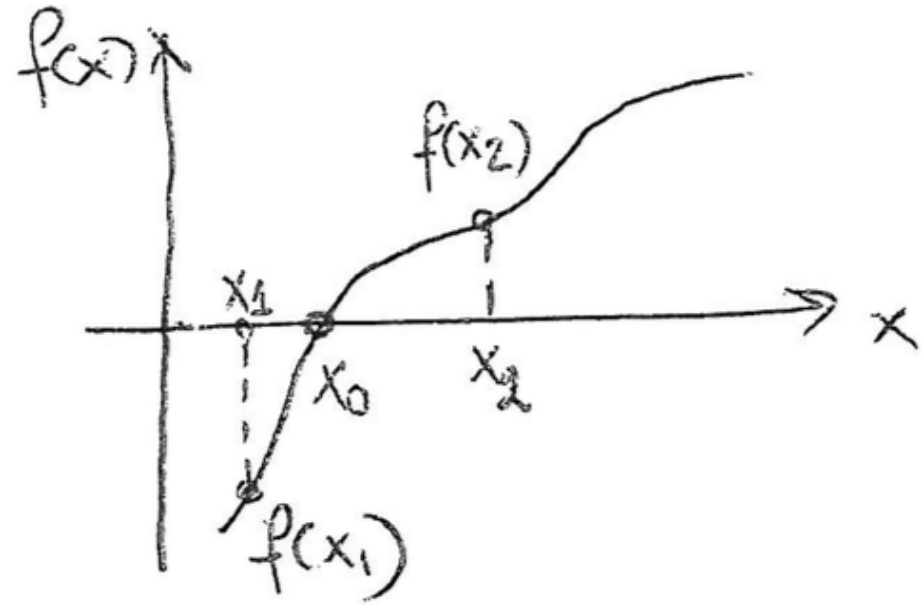


Note: $f(x_1)f(x_2) < 0$ is not a sufficient nor a necessary condition of the existence of the root.

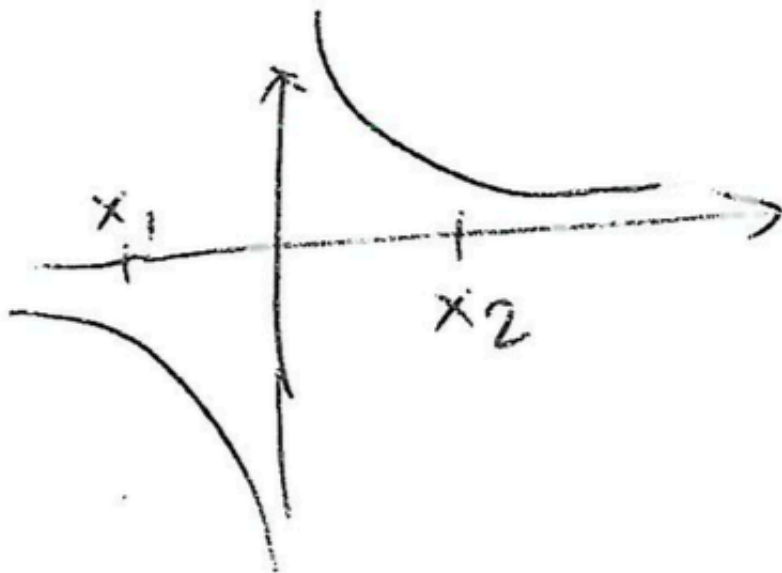
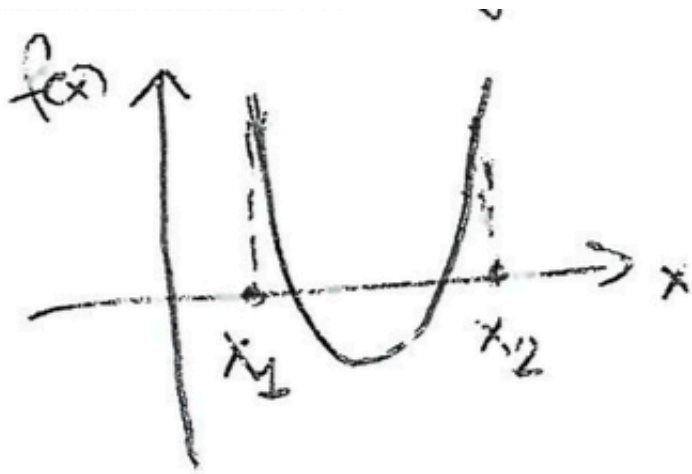
Bracketing:

Let be $f = f(x)$, $x \in \mathbb{R}$

then x_0 is bracketed in $[x_1, x_2]$ if $f(x_1)f(x_2) < 0$



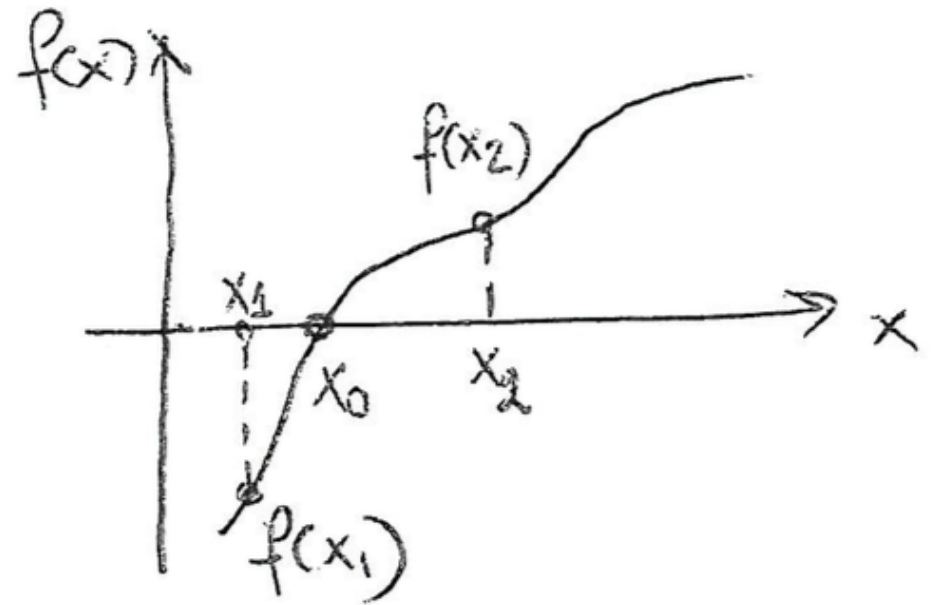
Note: $f(x_1)f(x_2) < 0$ is not a sufficient nor a necessary condition of the existence of the root.



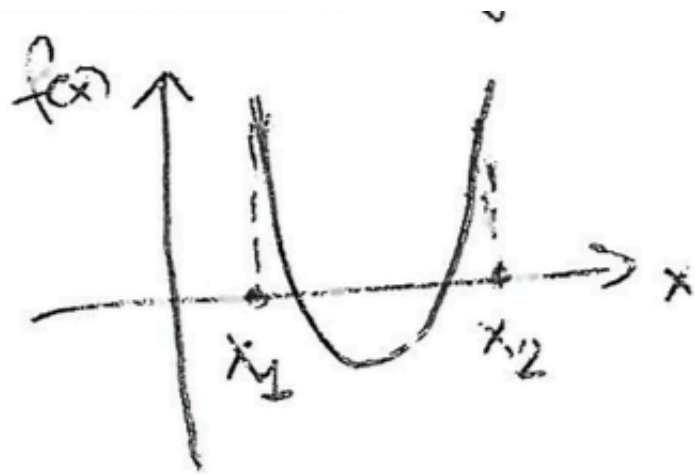
Bracketing:

Let be $f = f(x)$, $x \in \mathbb{R}$

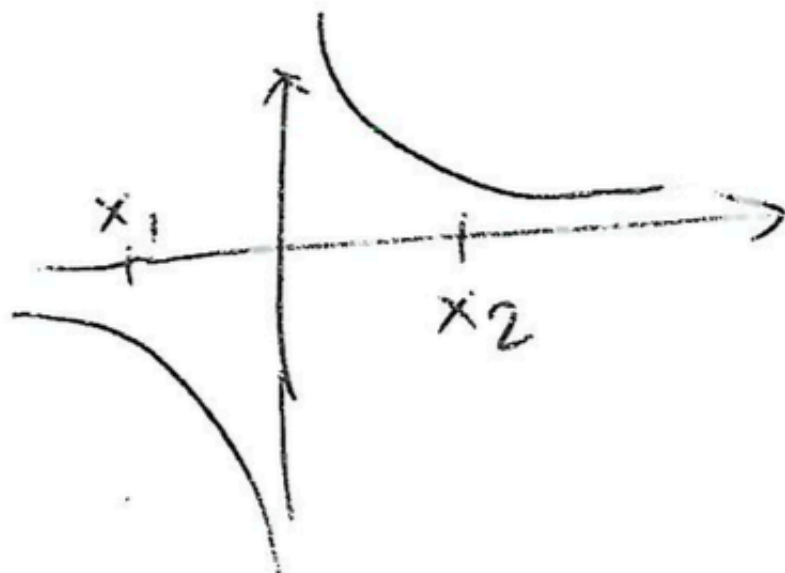
then x_0 is bracketed in $[x_1, x_2]$ if $f(x_1)f(x_2) < 0$



Note: $f(x_1)f(x_2) < 0$ is not a sufficient nor a necessary condition of the existence of the root.



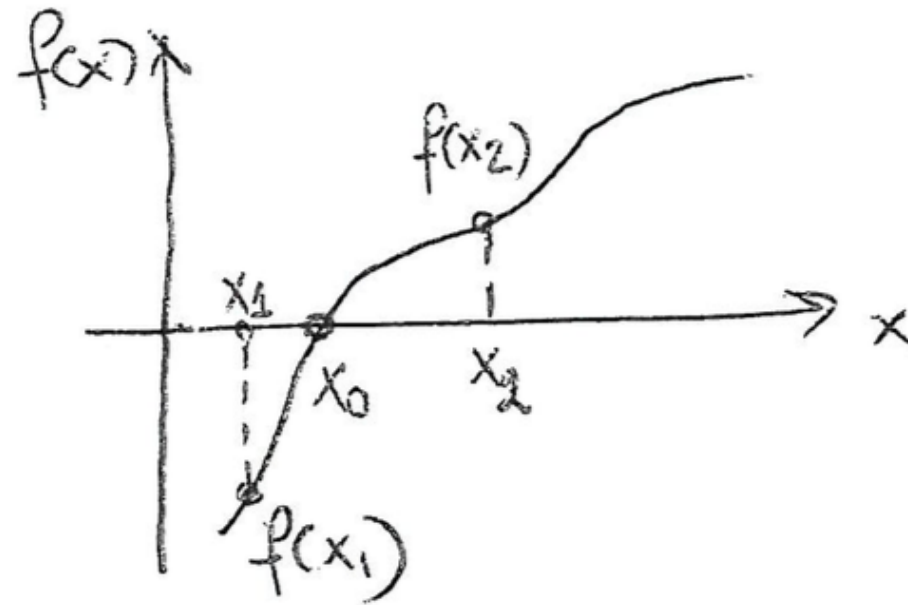
$f(x_1)f(x_2) > 0$ even though I have two roots
 \Rightarrow not a sufficient condition



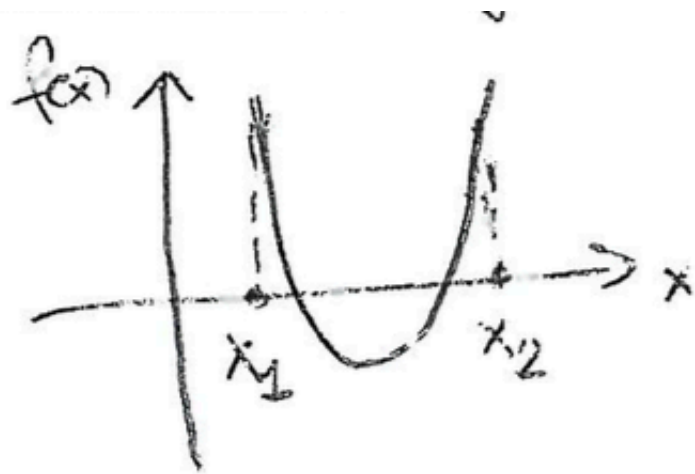
Bracketing:

Let be $f = f(x)$, $x \in \mathbb{R}$

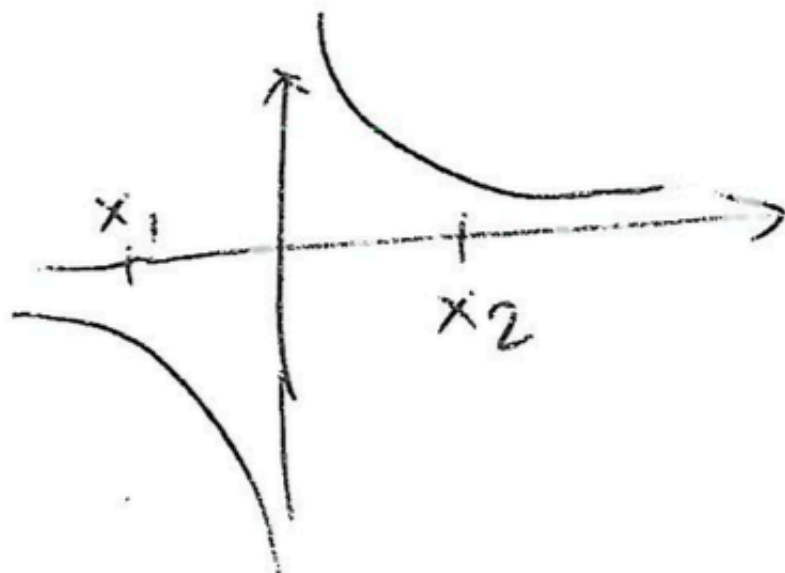
then x_0 is bracketed in $[x_1, x_2]$ if $f(x_1)f(x_2) < 0$



Note: $f(x_1)f(x_2) < 0$ is not a sufficient nor a necessary condition of the existence of the root.



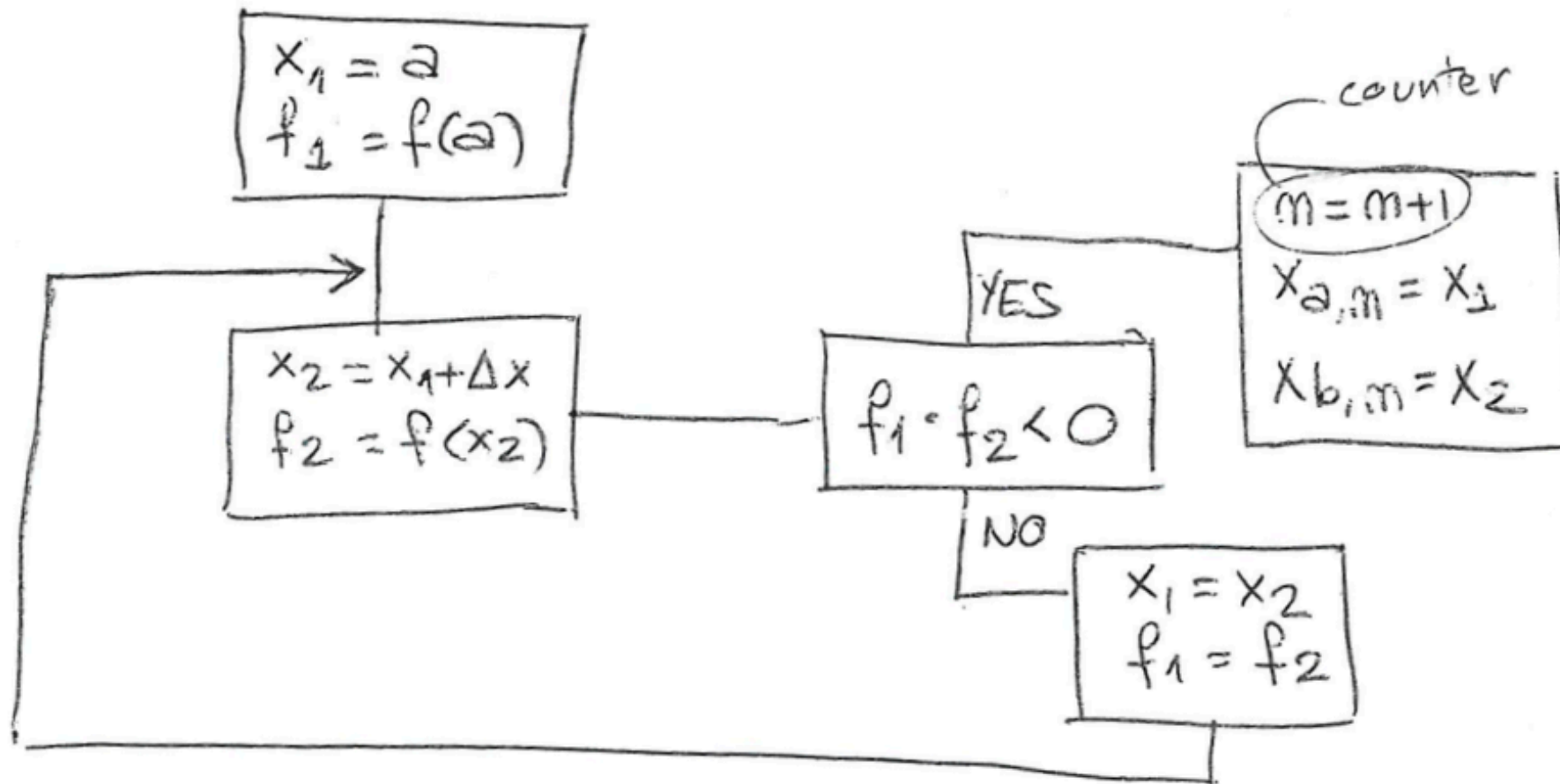
$f(x_1)f(x_2) > 0$ even though I have two roots
 \Rightarrow not a sufficient condition



$f(x_1)f(x_2) < 0$ even though there is no root
 \Rightarrow not a necessary condition

Algorithm for bracketing:

$I = [a, b]$ which I want to divide in N smallest intervals
 $\Delta x' = \frac{b-a}{N}$



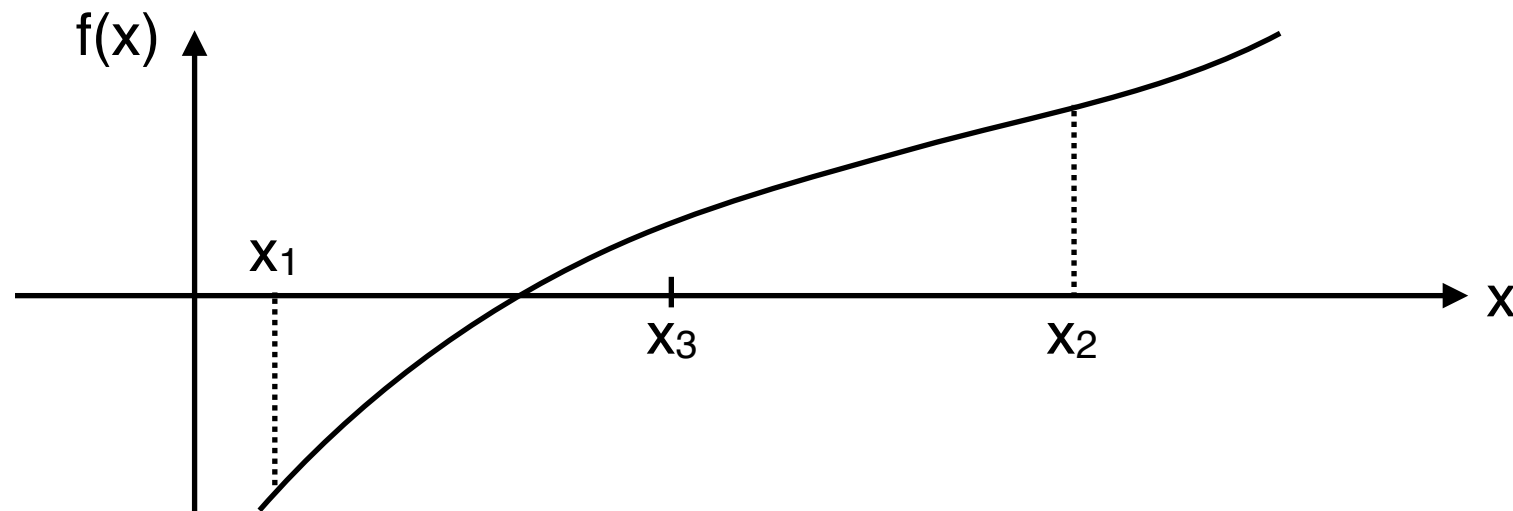
i.e., I found a root
and I store it

Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).

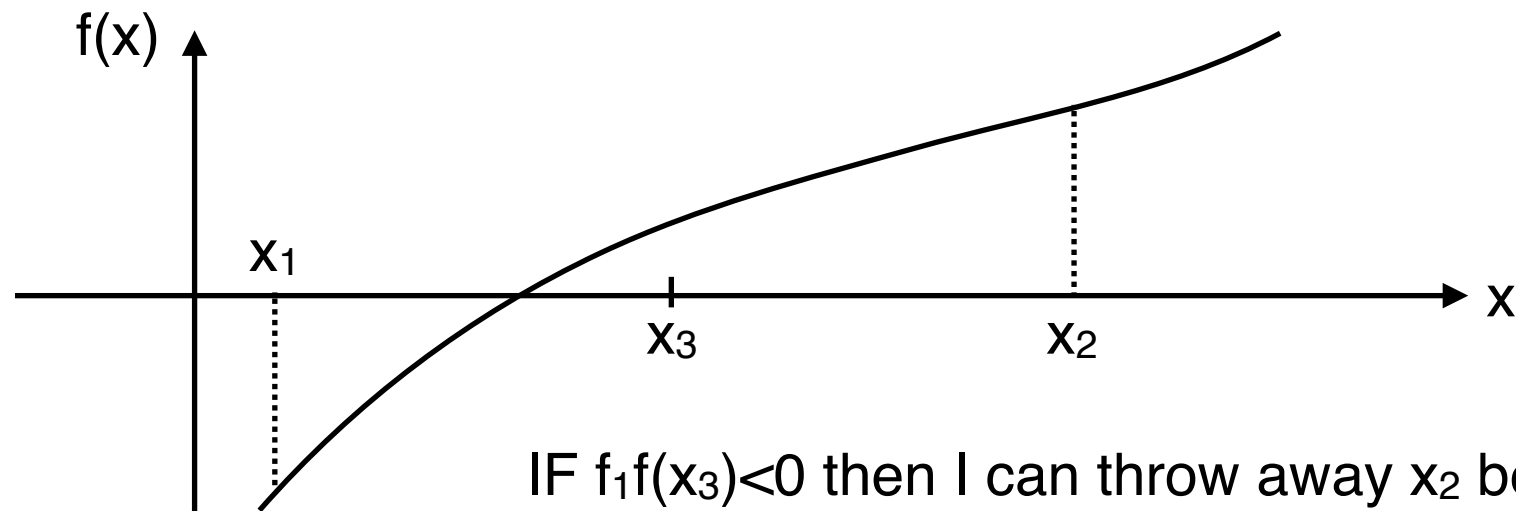
Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).



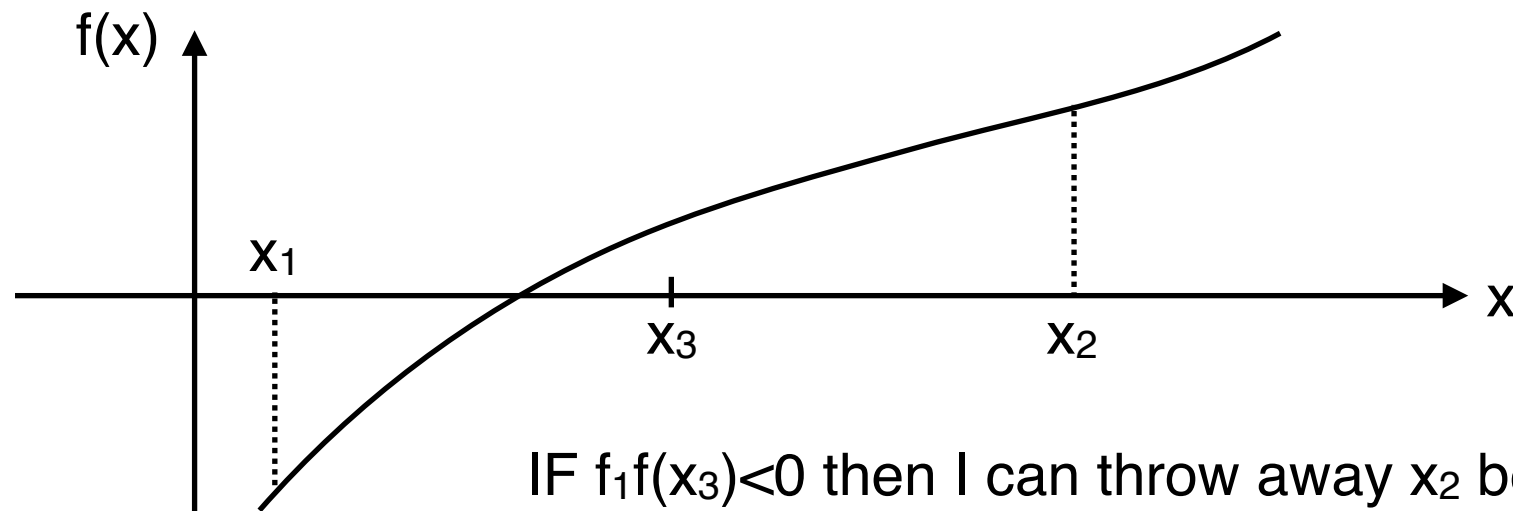
Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).



Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).

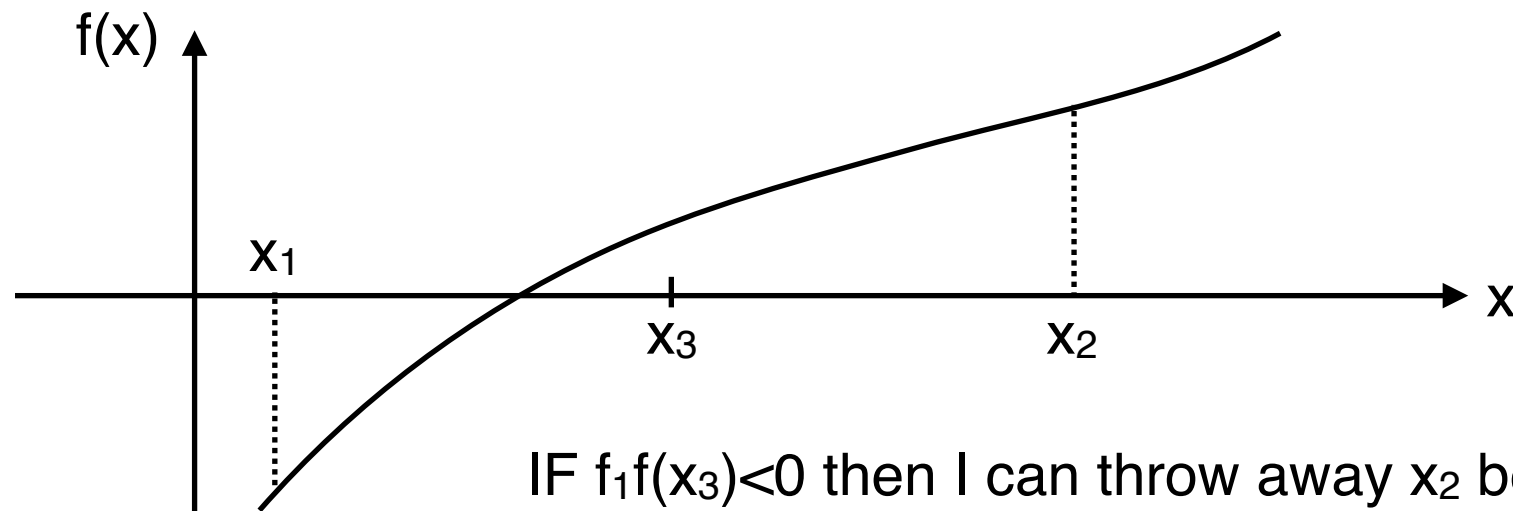


Pros: it cannot fail

Cons: not efficient (i.e., not quick)

Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).



Pros: it cannot fail

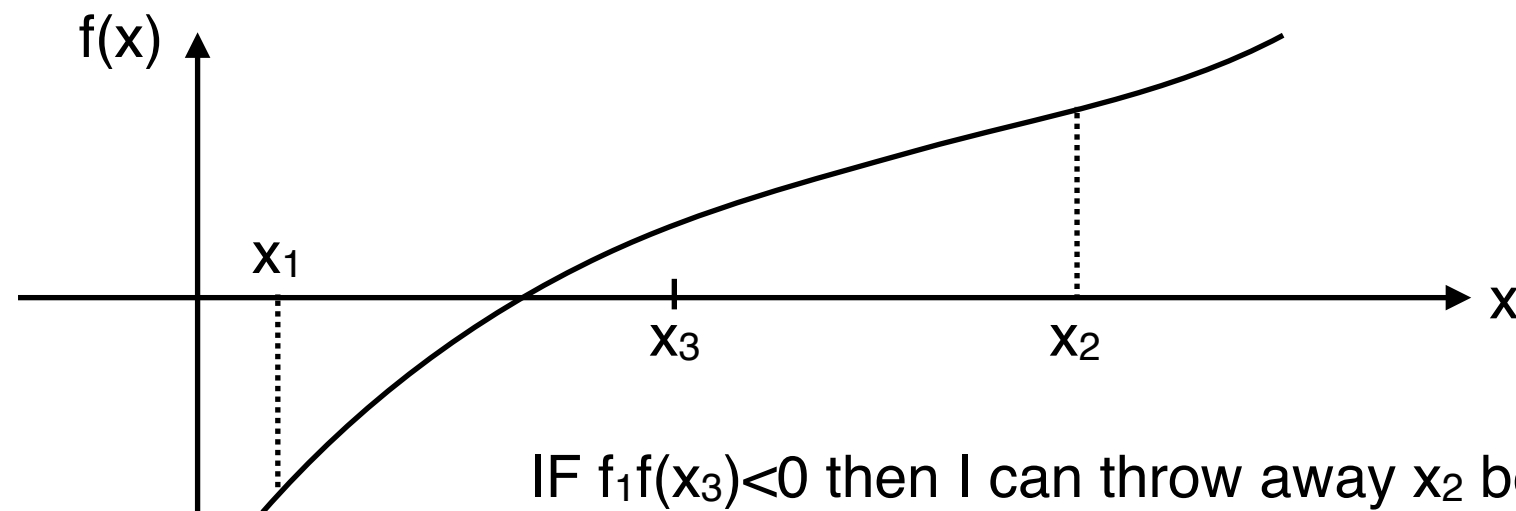
Cons: not efficient (i.e., not quick)

Set E_n the deviation from x_0 at each iteration:

$$E_n = \frac{1}{2} E_{n-1} = 2^{-n} E_0$$

Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).



Pros: it cannot fail

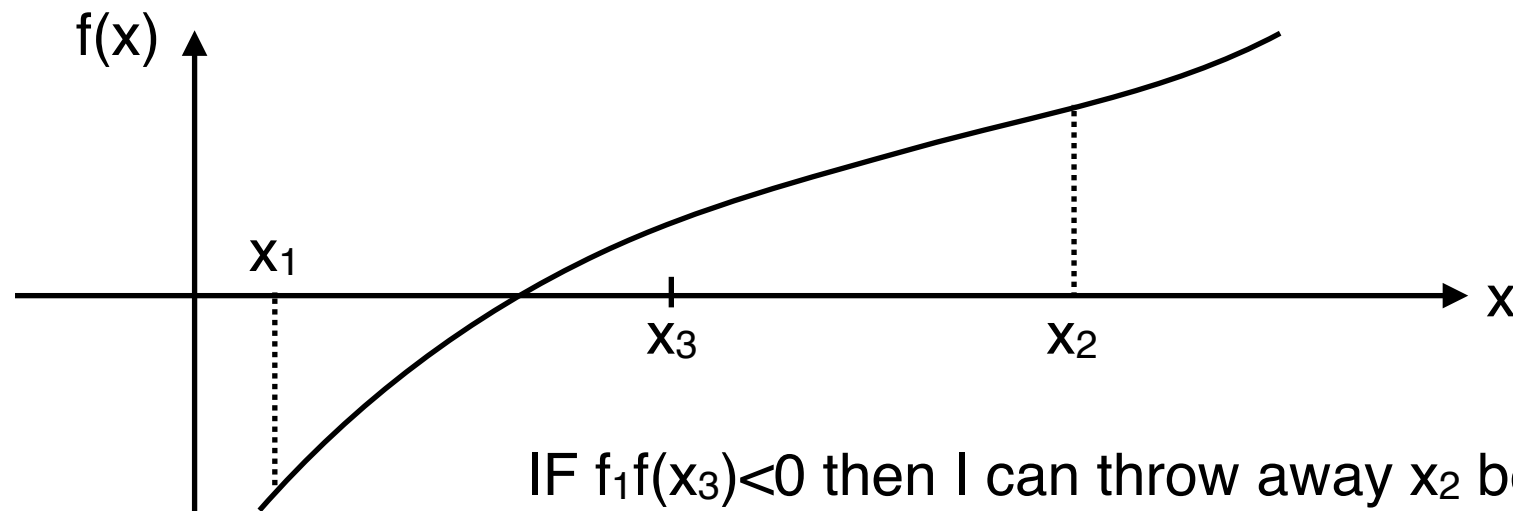
Cons: not efficient (i.e., not quick)

Set E_n the deviation from x_0 at each iteration:
$$E_n = \frac{1}{2} E_{n-1} = 2^{-n} E_0$$

If you fix E_0 (initial error) and E_F is the final error,
the number of steps N necessary to achieve $E_F \Rightarrow N = \log_2 \frac{E_F}{E_0}$

Bisection:

Given an interval $[x_1, x_2]$ bracketing x_0 , we evaluate the function of the mid-point and we use it to replace one of the previous limits (in particular the one which still satisfies $f_1 f(x_n) < 0$).



Pros: it cannot fail

Cons: not efficient (i.e., not quick)

Set E_n the deviation from x_0 at each iteration:
$$E_n = \frac{1}{2} E_{n-1} = 2^{-n} E_0$$

If you fix E_0 (initial error) and E_F is the final error, the number of steps N necessary to achieve $E_F \Rightarrow$
$$N = \log_2 \frac{E_F}{E_0}$$

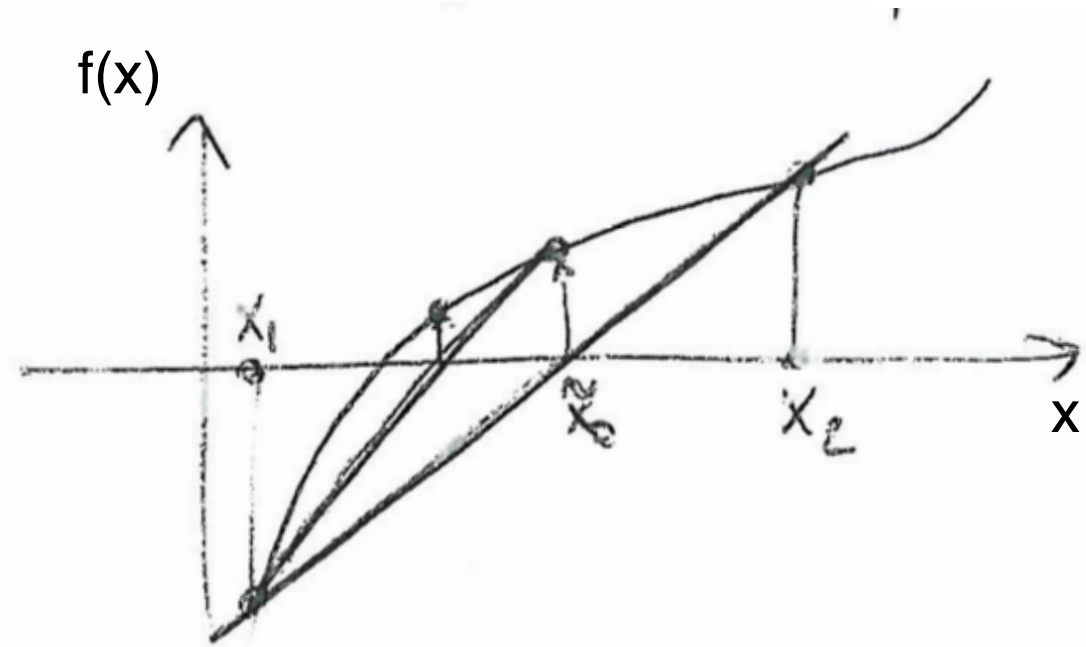
Note: $E_F=0$ is not possible, since N would be infinity

Secant method:

Let x_0 be a root of $f(x)$ and x_0 in $[a,b]$

a) approximate the function as a line through $f(x_1)$ and $f(x_2)$:

$$\frac{f(\tilde{x}_0) - f(x_1)}{\tilde{x}_0 - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$



b) $f(\tilde{x}_0) \approx 0$ then $\tilde{x}_0 = x_1 + \boxed{(x_2 - x_1) \frac{f(x_1)}{f(x_1) - f(x_2)}}$

new best guess

previous best guess

correction

c) iterate by using \tilde{x}_0 instead of x_1 or x_2

NOTE: this method is more efficient than bisection, i.e., after many iterations, you are getting faster and faster to the solution with the secant method.

CAVEAT: the secant method does not bother checking whether the root is always bracketed, i.e., **it can fail**

To avoid failure, one can use the **false positive method**.

False positive method:

basically, this is the secant method with a check on the cross product

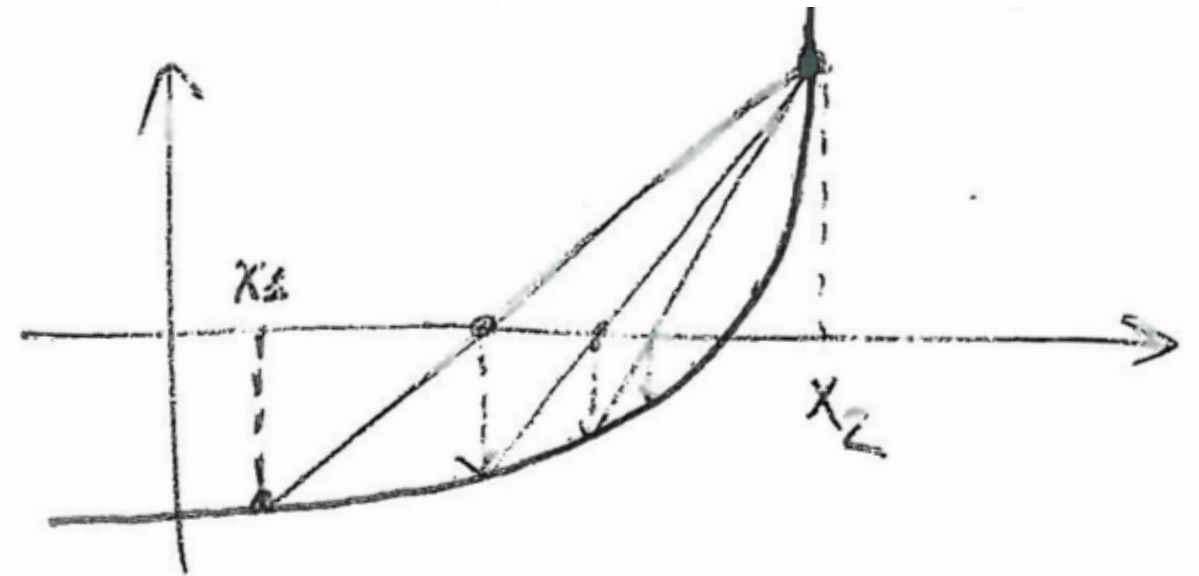
a) x_0 in $[x_1, x_2]$, $f(x_1) < 0$ and $f(x_2) > 0$

b) evaluate $\tilde{x}_0 = x_1 + (x_2 - x_1) \frac{f(x_1)}{f(x_1) - f(x_2)}$

c) check: if $f(\tilde{x}_0)f(x_2) < 0$, then retain x_2 ; if $f(\tilde{x}_0)f(x_2) > 0$, then retain x_1

NOTE: the false positive method is a bit slower (because of the check), but more secure.

NOTE: the secant / false positive methods can fail miserably; in this sample, they would be very slow.



Brent's method: standard (fast and safe)

The super-linear (fast) convergence rate is achieved by using an inverse quadratic interpolation among 3 points and estimating the root as the place where the interpolating function vanishes.

This method provides the certainty of the bisection method with the speed of the secant method.

a) $(x_1, f(x_1)), (x_2, f(x_2)), (x_3, f(x_3))$

b) $\tilde{x}_0 = x_2 + \frac{P}{Q}$ with $P = S[T(R-T)(x_3-x_1) - (1-R)(x_2-x_1)]$
 $Q = (T-1)(R-1)(S-1)$
 $R = f(x_2)/f(x_3), S = f(x_2)/f(x_1), T = f(x_1)/f(x_3)$

c) IF $|f(\tilde{x}_0)| - E_F < 0$ then you are getting close to the tolerance (good enough, happy);
otherwise, iterate

Newton-Raphson method:

Let f be a function with known derivative

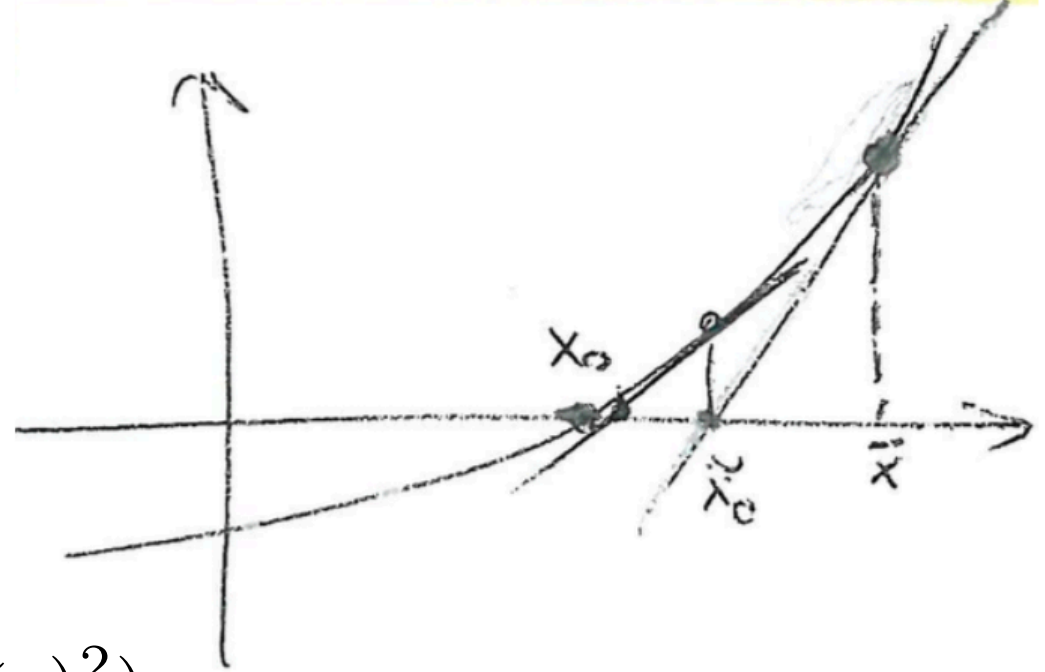
Consider the point \bar{x} and calculate $f'(\bar{x})$

Taylor expand around \bar{x} :

$$f(\tilde{x}_0) = f(\bar{x}) - f'(\bar{x})(\bar{x} - \tilde{x}_0) + o((\bar{x} - \tilde{x}_0)^2)$$

$$f(\tilde{x}_0) = 0 \longrightarrow \tilde{x}_0 = \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}$$

IF $|f(\tilde{x}_0)| < E_F$ then stop, otherwise
set $\bar{x} = \tilde{x}_0$ and reiterate.



Newton-Raphson method:

Let f be a function with known derivative

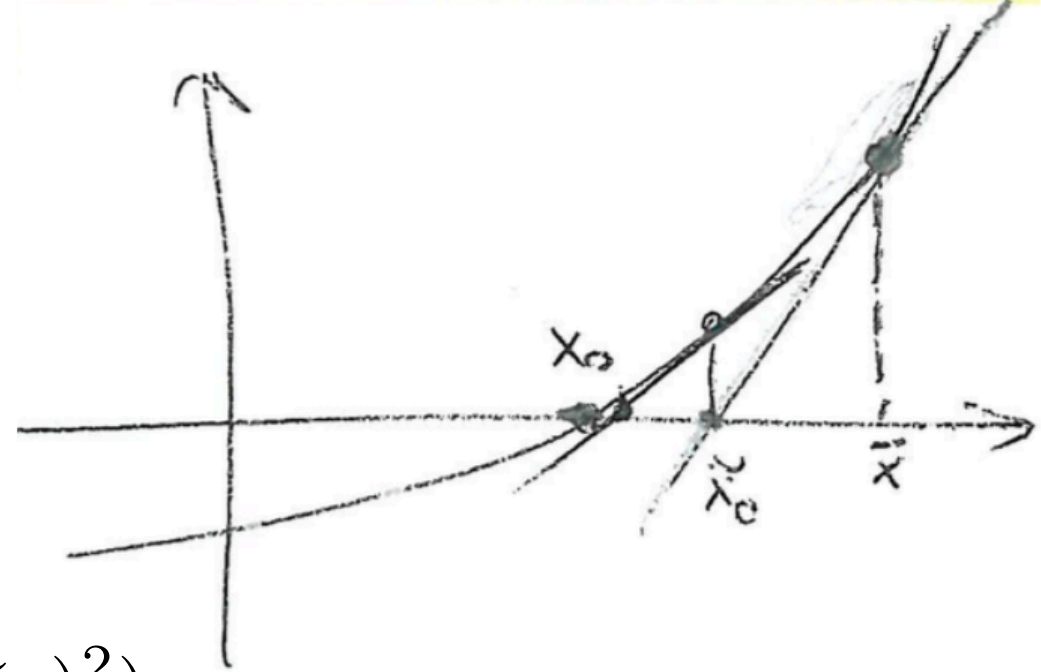
Consider the point \bar{x} and calculate $f'(\bar{x})$

Taylor expand around \bar{x} :

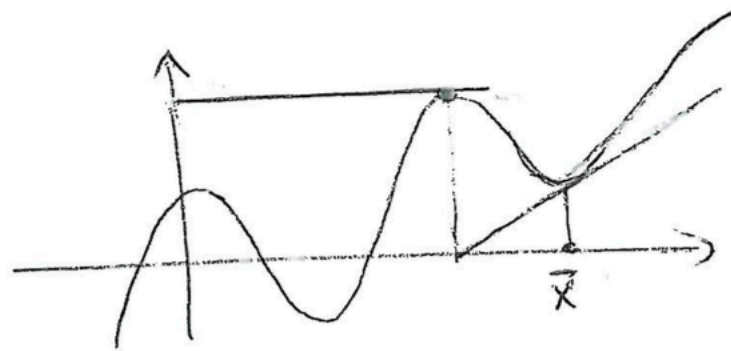
$$f(\tilde{x}_0) = f(\bar{x}) - f'(\bar{x})(\bar{x} - \tilde{x}_0) + o((\bar{x} - \tilde{x}_0)^2)$$

$$f(\tilde{x}_0) = 0 \longrightarrow \tilde{x}_0 = \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}$$

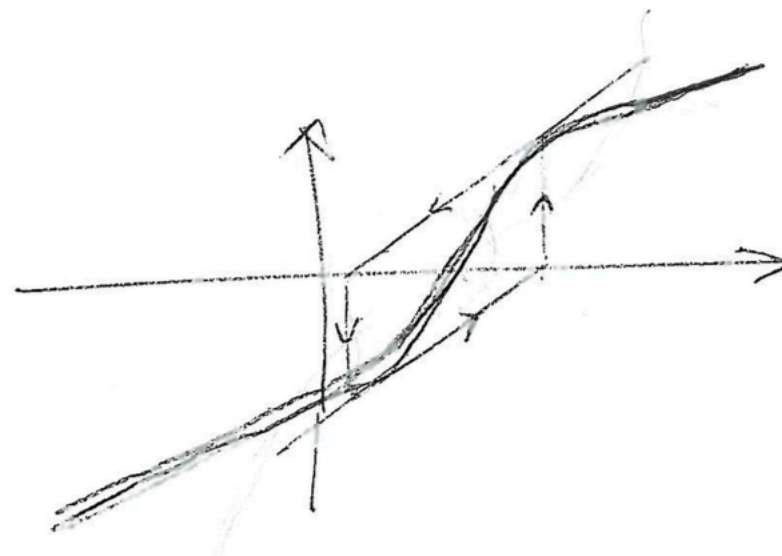
IF $|f(\tilde{x}_0)| < E_F$ then stop, otherwise
set $\bar{x} = \tilde{x}_0$ and reiterate.



This method is
very efficient, but it
can fail miserably:



It fails miserably because the derivative is
parallel to x (no intersection with the x axis)



i.e., loop or very slowly

Newton-Raphson method:

Let f be a function with known derivative

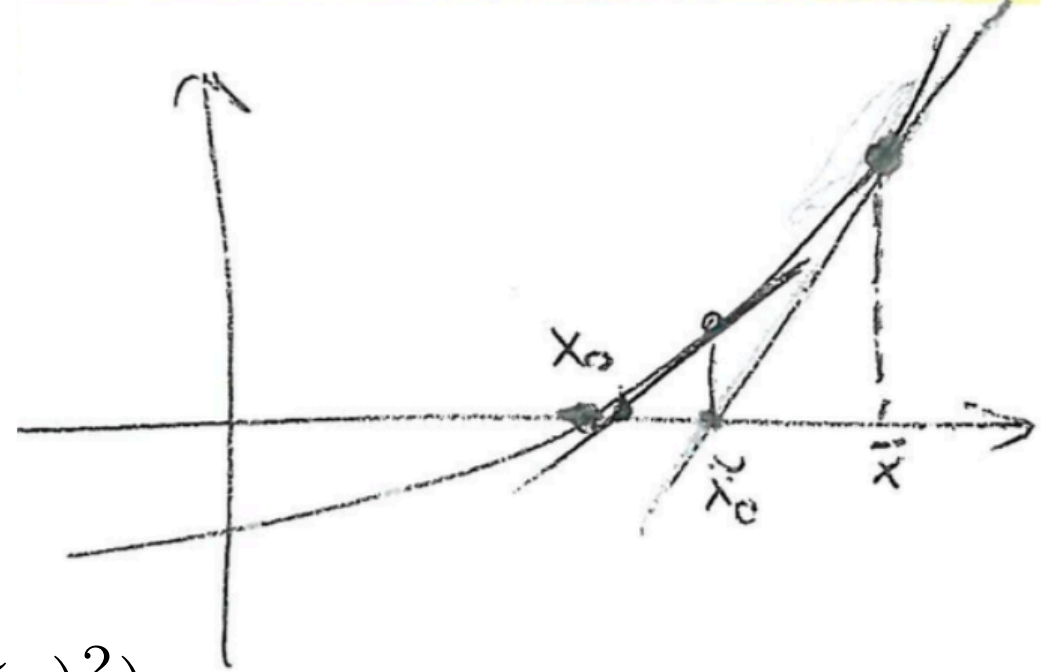
Consider the point \bar{x} and calculate $f'(\bar{x})$

Taylor expand around \bar{x} :

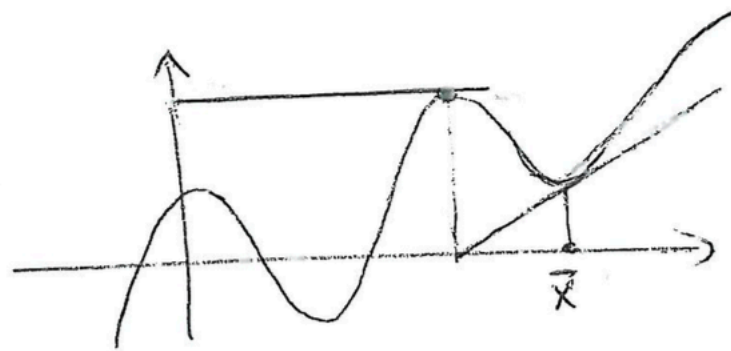
$$f(\tilde{x}_0) = f(\bar{x}) - f'(\bar{x})(\bar{x} - \tilde{x}_0) + o((\bar{x} - \tilde{x}_0)^2)$$

$$f(\tilde{x}_0) = 0 \longrightarrow \tilde{x}_0 = \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}$$

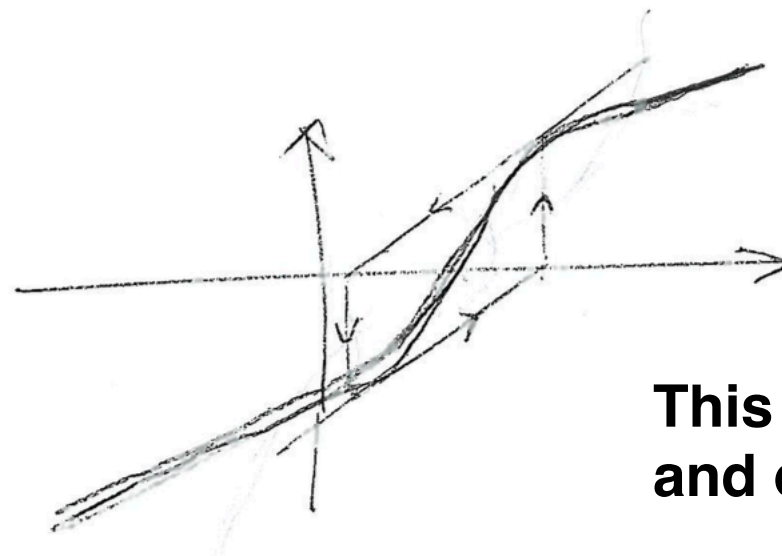
IF $|f(\tilde{x}_0)| < E_F$ then stop, otherwise
set $\bar{x} = \tilde{x}_0$ and reiterate.



This method is
very efficient, but it
can fail miserably:



It fails miserably because the derivative is
parallel to x (no intersection with the x axis)



i.e., loop or very slowly

**This method should be used with care,
and only if you know well your function**

SUMMARY:

Use Brent's method as standard (fast and safe)

Use Newton-Raphson method only for certain functions (e.g., parabola)

ALWAYS CHECK THE CODE with a test function:

$y = x^2 - x = x(x-1)$, with roots $x_{0,1}=0$ and $x_{0,2}=1$