

# Metodi Computazionali della Fisica

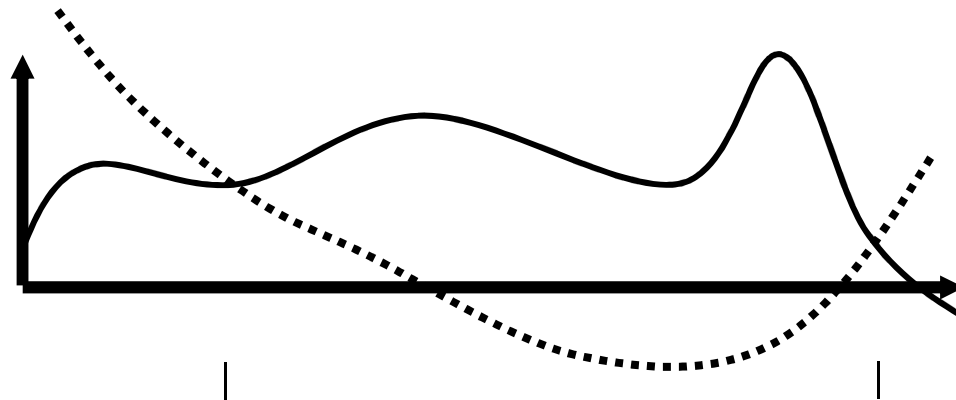
## **Radici di funzione**

# Motivazioni

La soluzione di molti problemi fisici può essere in parte ricondotta alla risoluzione di equazioni del tipo

$$f(x,y,z,\dots) = 0$$

$$f(x,y,z,\dots) = g(s,q,\dots)$$



cioè alla ricerca di **zeri di funzione**.

# Radice di una funzione

## Def

$r$  si dice *radice* della funzione  $f$  se  
$$f(r) = 0.$$

## Esempio:

La funzione  $f(x) = x^2 - 2x - 3$

ha due radici reali in  $r = -1$  e  $r = 3$ .

$$f(-1) = 1 + 2 - 3 = 0, \quad f(3) = 9 - 6 - 3 = 0$$

Note le sue radici,  $f$  può essere scritta in

**forma fattorizzata :**

$$f(x) = x^2 - 2x - 3 = (x + 1)(x - 3)$$

# Forma fattorizzata delle funzioni

*La forma fattorizzata non si limita ai polinomi*

Consideriamo la funzione:

$$f(x) = x \sin x - \sin x.$$

Avendo una radice in  $x = 1$  si potrà scrivere:

$$f(x) = (x - 1) \sin x$$

Analogamente, la funzione:

$$f(x) = \sin \pi x \quad \text{di radici } x=0,1,2,3,\dots$$

si scriverà

$$f(x) = x (x - 1) (x - 2) \dots$$

# Applicazione: Radici di potenze

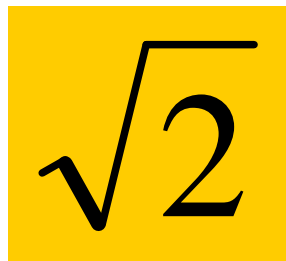
**Problema da risolvere:** Trovare  $x$ , tale che

$$x^p = c, \Rightarrow x^p - c = 0$$

In particolare, per  $p=2$ , si ha

$$x^2 - 2 = 0$$

ed il calcolo dello zero equivale al calcolo del  
numero irrazionale



# Algoritmi ricerca radici

- Tecniche “chiuse” o di confinamento
  - Bisezione
  - .....
- Tecniche “aperte”
  - Metodo di Newton
  - Metodo delle secanti

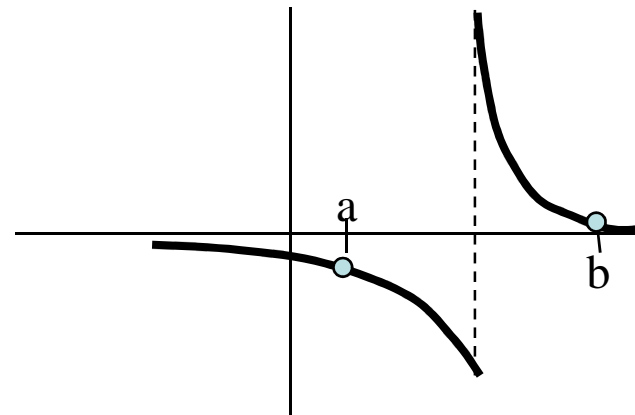
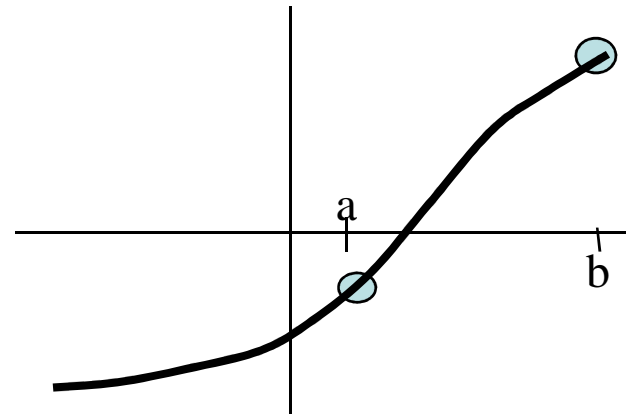
# Confinamento (bracketing)

*Def:*

Si dice che una radice  $r$  della funzione  $f(x)$  è *confinata* (**bracketed**) nell'intervallo  $[a,b]$  se  $f(a)$  e  $f(b)$  hanno segno opposto.

**N.B.** Se la funzione è continua almeno una radice deve trovarsi all'interno dell'intervallo (*teorema del valore intermedio*).

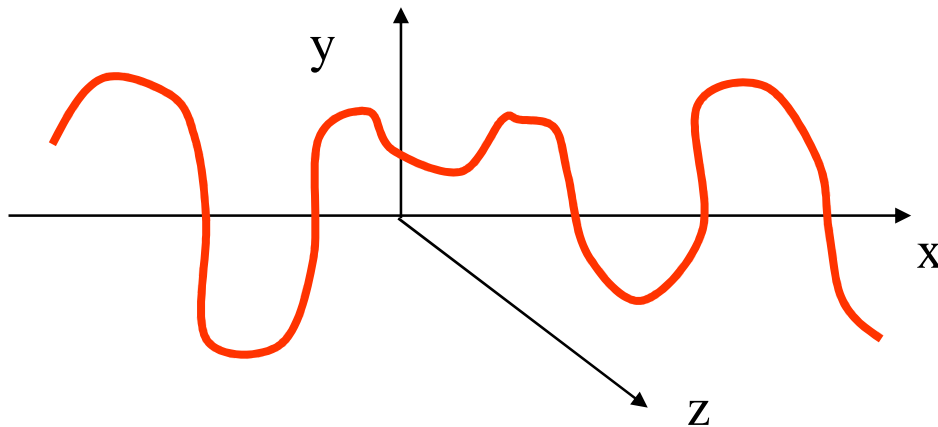
Se la funzione è discontinua, ma limitata, invece di una radice ci può essere una discontinuità finita che attraversa lo zero.



# Bracketing della radice

Già nel caso di un sistema di due equazioni, il metodo di bracketing di una radice non è possibile.

Esempio: il sistema  $y(x) = 0,$   
 $z(x) = 0$  definisce una curva del tipo;



e non è possibile delimitare una regione  $[x_1, x_2]$  in cui poter dire che esiste una radice del problema.



# Metodo di bisezione

*Si basa sul fatto che la funzione cambia segno ogni volta che passa da una radice.*

- Sia  $[a,b]$  tale che  $f(a) \cdot f(b) < 0$
- Una volta delimitata una radice da un intervallo  $[a,b]$  di lunghezza  $e_0 = b - a$ , si valuta la  $f$  nel punto medio

$$c = \frac{a+b}{2}$$

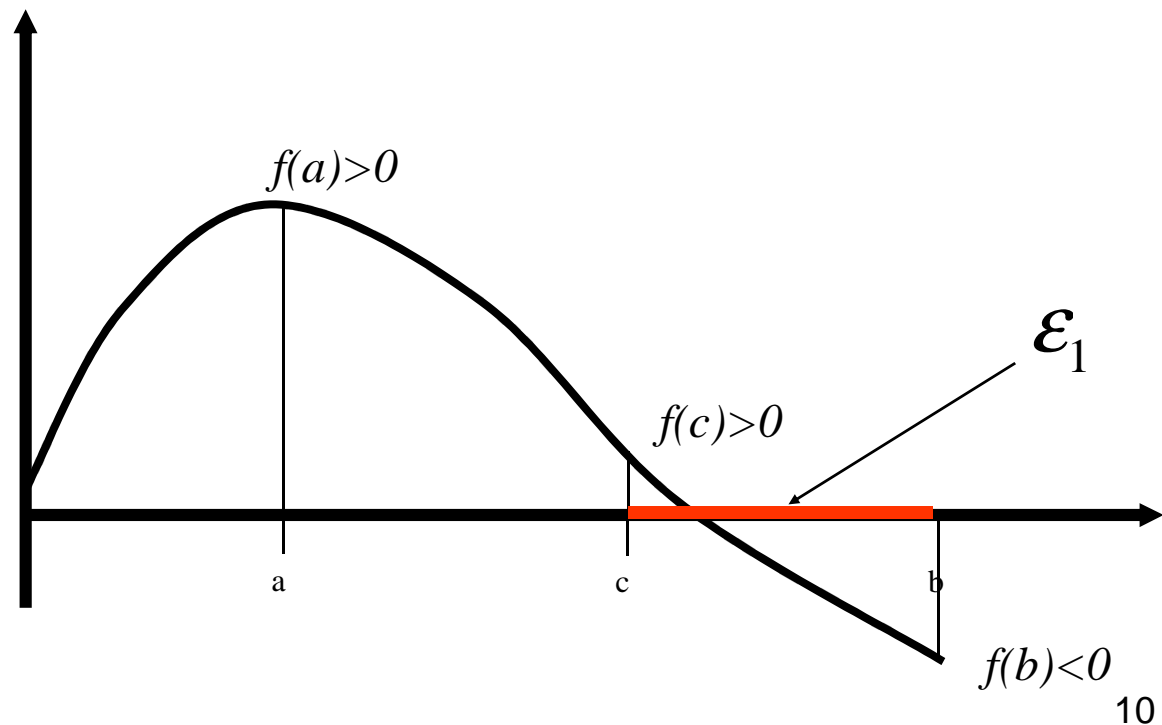
e si sceglie uno dei due intervalli  $[a,c]$  o  $[c,b]$  a seconda del segno di  $f(c)$  rapportato ai segni di  $f(a)$  e  $f(b)$

- Si ripete il punto precedente nel nuovo intervallo di lunghezza

$$\varepsilon_1 = \frac{\varepsilon_0}{2}$$

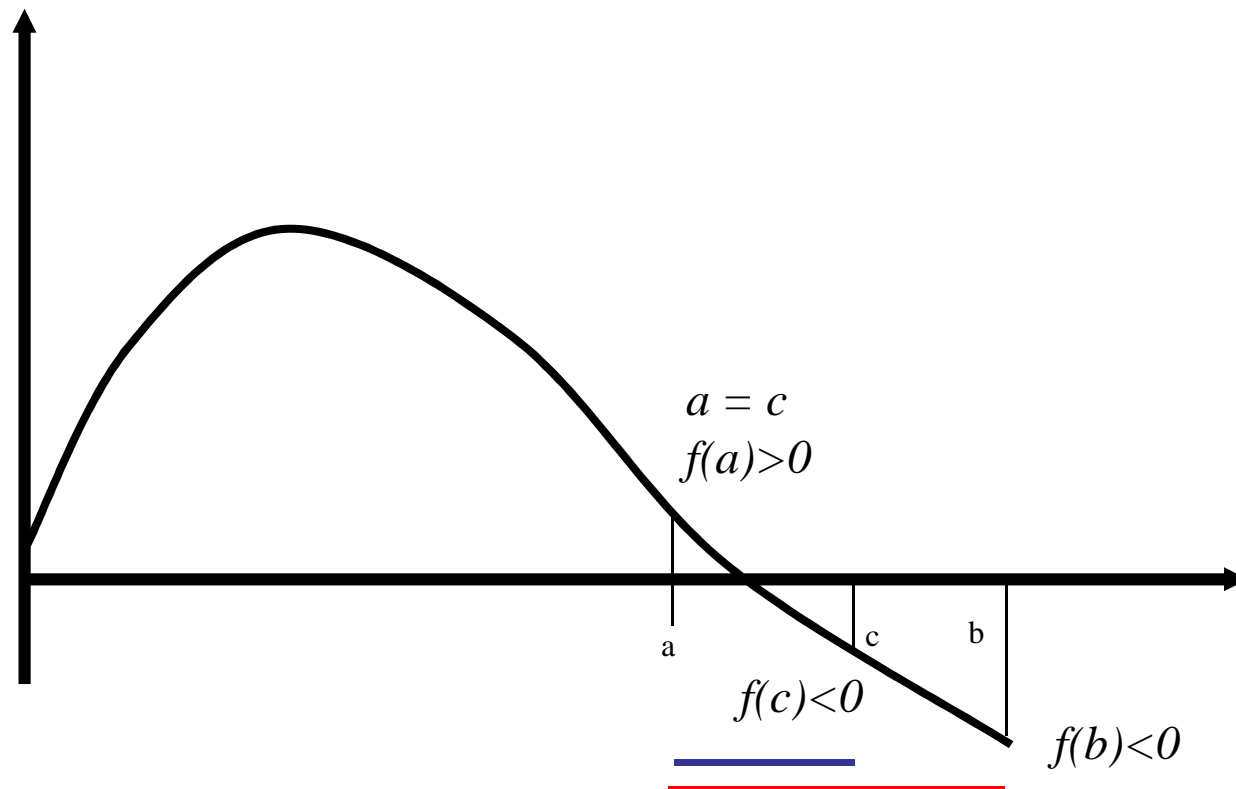
# Metodo di bisezione

- 1)  $c=(a+b)/2$ ;
- 2) Se  $f(c)*f(a)<0 \rightarrow [a,c]$  nuovo intervallo da considerare  
altrimenti  $\rightarrow [c,b]$  nuovo intervallo



# Metodo di bisezione

E' garantito convergere ad una radice se ne esiste una all'interno dell'intervallo di partenza  $[a,b]$



# Metodo di bisezione

## Algoritmo:

```
Input: float a, b    // tali che  $f(a)*f(b)<0$ 
Input: precisione, err

c ← (a+b)/2.0; // punto medio
While( |f(c)| > err ) {
    if( f(a)*f(c) < 0 ) // radice nella meta' sinistra
        b ← c;
    else                // radice nella meta' destra
        a ← c;
    c ← (a+b)/2.0; // Nuovo punto medio
}
return c;
```

## *bisection.cpp*

```
/* Bisection method */
#include <cmath>
#include <iostream>
#define MAX_ITER 40
double fof(double x); → Dichiarazione della funzione
int main(){
    using namespace std;
    double a,b,mid, f_a,f_b,f_mid;
    double error_bound,tolerance; } → Dichiarazioni delle variabili
    int iter;

    cout << "inserire l'intervallo (a,b) " << endl; } Input da schermo: estremi a e b di partenza
    cin >> a >> b;
    mid = 0.5 * (b+a);

    f_a = fof(a); } Calcolo di f(a) e f(b)
    f_b = fof(b);

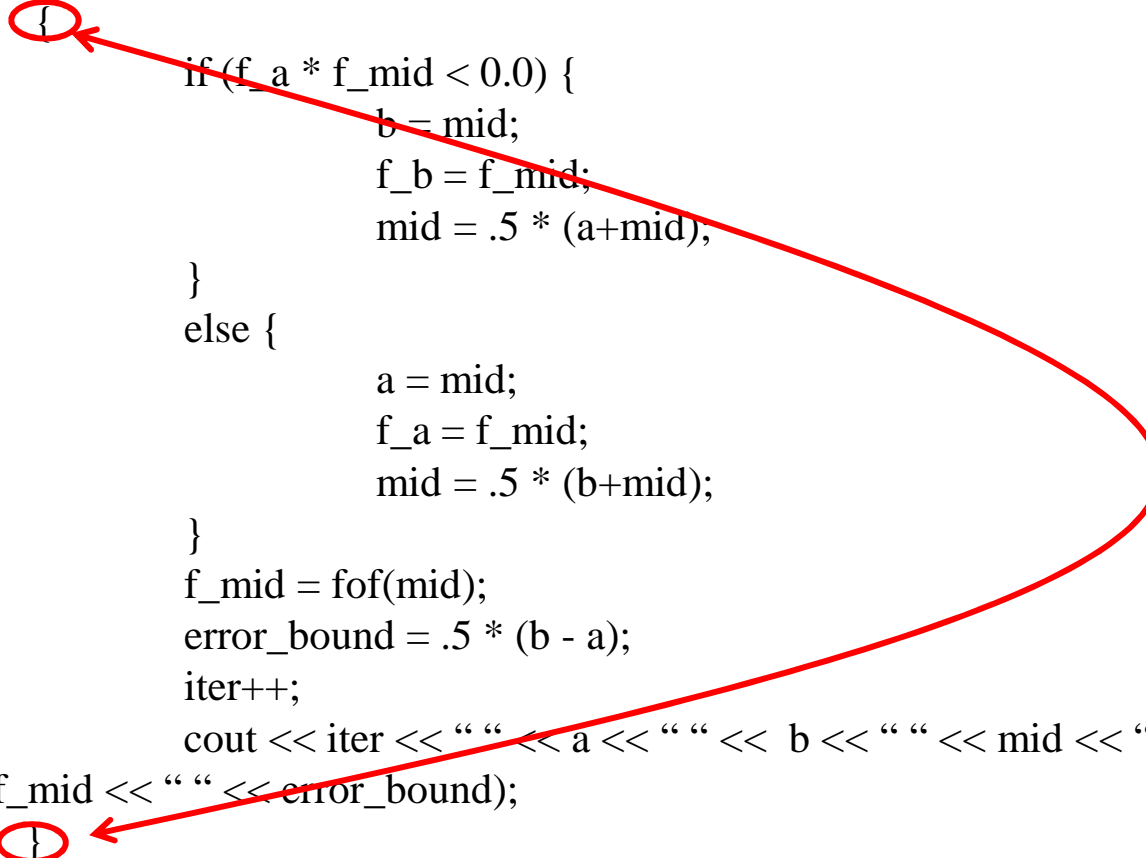
    if (f_a * f_b > 0.0) { cout << "Attenzione: l'intervallo scelto puo' non contenere una radice --
Riprova";exit(1);}
    f_mid = fof(mid);
    iter = 0; tolerance = .0001;
    error_bound = .5 * (b - a);
    cout << "i   a   b   mid   f_of_a   f_of_b   f_mid   err_bound" << endl ;
    cout << iter << " " << a << " " << b << " " << mid << " " << f_a << " " << f_b << " " << f_mid << " " <<
error_bound);
```

12/10/2009

```
while ((error_bound > tolerance) && (iter < MAX_ITER))
```

```
{  
    if (f_a * f_mid < 0.0) {  
        b = mid;  
        f_b = f_mid;  
        mid = .5 * (a+mid),  
    }  
    else {  
        a = mid;  
        f_a = f_mid;  
        mid = .5 * (b+mid);  
    }  
    f_mid = fof(mid);  
    error_bound = .5 * (b - a);  
    iter++;  
    cout << iter << " " << a << " " << b << " " << mid << " " << f_a << " " << f_b << " " <<  
f_mid << " " << error_bound);  
}  
    cout << "La radice e' x = " << mid << " la funzione = " << f_mid << " iterazioni = " << iter << "  
error_bound = " << error_bound << endl;  
} // Fine main
```

Corpo del while



```
double fof(double x)
{
    /* Si puo' scegliere la forma funzionale che si vuole e scriverla qui */
    /*return(x - tan(x));*/
    /*return(exp(x) - x*x + 3.0*x -2.0);*/
    /*return(x*x*x - 25.0); */
    return(x*x - 5.0);
}
```

```
g++ -o bisection bisection.cpp
```

```
./bisection
```

```
enter the range: a, b
```

```
1. 5.
```

```
i a b mid f_of_a f_of_b f_mid err_bound
```

```
0 1 5 3 -4 20 4 2
```

```
1 1 3 2 -4 4 -1 1
```

```
2 2 3 2.5 -1 4 1.25 0.5
```

```
3 2 2.5 2.25 -1 1.25 0.0625 0.25
```

```
4 2 2.25 2.125 -1 0.0625 -0.484375 0.125
```

```
5 2.125 2.25 2.1875 -0.484375 0.0625 -0.214844 0.0625
```

```
6 2.1875 2.25 2.21875 -0.214844 0.0625 -0.0771484 0.03125
```

```
7 2.21875 2.25 2.23438 -0.0771484 0.0625 -0.00756836 0.015625
```

```
8 2.23438 2.25 2.24219 -0.00756836 0.0625 0.0274048 0.0078125
```

```
9 2.23438 2.24219 2.23828 -0.00756836 0.0274048 0.00990295 0.00390625
```

```
10 2.23438 2.23828 2.23633 -0.00756836 0.00990295 0.00116348 0.00195312
```

```
11 2.23438 2.23633 2.23535 -0.00756836 0.00116348 -0.00320339 0.000976562
```

```
12 2.23535 2.23633 2.23584 -0.00320339 0.00116348 -0.00102019 0.000488281
```

```
13 2.23584 2.23633 2.23608 -0.00102019 0.00116348 7.15852e-05 0.000244141
```

```
14 2.23584 2.23608 2.23596 -0.00102019 7.15852e-05 -0.000474319 0.00012207
```

```
15 2.23596 2.23608 2.23602 -0.000474319 7.15852e-05 -0.000201371 6.10352e-05
```

```
Solution is x = 2.23602 function = -0.000201371 iterations = 15 error_bound = 6.10352e-05
```

```
RISULTATO ESATTO: x = sqrt(5.0) = 2.236067977
```

```
12/10/2009
```



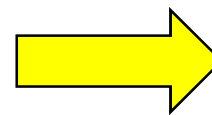
# Convergenza del Metodo di bisezione

Dopo ogni iterazione l'intervallo contenente la radice diminuisce di un fattore 2. Se dopo  $n$  iterazioni la radice si trova in un intervallo di grandezza  $\varepsilon_n$ , all'iterazione successiva sarà confinata in un intervallo di grandezza  $\varepsilon_{n+1} = \varepsilon_n / 2$  (**convergenza lineare**)

Se con  $\varepsilon_0$  indichiamo la grandezza dell'intervallo di partenza e con  $\varepsilon$  la tolleranza finale (precisione con cui si vuole localizzare la radice).

Poichè dopo  $n$  iterazioni:

$$\varepsilon_n = \frac{\varepsilon_0}{2^n}$$



$$n = \log_2 \frac{\varepsilon_0}{\varepsilon}$$

# Convergenza metodo di bisezione

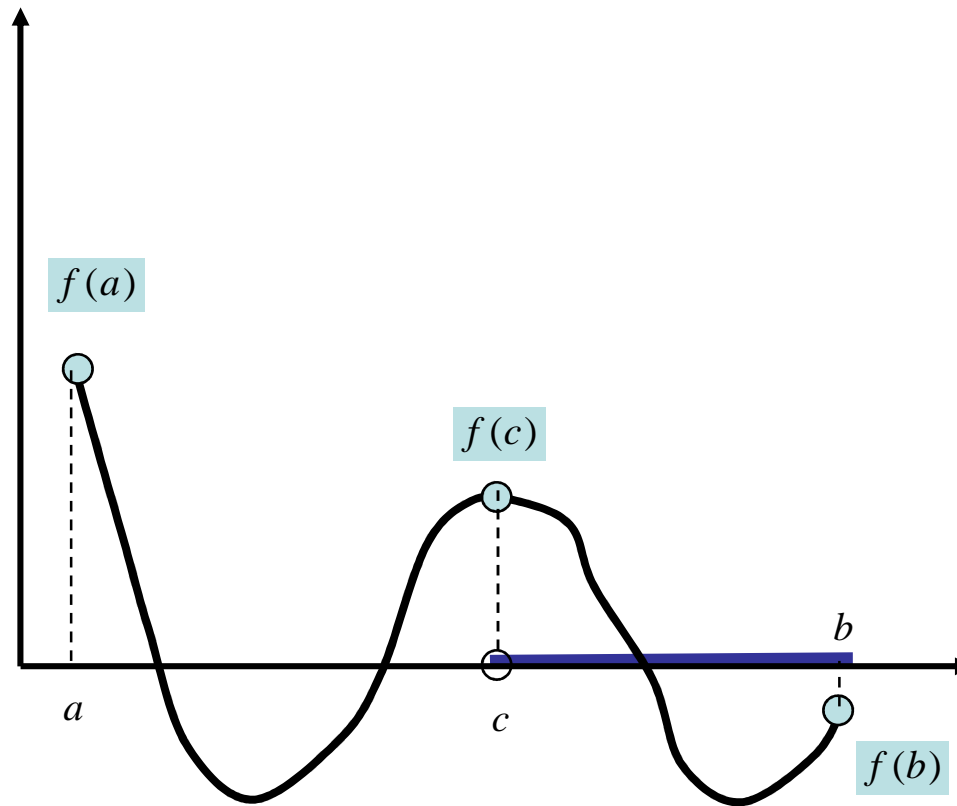
- Il metodo di bisezione converge **linearmente** alla radice.
- Se si ha bisogno di una precisione di 0.0001 e l'intervallo iniziale è  $(b-a)=1$ , allora:

$$2^{-n} < 0.0001 \Rightarrow 14 \text{ iterazioni}$$

- Metodo semplice da implementare su computer.
- Converge sempre.

# Problema

E se ci sono più radici in  $[a,b]$  ?



Il metodo ne sceglie una .....

# ***Problema:***

*Convergenza assicurata ma lenta !*

Le funzioni possono essere semplici, ma a volte si ha bisogno di valutarle tante volte.

Oppure la funzione può essere molto complicata da valutare e/o espressa in forma implicita (soluzione numerica di altri problemi)

## **Esempio:**

Supponiamo di essere interessati a quale sia la configurazione (posizione, orientazione, direzione del flusso, etc.) delle correnti d'aria che rendono la temperatura in una certa posizione dell'aula pari a 23°. E' una funzione questa ?

# Bisezione: Convergenza lenta

- Il calcolo di questa funzione può richiedere la soluzione di un'equazione del trasporto di calore accoppiata con l'equazione di Navier-Stokes → ore su un supercomputer !!!
- E' necessario a volte che la convergenza sia la più rapida possibile per evitare troppe valutazioni della funzione. In questo caso una convergenza lineare potrebbe non bastare.

# Metodi di confinamento

- Sono metodi robusti
- Convergono più lentamente di quelli aperti
- Si usano per trovare le radici in maniera approssimata
- La precisione si migliora con altri metodi
- Si basa sull'identificazione di due punti iniziali  $a, b$  tali che:
  - $f(a) f(b) < 0$
- E' garantita la loro convergenza.
- Non funzionano per sistemi di equazioni.

# Metodi non confinanti

# Metodo di Newton-Raphson

Consideriamo un punto  $x_0$ .

Se approssiamo  $f(x)$  con la tangente in  $x_0$ , allora si può cercare la radice della retta approssimante:

$$l(x) = f'(x_0)(x - x_0) + f(x_0)$$



# Newton-Raphson

Ciò porta alla seguente equazione:

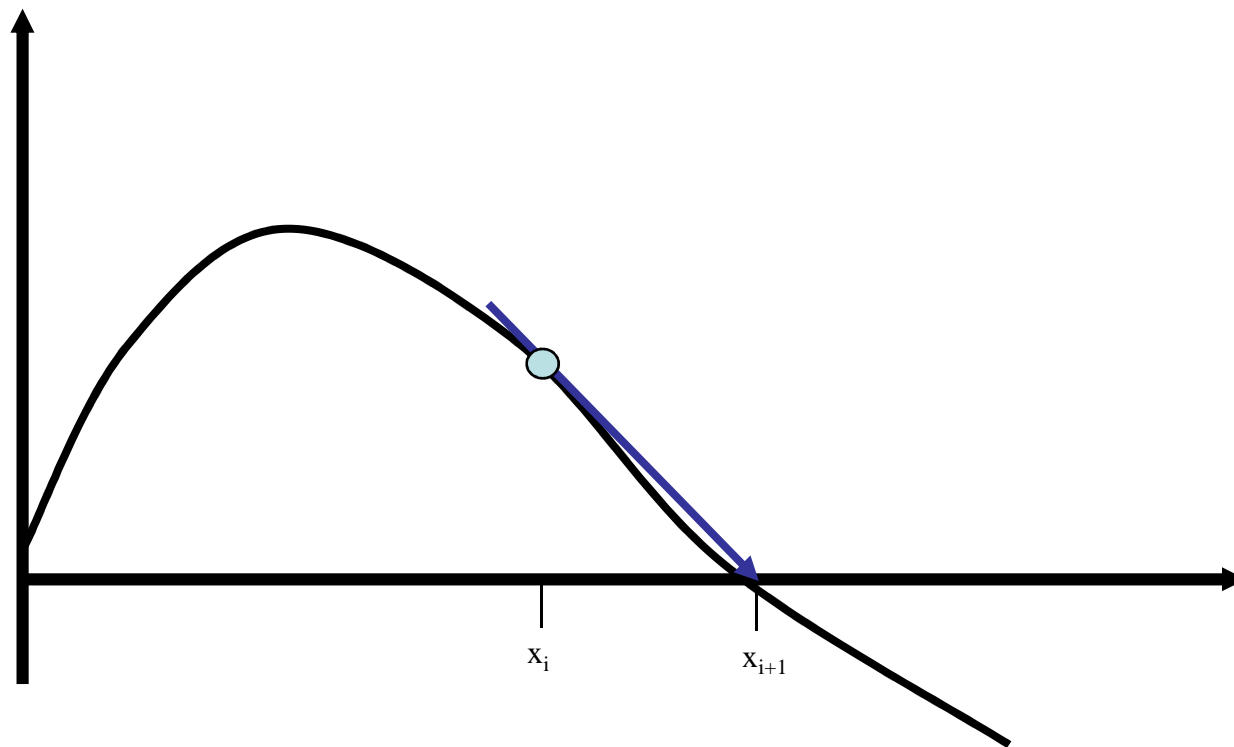
$$l(x) = 0$$
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

In generale, detto  $x_i$  l'approssimazione della radice  $r$  all'ordine  $i$ , la stima all'ordine  $i+1$  sarà :

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

# Newton-Raphson

Graficamente equivale a seguire il vettore tangente sino alla sua intersezione con l'asse x:



# Routine di Newton-Raphson in linguaggio C++

## *newton\_simple.cpp*

```
#include <iostream>
#include <cmath>
double newton(double x_0, double tol, int max_iters, int& iters_p, int& converged_p);
double f(double x); ← funzione
double f_prime(double x); ← derivata prima della funzione

int main() {
    using namespace std;
    double x_0;    /* Punto iniziale */ double x;    /* Radice approssimata */
    double tol;    /* Errore massimo */
    int max_iters; /* Numero massimo di iterazioni */
    int iters;     /* Numero di iterazioni */ int converged;

    cout << "Inserire x_0, tol, e numero massimo di iterazioni" << endl ;
    cin >> x_0 >> tol >> max_iters;

    x = newton(x_0, tol, max_iters, iters, converged); ← chiamata della routine Newton che fornisce  
Il valore approssimato x della radice

    if (converged) {
        cout << "Newton algorithm converged after " << iters << " steps." << endl;
        cout << "Radice approssimata" << x << endl;
        cout << "f(x) = " << f(x) << endl;
    } else {
        ← Check della convergenza e output su schermo
        cout << "Newton algorithm didn't converge after " << iters << " steps" << endl;
        cout << "Stima finale " << x << endl;
        cout << "f(x) = " << f(x) << endl;
    }
    return 0;
} // main
```

# Routine di Newton-Raphson in linguaggio C++

```
double newton(double x_0, double tol, int max_iters, int& iters_p, int& converged_p) {  
    double x = x_0;  
    double x_prev;  
    int iter = 0;  
  
    do {  
        iter++;  
        x_prev = x;  
        x = x_prev - f(x_prev)/f_prime(x_prev);  
    } while (fabs(x - x_prev) > tol && iter < max_iters);  
  
    if (fabs(x - x_prev) <= tol)  
        converged_p = 1;  
    else  
        converged_p = 0;  
  
    iters_p = iter;  
  
    return x;  
} // newton algorithm
```

← **Algoritmo iterativo di Newton**

} ← **Check della convergenza dell'algoritmo**

```
double f(double x) {  
    return x*x-5;  
} // f
```

} ← **Funzione**

```
double f_prime(double x) {  
    return 2*x; //the derivative  
} // f_prime
```

} ← **Derivata prima della funzione**

```
g++ -o newton_simple newton_simple.cpp
```

```
./newton_simple
```

```
Enter x_0, tol, and max_iters
```

```
1. 0.0001 100
```

```
Newton algorithm converged after 5 steps.
```

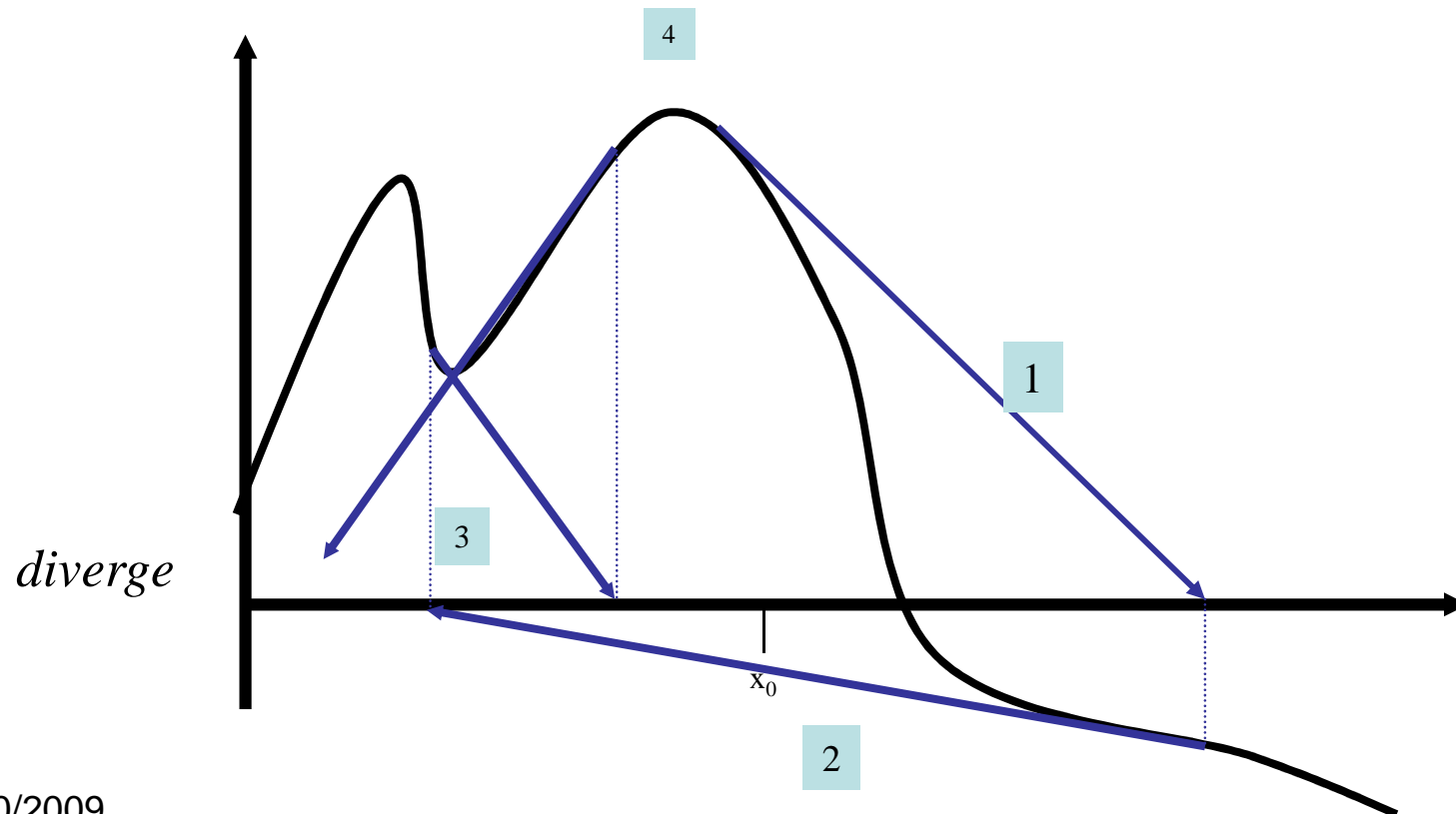
```
The approximate solution is 2.23607
```

```
f(x) = 8.42561e-13
```

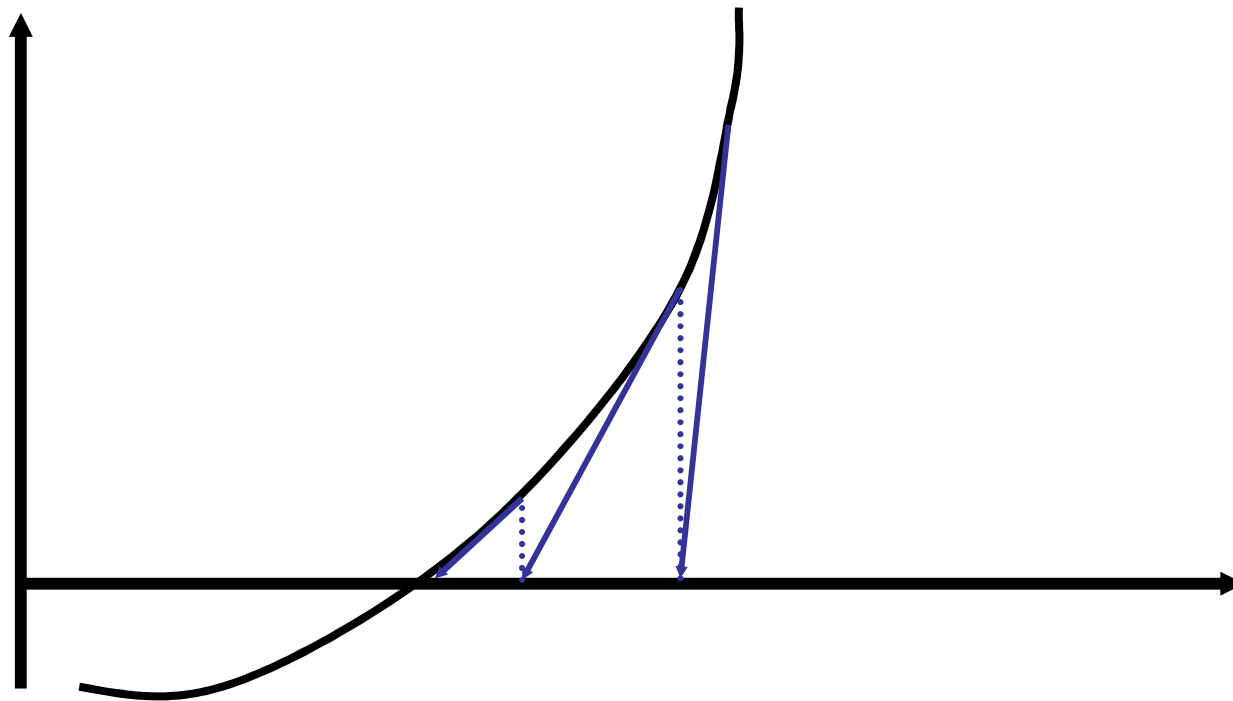
```
RISULTATO ESATTO: x = sqrt(5.0) = 2.236067977
```

# Problema con il metodo di Newton-Raphson

Se il punto iniziale  $x_0$  è lontano dalla radice  $r$  è possibile che il metodo non converga.



Occorre che il punto di partenza  $x_0$  sia *abbastanza* vicino alla radice, oppure, che la funzione si comporti *abbastanza linearmente* in quella zona.



# Applicazioni della routine in C++:

## 1) ricerca di una radice quadrata

Consideriamo le *radici* dell'equazione:

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

o, in generale dell'equazione:

$$\sqrt[p]{a} \Rightarrow x^p - a = 0, \quad p \in R$$



## p=2: ricerca di una radice quadrata

- Esempio:  $\sqrt{2} = 1.4142135623730950488016887242097$
- Sia  $x_0 = 1$  ed applichiamo il metodo di Newton:

$$f'(x) = 2x$$

$$x_{i+1} = x_i - \frac{x_i^2 - 2}{2x_i} = \frac{1}{2} \left( x_i + \frac{2}{x_i} \right)$$

$$x_0 = 1$$

$$x_1 = \frac{1}{2} \left( 1 + \frac{2}{1} \right) = \frac{3}{2} = 1.5000000000$$

$$x_2 = \frac{1}{2} \left( \frac{3}{2} + \frac{4}{3} \right) = \frac{17}{12} \approx 1.4166666667$$

# Ricerca di una radice quadrata

- Esempio:  $\sqrt{2} = 1.4142135623730950488016887242097$
- Notare la rapida convergenza

$$x_3 = \frac{1}{2} \left( \frac{17}{12} + \frac{24}{17} \right) = \frac{577}{408} \approx 1.41\cancel{4}215686$$

$$x_4 = 1.41421356237\cancel{4}6$$

$$x_5 = 1.41421356237309504880168\cancel{9}6$$

$$x_6 = 1.4142135623730950488016887242097$$

## 2) Zeri del polinomio

$$f(x) = x^3 - 2x^2 + x - 3, \quad x_0 = 4$$

n	$x_n$	$F(x_n)$
0	4	33
1	3	9
2	2.4375	2.03686523475
3	2.21303271631511	0.256363385061418
4	2.17555493872149	0.00646336148881306
5	2.17456010066645	4.47906804996122e-06
6	2.17455941029331	2.15717547991101e-12

# Convergenza del metodo

Sia  $e_n$  l'errore  
all'n-esima iterazione  
cioè:

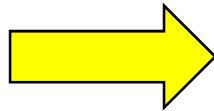
$$e_n = \bar{x} - x_n \text{ or } \bar{x} = x_n + e_n$$

$$0 \equiv f(\bar{x}) = f(x_n + e_n)$$

$$f(x_n + e_n) = f(x_n) + e_n f'(x_n) + \frac{1}{2} e_n^2 f''(\xi_n), \text{ for some } \xi_n \in (\bar{x}, x_n)$$

Espandendo secondo  
Taylor nell'intorno della  
radice :

$$\therefore f(x_n) + e_n f'(x_n) = -\frac{1}{2} e_n^2 f''(\xi_n)$$



$$\begin{aligned} e_{n+1} &= \bar{x} - x_{n+1} = \bar{x} - x_n + \frac{f(x_n)}{f'(x_n)} = e_n + \frac{f(x_n)}{f'(x_n)} \\ &= \frac{e_n f'(x_n) + f(x_n)}{f'(x_n)} \\ \therefore e_{n+1} &= -\frac{1}{2} \left( \frac{f''(\xi_n)}{f'(x_n)} \right) e_n^2 \end{aligned}$$

Convergenza quadratica !!!!

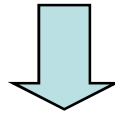
$$\text{Se } |e_n| \leq 10^{-k} \Rightarrow |e_{n+1}| \leq 10^{-2k}$$

# Metodo di Newton

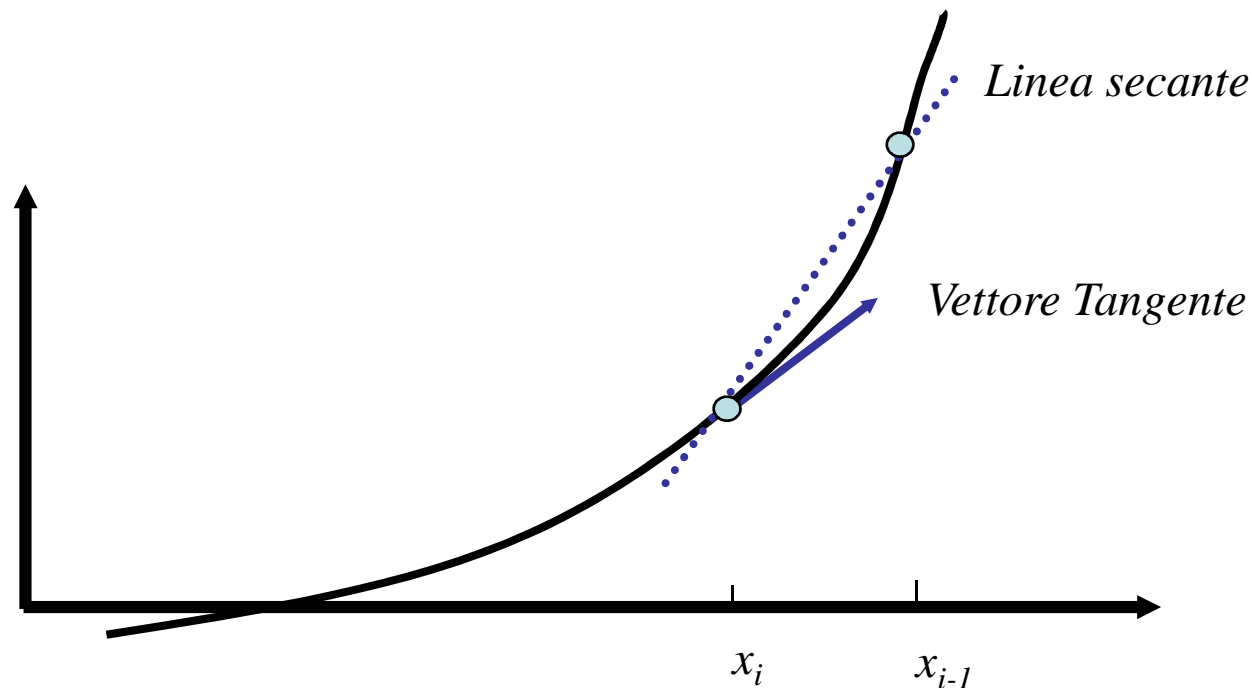
- Metodo **aperto** che richiede solo un'ipotesi iniziale sulla radice
- Non è necessario che la radice sia confinata in un intervallo (generalizzabile a sistemi di equazioni)
- Richiede la valutazione della derivata prima ad ogni iterazione.
- Poichè, se  $x_0$  è troppo lontana da  $r$  il metodo non converge, in tutte le implementazioni pratiche del metodo si pone un numero **massimo** d'iterazioni.
- L'uso più comune del *metodo di Newton* assume che il punto iniziale  $x_0$  sia abbastanza vicino alla radice e si fanno al più 3 iterazioni per una migliore precisione.
- $x_0$  può essere trovato con metodi di bracketing come la *bisezione*.

# Metodo della secante

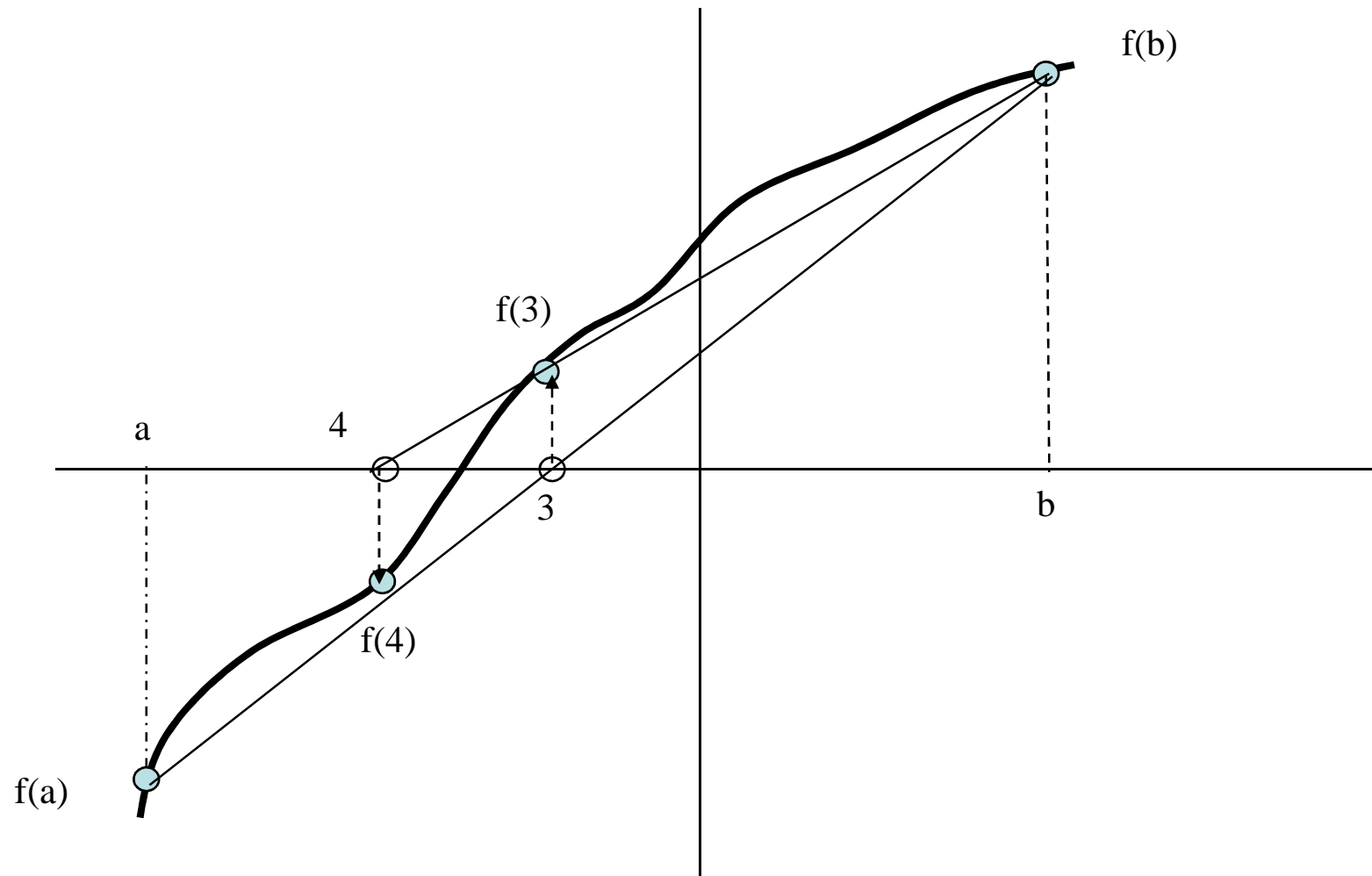
E se non conosciamo esplicitamente la derivata di  $f(x)$  in  $x_i$ ?



*la si approssima con la retta tra  $x_i$  e  $x_{i-1}$ :*



# Metodo della secante



# Metodo della secante

- Mentre converge alla radice, la linea secante convergerà alla tangente della funzione nella radice.
- Si può quindi usare la linea secante come stima della derivata e vedere dove interseca l'asse  $x$ .



# Metodo della secante

Lo si può anche ottenere dalla definizione di derivata:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$
$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Usando l'approssimazione discreta della derivata (metodo delle differenze finite) il metodo di Newton fornisce:

$$x_{k+1} = x_k - \left( \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \right) f(x_k)$$

che è appunto il metodo della **secante**.

# Convergenza del metodo della secante

Usando l'espansione in serie di Taylor, è possibile mostrare che:

$$\begin{aligned} e_{k+1} &= \bar{x} - x_{k+1} \\ &= -\frac{1}{2} \left( \frac{f''(\xi_k)}{f''(\zeta_k)} \right) e_k e_{k-1} \approx c \cdot e_k e_{k-1} \end{aligned}$$

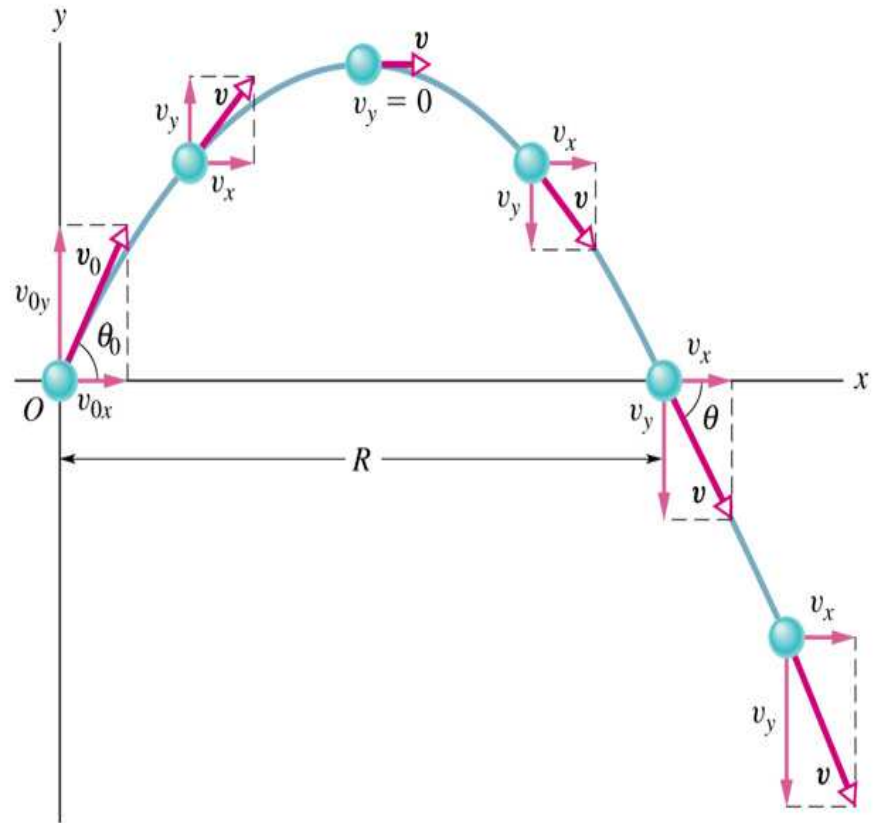
Espressione ricorsiva dell'errore.

Esplicitandola si ottiene:  $|e_{k+1}| \leq C |e_k|^\alpha$  Dove  $\alpha=1.618 \dots$   
↓  
golden ratio

Si parla di convergenza super-lineare

# Problema fisico: moto di un proiettile in presenza di attrito

Classico problema di **cinematica** (in 2D) del lancio di una massa in presenza di **gravità** e di **attrito**; il problema è concettualmente semplice ma **noioso** da risolvere *manualmente*...

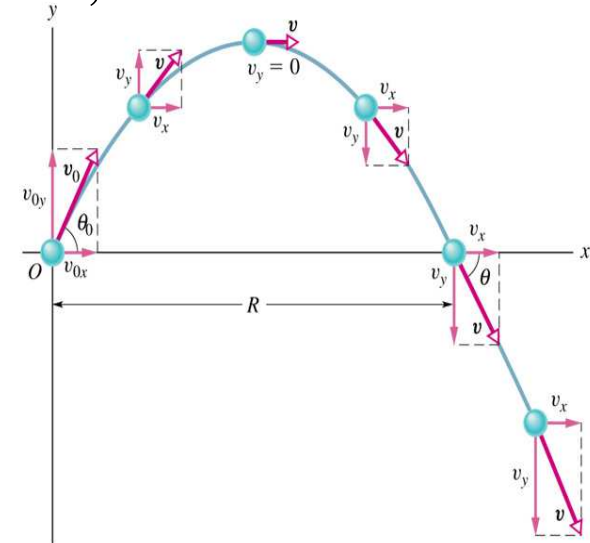


# Esercizio 1: moto di un proiettile

- **modificare** il programma precedente in modo che sia possibile tener conto dell'**attrito** (**viscoso**) dell'aria, con un'equazione del moto:

$$m\vec{a} = m\vec{g} - m\gamma\vec{v} \Rightarrow$$

$$\Rightarrow \begin{cases} x(t) - x_0 = \frac{v_{0x}}{\gamma} (1 - e^{-\gamma t}) \\ y(t) - y_0 = \left( v_{0y} + \frac{g}{\gamma} \right) \frac{1}{\gamma} (1 - e^{-\gamma t}) - \frac{g}{\gamma} t \end{cases}$$



dove  $\gamma$  (in unità di  $1/s$ ) indica il ***coefficiente di attrito***;

- stimare la **gittata** del proiettile  **$R$**  (la coordinata  **$x$**  per cui  **$y=0$** ) e le coordinate  **$x, y$**  ed il *tempo* corrispondenti al punto **più alto** della traiettoria, usando uno degli algoritmi descritti per il calcolo delle **radici di funzioni**.

# Esercizio 1: moto di un proiettile

Determinare l'influenza dell'**attrito**, considerando i casi:  $\gamma=0.1$  e  $\gamma=0.3 \text{ s}^{-1}$  ; in particolare, rispetto al moto in assenza di attrito (realizzare **grafici** illustrativi) :

- il punto **più alto** viene raggiunto in un **tempo inferiore** o **superiore**?
- il punto **più alto** è più **basso** o più **alto** ?
- per quale valore dell'**angolo** la gittata è **maggiore** ?
- la traiettoria è ancora **simmetrica** rispetto al punto più alto ?

# Esercizio 1: moto di un proiettile

**N.B.** storicamente i primi computer (ad es. *ENIAC*, sviluppato dal 1939 al 1946, per l'esercito americano) vennero utilizzati proprio per calcoli balistici !

