

# Zeri di una funzione

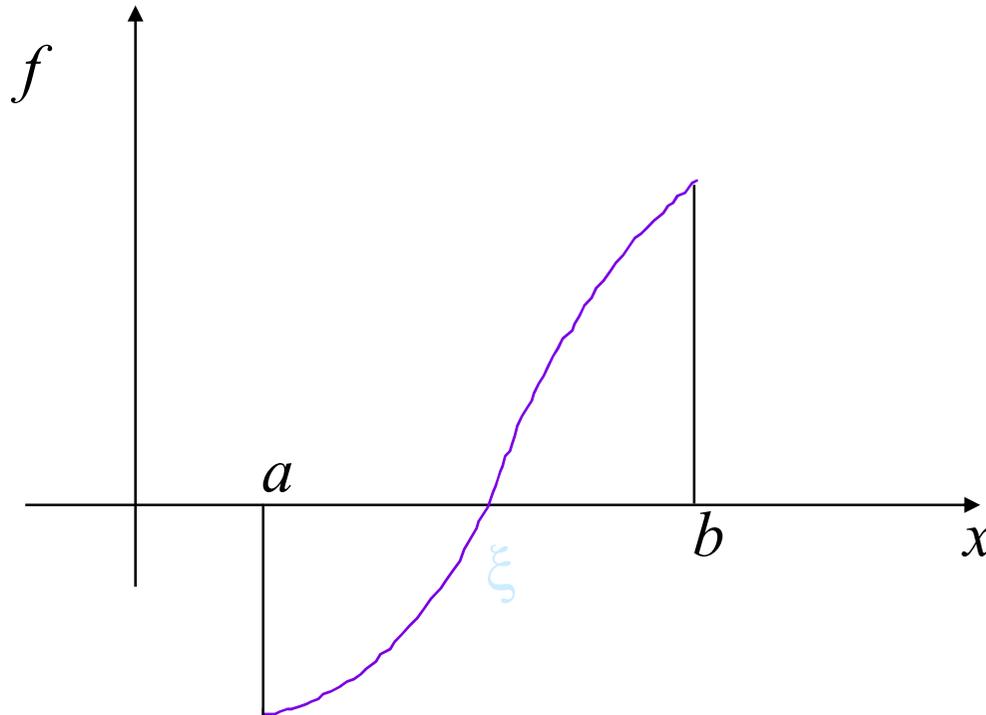
---

- Ricerca delle (eventuali) ***radici reali di una funzione*** che si supporrà definita e continua in un certo intervallo dell'asse  $x$
- La ricerca delle radici approssimate è composta da:
  - 1) separazione delle radici  $\rightarrow$  determinare gli intervalli  $a, b$  che contengono una sola radice
  - 2) **calcolo di un valore approssimato** della radice e **miglioramento** di tale valore fino ad ottenere la precisione desiderata (iterazione)

# Funzioni continue - Proprietà

---

- Se una funzione continua  $f(x)$  assume in due punti  $a$  e  $b$  valori di segno opposto, esiste almeno un valore  $\xi$  (o un numero dispari di punti) compreso fra  $a$  e  $b$  in cui  $f(x)=0$



# Metodo della bisezione

---

1. Dividere l'intervallo  $[a,b]$  a metà:  
 $c = (a + b)/2 \rightarrow$  calcolare  $f(c)$
2.  $f(c) = 0 \rightarrow$  radice trovata, fine procedimento.
3.  $f(c) > 0 \rightarrow$  trascurare l'intervallo destro  $[c,b] \rightarrow$   
imporre  $b = c$ , proseguire col punto 5
4.  $f(c) < 0 \rightarrow$  trascurare l'intervallo sinistro  $[a,c] \rightarrow$   
imporre  $a = c$ , procedere da 5
5. Se  $[a, b] < 2\varepsilon$ , terminare ( $\varepsilon$  è la precisione desiderata)  
Altrimenti proseguire da 1

In pratica, si approssima la funzione con la retta che passa per i punti  $(a, \text{sign}(f(a)))$ ,  $(b, \text{sign}(f(b)))$

# Problemi da affrontare...

---

...a parte l'implementazione dell'algoritmo

- Dove codificare la funzione di cui calcolare gli zeri?
  - Dentro l'algoritmo?
  - Da qualche altra parte?
- Come trattare gli eventuali errori?
  - Stampe dentro l'algoritmo?
  - Restituzione di un codice d'errore?
- Come organizzare il codice?
  - Tutto in un file?
  - Separazione di moduli funzionali in file diversi?

...le risposte sono abbastanza ovvie...

# Definizione della funzione

---

- Se definita internamente al modulo dove si definisce l' algoritmo, il modulo non può essere riutilizzato ☹
- Soluzione:
  - Dichiarazione in un header file (Funzione.h)
  - Definizione nel corrispondente file sorgente (Funzione.c)
- Riusabilità a livello di codice oggetto: con operazioni di linking diverse è possibile collegare definizioni diverse della funzione

# Definizione della funzione

---

## ■ Funzione.h

```
double funzione(double x);
```

## ■ Funzione.c

```
#include "Funzione.h"  
double funzione(double x)  
{  
    return x*x - 2;  
}
```

*Complicabile “a piacere”...*

# Trattamento degli errori

---

- Restituire un codice diverso a seconda del successo o del tipo di errore avvenuto
- I codici d' errore sono definiti “da qualche parte” e sono facilmente decodificabili
  - Può essere prevista una funzione che dato il codice stampa a video qualcosa che indichi che cosa è accaduto
- **Come** definire i codici d' errore?
  - Costanti descritte da commenti nel codice
  - Costanti simboliche
- **Dove** definire i codici d' errore?
  - Nel `main()` sperando che funzioni tutto...
  - In un header file specifico...
  - Vicino alla dichiarazione dell' algoritmo di calcolo degli zeri...

# Definizioni comuni – Common.h

---

```
#define BOOLEAN int  
#define TRUE 1  
#define FALSE 0
```

# Definizione codici d' errore – Zeri.h

---

```
#define CODICEUSCITA int
```

```
#define OK 0
```

```
#define TROPPEITERAZIONI 1
```

```
#define INTERVALLONONVALIDO 2
```

*Stampa un messaggio  
“amichevole” in base  
al codice in ingresso*

```
void printCodiceUscita(CODICEUSCITA code);
```

# Definizione degli errori – Zeri.c

---

```
#include <stdio.h>
#include "Zeri.h"

void printCodiceUscita(CODICEUSCITA code)
{
    switch (code)
    {
        case OK: printf("Ok.");
                break;
        case TROPPEITERAZIONI: printf("Troppe iterazioni.");
                break;
        case INTERVALLONONVALIDO: printf("Intervallo non valido.");
                break;
        default: printf("Codice sconosciuto.");
                break;
    }
}
```

# Algoritmo

---

- Valori in ingresso:
  - Estremi dell' intervallo:  $a, b$
  - Numero massimo di iterazioni
  - Precisione desiderata
- Valori in uscita:
  - Codice d' uscita
  - Valore dello zero

# Algoritmo – Interfaccia

---

*File Zer1.h*

```
#include "Common.h"
```

```
#include "Funzione.h"
```

```
CODICEUSCITA bisezione(double a, double b,  
    int maxIterazioni, double epsilon,  
    double *xZero);
```

# Algoritmo - Pseudocodice

---

- Se gli estremi non sono ordinati, ordinare gli estremi
- I valori della funzione calcolati agli estremi hanno lo stesso segno → Intervallo non valido
- Iterare fino a raggiungere il **numero massimo di iterazioni** o finché non si raggiunge la **precisione desiderata**:
  - Calcolare la funzione agli estremi correnti
  - Calcolare la funzione nel punto medio rispetto agli estremi correnti
  - Se la funzione calcolata nel punto medio ha lo stesso segno dell'estremo sinistro, il nuovo estremo sinistro è il punto medio
  - Altrimenti il nuovo estremo destro è il punto medio
  - Stop quando i due estremi sono abbastanza vicini oppure quando si è trovata la radice – in entrambi i casi la soluzione da restituire è il valore medio dell'intervallo

# Algoritmo – codifica

---

- Serve una funzione per il calcolo del valore assoluto di un `double`
- In C esiste solo quella per il calcolo del valore assoluto di un `int`

```
double doubleAbs(double value)
{
    return ((value<0) ? -value : value);
}
```

# Algoritmo – codifica

---

```
CODICEUSCITA bisezione(double a, double b, int maxIterazioni,  
    double epsilon, double *xZero)  
{  
    int i;  
    double xa, xb; //Estremi correnti  
    double fa, fb; //Valori di f agli estremi correnti  
    double xm, fm; //Valore medio estremi + corrisp. valore di f  
    BOOLEAN stop = FALSE;  
  
    if (a > b)  
    { //Estremi non ordinati --> scambiare  
        xb = a;  
        xa = b;  
    }  
    else  
    { xa = a;  
      xb = b;  
    }  
}
```

continua...

# Algoritmo – codifica

---

```
if (funzione(xa) * funzione(xb) >= 0)
{   return INTERVALLONONVALIDO;
}
for (i = 0; i < maxIterazioni && !stop; i++)
{   fa = funzione(xa);
    fb = funzione(xb);
    xm = (xa + xb) / 2;
    fm = funzione(xm);
    if (fm * fa < 0)
        xb = xm;
    else
        xa = xm;
    stop = ((fm == 0.0) || (doubleAbs(xb - xa) < epsilon));
}

if (stop)
{   *xZero = xm;
    return OK;
}
else
{   return TROPPEITERAZIONI;
}
}
```

# Algoritmo – main()

---

```
#include <stdio.h>
#include <math.h>
#include "Common.h"
#include "Zeri.h"

int main(void)
{
    double zero;
    CODICEUSCITA code;
    code = bisezione(0, 2, 30, 0.0001, &zero);
    if (code == OK)
    {
        printf("Zero: %.10f\n\n", zero);
    }
    else
    {
        printCodiceUscita(code);
        printf("\n\n");
    }
    return (0);
}
```

# Altri algoritmi di calcolo degli zeri

---

- Il procedimento base è sempre lo stesso...
- ...cambia solo il modo di avvicinarsi alla soluzione!
  - Detto  $\xi$  lo zero di  $f$  (appartenente all'intervallo  $[a,b]$ ), sia  $x_0$  una arbitraria approssimazione dello zero  $\xi$  nell'intervallo  $[a,b]$ :
  - Si approssima la funzione con una retta passante per il punto  $(x_0, f(x_0))$  la cui equazione è:
$$y = K_0(x - x_0) + f(x_0)$$
  - L'intersezione tra la retta e l'asse delle ascisse dà origine alla nuova approssimazione  $x_1$  della radice  $\xi$
  - Si itera fino a raggiungere la precisione desiderata

# Altri algoritmi di calcolo degli zeri

---

- Metodi:
  - Corde
  - Secanti
  - Newton – Rhapson
- Questi metodi si basano su approssimazioni successive della funzione  $f$  con rette del tipo:

$$y = K_i(x-x_i) + f(x_i)$$

Ogni metodo differisce dagli altri per la **scelta del coefficiente angolare  $K_i$**