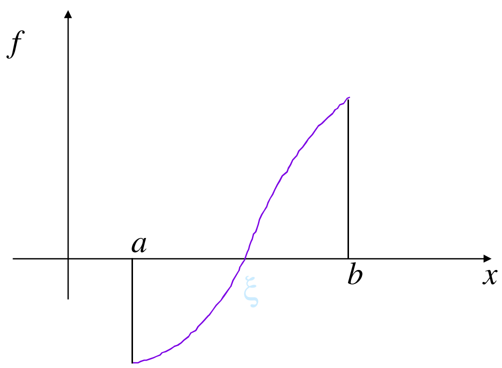
Zeri di una funzione

- 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 e' composta da:
  1) separazione delle radici → determinare degli intervalli a,b contenenti 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 ξ (o un numero dispari di punti) compreso fra a e b in cui *f(x)=0*



Metodo della bisezione

- Si divide l'intervallo [a,b] a meta' mediante il punto:
  - *c = (a + b)/2* → calcolare *f(c).*
- Se **f(c) = 0**, la radice e' trovata ed il procedimento finisce.
- Se ***f(c) > 0***,si trascura l'intervallo destro [c,b], si pone *b = c* e si ripete la procedura nel nuovo intervallo a,b.
- Se ***f(c) < 0***, si trascura l'intervallo sinistro [a,c], si pone *a = c* e si ripete la procedura nel nuovo intervallo [a,b].
- Proseguendo in questo modo o si trova la radice esatta o si raggiunge un intervallo:

  [a, b] < 2ε

  essendo ε e' la precisione desiderata.

<u>In pratica, si approssima la funzione con la retta che passa per i punti (a, sign(f(a))), (b, sign(f(b))).</u>

3

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…

4

Definizione della funzione

- Se definita internamente al modulo dove si definisce l'algoritmo, la modularità ed il riuso (per quanto possibile) crollano!
- Dichiarazione in un header file (Funzione.h)
- Definizione nel corrispondente file sorgente (Funzione.c)
- La riusabilità è a livello di codice oggetto: in link diversi è possibile collegare definizioni diverse!

5

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"…

6

Trattamento degli errori

- La funzione che implementa l'algoritmo restituisce 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 cosa è accaduto
- **Come** definire i codici d'errore?
  - Costanti "da tramandare verbalmente" o tramite commenti
  - Costanti simboliche
- **Dove** definire i codici d'errore?
  - Nel main sperando che funzioni tutto…
  - Un header file specifico

7

Definizione degli errori – `Defines.h`

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

#define CODICEUSCITA int

#define OK 0
#define TROPPEITERAZIONI 1
#define INTERVALLONONVALIDO 2

void printCodiceUscita(CODICEUSCITA code);
```

Stampa un messaggio "amichevole" in base al codice in ingresso

8

Definizione degli errori – `Defines.c`

```
#include <stdio.h>
#include "Defines.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;
    }
}
```

9

Algoritmo

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

10

Algoritmo – Interfaccia

File zeri.h

```
#include "Defines.h"

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

11

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

12

Algoritmo – codifica

- Cosa includere?
  - Libreria matematica
  - Nostre librerie

```
#include <math.h>
#include "Zeri.h"
#include "Funzione.h"
```

13

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;
}
```

14

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

15

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 == 0.0F || doubleAbs(xb - xa) < epsilon;
    }
```

16

Algoritmo – codifica

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

…manca ancora il main per un test!

17

Algoritmo – Test

```
#include <stdio.h>
#include <math.h>
#include "Defines.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 (1);
}
```

18

Notare che…

- L'algoritmo così codificato non interagisce mai con l'interfaccia utente (la console)
- Quindi potrebbe essere utilizzato anche in un mondo **diverso** rispetto quello della console…
- Risultato ottenuto:
  - **Disaccoppiando** il codice di calcolo dal codice di interazione
  - Cercando di **standardizzare l'interfaccia** di interazione – problema del trattamento degli errori

19

Da ricordare

- Oltre all'algoritmo che fa sempre bene…
- **Un principio:**
  - Se è possibile, è sempre meglio disaccoppiare l'interfaccia utente dagli algoritmi → maggiore riusabilità in altri contesti
- **Una pratica:**
  - Evitare di scrivere:

```
if (condizione)
        return TRUE;
else
        return FALSE;
```

ed equivalenti, piuttosto:

```
return condizione;
```

20

Da ricordare

Sempre relativamente alla pratica, è meglio:

```
if (fm == 0.0F || doubleAbs(xb – xa) < epsilon)
    stop = TRUE;
else
    stop = FALSE;
```

oppure

Occhio alla short-circuit evaluation!

```
stop = fm == 0.0F ||
    doubleAbs(xb – xa) < epsilon;
```
?

21

Altri algoritmi di calcolo degli zeri

- Il procedimento base è sempre lo stesso…
- …cambia solo il modo di avvicinarsi alla soluzione!
  - Detto ξ lo zero di *f* (appartenente all'intervallo [a,b]), sia x_0 una arbitraria approssimazione dello zero ξ nell'intervallo [a,b]:
  - Si approssima la funzione con una retta passante per il punto $(x_0, f(x_0))$ la cui equazione e`:

    $$y = K_0(x-x_0) + f(x_0)$$
  - L'intersezione tra la retta e l'asse delle ascisse da` origine alla nuova approssimazione x_1 della radice ξ.
  - Si itera fino a raggiungere la precisione desiderata

22

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**.

23