Algortimi e tecniche numeriche

- Calcolo di integrali definiti
- Ricerca degli zeri di una funzione
- Ricerca del minimo di una funzione
- Il metodo dei minimi quadrati
- Manipolazione di vettori e matrici
- Ordinamento dei vettori
- Liste e alberi
- Ricerca a chiave in liste e alberi

Calcolo Numerico

Capita spesso in Fisica di affrontare problemi complessi per i quali non si sa impostare una soluzione analitica, o non si è in grado di risolvere analiticamente le equazioni corrispondenti. In questo caso le tecniche di calcolo numerico consentono di ottenere soluzioni con il grado di precisione richiesto.

Il calcolo numerico è una branchia piuttosto complessa della matematica. Noi ci limiteremo a mostrare alcune tecniche di base, che hanno più che altro utilità dimostrativa.

Ricordatevi, prima di affrontare un problema (serio) di calcolo numerico di consultare la letteratura (ad esempio il classico ("Numerical recipes in C" che trovate disponibile anche su internet) e di verificare se qualcuno che ha già risolto lo stesso problema vi può fornire una libreria di funzioni.

Calcolo di integrali definiti

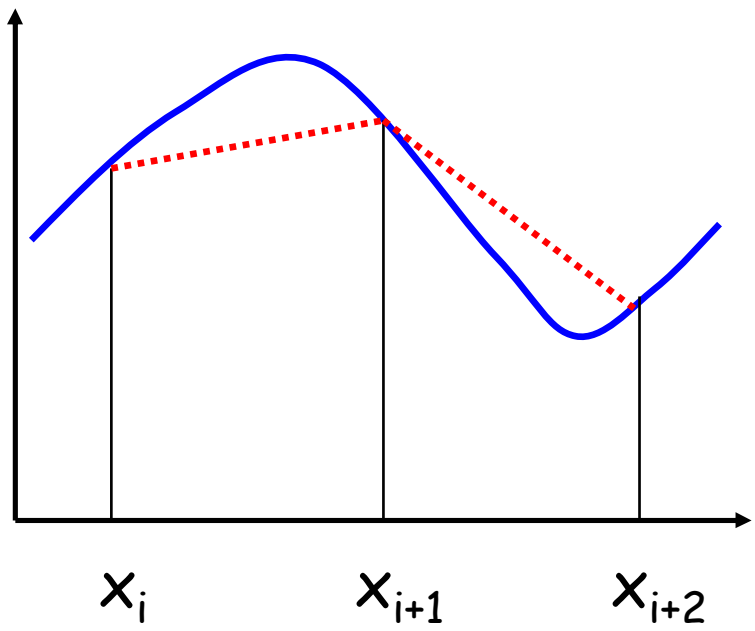
Il metodo più semplice per calcolare un integrale definito consiste nel suddividere l'intervallo di integrazione [a,b] in N parti uguali, delimitate dai punti:

$$x_i = x_0 + w*i; \quad i=0, 1,\ldots,N$$

dove $x_0 = a$ e $w = (b-a)/N$.

In ogni sottointervallo i sostituiremo l'integrale con l'area del trapezio delimitato dalla retta che congiunge i punti $(x_i, f(x_i))$ e $(x_{i+1}, f(x_{i+1}))$.

Questo metodo è esatto per una retta, e in generale l'errore è proporzionale a $w^3 f''(x)$.



$$\int_a^b f(x)dx = \sum_0^{N-1} \frac{w}{2}\left(f(x_i) - f(x_{i+1})\right)$$

L'errore sul calcolo dell'integrale e proporzionale al cubo della larghezza dell'intervallo, per cui, come facilmente intuibile, intervalli piccoli significa errori piccoli (e, naturalmente, tempi di calcolo lunghi).

Un ulteriore miglioramento del metodo consiste nell'utilizzare un doppio intervallo (cioè tre punti) ed approssimare l'integrale nel doppio intervallo con l'area sotto la parabola che passa per i tre punti. La formula da usare, detta formula di Simpson, è:

$$\int_{x}^{x+2w} f(x)dx = \frac{w}{3}\left(f(x_i)+4f(x_{i+1})+f(x_{i+2})\right)$$

In questo caso l'errore è proporzionale a $w^5 f^{(4)}(x)$.

```
double integr(double func(double x), double a, double b, int n) {
  double x, w = (b-a)/n;
  double integral=0;
  for (x=a; x<b; x+=w) {
    integral += w*(func(x) + func(x+w))/2;
  }
  return integral;
}

double interg_err(double func(double x), double a, double b, double err) {
  int n = 100;
  double rel, integral = integr(func,a,b,n);
  do {
    n*=2;
    double new = integr(func,a,b,n);
    rel = fabs(integral-new)/integral;
    integral = new;
  } while (rel > err);
  cout << "Numero intervalli = " << n << endl;
  return integral;
}
```

Calcolo con larghezza di intervallo fissata

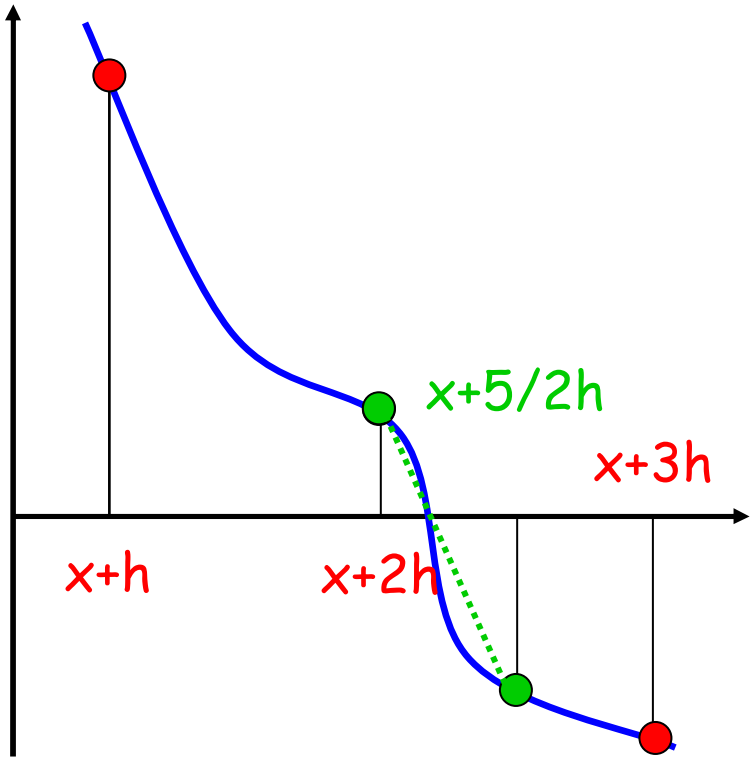
Si potrebbe ottimizzare…

Si dimezza la larghezza dell'intervallo fino ad ottenere una variazione relativa rispetto all'iterazione precedente al di sotto di un valore prefissato.

Zeri di una funzione

Se una funzione continua assume, agli estremi di un intervallo, valori discordi in segno si annulla in almeno un punto all'interno dell'intervallo.

Una tecnica di ricerca degli zeri consiste quindi nel fissare un punto di inizio della ricerca x_0 e un passo di campionamento h; si calcola la funzione nei punti x_0, x_1, con $x_1 = x_0 + h$. Se i valori $f(x_0)$ e $f(x_1)$ sono concordi si passa all' intervallo $[x_1, x_2]$ e così via fino a quando si trovano valori discordi. A questo punto si cambia segno al passo di campionamento e lo si dimezza. L'iterazione continua fino a che il passo di campionamento non scende sotto un valore prefissato.



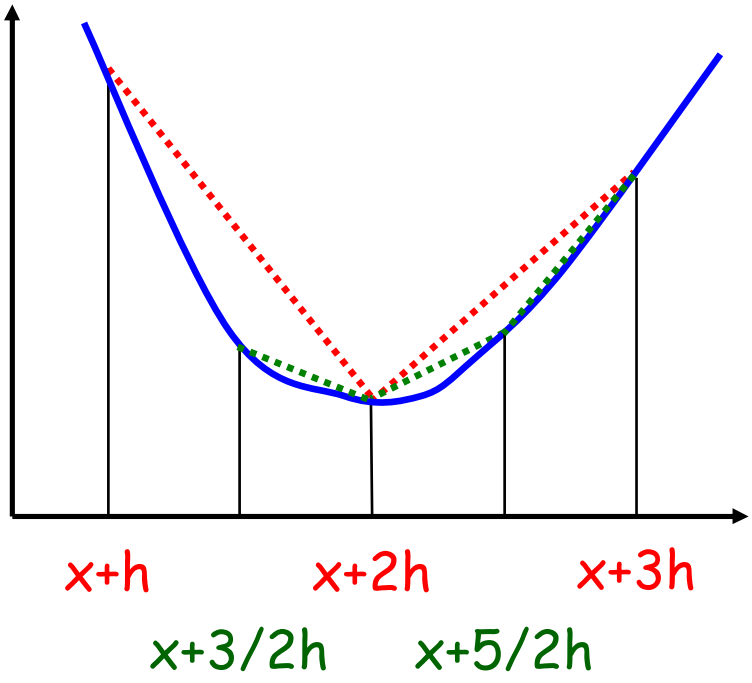
La miglior stima dello zero è data dallo zero della retta passante per gli estremi dell'intervallo

Osservazioni

- Il metodo funziona solo con funzioni continue e in intervalli limitati.
- La condizione di cambio segno è sufficiente ma non necessaria. Diventa necessaria solo nel caso in cui la funzione coincida con la spezzata passante per gli estremi degli intervalli.
- Di fatto il metodo funziona solo a patto che l'intervallo di campionamento sia sufficientemente piccolo da rendere accettabile l'approssimazione rettilinea.

Minimizzazione di funzioni

Un metodo per la ricerca del minimo di una funzione in casi non patologici è il seguente: si fissa un punto di inizio della ricerca x_0 e un passo di campionamento h; si calcola la pendenza della retta congiungente i punti della funzione agli estremi dell' intervallo $[x_0, x_1]$, con $x_1 = x_0 + h$. Si passa quindi all' intervallo $[x_1, x_2]$ e così via fino a quando la pendenza cambia segno. A questo punto si cambia segno al passo di campionamento e lo si dimezza. L'iterazione continua fino a che il passo di campionamento non scende sotto un valore prefissato.



La miglior stima del minimo è x_j se $[x_{j-1},x_j]$ e $[x_j,x_{j+1}]$ sono gli ultimi intervalli in cui c'è stato un cambio di pendenza.

Osservazioni

- Il metodo funziona solo con funzioni continue e in intervalli limitati.
- Il metodo illustrato non è altro che l'applicazione del metodo di ricerca di zeri precedentemente descritto, applicato alla derivata prima della funzione. Siccome la derivata prima non è in genere disponibile la si approssima con il rapporto incrementale.
- Anche in questo caso quindi è necessario che il passo di campionamento sia scelto in modo da rendere accettabile una approssimazione rettilinea della funzione in ogni intervallo.

Sviluppo in serie di una funzione

Come abbiamo visto nell'illustrazione dei semplici metodi di integrazione e ricerca di zeri, l'approccio del calcolo numerico consiste (spesso) nel sostituire una funzione complicata con una sua approssimazione più semplice. Si può dimostrare che, data una funzione $f(x)$ di classe $C(\infty)$ si può scrivere:

$$f(x) = f(0) + f'(0)x + \frac{1}{2} f''(0)x^2 + \ldots + \frac{1}{n!} f^{(n)}(0)x^n + \ldots$$

Questa serie, detta serie di Taylor, ci dice che, noti i valori di tutte le derivate di una funzione in un punto è possibile ricostruire la funzione in un punto qualsiasi.

Osservazioni sulla serie di Taylor

$$f(x) = f(0) + f'(0)x + \frac{1}{2}f''(0)x^2 + \ldots + \frac{1}{n!}f^{(n)}(0)x^n + \ldots$$

- Lo sviluppo di Taylor è polinomiale: gli $f^{(n)}(0)$ sono costanti numeriche.
- Si può partire in un punto diverso dall'origine; basta fare una traslazione dell'asse x.
- Se si è interessati all'intorno dell'origine (o del punto di sviluppo) si avrà $x \ll 1$, e quindi i contributi dei termini di ordine n andranno progressivamente decrescendo. Ciò consente di troncare lo sviluppo ad un dato termine n trascurando i contributi di ordine x^{n+1}.

Metodo dei minimi quadrati

Supponiamo di avere un set di misure $(x_i, y_i \pm \sigma_i)$ con $i=1,\ldots N$ e di una relazione funzionale teorica che lega le grandezze x e y

$$y = f(x, p_1, p_2, \ldots, p_M)$$

dove i p_j sono parametri liberi della teoria. Si pone il problema di determinare quale tra le possibili scelte dei p_j si adatti meglio ai punti misurati. Il metodo dei minimi quadrati consiste nell'affermare che la miglior scelta è quella che rende minima la funzione:

$$\chi^2(p_1, p_2, \ldots, p_M) = \sum_{i=1}^{N} \frac{(y_i - f(x_i, p_1, p_2, \ldots, p_M))^2}{\sigma_i^2}$$

Metodo dei minimi quadrati

Per fissare le idee possiamo pensare che la funzione sia una retta, cioè

$$f(x, p_1, p_2) = p_1 x + p_2$$

Il problema del best-fit consiste nello scegliere, tra le infinite possibili rette, quella che meglio si adatta alle nostre misure.

Per fare questo calcoliamo la funzione $\chi^2(p_1,p_2)$, che consiste nella somma dei quadrati delle distanze lungo y tra i punti e la retta diviso per il quadrato dell'errore. Ovviamente tale funzione è nulla se la retta passa per tutti i punti, e comunque è tanto più piccola quanto maggiore è l'accordo tra la retta e i punti, o quanto maggiore è l'errore di misura.

Metodo dei minimi quadrati

Il problema della minimizzazione del χ^2 nel caso di una retta possiede una soluzione analitica. Se però la funzione non è una retta ci si deve affidare ad un metodo numerico di ricerca del minimo. Se la funzione ha molti parametri la ricerca sarà complicata, e quindi è conveniente usare un algoritmo "serio" e non tentare il fai-da-te.

Il χ^2, oltre a indicarci i valori dei parametri di f che danno il migliore accordo ci fornisce altre informazioni:

- indica la qualità dell'accordo; intuitivamente χ^2 piccolo indica buon accordo, ma vedrete nel corso di Lab1 b che il χ^2 fornisce anche un indicazione statistica quantitativa.
- fornisce una stima dell'errore sui parametri.

Errori sui parametri

Visto che le misure sono affette da errore anche la determinazione dei parametri della funzione avrà una indeterminazione.

La regola dice che l'intervallo corrispondente all'errore sui parametri è quello in cui il χ^2 varia, rispetto al minimo, di 1

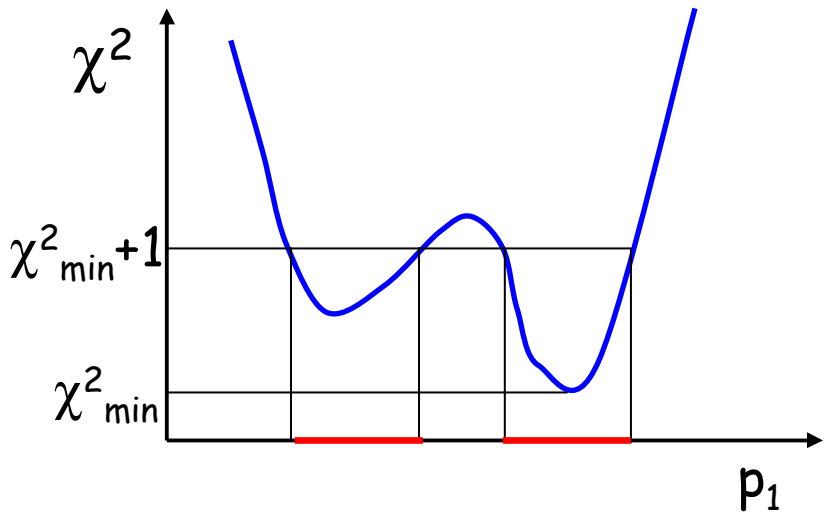
$$\chi^2(p_1,\ldots,p_M) - \chi^2_{min} < 1$$

Non possiamo dimostrare questa affermazione in generale. Se però si considera il caso particolare di una funzione con un solo parametro e un solo punto sperimentale (!!) si può fare una verifica diretta.

Sviluppo di χ^2 intorno al minimo

In generale la forma della regione corrispondente all'errore sui parametri è complessa, e non è necessariamente connessa, nel senso che può consistere di diverse regioni separate.

Possiamo però pensare di eseguire uno sviluppo in serie intorno al minimo (ad es. con due parametri):



$$\chi^2(p_1,p_2)=\chi^2_{\min}+\frac{\partial\chi^2}{\partial p_1}\Delta p_1+\frac{\partial\chi^2}{\partial p_2}\Delta p_2+\frac{1}{2}\frac{\partial^2\chi^2}{\partial p_1^2}\Delta p_1^2+\frac{\partial^2\chi^2}{\partial p_1\partial p_2}\Delta p_1\Delta p_2+\frac{1}{2}\frac{\partial^2\chi^2}{\partial p_1^2}\Delta p_1^2+O(\Delta p^3)$$

Siccome le derivate prime nel minimo sono nulle, restano solo le derivate seconde, e quindi la forma della regione corrispondente alla variazione di χ^2 uguale a uno è ellittica, e le lunghezze degli assi sono legate agli errori sui parametri.

$$\chi^2(p_1,p_2)-\chi^2_{\min}=1=\frac{1}{2}\frac{\partial^2\chi^2}{\partial p_1^2}\Delta p_1^2+\frac{\partial^2\chi^2}{\partial p_1\partial p_2}\Delta p_1\Delta p_2+\frac{1}{2}\frac{\partial^2\chi^2}{\partial p_1^2}\Delta p_1^2$$

La libreria Minuit

Minuit è una pacchetto specializzato per la minimizzazione di funzioni a molti parametri. Nella libreria CRoot c'è un'interfaccia semplificata a questo pacchetto. Vedremo i dettagli in laboratorio. Ciò che dovete fornire alla funzione che trovate in CRoot è:

- la funzione da minimizzare (il χ^2, che dipende da M parametri dati sotto forma di vettore)
- il valore iniziale dei parametri
- il valore iniziale del passo di ricerca del minimo (che puo essere zero se volete bloccare un parametro al valore iniziale)
- il massimo e il minimo per ciascun parametro

In output avrete:

- il valore dei parametri corrispondenti al minimo
- gli errori sui parametri

Buffers circolari (LIFO,FIFO)

```
#define BUFLEN 100

int nel = 0; /* numero di elementi presenti nel buffer */
int pointer = 0; /* indice dell'elemento successivo */
int buffer[BUFLEN];

void add(int i) {
  buffer[pointer] = i;
  nel++; if (nel > BUFLEN) nel = BUFLEN;
  pointer++; if (pointer >= BUFLEN) pointer = 0;
}

void read() {
  int i;
  int ip = pointer;

  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    cout << buffer[ip] << endl;
  }
}
```

I buffers circolari sono vettori nei quali è possibile scrivere quanto si vuole, nel senso che, giunti alla fine del vettore si ricomincia dalla prima posizione (perdendo ovviamente quello che c'era scritto prima).

Ricerca nei vettori

La ricerca nei vettori si esegue di solito in modo euristico, ovvero si controllano tutte le componenti fino a quando non si trova l'elemento cercato. Vediamo una routine che cerca un numero in una LIFO e ritorna la posizione se lo trova e -1 se non lo trova:

```
int find(int val) {
  int i;
  int ip = pointer;
  int pos = -1;

  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    if (buffer[ip] == val) {
      pos = ip;
      break;
    }
  }
  return pos;
}
```

Ordinamento di un vettore

L'ordinamento di un vettore di n elementi si può realizzare nel modo seguente: si cerca l'elemento più piccolo, lo si scambia con quello nella prima posizione, quindi si ripete la procedura nel vettore di n-1 elementi da 1 a n-1; la procedura va iterata fino a quando non si eseguono più scambi o si resta con un vettore di un solo elemento.

```
void sort(int *vect, int n) {
  int i,swap,top,temp;

  if (n < 2) return;

  top = 0;
  do {
    swap = 0;
    for (i=n-1; i>top; i--) {
      if (vect[i] < vect[i-1]) {
        swap = 1;
        temp = vect[i];
        vect[i] = vect[i-1];
        vect[i-1] = temp;
      }
    }
    top++;
  } while(swap != 0 && top < n-1);
}
```

Aggiunta di elementi a un vettore

Per aggiungere un elemento ad un vettore si devono fare due cose: variarne la dimensione e spostare tutti gli elementi posizionati dopo quello aggiunto:

```
int *vect = NULL;
int n_el = 0, max_el = 0;

void increase_size(int siz) {
  int i,*temp;
  /* Crea un nuovo vettore */
  max_el = max_el + siz;
  temp = new int[max_el]
  /* Copia il vecchio vettore nel nuovo */
  for (i=0; i<n_el; i++) temp[i] = vect[i];
  /* Cancella il vecchio vettore e riassegna il puntatore */
  if (vect != NULL) delete[] vect;
  vect = temp;
}
```

```
int add_element(int val) {
   int i;

  /* Controlla se c'e' ancora spazio */
  if (n_el + 1 > max_el) increase_size(100);

  /* Aggiungi il nuovo elemento al suo posto */
  for (i=n_el++; i>0; i--) {
    if (vect[i-1]>val) {
      vect[i] = vect[i-1];
    } else {
      vect[i] = val;
      return i;
    }
  }
  vect[0] = val;
  return 0;
}
```

Viene inserito un elemento rispettando l'ordine crescente degli elementi.

Liste

Una lista è un insieme di strutture collegate tra loro da links che connettono ogni elemento al precedente ed al successivo. Questa struttura consente di aggiungere elementi nel mezzo della lista modificando due soli links; inoltre non impone limiti di dimensione.

```
struct element {
  struct element *previous;
  struct element *next;
  string name;
  string city;
  int number;
};
```

Aggiunta di un elemento a una lista

```
void add(string name, string city, int number) {
  element *new_el,*i, *tmp;

  /* Per prima cosa creiamo un nuovo elemento copiamoci dentro i dati */
  new_el = new element;
  new->name = name;
  new->city = city;
  new->number = number;

  /* Adesso andiamo a cercare il primo elemento gia' presente nell'elenco
     con il nome successivo in ordine alfabetico a quello che stiamo
     inserendo */
  if (first == NULL) {
    first = new;
    new->previous = new->next = NULL;
    return;
  }
```

Inseriamo il primo elemento

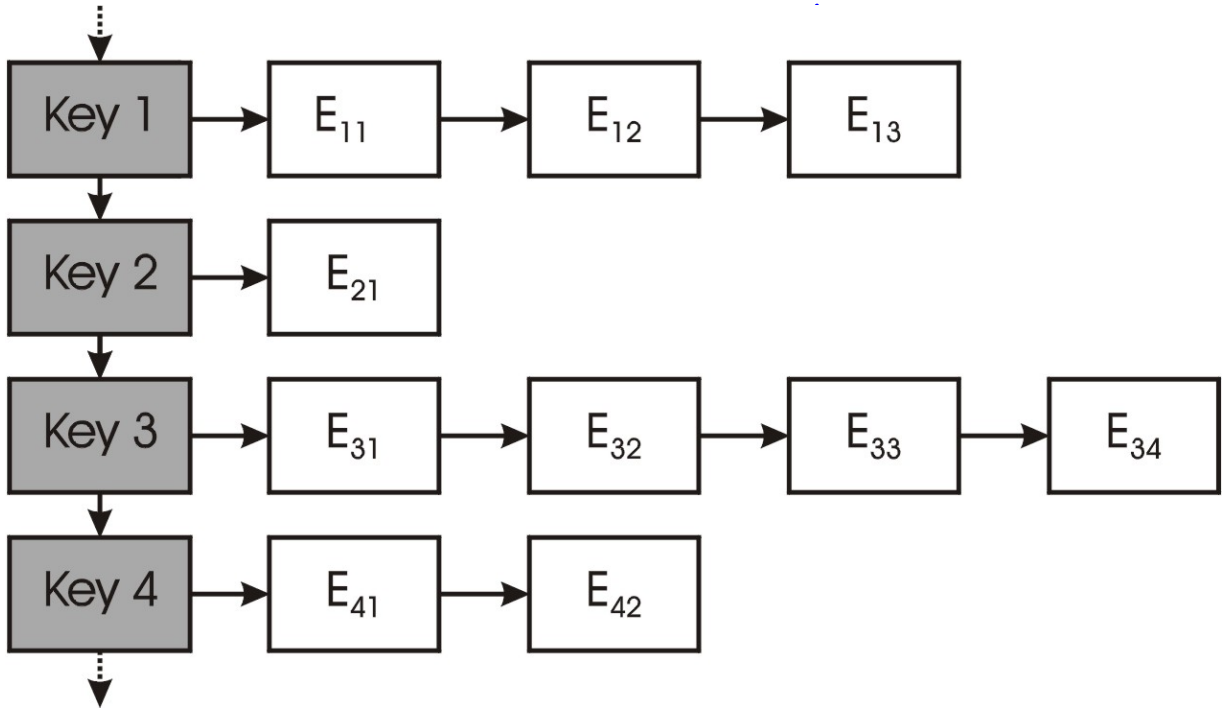
```
    i = first;
    while (i != NULL) {
      if (name < i->name) {
        /* A questo punto inseriamo il nuovo elemento subuto
            prima di quello trovato */
        new->previous = i->previous;
        i->previous = new;
        new->next = i;
        if (new->previous != NULL) (new->previous)->next = new;
        if (i == first) first = new;
        return;
      }
      if (i->next == NULL) {
        /* Se siamo arrivati all'ultimo elemento,  aggiungiamo
            il nuovo alla fine */
        i->next = new;
        new->previous = i;
        new->next = NULL;
      }
      i = i->next;
    }
}
```

Ricerca a chiave

Si esegue utilizzando liste di chiavi che a loro volta contengono liste di elementi:

```
struct key {
  struct key *previous;
  struct key *next;
  struct element *first;
  string keyval;
};
```

```
struct element {
  struct element *previous;
  struct element *next;
  string name;
  string city;
```



```
struct element* find_key(string name) {
  struct key* k;
  struct element* el;

  k = first;
  while (k != NULL) {
    if (k->keyval[0] == name[0]) {
      el = k->first;
      while (el != NULL) {
        if (name == el->name) return el;
        el = el->next;
      }
      break;
    }
    k = k->next;
  }
 return NULL;
}
```

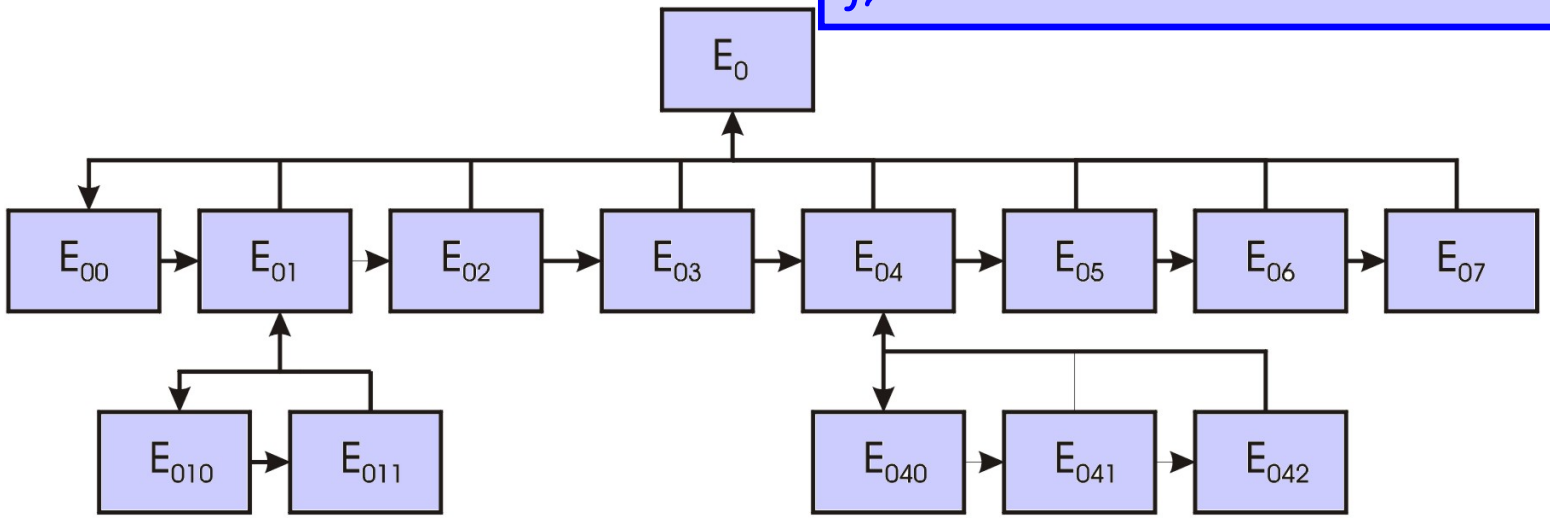
Loop sugli elementi

Loop sulle chiavi

Alberi

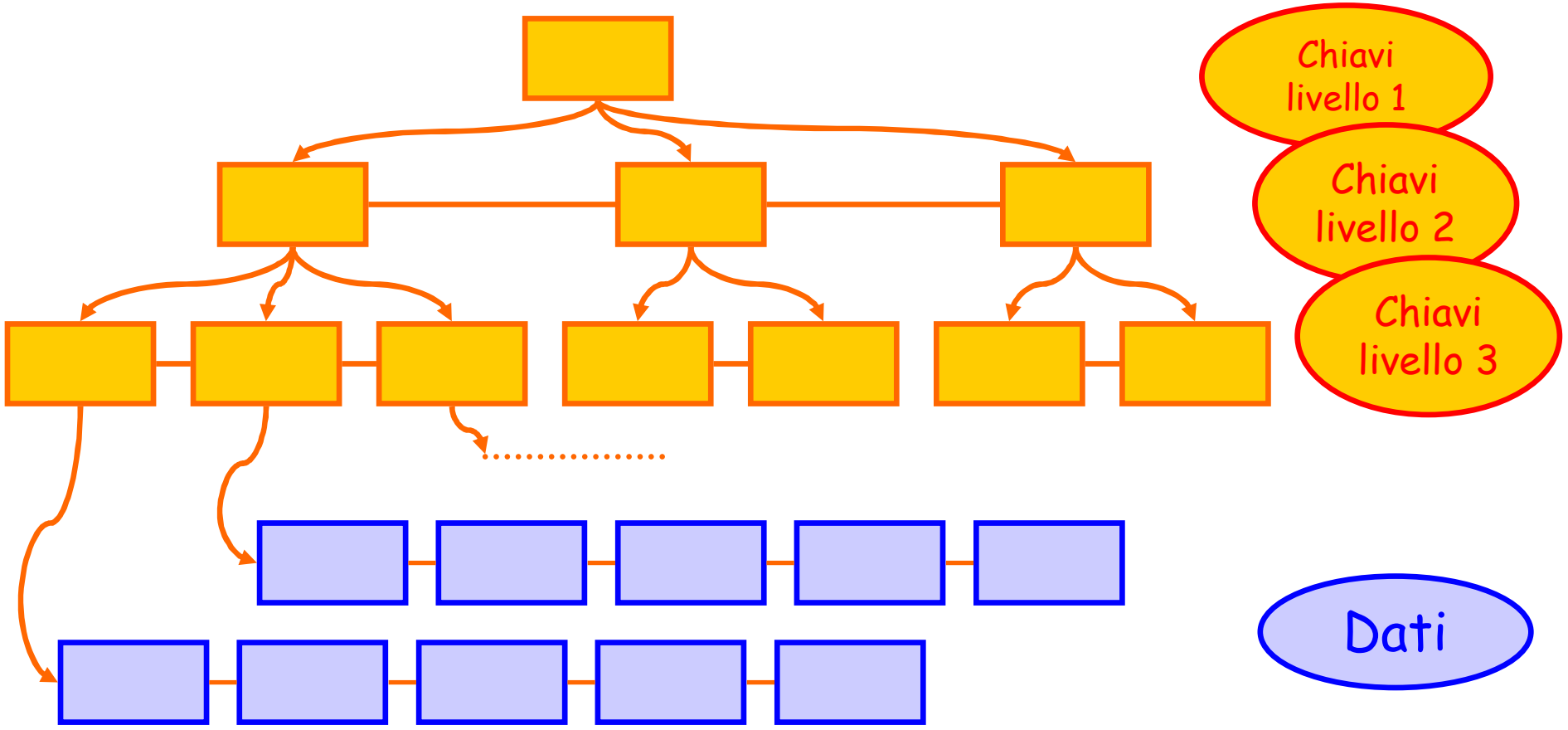
In un albero, ogni elemento possiede, oltre ai links previous e next, anche i links up e down. Gli elementi possono essere omogenei, o distinti in chiavi e dati; in questo caso una variabile deve indicare di che oggetto si tratta. In questo modo si possono implementare molti livelli di chiavi.

```
struct tree_element {
  struct tree_element *previous;
  struct tree_element *next;
  struct tree_element *up;
  struct tree_element *down;
  int type;
  string keyval;
  string name;
  string city;
  int number;
};
```



Ricerca a chiave in un albero

Gli elementi di un albero possono essere tutti dello stesso tipo, oppure dividersi in chiavi e dati.



Ricerca a chiave in un albero

La ricerca in un albero si implementa con il metodo recursivo: si scrive una function che riceve come parametro un elemento dell' albero e controlla se si tratta dell'elemento cercato. In caso affermativo ritorna l'elemento trovato, altrimenti esegue la ricerca in tutti i figli dell'elemento richiamando se stessa.

Implementando quindi un solo livello si ricerca e sfruttando la recursività si ottiene l'esplorazione dell'intera struttura dell'albero.

```
Struct tree_element* find_tree(string name, struct tree_element *el) {
 struct tree_element *i, *ret;
 /* Controlliamo se el e' una chiave o un dato */
 if (el->type == 0) {
   /* Se e' un dato, controlliamo se il nome e' quello cercato */
   if (name == el->name) return el;
 } else {
   /* Se e' una chiave, controlliamo se la lettera type-ima del nome
      cercato coincide con il valore della chiave */
   if (name[el->type-1] == el->keyval) {
     /* in caso affermativo esploriamo tutti gli elementi della lista figlia
        richiamando questa procedura */
     i = el->down;
     while (i != NULL) {
       ret = find_tree(name, i);
       if (ret != NULL) return ret;
       i = i->next;
     }
   }
 }
 return NULL;  /* vuol dire che non si e' trovato nulla in questo ramo */
}
```