Dispense di Esercizi per il corso di Laboratorio di Calcolo

Laurea Triennale in Fisica

Sapienza Università di Roma

Redatte da:

Stefano Campion, Michelangelo De Feo, Davide Germani, Flavio Giuliani

Autori e Autrici degli esercizi:

proff. Luciano Maria Barone, Lilia Boeri, Cristiano De Michele, Nicoletta Gnan, Giovanni Organtini, Shahram Rahatlou, Livia Soffi

A.A. 2021/22

Prefazione

Negli ultimi due anni, l'epidemia di COVID ci ha costretti a rivedere pratiche consolidate da molti anni. La necessità di adattare materiale e modalità didattiche è stata particolarmente significativa per i corsi di laboratorio, in cui è venuto a mancare il confronto diretto tra docente e insegnante che normalmente è cruciale ad acquisire competenze pratiche.
Queste dispense nascono dal tentativo di fornire agli studenti uno strumento in più per esercitarsi ad applicare le conoscenze di base di programmazione apprese durante il corso di Laboratorio di Calcolo del primo anno della Laurea Triennale in Fisica della Sapienza di Roma.
Le dispense, redatte con l'aiuto essenziale dei *tutor* che hanno risposto ai bandi di tutoraggio degli AA 2020/21 e 2021/22 e degli assegnisti di ricerca, raccolgono la soluzione dettagliata di diversi esercizi di esame, forniti dai docenti attuali e passati del corso.
Tutti i testi di esame comprendono una prova di programmazione in `C`, e un grafico in `Python`.
L'appendice contiene un riepilogo dei problemi più frequenti che si incontrano durante la soluzione.
Buon lavoro!

Gli autori:

Luciano Maria Barone, Lilia Boeri, Stefano Campion, Michelangelo De Feo, Davide Germani, Flavio Giuliani, Cristiano De Michele, Nicoletta Gnan, Giovanni Organtini, Shahram Rahatlou, Livia Soffi.

Indice

Parte I

Esercizi svolti

Capitolo 1

Triangolo di Sierpinski

1.1 Testo dell'esercizio

Il *triangolo di Serpinski*, raffigurato in figura, è un esempio di frattale, cioè di figura geometrica autosimile, ottenuta dividendo ricorsivamente un triangolo in triangoli via via più piccoli.

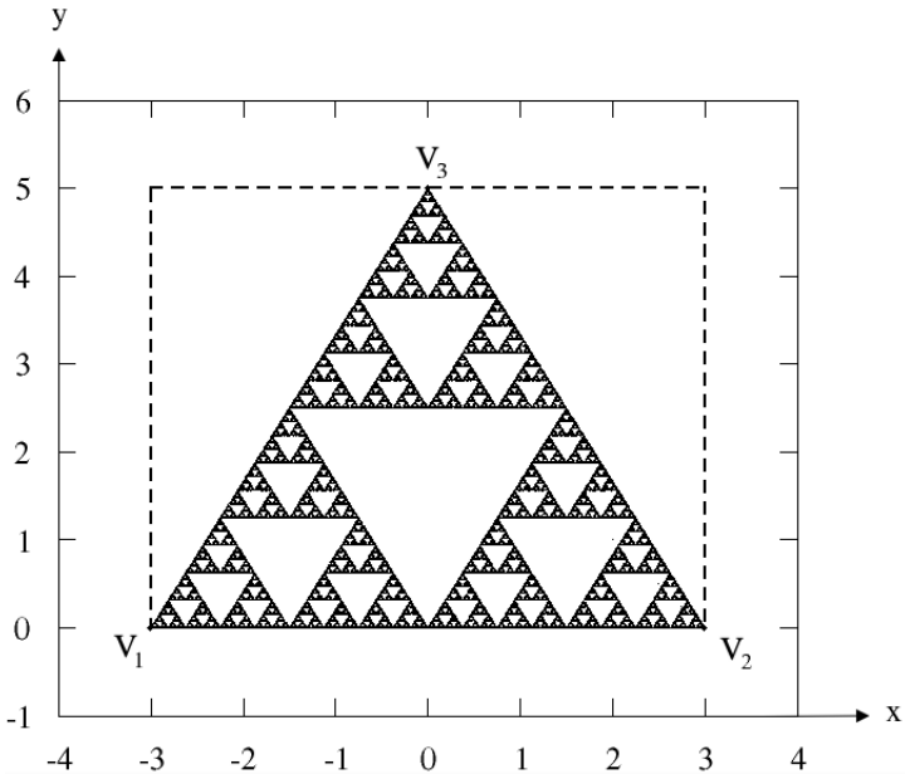


Figura 1.1: Triangolo di Serpinski di vertici $\mathbf{P}_1 = (-3, 0)$; $\mathbf{P}_2 = (3, 0)$; $\mathbf{P}_3 = (0, 5)$. Questa figura è stata generata iterando l'algoritmo descritto nel testo $N = 10000$ volte. La linea tratteggiata indica il rettangolo all'interno del quale va scelto il punto iniziale della successione $\mathbf{v}_n$.

Il triangolo di Serpinski può essere costruito mediante diversi algoritmi. In questo esercizio si userà una costruzione basata sul gioco del caos (*Chaos Game*), che funziona come segue:

1. Si sceglie come punto di partenza un punto $\mathbf{v}_1 = (x_1, x_2)$ qualsiasi, contenuto all'interno del triangolo di vertici $\mathbf{P}_1$,$\mathbf{P}_2$ e $\mathbf{P}_3$.

2. A questo punto si sceglie in maniera casuale uno dei tre vertici del triangolo.

3. Si genera un nuovo punto della successione come: $\mathbf{v}_2 = \frac{1}{2}(\mathbf{v}_1 + \mathbf{P}_R)$. Il simbolo $\mathbf{P}_R$, con $R = 1, 2$ o 3 indica il vertice scelto in modo casuale al punto 2.

4. Le operazioni 2. e 3. vengono ripetute N volte, generando la successione di punti:

$$\mathbf{v}_{n+1} = \frac{1}{2}(\mathbf{v}_n + \mathbf{P}_R) \tag{1.1}$$

5. Se il numero N è sufficientemente grande, i punti così generati ricostruiscono un triangolo di Serpinski.

Scrivere un programma che generi il triangolo di Serpinski di vertici $\mathbf{P}_1 = (-3, 0)$; $\mathbf{P}_2 = (3, 0)$; $\mathbf{P}_3 = (0, 5)$, mediante l'algoritmo descritto sopra, come segue:

1. Mediante una direttiva al precompilatore, si definisce il numero di punti `N` che compongono la successione – si consiglia di scegliere un numero `N` > 1000.

2. Il programma chiede all'utente di inserire le coordinate del punto $\mathbf{v}_1$ da cui far partire la successione, e controlla, tramite una funzione `IsInRect()`, che il punto corrispondente si trovi all'interno del rettangolo di base 6 e altezza 5, che contiene esattamente il triangolo di vertici $\mathbf{P}_1$, $\mathbf{P}_2$ e $\mathbf{P}_3$ – il rettangolo è tratteggiato in figura.[1] Altrimenti reitera la richiesta.

3. Le coordinate iniziali vengono quindi salvate in un vettore `vin[]` di tipo e dimensioni opportune.

4. A questo punto, il programma itera N volte l'algoritmo del *Chaos game* descritto sopra, tramite una funzione `NewPoint()` che riceve in input le coordinate del punto $\mathbf{v}_n$ contenute nel vettore `vin[]` e genera un nuovo punto della successione, scegliendo in modo casuale uno dei tre vertici $\mathbf{P}_R$ e calcolando $\mathbf{v}_{n+1}$ tramite l'Eq. 1.1. Le coordinate del punto $\mathbf{v}_{n+1}$ vengono salvate in un vettore `vout[]` di tipo e dimensioni opportune.

5. Ogni 10 passi, il programma stampa sullo schermo le seguenti informazioni: numero del passo; numero del vertice scelto per generare il nuovo punto; coordinate del punto di partenza $\mathbf{v}_n$; coordinate del punto di arrivo $\mathbf{v}_{n+1}$, nel formato:
   `It. n. 860 vertice n. 1 v(n) = (-2.38,0.83) v(n+1)=(-2.69,0.42)`

6. Le coordinate dei punti $\mathbf{v}_n$ che compongono la successione vengono salvate su un file di due colonne di nome `fractal.dat`.

7. Graficare con uno script in Python le coordinate dei punti contenuti nel file `fractal.dat`, per ottenere un grafico simile a quello mostrato in Fig 1.1.

[1] Questa condizione rimpiazza quella che il punto sia contenuto all'interno del triangolo di partenza, che non è strettamente necessaria.

1.2 Soluzione completa

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000
#define V1x -3.0
#define V1y 0.0
#define V2x 3.0
#define V2y 0.0
#define V3x 0.0
#define V3y 5.0

int IsInRect(float x, float y);
int NewPoint(float xn, float yn, float vout[2]);

int main(){
    int contr, i, R;
    float x1,y1, vin[2], vout[2];
    FILE *fp;
    // INSERIMENTO PUNTO INIZIALE
    printf("Inserire le coordinate x,y del punto iniziale separate da uno spazio
    :\n");
    do{
        scanf("%f %f",&x1,&y1);
        contr=IsInRect(x1,y1);
        if(contr==0){
            printf("Errore: il punto non appartiene al rettangolo delimitato dai
    vertici (%f,%f), (%f,%f).\n", V1x,V1y,V2x,V3y);
        }
    }while(contr==0);
    // CICLO PRINCIPALE
    srand48(time(NULL));
    fp=fopen("fractal.dat","w");
    vin[0]=x1;
    vin[1]=y1;
    fprintf(fp,"%.2f %.2f\n",vin[0],vin[1]);
    for(i=0;i<N;i++)
      {
      R=NewPoint(vin[0],vin[1],vout);
      if (i%10 == 0)
          printf("It. n. %d vertice n. %d p(n) = (%.2f,%.2f) p(n+1)=(%.2f,%.2f)\n
    ",i,R,vin[0],vin[1],vout[0],vout[1]);
      vin[0]=vout[0];
      vin[1]=vout[1];
```

```
      fprintf(fp,"%f %f\n",vin[0],vin[1]);
      }
    fclose(fp);
}

int IsInRect(float x, float y){
  int controllo=1;
    if(x<V1x || x>V2x){
        controllo=0;
    }else if(y<V1y || y>V3y){
        controllo=0;
    }
    return controllo;
}

int NewPoint(float xn, float yn, float* vout)
{
  float xnew,ynew;
  int r=1+drand48()*3;
  if(r==1){
    xnew=0.5*(xn+V1x);
    ynew=0.5*(yn+V1y);
  }else if(r==2){
    xnew=0.5*(xn+V2x);
    ynew=0.5*(yn+V2y);
  }else{
    xnew=0.5*(xn+V3x);
    ynew=0.5*(yn+V3y);
  }
  vout[0]=xnew;
  vout[1]=ynew;
  return r;
}
```

```
import numpy as np
import matplotlib.pyplot as plt

x,y = np.loadtxt("fractal.dat",unpack=True)
plt.plot(x,y,".",markersize=1)
plt.xlabel("x")
plt.ylabel("y")
plt.savefig("fractal.png")
```

1.3 Commento alla soluzione

Innanzitutto si include la libreria base di input/output `stdio.h`; per le parti successive serviranno anche `stdlib.h` (numeri casuali) e `time.h` (per inizializzare il generatore di numeri casuali).
La direttiva al precompilatore per definire un simbolo costante (o in generale una macro) è `#define`. Con la seguente sintassi viene definito il simbolo intero `N` pari a 10000, ancora prima che il codice sia eseguito, che non può essere alterata durante l'esecuzione.

```
#include <stdio.h> // funzioni di input/output
#include <stdlib.h> // generatori di numeri casuali: drand48(),srand48()
#include <time.h> // funzione time()
#define N 10000
#define V1x -3.0
#define V1y 0.0
...
```

Poiché le coordinate dei vertici del triangolo sono costanti, conviene dichiarare anch'esse con delle direttive al precompilatore. Un'alternativa è dichiararli come variabili globali (ad esempio `float V1x = -3.0;` fuori dal `main`), ma si consiglia di evitare l'uso di variabili gobali quando possibile.
Il programma deve inizialmente chiedere all'utente di inserire le coordinate del punto iniziale e leggerle da terminale. Il messaggio di richiesta viene stampato sul terminale tramite la funzione `printf()`; segue poi una chiamata alla funzione `scanf()` con argomenti opportuni che salva i valori letti da terminale in due variabili di tipo `float`, chiamate `x1`, `y1`. In questo caso il formato necessario è dato dalla stringa `"%f %f"`, che indica che si vuole leggere due variabili di tipo `float` separate da uno spazio.
Tramite una funzione `IsInRect()`, si deve poi verificare se il punto immesso appartiene al rettangolo. Avendo già definito le coordinate dei vertici del triangolo come direttive al precompilatore, è sufficiente passare come argomento alla funzione solo le coordinate `x` e `y` del punto inserito, dichiarando quindi la funzione come: `int IsInRect(float x, float y)`. Nel seguito sarà chiaro perché si è scelto di dichiararla di tipo `int`. Affinché il punto appartenga al rettangolo indicato in Fig. 1.1, la coordinata x deve essere compresa nell'intervallo [`V1x`,`V2x`] e la y in [`V1y`,`V3y`]. Se almeno una di queste due condizioni non è soddisfatta, la funzione deve restituire un messaggio negativo. In questi casi è utile dichiarare la funzione `IsInRect()` di tipo `int`, in modo da restituire alla funzione chiamante il valore 1 ("va bene") o 0 ("non va bene"). Il valore da restituire viene calcolato all'interno della funzione, salvato in una variabile di controllo e restiuito alla funzione chiamante tramite l'istruzione `return`:

```
int IsInRect(float x, float y){
  int controllo=1; // a priori il controllo sarà "va bene"
    if(x<V1x || x>V2x){
        controllo=0; // se l'ascissa cade fuori dalla base, "non va bene"
    }else if(y<V1y || y>V3y){
        controllo=0; // se l'ordinata cade fuori dall'altezza, "non va bene"
    }
    return controllo;
}
```

Se `IsInRect()` restituisce un valore pari a 0 nel `main()`, bisogna reiterare la richiesta delle coordinate del punto iniziale, tramite un'istruzione `do...while()` avente come condizione che il valore restituito dalla funzione `IsInRect()` sia pari a 0:

```
int main(){
    float x1,y1;
    int contr;
    do{
        printf("Inserire le coordinate x,y del punto iniziale separate da uno spazio:\n");
        scanf("%f %f",&x1,&y1);
        contr=IsInRect(x1,y1);
        if(contr==0){
            printf("Errore: il punto non appartiene al rettangolo delimitato dai vertici (%f,%f), (%f,%f).\n", V1x,V1y,V2x,V3y);
        }
    }while(contr==0);
}
```

Per una gestione più avanzata dell'input, si veda l'Appendice 12 . **Nota**: in alternativa si può utilizzare anche solo l'istruzione `while()`, purché si inizializzi `contr` a 0 per entrare nel ciclo almeno una volta.
Inizia ora la parte principale del codice. Le coordinate iniziali (`x1`,`y1`) devono essere salvate in un vettore. Si dichiara quindi un array `vin[]` di tipo `float` e dimensione 2; nella prima componente si salva la coordinata `x1` e nella seconda la coordinata `y1`: `vin[0]=x1`, `vin[1]=y1`. La successione dei punti $\mathbf{v}_n$ viene generata chiamando iterativamente una funzione `NewPoint()`, che deve calcolare $\mathbf{v}_{n+1}$ a partire da $\mathbf{v}_n$, usando la formula descritta dall'Eq. 1.1. Supponendo di aver già definito la funzione `int NewPoint(...)`, di tipo `int` in modo che possa restituire l'indice del vertice estratto, nel `main()` è sufficiente implementare un ciclo in cui si chiama `NewPoint()` per N volte. Inoltre bisogna stampare dei sommari ogni 10 passi, ovvero ogni volta che `i` è multiplo di 10, ovvero ogni volta che la divisione per 10 da come resto 0. Si può quindi controllare quando questa condizione si avvera utilizzando l'istruzione `if((i%10)==0)`, dove l'operatore binario *modulo* % restituisce appunto il resto della divisione dell'intero a sinistra per l'intero a destra di esso.

```
int main(){
    int contr,i,R;
    float x1, y1, vin[2], vout[2];
    //... inserimento di x1,y1
    vin[0]=x1;
    vin[1]=y1;
    for(i=1;i<=N;i++){
        // calcolo il nuovo punto e salvo R=1,2,3
        R=NewPoint(vin[0],vin[1],vout); //NewPoint() è definita nel riquadro successivo
        // stampo il sommario
        if (i%10 == 0)
```

```
12          printf("It. n. %d vertice n. %d p(n) = (%.2f,%.2f) p(n+1)=(%.2f,%.2f)\n
    ",i,R,vin[0],vin[1],vout[0],vout[1]);
13      // sovrascrivo le vecchie coordinate con le nuove coordinate prima di
    iniziare il ciclo successivo
14      vin[0]=vout[0];
15      vin[1]=vout[1];
16    }
17 }
```

Una prima traccia del corpo della funzione `NewPoint()` è la seguente:

1. si sceglie casualmente uno dei tre vertici, estraendo un intero casuale `r` fra 1,2,3 come descritto nell'Appendice 14;
2. in base al valore estratto, si decide tramite l'istruzione `if..else` su quale vertice eseguire l'algoritmo;
3. le nuove coordinate vengono salvate in delle variabili `xnew`, `ynew` di tipo `float`.

Si vorrebbe che la funzione restituisca `xnew`, `ynew` e anche l'indice `r` del vertice estratto casualmente (come richiesto dal punto 5). Ma una funzione in C può restituire un solo valore in output! Per aggirare l'impasse si può passare come argomento alla funzione un array `vout[2]`, dichiarato di tipo `float` nel `main()`, che contenga le coordinate del punto generato, in modo tale che esse risultino anche nel `main()`. La definizione di `NewPoint()` che soddisfa tutte le richieste del testo è quindi la seguente:

```
1  int NewPoint(float xn, float yn, float* vout)
2  {
3    float xnew,ynew;
4    int r=1+drand48()*3; //estrazione dell'indice del vertice
5    if(r==1){
6      xnew=0.5*(xn+V1x);
7      ynew=0.5*(yn+V1y);
8    }else if(r==2){
9      xnew=0.5*(xn+V2x);
10     ynew=0.5*(yn+V2y);
11   }else{
12     xnew=0.5*(xn+V3x);
13     ynew=0.5*(yn+V3y);
14   }
15   vout[0]=xnew;
16   vout[1]=ynew;
17   // non è necessario restituire xnew,ynew; rimane solo r
18   return r;
19 }
```

Nota: in questo caso non è necessario passare fra gli argomenti la dimensione dell'array `vout[]`, perché si sa a priori che è 2.

Si ricorda inoltre che, affinché il programma generi una sequenza diversa ad ogni esecuzione, bisogna inizializzare il seme del generatore di numeri pseudocasuali, tramite il comando `srand48(time(NULL))`. Infine, si vuole salvare su un file le coordinate di tutti i punti. Bisogna fare attenzione al fatto che il ciclo comincia già con la prima iterazione, pertanto le coordinate di partenza devono essere stampate prima del ciclo.

```
int main(){}
    float x1,y1, vin[2], vout[2];
    int contr,i,R;
    FILE *fp;
    srand48(time(NULL)); // Inizializzazione del seed della sequenza
    //... inserimento di x1,y1 (omesso)
    fp=fopen("fractal.dat","w"); // creazione file output
    vin[0]=x1;
    vin[1]=y1;
    fprintf(fp,"%.2f %.2f\n",vin[0],vin[1]); // stampa i dati iniziali
    for(i=1;i<=N;i++){
      //... calcolo della sequenza (omesso)
      fprintf(fp,"%f %f\n",vin[0],vin[1]); // stampa il nuovo punto nel file di
   output
    }
    fclose(fp);
}
```

Nello script in Python, si segue la procedura standard spiegata in Appendice 16.1 per effettuare un grafico di ordinate in funzione di ascisse a partire da un file di dati disposti in due colonne. **Suggerimento**: per rendere più in dettaglio l'immagine frattale conviene utilizzare nella funzione `plt.plot()` il formato `"."` e l'argomento opzionale `markersize=1`, che riduce al minimo, cioè 1 pixel, la dimensione del punti graficati.

```
import numpy as np
import matplotlib.pyplot as plt

x,y = np.loadtxt("fractal.dat",unpack=True)
plt.plot(x,y,".",markersize=1)
plt.savefig("fractal.png")
plt.show()
```

Capitolo 2

Cifratura di Vigenère

2.1 Testo dell'esercizio

► **Introduzione:**
Il metodo di Vigenère è un metodo di crittografia a sostituzione alfabetica in cui ogni lettera del testo da decrifrare è traslata di un numero di posti variabile, determinato in base ad una breve parola chiave, detta *verme*, da concordarsi tra mittente e destinatario. La parola è detta verme perché, essendo in genere molto più corta del messaggio, per formare la *chiave* vera e propria essa deve essere ripetuta molte volte per poter essere confrontata col messaggio, carattere per carattere, come nel seguente esempio:
Si noti che in questo caso il messaggio contiene degli spazi ai quali sono associate delle lettere della

testo in chiaro= N E L ME Z Z O D E L C A MM I N D I N O S T R A V I T A
chiave= V E R ME V E R ME V E R ME V E R ME V E R ME V E R ME V E R ME
testo cifrato= I I C Q Z D Q A Y I C G V Q D U R H Z R J WK D E Z Z F E

chiave. Nonostante questo gli spazi non devono essere criptati (e quindi rimangono tali).
La crittografia con il metodo di Vigenère avviene nel seguente modo: si sposta ciascuna lettera del messaggio in chiaro (spazi esclusi) di un numero di caratteri pari al numero ordinale della lettera corrispondente del verme ($A = 0, B = 1, C = 2..., Z = 25$). Di fatto si esegue una somma aritmetica tra l'ordinale del testo in chiaro e quello del verme; se la somma supera l'ultima lettera, la Z, si ricomincia a contare dalla A. Matematicamente l'operazione si riduce a

$$\texttt{CIFRATO}_i = (\texttt{CHIARO}_i + \texttt{CHIAVE}_i) \quad \% \quad 26 \tag{2.1}$$

dove $\texttt{CHIARO}_i$,$\texttt{CIFRATO}_i$ e $\texttt{CHIAVE}_i$ sono, rispettivamente, il carattere i-esimo del testo in chiaro, cifrato e della chiave e % è l'operatore modulo. L'espressione sopra è valida solo se il carattere i-esimo non è uno spazio.
Analogamente, a partire da un messaggio criptato con una chiave nota, è possibile ricostruire il messaggio in chiaro attraverso la seguente operazione

$$\texttt{CHIARO}_i = (\texttt{CIFRATO}_i - \texttt{CHIAVE}_i + 26) \quad \% \quad 26 \tag{2.2}$$

Lo scopo dell'esercitazione è creare un cifrario di Vigenère che permetta di criptare e decriptare una frase usando un verme generato in maniera casuale. Sia il codice da cifrare/decifrare che la chiave devono essere lettere dell'alfabeto maiuscole (no accenti né caratteri speciali); inoltre il verme dovrà essere un'unica parola (senza spazi).

► **Prima parte:**
Si crei un programma chiamato `cifra.c` che esegua la cifratura di un messaggio inserito da terminale. Il programma:

- Chiede in input una frase da criptare (le lettere devono essere tutte maiuscole). Attraverso una funzione chiamata `ReadString()` la frase viene letta e memorizzata in un array chiamato `frase[]` passato in input alla funzione. Oltre a memorizzare la frase in un array, `ReadString()` restituisce il numero di caratteri contenuti nella stringa (spazi inclusi). Durante il processo di lettura della frase viene controllato che la lunghezza massima della frase non ecceda `LEN`, una costante simbolica definita attraverso la direttiva al preprocessore, pari a 100. Se la frase eccede in lunghezza `LEN` il programma stampa un messaggio di errore e termina.

- Genera casualmente il verme di 5 caratteri tramite una funzione `CreaVerme()` di tipo `void` che prende in ingresso un array di caratteri. La funzione estrae 5 numeri casuali interi compresi tra 0 e 25 e li salva nell'array di caratteri `verme[]` passato come input alla funzione dal `main()`. Ogni numero estratto va sommato al codice ASCII della lettera 'A' (codice 65) in modo che i numeri risultanti corrispondano a lettere dell'alfabeto maiuscole (si può semplicemente sommare il numero estratto direttamente al carattere 'A', ad esempio il carattere 'C' è uguale a 'A'+2, mentre il carattere 'Z' si ottiene sommando 'A'+25).

- Si crei una funzione chiamata `ReplicaVerme()`, di tipo `void`, che prenda in input due array di caratteri – uno contenete il verme e l'altro che conterrá la chiave – e la lunghezza del messaggio in chiaro. La funzione deve replicare il verme lungo tutta la lunghezza del messaggio, come nell'esempio iniziale, e salvare il risultato viene salvato in un array chiamato `chiave[]`, di lunghezza opportuna.

- Si crei una funzione `Cifra()` di tipo `void` che prenda in input l'array contenente il testo in chiaro, la chiave replicata, la lunghezza della stringa e un array chiamato `cfrase[]` di lunghezza opportuna, che conterrà il messaggio cifrato. La funzione effettua la cifratura usando l'eq.(2.1).

- Nel `main()` si dovrà stampare su schermo nello stesso formato dell'esempio dell'esercitazione le tre stringhe relative al messaggio in chiaro, la chiave (cioè il verme replicato) e il testo cifrato. Per controllare che la cifratura sia andata a buon fine, si può inserire come testo in chiaro quello dell'esempio e manualmente come verme la parola VERME e il risultato deve essere un testo cifrato identico a quello dell'esempio.

► **Seconda parte:**
Si copi il programma `cifra.c` in uno chiamato `decifra.c` e lo si modifichi in maniera tale che esegua la decifratura di un messaggio inserito da terminale. Il programma

- Chiede in input una frase da decifrare (ad esempio una frase cifrata in precedenza con il programma `cifra.c`). Attraverso la funzione `ReadString()` la frase viene letta e memorizzata in un array chiamato `cfrase[]`, di lunghezza opportuna, passato in input alla funzione;

- Chiede in input da terminale il `verme` per la decriptazione del messaggio (di 5 caratteri). Il programma deve leggere tutta la stringa immessa da terminale, controllare che la lunghezza della stringa sia esattamente 5 e in caso contrario viene richiesto il verme in input finché la condizione non sia soddisfatta.
- Chiama la funzione `ReplicaVerme()` che replica il verme e lo memorizza nell'array `chiave[]`.
- Nel `main` chiama la funzione `Decifra()` che prende in input l'array contenente il testo cifrato, l'array contenente la chiave replicata e decripta il testo usando l'eq.(2.2). Il risultato deve essere salvato in un nuovo array chiamato `frase[]`.
- Nel `main()` stampa su schermo nello stesso formato dell'esempio le tre stringhe relative al messaggio in chiaro, la chiave e il testo cifrato. Per controllare che la decifratura sia andata a buon fine, si inserisca come testo in cifrato quello dell'esempio e come chiave la parola `VERME`.

Provare a decifrare la seguente frase:
`UO OVCS HGIQUJFQ GQGUDE OGCIWIISE XWYB FKBU EJO JQP QRTCS LTQLBAOK Q UWZUNCTO H GCQIG`
usando come verme:
`CKQWC`

► **Terza parte:**

Mettere insieme le funzioni create nella prima e seconda parte per creare un codice chiamato `vigenere.c` che cripti/decripti delle frasi. Il programma avrà un menu interattivo che permetterà all'utente di scegliere se criptare o decriptare una frase.

2.2 Soluzione completa

► Soluzione della prima parte

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h> // per l'inizializzazione del seed

#define LEN 100 // lunghezza massima della frase

int ReadString(char string[]);
void CreaVerme(char v[]);
void ReplicaVerme(char v[], char k[], int N);
void Cifra(char cmsg[], char msg[], char chiave[], int N);

int main(){
    // Dichiara gli array di char con un carattere in più del necessario,
    // per tener conto del carattere di fine stringa '\0'.
    // Inizializza frase[],chiave[],cfrase[] con stringhe vuote
    char frase[LEN+1]={0},chiave[LEN+1]={0},cfrase[LEN+1]={0};
    char verme[6];
    int i,msg_len;
    srand48(time(NULL)); //usato da CreaVerme()

    printf("Questo programma usa il metodo di Vigénere per cifrare una frase immessa dall'utente usando un verme casuale di 5 lettere.\n");
    msg_len=ReadString(frase);
    CreaVerme(verme);
    ReplicaVerme(verme,chiave,msg_len);
    Cifra(cfrase,frase,chiave,msg_len);
    printf("testo chiaro:\t%s\nchiave:\t%s\ntesto cifrato:\t%s\n",frase,chiave,cfrase);
}

int ReadString(char string[]) {
    int N=0;
    char ch;
    printf("Inserire il messagio (solo maiuscole e spazi, max %d caratteri): ",LEN);
    while((ch=getchar())!='\n'){
        if(N==LEN) {
            printf("Errore: la stringa inserita supera la lunghezza massima consentita %d.\n",LEN);
            exit(1);
        }
        string[N++]=ch; //incrementa N dopo aver effettuato il comando
```

```
    }
    string[N]='\0'; // carattere di fine stringa!
    return N;
}
void CreaVerme(char v[]){
    int i;
    for(i=0;i<5;i++)
        v[i]=(drand48()*26)+'A';
    v[i]='\0'; // v[5] = carattere di fine stringa!
}
void ReplicaVerme(char v[], char c[], int N){
    int i;
    for(i=0;i<N;i++){
        c[i]=v[i%5];
    }
    c[i]='\0';//carattere di fine stringa!
}
void Cifra(char cmsg[],char msg[], char chiave[], int N){
    int i;
    for(i=0;i<N;i++){
        if(msg[i] !=' ' && msg[i]!='\0'){
            cmsg[i]=((msg[i] + chiave[i])%26) + 'A'; // uso l'eq.1 del testo
        }else{
        // non cifrare gli spazi o il carattere di fine stringa
            cmsg[i]=msg[i];
        }
    }
}
```

► **Soluzione della seconda parte**

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h> //per le funzioni strlen(),strcpy()

#define LEN 100 // lunghezza massima della frase

int ReadString(char string[]); //uguale alla prima parte
void LeggiVerme(char v[]);
void ReplicaVerme(char v[], char k[], int N); //uguale alla prima parte
void Decifra(char cmsg[], char msg[], char chiave[], int N);

int main(){
    char verme[6],frase[LEN+1]={0},chiave[LEN+1]={0},cfrase[LEN+1]={0};
    int i,msg_len;
```

```
    printf("Questo programma usa il metodo di Vigénere per decifrare una frase immessa dall'utente usando un verme di 5 lettere inserito dall'utente.\n");
    msg_len=ReadString(cfrase);
    LeggiVerme(verme);
    ReplicaVerme(verme,chiave,msg_len);
    Decifra(cfrase,frase,chiave,msg_len);
    printf("testo cifrato:\t%s\nchiave:\t%s\ntesto chiaro:\t%s\n",cfrase,chiave,frase);
}

void LeggiVerme(char v[]){
  char str[LEN]={0};
  int len=0;
  while(len!=5){
    printf("Inserire il verme di 5 caratteri maiuscoli:\n");
    scanf("%s",str);
    len=strlen(str);
    if(len!=5) printf("Errore: inseriti %d caratteri.\n",len);
  }
  strcpy(v,str);
}
void Decifra(char cmsg[],char msg[], char chiave[], int N){
    int i;
    for(i=0;i<N;i++){
        if(cmsg[i]!=' ' && cmsg[i]!='\0'){
            msg[i]=((cmsg[i] - chiave[i] + 26)%26) + 'A'; // eq.(2) del testo
        }else{
        // non cifrare gli spazi o il carattere di fine stringa
            msg[i]=cmsg[i];
        }
    }
}
```

► **Soluzione della terza parte (non commentata)**

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

#define LEN 100 // lunghezza massima della frase

int ReadString(char string[]);
void CreaVerme(char v[]);
void LeggiVerme(char v[]);
```

```c
void ReplicaVerme(char v[], char k[], int N);
void Cifra(char cmsg[], char msg[], char chiave[], int N);
void Decifra(char cmsg[], char msg[], char chiave[], int N);

int main(int argc, char* argv){
    char verme[6],frase[LEN+1]={0},chiave[LEN+1]={0},cfrase[LEN+1]={0};
    int i,msg_len, cifra_mode,res;
    srand48(time(NULL));
    printf("Questo programma usa il metodo di Vigénere per cifrare/decifrare una frase immessa dall'utente usando un verme di 5 lettere. L'utente può scegliere se cifrare il messaggio con un verme casuale oppure decifrare il messaggio con un verme immesso dall'utente.\n");
    /* Scelta fra cifratura/decifratura */
    /* (si veda l'Appendice sull'inserimento di un numero intero) */
    do{
        printf("Scegli se cifrare (1) o decifrare (2): ");
        res=scanf("%d",&cifra_mode);
        while(getchar()!='\n');
        if(res==0) printf("Errore nella conversione dell'input.\n");
        else if(cifra_mode<1 || cifra_mode>2) printf("Errore: valore dell'input non consentito.\n");
    } while(res==0 || cifra_mode<1 || cifra_mode>2);
    /* CIFRATURA */
    if (cifra_mode==1){
        msg_len=ReadString(frase);
        CreaVerme(verme);
        ReplicaVerme(verme,chiave,msg_len);
        Cifra(cfrase,frase,chiave,msg_len);
        printf("testo chiaro:\t\t%s\nchiave:\t\t%s\ntesto cifrato:\t\t%s\n",frase,chiave,cfrase);
    }
    /* DECIFRATURA */
    else{
        msg_len=ReadString(cfrase);
        LeggiVerme(verme);
        ReplicaVerme(verme,chiave,msg_len);
        Decifra(cfrase,frase,chiave,msg_len);
        printf("testo cifrato:\t\t%s\nchiave:\t\t%s\ntesto chiaro:\t\t%s\n",cfrase,chiave,frase);
    }
}
/* Si vedano le parti 1 e 2 per le definizioni delle funzioni */
...
```

2.3 Commento alla soluzione

▶ Prima parte:
Si includono le librerie: `stdio.h` per le funzioni di input/output; `stdlib.h` per la generazione di numeri casuali e la funzione `exit()`; `time.h` per l'inizializzazione casuale del *seed* del generatore di numeri casuali; `string.h` per le funzioni `strlen()` e `strcpy()`.
Si dichiara tramite una direttiva al precompilatore la lunghezza massima consentita `LEN` per la frase da decifrare, pari a 100. Di conseguenza, gli array di caratteri `frase[],cfrase[],chiave[]` devono essere dichiarati di dimensione pari a `LEN+1` per includere il carattere di fine strina `'\0'`. Tali array devono essere inoltre assegnati a `{0}`, ovvero inizializzati a stringhe vuote, in modo da poter contenere un messaggio di lunghezza inferiore a `LEN`.
Lettura della frase in chiaro tramite la funzione `ReadString()`:
Il primo punto chiede di implementare una funzione `ReadString()` che legga una stringa dal terminale, la salvi nell'array `frase[]` e restituisca il numero di caratteri letti. Nel listato seguente è mostrato il prototipo della funzione `ReadString()`, dove `char string[]` è un parametro formale e corrisponde ad un puntatore ad un array di `char`. Ci'o significa che se all'interno della funzione viene modificato il contenuto delle caselle di memoria dell'array passato in input, esse verranno modificate in maniera permanente (passaggio *by reference*).

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define LEN 100 //massima lunghezza del messaggio

void ReadString(char string[]);

int main(){
    char frase[LEN+1]={0}, chiave[LEN+1]={0}, cfrase[LEN+1]={0};
    int msg_len;
    printf("..."); //messaggio iniziale
    msg_len=ReadString(frase);
}
```

All'interno di `ReadString()` la frase può essere letta attraverso un ciclo `while()` che chiama la funzione `getchar()`, per leggere uno alla volta i caratteri inseriti nel terminale, finché non si incontra la *newline* `'\n'`. I caratteri vengono assegnati ad una variabile temporanea `ch` in modo da permettere sia il confronto con `'\n'`, sia l'assegnazione all'elemento `string[N]`, dove `N` é un contatore intero inizializzato a zero ed incrementato di 1 dopo ogni assegnazione. Se il numero di caratteri inseriti dall'utente eccede `LEN`, il programma deve terminare con `exit(1)` preceduto da un messagio di errore. Questa eventualitá deve essere verificata inserendo un'istruzione `if(N==LEN)` all'interno del ciclo `while()` e prima dell'assegnazione del carattere all'elemento dell'array. Infatti, se al limite l'utente inserisce esattamente `LEN` caratteri seguiti dalla *newline*, il ciclo `LEN`-esimo deve concludersi per consentire di leggere la *newline* all'inizio del ciclo `(LEN+1)`-esimo ed uscire. La definizione di `ReadString()` é dunque:

```
int ReadString(char string[]) {
```

```
    int N=0;
    char ch;
    printf("Inserire il messagio (solo maiuscole e spazi, max %d caratteri): ",
    LEN);
    while((ch=getchar())!='\n'){
        if(N==LEN) {
            printf("Errore: la stringa inserita supera la lunghezza massima
    consentita %d.\n",LEN);
            exit(1);
        }
        string[N++]=ch; //incrementa N dopo aver effettuato il comando
    }
    string[N]='\0'; // carattere di fine stringa!
    return N;
}
```

Creazione del verme mediante `CreaVerme()`:
Si vuole generare un verme casuale attraverso la funzione `CreaVerme()`. Essa non restituisce alcun valore ma prende in input un puntatore ad un array di `char` che conterrà il verme di 5 caratteri (più il carattere di fine stringa `'\0'`). La dichiarazione e chiamata di `CreaVerme()` saranno dunque:

```
void CreaVerme(char v[]); // equivalente: char *v oppure char v[6]

int main(){
    char verme[6];
    ...
    CreaVerme(verme);
}
```

La funzione genera 5 numeri random tra 0 e 25 (si veda l'Appendice 14) e li memorizza nelle caselle di memoria di `v[]` dopo averli sommati al codice ASCII di `'A'` (codice 65), come spiegato nel testo. In questo modo si ottengono 5 lettere maiuscole casuali:

```
void CreaVerme(char v[]) {
    int i;
    for(i=0;i<5;i++)
        v[i]=(drand48()*26)+'A';
    v[i]='\0'; // v[5]=carattere di fine stringa !!
}
```

Replica del verme mediante la funzione `ReplicaVerme()`:
Il verme deve essere replicato per la lunghezza della frase, all'interno dell' array `chiave[]`. La funzione `ReplicaVerme()` non restituisce alcun valore; riceve invece come parametri di input i puntatori all'array contenente il verme e all'array che dovrà contenere il verme replicato, ed il numero N di caratteri della frase in chiaro. La sua dichiarazione e chiamata saranno dunque:

```c
void ReplicaVerme(char v[], char k[], int N);

int main(){
    char verme[6], chiave[LEN+1]={0}, frase[LEN+1]={0};
    int msg_len;
    ...
    msg_len=ReadString(frase);
    CreaVerme(verme);
    ReplicaVerme(verme,chiave,msg_len);
}
```

La corrispondenza che si vuole ottenere fra i caratteri della chiave (lunga N) e i caratteri del verme (lungo 5) è la seguente:

```
indice della chiave: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ...
indice del verme:    0 1 2 3 4 0 1 2 3 4 0  1  2  3  ...
```

ovvero si vuole scrivere nella chiave il carattere corrispondente all'indice 0 del verme, ogni volta che l'indice della chiave è un multiplo di 5, scrivere 1 se è multiplo di 5 con resto 1, eccetera. Questo si puó ottenere utilizzando la funzione modulo 5 (`%5`), che restituisce il resto della divisione per 5; in `C` questa funzione é rappresentata dall'operatore modulo `%`. L'operatore modulo 5 (`%5`) mappa la successione degli interi nei numeri da 0 a 4, ciclicamente: 0%5=0, 1%5=1, 2%5=2, 3%5=3, 4%5=4, 5%5=0, 6%5=1, e così via. La definizione di `ReplicaVerme()` è dunque la seguente:

```c
void ReplicaVerme(char v[],char k[], int N){
    int i;
    for(i=0;i<N;i++){
        k[i]=v[i%5];
    }
    k[i]='\0';// k[N]=carattere di fine stringa!
}
```

Cifratura mediante la funzione `Cifra()`:
La funzione `Cifra()` non restituisce nulla e vuole in input, in ordine: un puntatore all'array che conterrà il messaggio cifrato, un puntatore all'array che contiene il messaggio in chiaro, la chiave (cioè il verme replicato) e la lunghezza del messaggio da cifrare. La sua dichiarazione e la sua chiamata nel `main()` saranno dunque:

```c
void Cifra(char cmsg[], char msg[], char chiave[], int N);

int main(){
    ...
    Cifra(cfrase,frase,chiave,msg_len);
}
```

La funzione cifra il messaggio usando l'eq.(2.1) del testo, facendo attenzione a non cifrare gli spazi (che rimarranno tali) né il carattere di fine stringa. Quindi:

```
void Cifra(char cmsg[],char msg[], char chiave[], int N){
    int i;
    for(i=0;i<N;i++){
        if(msg[i] !=' ' && msg[i]!='\0'){
            cmsg[i]=((msg[i] + chiave[i])%26) + 'A'; // eq.1 del testo
        }else{
        //non cifrare gli spazi né il carattere di fine stringa
            cmsg[i]=msg[i];
        }
    }
}
```

Infine, si stampa un messaggio nello stesso formato del testo, contenente frase in chiaro, chiave e frase cifrata:

```
printf("testo chiaro:\t%s\nchiave:\t%s\ntesto cifrato:\t%s\n",frase,chiave,cfrase
    );
```

► **Seconda parte:**
Rispetto al programma della prima parte, bisogna operare due cambiamenti principali:

- Il verme è letto dal terminale, attraverso una funzione che si chiamerà `LeggiVerme()`; é utile usare le funzioni `strlen()` e `strcpy()` della libreria `string.h`.
- La funzione `Decifra()` differisce dalla funzione `Cifra()` perché utilizza l'eq.(2.2) anziché l'eq.(2.1).

Lettura del verme mediante la funzione `LeggiVerme()`:
La funzione deve leggere una stringa di 5 caratteri maiuscoli (più il carattere di fine stringa `'\0'`) dal terminale e salvarla nell'array `verme`. Non restituisce nulla ed ha come unico input un puntatore al primo elemento dell'array `verme[]`, per cui il prototipo e la chiamata nel `main()` sono:

```
#include<string.h>
...
void LeggiVerme(char v[]);

int main(){
    char verme[6]; // deve contenere 5 caratteri più '\0'
    ...
    LeggiVerme(verme);
}
```

La funzione legge tutta la stringa in input con il comando `scanf("%s",str)`, dove si usa un array temporaneo `str[]` inizialmente vuoto, di lunghezza molto maggiore di 6, ad esempio `LEN`, in modo da poter contenere anche immissioni erronee di più di 5 caratteri. Una chiamata alla funzione `strlen()`

verifica se la lunghezza del messaggio sia pari a 5; in caso nefativo, si itera la richiesta all'interno di un ciclo `while()` e alla fine del ciclo si copia il verme su `v[]` mediante la funzione `strcpy()` (in alternativa si può usare un ciclo `for()` che copi i due array carattere per carattere).

```
1  void LeggiVerme(char v[]){
2    char str[LEN]={0};
3    int len=0;
4    while(len!=5){
5      printf("Inserire il verme di 5 caratteri maiuscoli:\n");
6      scanf("%s",str);
7      len=strlen(str); //restituisce la lunghezza della stringa
8      if(len!=5) printf("Errore: inseriti %d caratteri.\n",len);
9    }
10   strcpy(v,str); //copia gli elementi di str[] su v[]
11 }
```

Decifratura mediante `Decifra()`:
A differenza della funzione `Cifra()`, nel corpo di `Decifra()` il ruolo di `msg[]` e `cmsg[]` è scambiato, e nella riga 7 si usa l'eq.(2.2) del testo:

```
1  msg[i]=((cmsg[i] - chiave[i] + 26)%26) + 'A';
```

Capitolo 3

Tunnel con pioggia

3.1 Testo dell'esercizio

Una società di costruzioni sta realizzando un tunnel di 57 km attraverso una montagna per consentire il passaggio di una ferrovia. Lo scavo è eseguito con l'esplosivo. Ogni detonazione provoca lo sgretolamento di una parte della montagna corrispondente a una lunghezza del tunnel compresa tra i 24 e i 48 m.
Il programma deve simulare la realizzazione del tunnel per stimare la durata delle operazioni necessarie al suo completamento, tenendo conto di potenziali imprevisti e del fatto che le operazioni destinate alla realizzazione sono parzialmente determinate da fenomeni stocastici.

1. Nel programma dev'esserci una **funzione di nome `boom` che restituisce la lunghezza del tratto di tunnel abbattuto** in seguito a un'esplosione. Questa lunghezza è un **numero casuale compreso tra 24 e 48 m distribuito uniformemente nell'intervallo**.

2. Poiché le condizioni meteo avverse rallentano il processo, il programma **inizia chiedendo all'utente di inserire il numero intero n di giorni di maltempo previsti in media ogni mese**. Tale numero dev'essere **compreso tra 3 e 12**. Se tali condizioni non sono rispettate il programma deve reiterare la richiesta fino a quando l'utente inserisce un valore accettabile.

3. Il numero di giorni di maltempo dipende, in effetti, dalla stagione. **Per simulare la stagionalità si costruisce un array `stagione` di quattro elementi che contiene la probabilità di pioggia per ciascuna delle stagioni** primavera, estate, autunno e inverno, in questa sequenza. La probabilità p_i della stagione i è data dal rapporto $\alpha_i n/N$ con $N = 30$, n è il numero medio di giorni di maltempo per mese inserito dall'utente, e α_i che dipende dalla stagione: $\alpha_{primavera} = 0.92$, $\alpha_{estate} = 0.94$, $\alpha_{autunno} = 0.65$ e $\alpha_{inverno} = 0.58$. Per semplicità **ogni stagione ha una durata di 90 giorni**.

4. Attraverso la funzione `boom` il programma simula, **a partire dal primo giorno di primavera**, il cedimento di una parte di tunnel di lunghezza l.

5. Si definisce una **funzione `pioggia` che, sulla base del giorno corrente e della probabilità di pioggia, restituisce 1 se ha piovuto e 0 altrimenti**.

6. **La successiva detonazione si svolge quindi il giorno seguente se non ha piovuto e due giorni dopo se invece il tempo non è stato clemente**.

7. Le operazioni di scavo, a partire dal punto 4, sono reiterate fino a quando l'intero tunnel non è stato scavato.

8. **Ogni trenta giorni**, il programma informa l'utente circa il **tempo trascorso** e la **lunghezza scavata** fino a quel momento espressa in km.

9. Il programma, alla fine, indica all'utente il **numero di giorni trascorsi dall'inizio delle operazioni**.

3.2 Soluzione completa

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 30 // giorni mese
#define L_TOT 57000 // lunghezza totale in m
#define L_MIN 24 // abbattimento minimo
#define L_MAX 48 // abbattimento massimo
#define G_MIN 3 // numero minimo giorni maltempo
#define G_MAX 12 // numero massimo giorni maltempo
#define PRI 0.92 // parametro primavera
#define EST 0.94 // parametro estate
#define AUT 0.65 // parametro autunno
#define INV 0.58 // parametro inverno

float boom();
int pioggia(int giorno, float probabilita[]);

int main(){
  int i,n; // n=giorni medi di brutto tempo
  float l=0; // metri scavati
  float stagione[4]; // probabilità di pioggia
  float alpha[]={PRI,EST,AUT,INV}; // parametri stagionali
  srand48(time(NULL)); // inizializzazione seed numeri random
  ////// Inserimento n e riempimento probabilità ///////
  printf("Inserire il numero medio di giorni di maltempo previsti al mese nell'
   intervallo [%d,%d]:\n",G_MIN,G_MAX);
  do{
    scanf("%d",&n);
    if(n<G_MIN||n>G_MAX){
      printf("ERRORE, IL NUMERO NON APPARTIENE ALL'INTERVALLO [%d,%d]. REINSERIRE
   .\n",G_MIN,G_MAX);
    }
  }while(n<G_MIN||n>G_MAX); // controllo sul numero medio di giorni di maltempo
  for(i=0;i<4;i++){
    stagione[i]=alpha[i]*n/(double)N; // riempimento array probabilità
    printf("p=%f\n",stagione[i]);
  }
  ////// CICLO PRINCIPALE ////////
  i=0; // riutilizzo i per contare i giorni
  while(l<L_TOT){
    i++; // i giorni cominciano da 1
    l+=boom(); // aumento lunghezza tunnel
    if((i%N)==0){
```

```
      printf("Sono trascorsi %d giorni;\n"
       "Lunghezza scavata %.3f km.\n\n",i,l/1000.0); // stampa resoconto
    }
    // controllo pioggia (solo se il tunnel non è ancora finito)
    if((l<L_TOT) && pioggia(i+1,stagione)){
      i++; // giorno nullo
    }
  }
  // Fine ciclo di scavo
  printf("Tunnel completato: %d giorni complessivi.\n",i); // resoconto finale
}

float boom(){
  float r=L_MIN+(drand48()*(L_MAX-L_MIN)); // numero casuale tra [L_MIN,L_MAX]
  return(r);
}

int pioggia(int g, float p[]){
  int s;
  float r;
  g=(g-1)%360; //normalizzazione giorno a [0,359]
  s=g/90; //indice della stagione
  r=drand48();
  return (r<p[s]); // restituisce 1 con probabilità p[s], 0 altrimenti
}
```

3.3 Commento alla soluzione

Si includono la librerie: `stdio.h` per le funzioni di input/output; `stdlib.h` per la generazione di numeri casuali; `time.h` per l'inizializzazione casuale del seed.
Si inizia definendo con delle direttive al precompilatore tutte le costanti di interesse per il problema:

```
#define N 30 // giorni mese
#define L_TOT 57000 // lunghezza totale in metri
#define L_MIN 24 // abbattimento minimo in metri
#define L_MAX 48 // abbattimento massimo in metri
#define G_MIN 3 // numero minimo giorni maltempo
#define G_MAX 12 // numero massimo giorni maltempo
#define PRI 0.92 // parametro primavera
#define EST 0.94 // parametro estate
#define AUT 0.65 // parametro autunno
#define INV 0.58 // parametro inverno
```

1. La funzione `boom` non ha bisogno di alcun input e deve restituire un numero reale distribuito uniformemente tra `L_MIN` e `L_MAX`, dunque di tipo `double`. Si veda la Sez. 14 dell'appendice per un ripasso sulla generazione di numeri pseudo-casuali. Prima di utilizzare la funzione `drand48` bisogna sempre **inizializzare il suo seed chiamando** `srand48` **nel** `main`. La definizione della funzione è la seguente:

```
double boom(void){
    double r=L_MIN+(drand48()*(L_MAX-L_MIN));
    return(r);
}
```

2. La parte di inserimento del numero di giorni può essere effettuata agilmente con poche di righe di codice:

```
int main(){
    int n; //giorni medi di brutto tempo
    printf("Inserire il numero medio di giorni di maltempo previsti al mese
    nell'intervallo [%d,%d]:\n",G_MIN,G_MAX);
    scanf("%d",&n);
}
```

   Più sofisticata è la richiesta di dover iterare l'inserimento, producendo un messaggio d'errore, finché il numero di giorni non è compreso tra `G_MIN` e `G_MAX`. A questo scopo si può utilizzare il costrutto `do-while` avente come condizione di uscita dal ciclo l'appartenenza all'intervallo di giorni prefissato:

```
int main(){
    int n;
```

```
    printf("Inserire il numero medio di giorni di maltempo previsti al mese
    nell'intervallo [%d,%d]:\n",G_MIN,G_MAX);
    do{
        scanf("%d",&n);
        if(n<G_MIN||n>G_MAX){
            printf("ERRORE, IL NUMERO NON APPARTIENE ALL'INTERVALLO [%d,%d].
    REINSERIRE.\n",G_MIN,G_MAX);
        }
    }while(n<G_MIN||n>G_MAX); // ripetere finché n è FUORI dall'intervallo
}
```

3. Questo punto è triviale e può essere risolto in vari modi. Una prima possibilità consiste nel definire un array di 4 elementi, per poi riempirlo inserendo manualmente il valore di ogni elemento. Durante l'operazione n/N si ricordi di effettuare il **casting a** `double`, altrimenti il risultato sarà il quoziente della divisione intera di n per N.

```
int main(){
    double stagione[4] // probabilità di pioggia
    stagione[0]=PRI*n/(double)N;
    stagione[1]=EST*n/(double)N;
    ...
}
```

   Un metodo alternativo, facilmente generalizzabile al caso di più di 4 parametri, prevede di definire un secondo array contente i parametri stagionali α_i; il riempimento dell'array di probabilità può allora essere fatto all'interno di un ciclo `for` sulle 4 stagioni:

```
int main(){
    double stagione[4];
    double alpha[]={PRI,EST,AUT,INV} // parametri stagionali
    int i;
    for(i=0;i<4;i++){
        stagione[i]=alpha[i]*n/(double)N;
    }
}
```

4. Il tunnel viene costruito tramite esplosioni ripetute all'interno di un ciclo. Non si può usare un `for` in quanto non è noto a priori quanti passi sono necessari per finire l'opera; bisogna dunque optare per un ciclo `while` avente come condizione d'uscita il raggiungimento (o anche superamento) di 57 km di tunnel scavati. Definendo una variabile `l` di tipo `double` che tenga conto della lunghezza del tunnel, una prima versione del ciclo è la seguente:

```
int main(){
    double l=0; // metri di tunnel scavati
```

```
    ...
    while(l<L_TOT){ // gli scavi finiscono quando l>=57km
        l+=boom(); // esplosione: aumento casuale di l
    }
}
```

5. Il punto più pernicioso del programma è questo. La funzione che `pioggia` deve restituire un `int` pari a 0 (non piove) o 1 (piove), in base al giorno e alla probabilità del mese corrente; pertanto il prototipo della funzione deve essere (prestare attenzione al modo in cui si passa un array ad una funzione):

```
int pioggia(int giorno, double probabilità[]);
```

   Nel corpo della funzione, innanzitutto bisogna capire a quale stagione appartiene il giorno g ricevuto in input, con $g \geq 1$. Il testo assume che gli scavi comincino in primavera e che per semplicità ogni stagione sia composta da $N = 90$ giorni, pertanto un anno da 360 giorni. Si possono usare diversi approcci; se ne riportano 2 a titolo di esempio:

   - Una possibile idea consiste nel normalizzare $1 \leq g < \infty$ all'intervallo dell'anno $1 \leq g \leq 360$, ad esempio sottraendo ripetutamente 360 a g in un ciclo `while` finché non risulta $d \leq 360$. *Nota: non serve creare una seconda variabile, poiché le operazioni su g all'interno della funzione non cambiano il suo valore nel* `main`. Dopodiché, è sufficiente dividere l'anno in gruppi di 90 giorni per individuare la stagione di appartenenza, partendo dalla primavera:

```
int pioggia(int g, float prob[]){
    int stagione;
    while(g>360){
        g-=360;
    } // giorno normalizzato a [1,360]
    if(g<=90){
        stagione=0; // è primavera
    }else if(g>90 && g<=180){
        stagione=1; // è estate
    }else if(g>180 && g<=270){
    ...
}
```

   - Un'alternativa più elegante consiste nell'utilizzare l'operatore **modulo 360** per normalizzare il giorno nel primo anno. Dopodiché si divide d per 90, sfruttando il fatto che il risultato della divisione fra due variabili `int` è una variabile `int` pari alla parte intera del quoziente. Attenzione: il modulo 360 **restituisce il resto della divisione intera per 360**, che appartiene all'intervallo $[0, 359]$; dato che i giorni cominciano da 1, anziché da 0, queste operazioni devono essere applicate a **g-1**.

Esempio: si supponga di essere al giorno $g = 360$, l'ultimo dell'anno, che si trova in inverno; esso corrisponde al giorno $(g-1)\%360 = 359$. Il risultato della divisione $359/90$ fornisce un quoziente intero pari a 3, che è effettivamente l'indice della stagione giusta.
Per implementare questo metodo possiamo utilizzare il seguente listato:

```
int pioggia(int g, float p[]){
    int stagione;
    g=(g-1)%360; // normalizzazione del giorno
    stagione=d/90;
    ...
}
```

Una volta individuata la stagione bisogna determinare se ha piovuto o meno, con probabilità `p[stagione]` che in un determinato giorno piova. Per decidere se l'evento avviene o meno si estrae un numero casuale compreso tra $[0, 1]$, tramite la funzione `drand48`: se il numero è $\leq$`p[stagione]` allora ha piovuto, altrimenti no; infatti la probabilità di estrarre un numero causale compreso tra $[0, p]$ è esattamente p.

```
int pioggia(int g, float p[]){
    int stagione;
    g=(g-1)%360;
    stagione=g/90; // stagione
    if(drand48()<p[stagione]){ \\ piove?
        return(1);
    }
    return(0);
}
```

6. In questa parte bisogna inserire la funzione `pioggia` all'interno del `main` e implementare il conteggio dei giorni di lavoro, con il salto di un giorno in caso di pioggia. Ripartendo dal ciclo della parte 3, una variabile giorno `int g` viene incrementata ad ogni ciclo per contare i giorni:

```
int main(){
    int l=0; // metri scavati
    int g=0; // giorni di scavo
    srand48(time(NULL)); // inizializzazione seed numeri random
    ...
    while(l<L_TOT){
        g++; // parte da 1
        l+=boom(); // aumento lunghezza tunnel
    }
}
```

Dopo la chiamata alla funzione `boom` si controlla se il giorno successivo pioverà chiamando la funzione `pioggia` e usando il suo output come condizione per un `if`. In caso affermativo, si

incrementa ulteriormente la variabile `g` di 1 per tenere conto del giorno di inabilità.

```
int main(){
    int l=0; // metri scavati
    int g=0; // giorni di scavo
    srand48(time(NULL)); // inizializzazione seed numeri random
    ...
    while(l<L_TOT){
        g++; // contiamo i giorni partendo da 1
        l+=boom(); // aumento lunghezza tunnel
        if(pioggia(g,stagione)){
            g++; // giorno nullo
        }
    }
}
```

Si noti che `pioggia` deve ricevere in input il giorno in cui si vuole sapere se pioverà e **non** il giorno corrente.
Osservare come venga passato come argomento dell'`if` direttamente il valore restituito da `pioggia` senza utilizzare `==` o altre istruzioni. Ciò deriva dal fatto che per l'`if` un argomento intero è vero se e solo se è diverso da zero.

7. Questo punto è già stato risolto utilizzando il ciclo `while` che termina quando `l>=LTOT`.

8. Per individuare i giorni che sono multipli di $N = 30$ si usa ancora l'operatore modulo `%` controllando quando il resto della divisione sia 0. Una volta effettuato questo passaggio, bisogna solo prestare attenzione ai dettagli. Ad esempio, la stampa del resoconto deve essere fatta prima della chiamata a `pioggia`, altrimenti il giorno sarà sbagliato. Inoltre la lunghezza del tunnel deve essere espressa in km anziché in metri, quindi bisogna dividere la variabile intera `l` per 1000; per avere una divisione con decimali bisogna inserire un cast a `float`, oppure scrivere `1000.` (o `1000.0`) in modo che sia riconosciuto come `float`.

```
int l=0; // metri scavati
int g=0; // giorni di scavo
while(l<L_TOT){
    g++; // contiamo i giorni partendo da 1
    l+=boom(); // aumento lunghezza tunnel
    if((g%N)==0){
        printf("Sono trascorsi %d giorni;\n"
               "Lunghezza scavata %.3f km.\n",g,l/1000.); // stampa resoconto
    }
    if(pioggia(g+1,stagione)){
        g++; // giorno saltato
    }
}
```

9. Il messaggio finale va collocato dopo il ciclo:

```
printf("Tunnel completo: %d giorni complessivi.\n",g);
```

ATTENZIONE: un errore comune è dimenticarsi che dopo aver effettuato l'ultima detonazione viene richiamata la funzione `pioggia`, la quale può cambiare il giorno finale prima dell'uscita dal ciclo! Se il tunnel è completato è di fatto inutile vedere cosa succederà il giorno dopo: questa osservazione si può implementare con un doppio controllo nell'`if` della pioggia:

```
int main(){
    ...
    while(l<L_TOT){
        g++; // contiamo i giorni partendo da 1
        l+=boom(); // aumento lunghezza tunnel
        if((g%N)==0){
            printf("Sono trascorsi %d giorni;\n"
                   "Lunghezza scavata %.3f km.\n",g,l/1000.); // stampa
    resoconto
        }
        //non controllare il meteo se il tunnel è già finito!
        if((l<L_TOT) && pioggia(g+1,stagione)){
            g++; // giorno saltato
        }
    }
    // Fine ciclo di scavo
    printf("Tunnel completo: %d giorni complessivi.\n",d);
}
```

Capitolo 4

Ricerca del bosone di Higgs

4.1 Testo dell'esercizio

Il bosone di Higgs e' una particella fondamentale prevista dal modello standard delle particelle elementari, ed e' stata scoperta nel 2012 dagli esperimenti CMS ed ATLAS al Large Hadron Collider di Ginevra. Il bosone di Higgs ha una massa di 125 GeV. Se si studiassero N collisioni di LHC, in cui venisse prodotto sicuramente un bosone di Higgs, e di questo se ne ricostruisse la massa, si osserverebbe che gli N valori di massa misurati si distribuiscono come una gaussiana attorno al valore di $\mu = 125$ GeV e una larghezza $\sigma = 2.5$ Gev. Nella vita vera, nelle collisioni di LHC non si produce sempre e un bosone di Higgs, ma anche altre particelle non interessanti, chiamate eventi di fondo, per i quali, se si ricostruisce la massa, si osserva una distribuzione in prima approssimazione esponenziale nell'intorno dei 125 GeV. Sperimentalmente si definisce significanza del segnale del bosone di Higgs il rapporto $S/\sqrt{B}$, dove S rappresenta l'integrale della distribuzione di segnale, gaussiana, in un determinato intorno dei 125 GeV, e per B l'integrale della distribuzione esponenziale nel medesimo intorno.

Si vuole simulare la ricerca di un segnale gaussiano $G(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ dove μ è il valore centrale e σ la larghezza della Gaussiana, e con il fondo che ha una distribuzione esponenziale del tipo $B(x|\lambda) = \lambda e^{-\lambda x}$.
Il programma deve:

1. Implementare una funzione `Segnale()` che prenda come argomenti due valori `m` e `s` e restituisca una variabile con distribuzione Gaussiana con i parametri μ=`m` e σ=`s`. A tal fine, generare due numeri razionali casuali y_1 e y_2 distribuiti uniformemente nell'intervallo $(0, 1]$, e calcolare $x = \mu + \sigma \times \cos(2\pi y_1)\sqrt{-2\ln(y_2)}$ che avrà la distribuzione Gaussiana desiderata.

2. Implementare una funzione `Fondo()` che prenda come argomenti `lambda, a` e `b` e generi una distribuzione esponenziale con il parametro `lambda` nell'intervallo [`a`,`b`]. A tal fine, è necessario generare un numero casuale `p` distribuito uniformemente nell'intervallo $(e^{-\lambda b}, e^{-\lambda a})$ e restituire $x = -\frac{\ln p}{\lambda}$ che avrà la distribuzione esponenziale desiderata.

3. Acquisire dall'utente un numero intero `Nsig` di eventi di segnale aspettato, con `Nsig` nell'intervallo $[10, 30]$, e ripetere l'acquisizione in caso di errore.

4. Con un opportuno ciclo, variare il numero `Nbkg` di eventi di fondo da un minimo di 50 fino a un massimo di 200, con passo di 10.

5. Per ciascun valore di `Nbkg`, simulare `NEXP=100` esperimenti, ciascuno dei quali consiste in:
   a. Generare `Nsig` eventi di segnale distribuiti secondo una gaussiana con $\mu = 125$ e $\sigma = 2.5$;
   b. Generare `Nbkg` eventi di fondo nell'intervallo $[100, 150]$ secondo una distribuzione esponenziale con $\lambda = 0.0025$;
   c. Contare il numero di eventi di segnale e di fondo che cadono nell'intervallo $[122, 128]$ e salvarli in opportuni array `scut` e `bcut`;
6. Al termine degli esperimenti, tramite un'opportuna funzione `Analisi()` che prenda in input i due array `scut` e `bcut`, calcolare la significanza $S_{cut}/\sqrt{B_{cut}}$ per tutti gli esperimenti, e restituire il valore minimo, massimo e medio su tutti gli esperimenti.
7. Nella funzione `Main()`, stampare sullo schermo i valori restituiti dalla funzione `Analisi()`, con un formato simile a quello riportato:

```
Ricerca di segnale gaussiano (mu=125.0, sig=2.5) e fondo esponenziale(lambda=2.5e-03) in [100,150]
Numero di eventi di segnale in [10,30]:13
segnale: 13     fondo: 50      Significanza min: 2.67         max: 6.50     media: 4.00
segnale: 13     fondo: 60      Significanza min: 2.29         max: 5.81     media: 3.61
```

8. Scrivere questi valori numerici (una riga per ciascun valore `Nbkg`) su un file `risultati.txt` per fare i grafici con Python. Graficare i valori min, max e medio della significanza in funzione di `Nbkg`.

4.2 Soluzione completa

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>

// Costanti
#define LAMBDA 0.0025
#define NSMIN 10
#define NSMAX 30
#define NBMIN 50
#define NBMAX 200
#define dNBKG 10
#define XMIN 100.
#define XMAX 150.
#define TAGLIO 5.
#define NEXP 100
#define SIGMA 2.5
#define M0 125.

// Definizione funzioni
float Segnale(float mu, float sig);
float Fondo(float lambda, float xmin, float xmax);
void Analisi(int* s, int* b, int n, float* smin, float* smax, float* smedia);

//INIZIO PROGRAMMA
int main(){
    int Nsig, Nbkg;
    int i, j, Bcut[NEXP], Scut[NEXP];
    float massa;
    float smedia, smin, smax;
    FILE *fp;

    srand48(time(NULL)); // Inizializzazione seed
    printf("Ricerca di segnale gaussiano (mu=%.1f, sig=%.1f) e fondo esponenziale
    "
        "(lambda=%.1e) in [%.0f,%.0f]\n",M0,SIGMA,LAMBDA,XMIN, XMAX); //
    Messaggio iniziale
    printf("Numero di eventi di segnale in [%d,%d]:",NSMIN,NSMAX); // Messaggio
    richiesta input
    // Inserimento input
    do{
        scanf("%d",&Nsig);
        // Messaggio di errore
        if(Nsig<NSMIN || Nsig>NSMAX){
```

```
42              printf("ERRORE: VALORE INSERITO FUORI DAL RANGE [%d,%d]",NSMIN,NSMAX)
    ;
43      }
44      }while(Nsig<NSMIN || Nsig>NSMAX); // Controllo per reimmissione
45      fp=fopen("risultati.txt","w"); // Apertura file di scrittura
46      fprintf(fp,"#segnale \t fondo \t Sign. min. \t Sign. max. \t Sign. media\n");
        // Stampa nomi delle colonne (preceduti da #)
47
48      // Ciclo incremento fondo
49      for(Nbkg=NBMIN; Nbkg<=NBMAX; Nbkg+=dNBKG) {
50          // Ciclo esperimenti
51          for(i=0; i<NEXP; i++) {
52          Scut[i] = Bcut[i] = 0; // Inizializzazione array
53              // Ciclo segnale
54              for(j=0; j<Nsig; j++) {
55                  massa = Segnale(M0, SIGMA); // Generazione segnale
56                  if( fabs(massa-M0) <= TAGLIO ) Scut[i]++; // Controllo sul
    segnale
57              }
58              // Ciclo fondo
59              for(j=0; j<Nbkg; j++) {
60                  massa = Fondo(LAMBDA, XMIN, XMAX); // Generazione fondo
61                  if( fabs(massa-M0) <= TAGLIO ) Bcut[i]++; // Controllo sul fondo
62              }
63          }
64          Analisi(Scut, Bcut, NEXP, &smin, &smax, &smedia); // Calcolo significanza
65          printf("segnale: %d \t fondo: %d \t Significanza min: %.2f \t max: %.2f \
    t media: %.2f\n", Nsig, Nbkg, smin, smax, smedia); // Stampa dei risultati
66          // Stampa su file
67          fprintf(fp,"%d \t %d \t %.2f \t %.2f \t %.2f\n",Nsig, Nbkg, smin, smax,
    smedia);
68      }
69      // Fine cicli
70      fclose(fp); // Chiusura file
71      return 0;
72  }
73  // FINE PROGRAMMA
74
75  // FUNZIONI
76  float Segnale(float mu, float sig){
77  // Funzione per generare numeri distribuiti gaussianamente (mu,sig)
78      float y1, y2;
79      do{
80          y1 = (float) lrand48()/RAND_MAX;
81      } while(y1==0.); // Controllo numero diverso da 0
82      do{
```

```
83          y2 = (float) lrand48()/RAND_MAX;
84      } while(y2==0.); // Controllo numero diverso da 0
85      return mu + sig*cos(2*M_PI*y1)*sqrt(-2*log(y2));
86  }
87
88  float Fondo(float lambda, float a, float b){
89  // Funzione per generare numeri distribuiti esponenzialmente con parametro lambda
        tra [a,b]
90      float x,y;
91      float pdfa = exp(-lambda*a);
92      float pdfb = exp(-lambda*b);
93      do{
94          y = pdfb+(pdfa-pdfb)*lrand48()/RAND_MAX;
95      } while( y==pdfa || y==pdfb );
96      x = - log(y)/lambda; return x;
97  }
98
99  void Analisi(int* S, int* B, int n, float* min, float* max, float* avg){
100     int i;
101     float sig;
102     *min = 100;
103     *max = 0.;
104     *avg = 0.0;
105     for(i=0; i<n; i++){
106         sig = S[i]/sqrt(B[i]); // Significanza
107         *avg += sig; // Somma dei segnali
108         if(sig>*max) *max = sig; // Ricerca massimo
109         if(sig<*min) *min = sig; // Ricerca minimo
110     }
111     *avg /= n; return; // Calcolo media
112 }
```

► **Listato in Python per il grafico**

```python
import numpy as np
import matplotlib.pyplot as plt

Nsig, Nbkg, Smin, Smax, Savg = np.loadtxt("higgs.dat", unpack=True)

plt.plot(Nbkg, Smin, "o", label="min")
plt.plot(Nbkg, Smax, "o", label="max")
plt.plot(Nbkg, Savg, "o", label="average")

plt.xlabel("N. of background events $N_{bkg}$")
plt.ylabel("Significance")
plt.title("N. of signal events $N_{sig}=$%d"%Nsig[0])
plt.legend()
plt.savefig("higgs.png")
```

4.3 Commento alla soluzione

1. Il primo punto è alquanto semplice, l'importante è l'uso della funzione `drand48()` che, ricordiamo, genera un numero `double` casuale compreso tra $[0, 1]$. Siccome vogliamo un numero tra $(0, 1]$ bisogna inserire un controllo per escludere lo 0 nel caso venga estratto (evento raro ma non impossibile). La funzione può essere scritta in questo modo:

```
float Segnale(float mu, float sig){
    double y1,y2;
    do{
        y1 = drand48();
    }while(y1==0.); // Controllo numero diverso da 0
    do{
        y2 = drand48();
    } while(y2==0.); // Controllo numero diverso da 0
    return mu + sig*cos(2*M_PI*y1)*sqrt(-2*log(y2));
}
```

2. Questo punto è praticamente identico al precedente, bisogna solo prestare attenzione alla generazione del numero tra $\left[e^{-\lambda b}, e^{-\lambda a}\right]$ (notare gli estremi esclusi):

```
float Fondo(float lambda, float a, float b){
    float x,y;
    float pdfa = exp(-lambda*a);
    float pdfb = exp(-lambda*b);
    do{
        y=pdfb+(pdfa-pdfb)*drand48();
    }while( y==pdfa || y==pdfb );
    x = - log(y)/lambda; return x;
}
```

3. Cominciamo a costruire il `main()` del programma. Definiamo per comodità due costanti `NSMIN` e `NSMAX` che valgono rispettivamente 10 e 30. Una possibile soluzione per questo punto può essere:

```
#define NSMIN 10
#define NSMAX 30

int main(){
    int Nsig;
    do{
        scanf("%d",&Nsig);
        // Messaggio di errore
        if(Nsig<NSMIN || Nsig>NSMAX){
            printf("ERRORE: VALORE INSERITO FUORI DAL RANGE [%d,%d]",NSMIN,
   NSMAX);
```

```
        }
    }while(Nsig<NSMIN || Nsig>NSMAX); // Controllo per reimmissione
}
```

4. Definiamo le costanti `NBMIN`, `NBMAX` e `dNBKG` che valgono rispettivamente 50, 200 e 10. Scriviamo il ciclo per incrementare la variabile `Nbkg` con un `for`:

```
#define NSMIN 10
#define NSMAX 30
#define NBMIN 50
#define NBMAX 200
#define dNBKG 10

int main(){
    ...
    int Nbkg;
    ...
    // Ciclo incremento fondo
    for(Nbkg=NBMIN;Nbkg<=NBMAX;Nbkg+=dNBKG){

    }
}
```

Alternativamente si può utilizzare anche un ciclo `while/do` nel seguente modo:

```
    ...
int main(){
    ...
    Nbkg=NBMIN;
    while(Nbkg<=NBMAX){
        Nbkg+=dNBKG;
    }
}
```

5. Entriamo nel cuore del programma. Definiamo la costante `NEXP` e diamole valore 100. Costruiamo un ciclo `for`, interno a quello precedente che scandisce ogni esperimento:

```
int main(){
    ...
    int i;
    // Ciclo incremento fondo
    for(Nbkg=NBMIN;Nbkg<=NBMAX;Nbkg+=dNBKG){
        // Ciclo esperimento
        for(i=0;i<NEXP;i++){
```

```
            // Codice Esperimento
        }
        // Fine esperimento
    }
    // Fine ciclo fondo
}
```

Vediamo passo passo come implementare questa parte.

a. Per svolgere questa parte possiamo utilizzare la funzione `Segnale()` implementata al punto 1. All'interno del ciclo `for()` della variabile `i` inseriamo un ulteriore ciclo per la generazione del "segnale" (la variabile viene chiamata, non casualmente, `massa`). Definiamo anche qui per comodità `M0` e `SIGMA` come 125 e 2.5, utilizzando una direttiva `define`:

```
...
int i,j;
float massa;
for(i=0;i<NEXP;i++){
    for(j=0;j<Nsig;j++){
        massa=Segnale(M0,SIGMA); // Generazione segnale
    }
}
```

b. Questo punto è identico al precedente solo che va usata la funzione `Fondo()`. Come sopra, si definiscono tramite una direttiva `define` le costanti `LAMBDA`, `XMIN` e `XMAX` rispettivamente 0.0025, 100 e 150. Vedremo che non è necessaria definire una nuova variabile per il rumore, ma si può utilizzare sempre `massa`, sovrascrivendo il valore precedente dopo aver consegnato l'evento::

```
...
int i,j;
float massa;
...
for(i=0;i<NEXP;i++){
    // Ciclo segnale
    for(j=0;j<Nsig;j++){
        massa=Segnale(M0,SIGMA); // Generazione segnale
    }
    // Ciclo fondo
    for(j=0; j<Nbkg; j++) {
        massa=Fondo(LAMBDA, XMIN, XMAX); // Generazione fondo
    }
}
...
```

c. Vogliamo ora salvare i valori generati nei punti precedenti in due array separati. Innanzitutto definiamo i due array che siccome devono contenere i dati di tutti gli esperimenti avranno lunghezza pari a `NEXP`. Inoltre questi array servono per tenere il conto degli eventi significativi quindi possiamo definirli anche come interi (`int`):

```
int main(){
...
int Scut[NEXP],Bcut[NEXP];
}
```

Dopo aver generato il numero casuale e averlo salvato in `massa`, inseriamo con un `if` il controllo se il valore rientra nell'intervallo richiesto. In caso affermativo incrementiamo il contatore `Scut` o `Bcut` di uno, a seconda che si tratti di un evento di segnale o di fondo. Siccome il valore dell'array deve essere incrementato è fondamentale ricordarsi di inizializzare l'array a zero prima di iniziare il conteggio. Questo può essere fatto o in fase di dichiarazione o all'inizio di ogni ciclo di esperimenti come in questo caso:

```
...
for(i=0; i<NEXP; i++) {
    Scut[i] = Bcut[i] = 0;
    for(j=0; j<Nsig; j++) {
        massa = Segnale(M0, SIGMA);
        if(massa>=122 && massa<=128) Scut[i]++; // Controllo sul segnale
    }
    for(j=0; j<Nbkg; j++) {
        massa = Fondo(LAMBDA, XMIN, XMAX);
        if(massa>=122 && massa<=128) Bcut[i]++; // Controllo sul fondo
    }
}
```

Come si può notare non è necessario definire due variabili diverse per salvare i risultati di `Segnale()` e `Fondo()` in quanto servono solo come input per i relativi `if` e non vengono utilizzate al di fuori dei cicli. Fare inoltre attenzione al fatto che la condizione dell'`if` è espressa utilizzando l'operatore `&&` (`AND`) e non l'operatore `||` (`OR`). Questo perché si vuole che il numero sia contemporaneamente ≥ 122 e ≤ 128.
Si può anche implementare in modo leggermente più elegante la condizione espressa dai due `if`, notando che l'intervallo dato è centrato esattamente intorno a 125, che è il valore di `M0`. Il valore generato casualmente va conteggiato se non eccede `M0` di 3 in eccesso o in difetto, ovvero se `|massa-M0|<=3`, il che può essere scritto utilizzando la funzione `fabs()`:

```
...
for(i=0; i<NEXP; i++) {
    Scut[i] = Bcut[i] = 0;
    for(j=0; j<Nsig; j++) {
        massa = Segnale(M0, SIGMA);
        if(fabs(massa-M0)<=TAGLIO) Scut[i]++; // Controllo segnale
```

```
    }
    for(j=0; j<Nbkg; j++) {
        massa = Fondo(LAMBDA, XMIN, XMAX);
        if(fabs(massa-M0)<=TAGLIO) Bcut[i]++; // Controllo fondo
    }
}
```

dove la costante `TAGLIO` è stata precedentemente posta $= 3$ tramite una direttiva `define`. Per chi volesse fare ancora più economia di variabili, la variabile `massa` può essere omessa, utilizzando direttamente la chiamata a funzione all'interno dell'`if`.

6. Implementiamo la funzione `Analisi()` e poi vediamo dove posizionarla nel `main()`. Già leggendo il testo è evidente che i valori richiesti non possono essere forniti tutti quanti come output dalla funzione, che può restituirne al massimo uno. In questi casi la cosa più immediata è passare alla funzione i puntatori a tre variabili e riempire queste tre variabili con i valori richiesti . Nel nostro caso dichiariamo il prototipo di `Analisi()` in questo modo:

```
void Analisi(int* s, int* b, int n, float* smin, float* smax, float* smedia);
```

I primi due parametri di input sono quelli richiesti dal testo e il terzo è la dimensione degli array. La funzione non è particolarmente difficile da scrivere. Bisogna effettuare un ciclo `for` su tutti gli esperimenti (quindi da 1 fino a `NEXP`, o meglio da 0 fino a `NEXP-1` in modo che l'indice coincida con l'indice degli array) e calcolare ciascuna significanza S_{cut}`[i]`$/\sqrt{B_{cut}}$`[i]`. I valori `massimo, minimo e medio` possono essere calcolati all'interno dello stesso ciclo; questo consente di salvare le significanze in una singola variabile temporanea anziché in un array lungo `NEXP`.
Un possibile modo di ottenere il minimo e massimo è partire da due valori `smin`,`smax` molto maggiori o molto minori del valore atteso, ad esempio 100,0 che sono casi estremi opposti e ogni volta confrontare il valore ottenuto con questi, per stabilire se sia più piccolo o più grande, e in caso affermativo sostituirli. Per la media si può inizializzare una variabile `media` a 0, sommare ad ogni iterazione la significanza, e dividere la somma alla fine del ciclo per il numero di termini (nel nostro caso `NEXP`).

```
void Analisi(int* S, int* B, int n, float* smin, float* smax, float* smedia){
    int i;
    float sig;
    *smin = 100;
    *smax = 0.;
    *smedia = 0.0;
    for(i=0; i<n; i++){
        sig = S[i]/sqrt(B[i]);
        *smedia += sig;
        if(sig>*smax) *smax = sig; // Ricerca massimo
        if(sig<*smin) *smin = sig; // Ricerca minimo
    }
    *smedia /= n; return; // Calcolo media
```

```
}
```

I valori assegnati inizialmente a `smin` e `smax` servono solo a fare in modo che al primo ciclo le condizioni degli `if` siano soddisfatte, altrimenti non avremo controllo sul valore del massimo e del minimo.
La funzione va inserita alla fine del ciclo degli esperimenti, ma all'interno di quello che varia `Nbkg` in modo da poter salvare i valori di interesse prima del prossimo ciclo di esperimenti:

```
int main(){
...
    float smedia, smin, smax;
...
    for(Nbkg=NBMIN; Nbkg<=NBMAX; Nbkg+=dNBKG) {
        for(i=0; i<NEXP; i++) {
            ... // Codice esperimento
        }
        Analisi(Scut, Bcut, NEXP, &smin, &smax, &smedia); // Calcolo
    significanza
    }
}
```

7. Per come è stata impostata la funzione `Analisi()`, la stampa dei valori va effettuata nella riga successiva; all'interno della stringa di formato si utilizza il separatore `\t` per tabulare le colonne come richiesto dal testo:

```
for(Nbkg=NBMIN; Nbkg<=NBMAX; Nbkg+=dNBKG) {
    for(i=0; i<NEXP; i++) {
        ... //Codice esperimento
    }
    Analisi(Scut, Bcut, NEXP, &smin, &smax, &smedia);
    printf("segnale: %d \t fondo: %d \t Significanza min: %.2f \t max: %.2f \
    t media: %.2f\n", Nsig, Nbkg, smin, smax, smedia); // Stampa dei
    risultati
}
```

Nel caso si siano usati gli array, si possono stampare i valori anche alla fine con un ulteriore ciclo `for`, ma questa rimane comunque la scelta migliore in termini di tempo.
La parte iniziale del messaggio va stampata all'inizio del codice e prima di chiedere l'immissione di `Nsig` (punto 3):

```
... // Parte di definizione delle variabili
printf("Ricerca di segnale gaussiano (mu=%.1f, sig=%.1f) e fondo
    esponenziale"
      "(lambda=%.1e) in [%.0f,%.0f]\n",M0,SIGMA,LAMBDA,XMIN, XMAX); //
    Messaggio iniziale
```

```
printf("Numero di eventi di segnale in [%d,%d]:",NSMIN,NSMAX); // Messaggio
    richiesta input
do{
    scanf("%d",&Nsig);
    if(Nsig<NSMIN || Nsig>NSMAX){
        printf("ERRORE: VALORE INSERITO FUORI DAL RANGE [%d,%d]",NSMIN,NSMAX)
    ;
    }
}while(Nsig<NSMIN || Nsig>NSMAX);
```

8. Per questa parte può essere utile rivedere la Sezione 11. Dichiariamo un puntatore a file in modo da stampare i valori che vogliamo su un file che chiameremo ad esempio `risultati.txt`. Per stampare i valori possiamo inserire `fprintf` dopo il `printf` del punto precedente con il quale stampiamo i vari resoconti. Nel file devono comparire solo le colonne con i dati senza testo.

```
int main(){
    ...
    FILE *fp;
    ...
    fp=fopen("risultati.txt","w"); // Apertura file
    for(Nbkg=NBMIN; Nbkg<=NBMAX; Nbkg+=dNBKG){
        for(i=0; i<NEXP; i++) {
        ... // Codice esperimento
        }
    Analisi(Scut, Bcut, NEXP, &smin, &smax, &smedia);
    printf("segnale: %d \t fondo: %d \t Significanza min: %.2f \t max: %.2f \
    t media: %.2f\n", Nsig, Nbkg, smin, smax, smedia);
    fprintf(fp,"%d \t %d \t %.2f \t %.2f \t %.2f\n",Nsig, Nbkg, smin, smax,
    smedia); // Stampa su file
    }
    fclose(fp); // Chiusura file
}
```

Eventualmente, per chiarezza, si può inserire una prima riga con i titoli delle colonne premettendo il simbolo # (si veda il listato completo).
Per creare il grafico con Python si veda il listato a fine Sezione 4.2, e/o la Sezione 16.

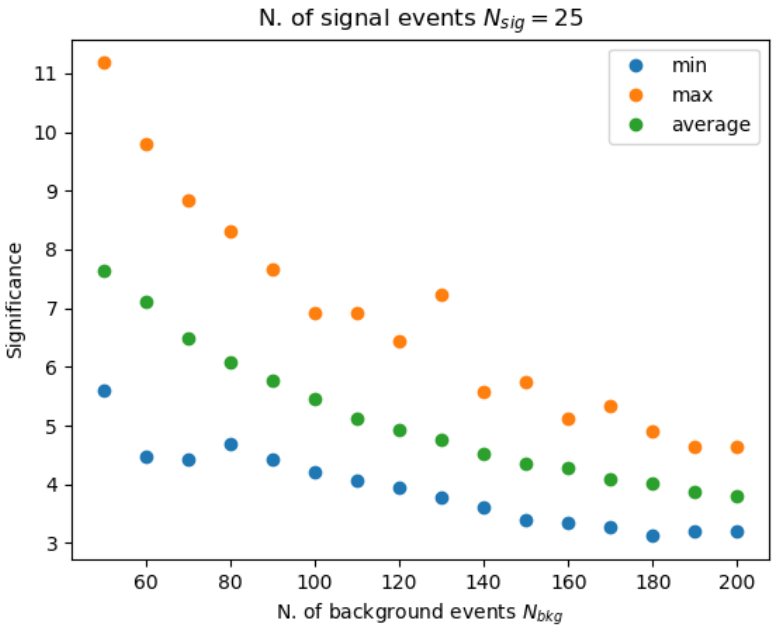


Figura 4.1: Esempio di grafico finale.

Capitolo 5

Esperimento Auger

5.1 Testo dell'esercizio

▸ Prima parte:
Auger è un esperimento per rivelare raggi cosmici di alte energie ($> 10^{17}$ eV) che si trova nella Pampa Argentina ed ha una superficie di rivelazione di $60 \cdot 60$ km^2, grande cioè quasi come l'intera Val d'Aosta. I rivelatori consistono in *tank* cilindrici di acqua di 1 m di raggio, distanziati 2 km l'uno dall'altro. Pertanto sono disponibili un totale di 900 rivelatori disposti su una griglia di 30 righe per 30 colonne. Ciascun rivelatore registra un segnale se viene colpito da un raggio cosmico. Assumendo

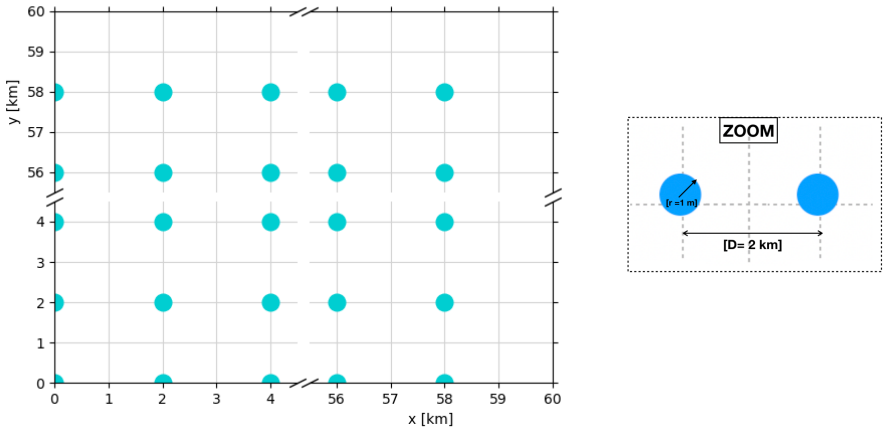


Figura 5.1: Disposizione dei rivelatori per l'esperimento Auger. Si noti che le dimensioni dei rivelatori non sono realistiche: il loro raggio dovrebbe essere un millesimo della quadrettatura.

che i raggi cosmici arrivino Uniformmente sull'intera superficie dell'osservatorio, determinare quanti segnali vengono registrati in funzione del numero di raggi cosmici incidenti. Si consideri il territorio di 3600 km^2 in un sistema di riferimento (x, y), avente l'origine in $(0, 0)$ e i lati sui semiassi positivi. I rivelatori sono posti nelle coordinate (i, j) come indicato in figura.

Il programma `auger.c` deve:

- Scrivere una breve descrizione di cosa si vuole calcolare.
- Chiedere all'utente di inserire attraverso la tastiera il numero N_0 di raggi cosmici incidenti, con $N_0 \in [1000, 1100]$.
- Nel caso in cui il numero N_0 non appartenga all'intervallo richiesto, stampare un messaggio d'errore e consentire all'utente di reimmettere il numero.
- Contenere una funzione `PlaceDetectors()` che prenda in ingresso due array vuoti, `x[900]` e `y[900]` e li riempia appropriatamente con le coordinate x_i e y_i di ciascuno dei 900 rivelatori mostrati in figura.
- Per ciascuno degli N_0 raggi cosmici:
  - Generare tramite una funzione `GenPosition()` la posizione (x, y) del raggio cosmico all'interno della superficie dell'osservatorio $60 \cdot 60$ km^2. Le coordinate x e y devono essere variabili di tipo `double`.
  - Tramite una funzione `DetectCosmic()` che riceva in input le coordinate del raggio generato e gli array con le coordinate dei rivelatori, determinare se il raggio colpisce un rivelatore o meno; la funzione deve restituire 1 in caso affermativo, altrimenti 0. Il raggio colpisce il rivelatore i-esimo se cade all'interno della sua superficie, ovvero nel cerchio di raggio 1 m centrato nelle coordinate x_i, y_i.
- Calcolare quanti raggi cosmici S_0 sono stati rivelati e stampare su schermo il risultato.

► **Seconda parte:**

- A partire da N_0 variare il numero di raggi cosmici incidenti per $N_i = N_0 + i \cdot 200$ con $i \in [0, 10]$ e calcolare per ciascun N_i il numero di segnali registrati S_i.
- Scrivere in un file `eventi_misurati.txt` l'indice i, il numero di raggi cosmici prodotti N_i e il corrispondente numero di rivelatori accesi S_i.
- Graficare infine S_i in funzione di N_i tramite uno script in python chiamato `auger.py`. In questo grafico deve essere presente una legenda, un titolo e dei nomi opportuni per gli assi. Salvare l'immagine in un file `auger.png`.

5.2 Soluzione completa

▶ **Listato di** `auger.c`

```c
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<math.h>

#define L 60.0 //km, lato del quadrato
#define D 2.0 //km, distanza minima fra i centri di 2 rivelatori
#define R 0.001 //km, raggio di ciascun rivelatore
#define ND 900 //numero di rivelatori
#define NMIN 1000 //n.min. di raggi cosmici
#define NMAX 1100 //n.max. di raggi cosmici
#define NEXP 10 //n. di esperimenti (parte 2)

void PlaceDetectors(int*, int*);
double Uniform(double, double);
void GenPosition(double*, double*);
int DetectCosmic(double, double, int*, int*);

int main() {
  int N0,S0=0, res=0, xD[ND], yD[ND], r, hit, i,N,S;
  double x,y;
  FILE* fp;
  srand48(time(NULL));

  /* PRIMA PARTE */
  printf("Benvenuti alla simulazione dell'esperimento Auger!\n"
         "Questo programma genera N raggi cosmici incidenti su una superficie
    quadrata di lato L=%.0f km, su cui sono disposti in un reticolo quadrato ND=%d
     rivelatori circolari di raggio R=%.3f km.\n"
         "Restituisce il numero S di raggi cosmici che hanno colpito i rivelatori
    .\n\n",L,ND,R);

  while( res==0 || N0<NMIN || N0>NMAX ){
    printf("Inserire il numero N0 di raggi cosmici incidenti, compreso fra %d e %
    d: ", NMIN, NMAX);
    res=scanf("%d", &N0);
    if(res==0) printf("Errore nella conversione dell'input.\n");
    else if (N0<NMIN || N0>NMAX) printf("Errore: valore di N0 non accettabile.\n"
    );
    while(getchar()!='\n'); //svuota il buffer
  }
  /* Posiziona le coordinate dei rivelatori nella griglia */
  PlaceDetectors(xD,yD);
```

```
  /* Genera N0 raggi cosmici e conta il numero di segnali */
  fp=fopen("raggi.dat","w");
  for(r=0; r<N0; r++) {
    GenPosition(&x,&y);
    fprintf(fp,"%f %f\n",x,y);
    hit = DetectCosmic(x,y, xD,yD);
    if(hit) S0++;
  }
  fclose(fp);
  printf("Fine dell'esperimento: Con %d raggi cosmici incidenti si accendono %d
   rivelatori.\n\n", N0,S0);

  /* SECONDA PARTE */
  /* Varia N a partire da N0 e stampa su file i,N,S */
  fp=fopen("eventi_misurati.txt", "w");
  fprintf(fp, "%d %d %d\n",0,N0,S0); // i=0
  for(i=1; i<NEXP; i++){// i>=1
    N=N0+i*200;
    S=0;
    for(r=0; r<N; r++) {
      GenPosition(&x,&y);
      hit = DetectCosmic(x,y, xD,yD);
      if(hit) S++;
    }
    fprintf(fp, "%d %d %d\n",i,N,S);
  }
  fclose(fp);
}

void PlaceDetectors(int* xarr, int* yarr){
  int i,j,index=0;
  FILE *fp;
  fp=fopen("detectors.dat","w");
  for(i=0;i<L;i+=D){
    for(j=0;j<L;j+=D){
      xarr[index] = i;
      yarr[index] = j;
      fprintf(fp,"%d %d\n",i,j);
      index++;
    }
  }
  fclose(fp);
}
double Uniform(double a, double b) {
  return a + (b-a)*drand48();
}
```

```
void GenPosition(double* x, double* y) {
  *x = Uniform(0.,L); // assegnazione tramite puntatore
  *y = Uniform(0.,L);
}
int DetectCosmic(double x, double y, int* xD, int* yD) {
  float dist;
  for(int i=0; i<ND; i++) {
    dist = sqrt( (x-xD[i])*(x-xD[i]) + (y-yD[i])*(y-yD[i]) );
    if(dist<R) return 1;
  }
  return 0; // nessun rivelatore colpito
}
```

► **Listato di** `auger.py`

```
import numpy as np
import matplotlib.pyplot as plt

i,N,S = np.loadtxt("eventi_misurati.txt", unpack=True)

plt.plot(N,S,"o",label="esperimento")
plt.xlabel("N (n. di raggi incidenti)")
plt.ylabel("S (n. di raggi misurati)")
plt.title("Esperimento Auger")
plt.legend()
plt.savefig("auger.png")
plt.show()
```

5.3 Commento alla soluzione

► **Prima parte:**
L'obiettivo della prima parte è calcolare quanti raggi cosmici S_0 colpiscono i rivelatori, se sulla superficie totale ne arrivano N_0 distribuiti Uniformmente. In particolare, il programma deve calcolare per ogni raggio cosmico se il suo punto d'impatto (x, y) cade all'interno di un rivelatore di coordinate (i, j). Poiché i rivelatori coprono un'area circolare di raggio 1 m, è sufficiente verificare se la distanza $d = \sqrt{(x-i)^2 + (y-j)^2}$ sia minore di 1 m.
Si includono innanzitutto le librerie: `stdio.h`; `stdlib.h` e `time.h` per generare numeri pseudocasuali; `math.h` per la funzione `sqrt()` che restituisce la radice quadrata dell'argomento. É sempre conveniente dichiarare al precompilatore i parametri costanti della simulazione attraverso la direttiva `define`: ad esempio i limiti dell'intervallo per N_0, il lato $L = 60$ km della regione quadrata contenente i rivelatori, la distanza fra i rivelatori $D = 2$ km ed il loro raggio $R = 1$ m.

```
#include <stdlib.h>
#include <stdio.h>
```

```c
#include <time.h>
#include <math.h>

#define NMIN 1000
#define NMAX 1100
#define L 60 //km, lato del quadrato contenente i rivelatori
#define D 2 //km, distanza minima fra i rivelatori
#define R 0.001 //km, raggio dei rivelatori
```

Per l'inserimento del valore di N_0 da terminale si rimanda all'Appendice 12.
La funzione `PlaceDetectors()` riceve in ingresso due array `xD[]`, `yD[]` e li riempie con le coordinate dei rivelatori. Poiché non restituisce alcun valore, deve essere dichiarata di tipo `void`. Dalla descrizione dell'apparato e dall'immagine si evince che i detector sono disposti su una griglia rettangolare 30×30 km^2, sui punti aventi coordinate intere (i, j), con $i, j = 0, 2, 4, \ldots, 58$ km. Quindi tali array possono essere dichiarati equivalentemente di tipo intero o a virgola mobile, con una lunghezza prefissata $N_D = 900$ definita anch'essa tramite la direttiva `define`. Come parametri di `PlaceDetectors()` é sufficiente passare i due array `xD[]`, `yD[]`, dal momento che la lunghezza di entrambi è accessibile globalmente.

```c
#define ND 900 //numero di rivelatori
void PlaceDetectors(int *, int *);

int main(){
    int N,xD[ND], yD[ND]; //coordinate dei rivelatori
    ... //inserimento di N0
    PlaceDetectors(xD,yD);
}
```

La funzione `PlaceDetectors()` deve scorrere lungo 3 variabili: l'indice del rivelatore `index`$= 0, 1, \ldots, 899$ e le coppie di coordinate $i, j = 0, 2, \ldots, 58$. Un possibile metodo prevede due cicli `for()` innestati in cui si incrementano i e j ciascuna di un passo D per volta, mentre `index` viene inizializzata a 0 ed incrementata di 1 nel loop più interno; `index` assume cosí tutti i valori interi fino a $30 \times 30 - 1 = 899$, come desiderato. Le coordinate i, j dell'`index`-esimo rivelatore vengono salvate rispettivamente in `x[index]` e `y[index]`.

```c
#define L 60 //km
#define D 2  //km
void PlaceDetectors(int* xarr, int* yarr){
  int i,j,index=0;
  for(i=0;i<L;i+=D){
    for(j=0;j<L;j+=D){
      xarr[index] = i;
      yarr[index] = j;
      index++;
    }
  }
```

```
}
```

La funzione `GenPosition()` deve assegnare un valore pseudocasuale a 2 variabili `double` x, y definite nel `main`. Tuttavia in C una funzione non può restituire più di una variabile in output. Bisogna optare piuttosto per una funzione `void` che assegni il valore ad x e y al suo interno *by reference*, cioè tramite puntatori. Si fa riferimento all'Appendice 14 per l'implementazione della funzione, di cui é mostrata di seguito la definizione e la chiamata dal `main()`. Si ricordi di inizializzare il *seed* del generatore di numeri pseudocasuali una volta sola all'interno del `main()`, prima della chiamata di `GenPosition()`.

```
double Uniform(double, double);
void GenPosition(int*, int*);
int main(){
    int r;
    double x,y;
    srand48(time(NULL));//inizializzazione del seed
    ...
    for(r=0;r<N0;r++){
        GenPosition(&x,&y); //passaggio 'by reference' di x e y
        ...
    }
}
```

Per migliorare la leggibilitá e la portabilitá del codice, si é scelto di utilizzare una funzione ausiliaria `double Uniform(float a, float b)` che genera un numero reale pseudocasuale distribuito uniformemente nell'intervallo $[a, b]$ a partire da un numero Uniform fra 0 e 1.
Suggerimento: per verificare di aver generato correttamente le posizioni dei rivelatori e dei raggi cosmici, si consiglia di salvare le coordinate di entrambi, in due file separati, e di graficarli con `Python`.
La funzione `DetectCosmic()` riceve in input gli array `xD[]`, `yD[]` e le coordinate x, y del raggio e restituisce un numero intero (1 se il raggio colpisce un rivelatore, 0 altrimenti). Le coordinate x, y possono essere passate *by value*, dato che non vengono modificate. Nel `main()` si inizializza un contatore $S_0 = 0$ e lo si incrementa nel loop principale di N_0 cicli, ogni volta che `DetectCosmic()` restituisce un 1 (si noti che le istruzioni `if(hit==1)` e `if(hit)` sono equivalenti se `hit` puó assumere solo i valori 0,1).

```
...
int int DetectCosmic(double, double,int*, int*);
int main(){
    int S0=0, hit, xD[ND],yD[ND];
    ...
    for(int r=0;r<N0;r++){
        GenPosition(&x,&y);
        hit=DetectCosmic(x,y,xD,yD);
        if(hit) S0++;
    }
    printf("Fine dell'esperimento: Con %d raggi cosmici incidenti si accendono %d
     rivelatori\n", N0,S0);
```

```
}
```

Come anticipato all'inizio del commento alla soluzione, all'interno di `DetectCosmic()` bisogna calcolare la distanza d del punto d'impatto (x, y) da ciascun rivelatore. La funzione ritorna 1 se esiste almeno un rivelatore tale che $d < R = 1$ m, altrimenti ritorna 0.

```
#define R 0.001 //km, raggio dei rivelatori
int DetectCosmic(double x, double y, int * xD, int * yD) {
  double dist;
  for(int i=0; i<ND; i++) {// scorro tutti i rivelatori
   dist = sqrt((x-xD[i])*(x-xD[i])+(y-yD[i])*(y-yD[i]));
   if(dist<R) return 1;
  }
  //se esce dal loop significa che non ha colpito alcun rivelatore
  return 0;
}
```

► **Seconda parte:**
Si definisce il numero di esperimenti `NEXP` pari a 10 con una *macro*. La serie di esperimenti può essere facilmente realizzata con un ciclo `for()` con `NEXP` iterazioni; in ciascuna iterazione si ripete il codice scritto nella prima parte utilizzando N_i, S_i al posto di N_0, S_0, ricordandosi di aggiornare $N_i = N_0 + i \cdot 200$ e di inizializzare il conteggio dei raggi a 0 prima di ogni esperimento.
Poiché l'unica operazione che utilizza N_i e S_i consiste nella stampa su un file, è permesso di usare due sole variabili `N`, `S` anziché due array `N[NEXP] S[NEXP]`: le variabili `N`, `S` possono infatti essere sovrascritte nell'iterazione successiva, una volta stampato il loro valore precedente su file. Non è un errore utilizzare gli array, ma si occupa memoria inutilmente.
Infine, la procedura per stampare dati su un file di testo prevede tre passi: 1) creare ed inizializzare un puntatore a variabile di tipo `FILE` con `fopen()`, in modalità scrittura "w"; 2) stampare ciascun gruppo di dati con `fprintf()`; 3) chiudere il file usando `fopen()` alla fine dell'operazione. Le variabili i, N_i, S_i sono interi, pertanto vengono stampati nel formato "%d %d %d\n".
Per il commento dello *script* in Python si rimanda all'Appendice 16.

Capitolo 6

Game of life

6.1 Testo dell'esercizio

Scrivere un programma che simuli il *Gioco della vita* unidimensionale, in cui vedremo come evolve una popolazione iniziale di 3 persone in base a delle regole precise. Per fare questo:

1. Dichiarare un array `life[]` di lunghezza massima 100. Chiedere all'utente di inserire un intero positivo `N`$\leq$ 100 e nel caso in cui non soddisfi la condizione, reiterare la richiesta. Dopodiché,azzerare tutti gli `N` elementi dell'array;

2. Estrarre casualmente 3 diverse locazioni dell'array in cui inserire il valore 1;

3. Chiedere all'utente di inserire un valore `T` e controllare che `T` sia ≤ 1000. Se la condizione non è rispettata, reiterare la richiesta;

4. Ripetere `T` volte le seguenti operazioni:

   i. A partire dall'array `life[]`, creare un nuovo array `newlife[]` della stessa dimensione, secondo le seguenti regole per l'elemento `j`-esimo:

      a. Se entrambe le caselle adiacenti a `life[j]` contengono il valore `1`, impostare il valore di `newlife[j]` a 0.
      b. Se una sola casella adiacente a `life[j]` contiene il valore `1`, impostare il valore di `newlife[j]` a `1`.
      c. Se entrambe le caselle adiacente a `life[j]` contengono il valore `0`, copiare il valore di $\widehat{\text{life}}$[j] in `newlife[j]`.

      L'array deve essere considerato *ad anello*, cioè la prima casella (`life[0]`) è preceduta dall'ultima casella (`life[N-1]`), e viceversa l'ultima casella è seguita dalla prima.

   ii. A ogni passo, salvare in un array `population[]` il numero di `1` presenti nell'array `life[]` all'iterazione corrente. Salvare anche il valore della popolazione iniziale.

   iii. Stampare ad ogni passo l'array `newlife[]` su una riga, quindi copiare su `life[]` i valori di `newlife[]` e riazzerare il contenuto di `newlife[]`;

5. Alla fine del ciclo di `T` passi, cercare il valore minimo, il massimo e l'ultimo valore contenuti nell'array `population[]` e stamparli sullo schermo.

6.2 Soluzione completa

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NMAX 100
#define TMAX 1000

void Inserimento(int *, int, int, char []);

int main(){
  int N,T, i,j,prec,succ, min,max;
  int life[NMAX]={0}, newlife[NMAX]={0}, population[TMAX+1]={0};
  printf("\nQuesto programma simula il gioco della vita"
   "su un array di dimensione N, per un numero di passi T, "
   "partendo da una popolazione di 3 elementi in posizioni casuali.\n\n");
  /* Inizializzazione seed numeri random */
  srand48(time(NULL));
  /* Inserimento N,T */
  Inserimento(&N,3,NMAX,"N");
  Inserimento(&T,1,TMAX,"T");
  /* Inizia con 3 elementi casuali di life[] pari a 1 */
  i=0;
  while(i<3){
    j=drand48()*N;
    if(life[j]==0){
      life[j]=1;
      i++;
    }
  }
  /* Stampa life[] iniziale e salva popolazione iniziale */
  for(j=0;j<N;j++) printf("%d",life[j]);
  printf("\n");
  population[0]=3;
  /* Ciclo su T passi temporali */
  for(i=1;i<=T;i++){
    /* Ciclo su N elementi di newlife[] */
    for(j=0;j<N;j++){
      /* Condizioni periodiche al bordo */
      prec=(j-1+N)%N;
      succ=(j+1)%N;
      /* Regole per creare newlife[j] */
      if((life[prec]==0 && life[succ])==0) // entrambi 0 -> non cambiare
  newlife[j]=life[j];
      else if(life[prec]==1 || life[succ]==1) // 1 0 oppure 0 1 -> 1
```

```
45    newlife[j]=1;
46        /* non serve implementare 1 1 -> 0, dato che newlife[] e' inizializzato a 0
       */
47      }
48      for(j=0;j<N;j++) {
49        printf("%d",newlife[j]); // Stampa newlife[] in riga
50        population[i]+=newlife[j]; // conta popolazione di newlife[]
51        life[j]=newlife[j]; // Copia newlife[] in life[]
52        newlife[j]=0; // Azzera newlife[] per il passo successivo
53      }
54      //printf("= %d",population[i]);
55      printf("\n"); // A capo finale
56    }
57    /* Calcolo di max,min della popolazione */
58    min=population[0]; // Inizializzazione min
59    max=population[0]; // Inizializzazione max
60    // Ciclo di ricerca del massimo e del minimo di population
61    for(i=0;i<=T;i++){
62      if(population[i]<min){
63        min=population[i];
64      }
65      if(population[i]>max){
66        max=population[i];
67      }
68    }
69    /* Stampa max,min e l'ultima popolazione */
70    printf("Popolazione minima: %d\n"
71     "Popolazione massima: %d;\n"
72     "Ultima popolazione: %d\n",min,max,population[T]);
73  }
74
75  void Inserimento(int *ptr, int min, int max, char name[]) {
76    int res, n;
77    do {
78      printf("Inserire %s (intero compreso fra %d e %d): ",name,min,max);
79      res=scanf("%d",&n);
80      while(getchar()!='\n'); //svuota il buffer
81      if(res==0) printf("Errore nella formattazione dell'input.\n");
82      else if((n<min)||(n>max)) printf("Errore: il valore inserito è fuori dall'
       intervallo [%d, %d].\n",min,max);
83    } while((res==0)||(n<min)||(n>max));
84    (*ptr)=n;
85  }
```

6.3 Commento alla soluzione

Si includono la libreria `stdio.h` per le funzioni di input/output, e le librerie `stdlib.h` e `time.h` per la generazione di numeri pseudocasuali. Con una direttiva al precompilatore si definiscono le costanti `NMAX 100` e `TMAX 1000`. Si dichiarano le variabili intere `N` (taglia della popolazione) e `T` (numero di passi temporali), alcune variabili intere ausiliarie, ed infine gli array interi `life[NMAX]`, `newlife[NMAX]`, e `population[TMAX+1]`; quest'ultimo ha dimensione `TMAX+1` perché deve contare anche la popolazione al tempo zero, oltre alla popolazione a ciascuno dei `T` passi.
Si fa riferimento all'Appendice 12 per la lettura da terminale di un numero intero in un intervallo prefissato. Siccome questa operazione deve essere effettuata 2 volte (per `N` e per `T`), su intervalli diversi, è conveniente dichiarare una funzione `Inserimento()` che contenga un adattamento del codice dell'Appendice e che riceva in ingresso: un puntatore alla variabile intera su cui salvare il valore finale; due interi contenenti i limiti dell'intervallo; un array di caratteri contenente il nome della variabile, da stampare durante la richiesta di inserimento. Si veda la soluzione completa per la definizione di `Inserimento()`.
Durante la dichiarazione, si possono assegnare gli array a `{0}` per inizializzarne immediatamente tutti gli elementi a `0`; oppure, dopo aver letto il valore di `N`, si può utilizzare un ciclo `for()` di `N` passi per inizializzare a `0` solo gli `N` elementi di `life[]` e `newlife[]` che saranno utilizzati nel codice.
Nota: il limite inferiore per `N` è 3.

```
...
#define NMAX 100
#define TMAX 1000

void Inserimento(int *ptr, int min, int max, char name[]);

int main(){
    int N,T, life[NMAX]={0}, newlife[NMAX]={0}, population[TMAX+1]={0};
    printf("...\n"); //stampa una breve spiegazione del programma
    /* Inserimento di N,T */
    Inserimento(&N,3,NMAX,"N"); // N>=3 !!
    Inserimento(&T,1,TMAX,"T");
 }
```

Per scegliere casualmente 3 locazioni `j` diverse in cui inizializzare `life[j]` ad `1`, bisogna usare un contatore `i` ed un generatore di numero casuali. All'interno di un ciclo `while(i<3)`:

- si genera un intero `j` pseudocasuale compreso fra `0` e `N-1` inclusi, ad esempio tramite il comando `drand()48*N` (si veda l'Appendice 14);
- se l'elemento `life[j]` è nullo, si pone `life[j]=1` e si incrementa il contatore; altrimenti non si fa nulla;
- si ripete estraendo un nuovo numero pseudocasuale.

Il ciclo termina quando il contatore è pari a 3.

```
int main(){
```

```
    int N,T,i,j;
    ...
    srand48(time(NULL)); // inizializzazione del seed del generatore!
    i=0; // inizializza il contatore a 0
    while(i<3){
        j=drand48()*N; // intero pseudocasuale in [0,N-1]
        if(life[j]==0){ // se il valore in r è 0...
            life[j]=1; //... assegna 1
            i++; //... e aumenta il contatore
        }
    }
}
```

Controllare che la locazione estratta sia libera è fondamentale soprattutto nel caso in cui `N` sia piccolo e quindi è facile che venga estratto lo stesso numero più volte di seguito.
Comincia ora la parte principale del codice. Si usa un ciclo `for(i=1;i<=T;i++)` su un indice `i` per ripetere `T` volte le operazioni seguenti:

- All'interno di un ciclo innestato `for(j=0;j<N;j++)` si scorrono gli elementi `life[j]` e si calcolano i corrispondenti `newlife[j]` in base al valore degli elementi adiacenti a `life[j]`.
- Con un secondo ciclo `for()` identico al primo, si stampano gli elementi di `newlife[]`, si conta la loro popolazione incrementando `popolazione[i]`, si copia `newlife[]` su `life[]` e lo si reimposta a zero.

Nota: per come si è scelto di utilizzare l'indice `i`, l'elemento `popolazione[0]` non viene considerato; deve essere posto pari a 3 con un ulteriore comando.
Nota: si utilizzano due cicli consecutivi sugli `N` elementi, perché scrivere tutto in un unico ciclo non è immediato: bisogna tener conto che quando si copia `newlife[j]` su `life[j]` si modifica il vicino di `life[j+1]`, cambiando il risultato delle regole per `newlife[j+1]`.
In dettaglio, nel primo ciclo bisogna calcolare il valore degli indici precedente e successivo all'indice `j` corrente, tenendo conto della proprietà di *anello* dell'array (che prende il nome di *condizioni periodiche al bordo*). Si dichiarano due variabili intere `prec=j-1` e `succ=j+1`. Il modo più semplice di implementare la condizione ad anello è inserire due istruzioni `if()`, che pongano `prec=N-1` se `j` è pari a `0`, oppure `succ=0` se `j` è pari a `N-1`. Dopodiché si applicano le regole per il calcolo di `newlife[j]`, verificando prima il caso in cui entrambi i vicini sono pari a zero e poi il caso in cui almeno due è pari ad uno; il caso rimanente è inutile da verificare perché `newlife[j]` è già inizializzato a `0`.
Nota: un metodo equivalente per tenere conto delle condizioni periodiche al bordo, come implementato nella soluzione completa, consiste nel prendere `succ=(j+1)%N` e `prec=(j-1+N)%N`; nel secondo caso bisogna sommare `N` per riportare `j-1` fra gli interi positivi (il resto della divisione di un numero negativo per `N` restituirebbe un numero negativo).

```
int main() {
    ...
    population[0]=3;
    /* Ciclo su T passi temporali */
    for(i=1;i<=T;i++){
```

```
        /* Ciclo su N elementi di newlife[] */
        for(j=0;j<N;j++){
            prec=j-1;
            succ=j+1;
            /* Condizioni periodiche al bordo */
            if(j==0) prec=N-1;
            else if(j==N-1) succ=0;
            /* Regole per creare newlife[j] */
            if((life[prec]==0 && life[succ])==0) // entrambi 0 -> non cambiare
              newlife[j]=life[j];
            else if(life[prec]==1 || life[succ]==1) // (1 0 oppure 0 1) -> 1
              newlife[j]=1;
            /* non serve implementare 1 1 -> 0, dato che newlife[] e'
    inizializzato a 0 */
        }
    }
}
```

Nota: quando si usa il costrutto `if()...else if()` nel modo mostrato, è importante verificare per prima la condizione `x==0&&y==0`, perché è più stringente della condizione `x==1||y==1`.
Nel secondo ciclo, possiamo riutilizzare la stessa variabile `j` per scorrere di nuovo gli elementi di `life[]` e di `newlife[]` ed effettuare le operazioni spiegate precedentemente:

```
int main(){
    ...
    population[0]=3;
    for(i=1;i<=T;i++) {
        for(j=0;j<N;j++){...} //Calcola newlife[]
        /* Secondo ciclo */
        for(j=0;j<N;j++) {
            printf("%d",newlife[j]); //Stampa newlife[] in riga, senza spazi
            population[i]+=newlife[j]; //Conta la popolazione di newlife[] al
    passo i
            life[j]=newlife[j]; //Copia ciascun newlife[j] in life[j]
            newlife[j]=0; //Azzera newlife[]
        }
        printf("\n"); //Vai a capo a fine riga (fuori dal ciclo!)
    }
}
```

Alla fine del ciclo sui passi bisogna stampare sul terminale il valore minimo, massimo e ultimo assunto dalla popolazione al variare dei passi. La ricerca del massimo e del minimo di un array è un processo standard, spiegato in Appendice 13. Si veda la soluzione completa per l'applicazione a questo esercizio.

Capitolo 7

Conversione di una sequenza di bit in un numero intero

7.1 Testo dell'esercizio

Lo scopo di questo esercizio è di scrivere un programma che generi sequenze casuali di bit e che converta tali sequenze in numeri interi con segno, assumendo che ogni sequenza casuale generata sia una rappresentazione in complemento a 2 di un intero con segno.

▶ **Prima parte:**
Si scriva un programma chiamato `generabits.c` che:

1. Richieda in input un numero intero positivo $N_b \leq 16$ e generi una sequenza di N_b bit, ovvero N_b numeri interi pari a 0 o 1, che vanno memorizzati in un opportuno array unidimensionale.

2. Converta la sequenza di bit precedentemente generata in un numero intero con segno, assumendo che la sequenza rappresenti un numero binario in complemento a due (*Suggerimento: convertire la sequenza di bit nell'intero positivo corrispondente e poi eventualmente calcolarne il complemento a 2*).

3. Stampi su schermo la sequenza di bit generata a partire dal bit più significativo, posto a sinistra, a quello meno significativo, posto a destra, ed il numero intero ottenuto dalla conversione.

Nello scrivere il programma si richiede che vengano implementate le seguenti funzioni:

- `Inserimento(…)`, che richiede l'inserimento di un numero intero positivo.
- `Genera_Bit()`, che restituisce un numero intero casuale di valore 0 o 1.
- `IntPow(x,y)`, che eleva il numero intero `x` alla potenza intera `y`.
- `Convert_Int(…)` che richieda come argomento un array di interi (cioè la sequenza di bit) e il numero di bit memorizzati nell'array e che restituisca l'intero con segno che corrisponde alla rappresentazione in complemento a due della sequenza fornita.

► **Seconda parte:**
Dopo aver copiato il file creato nella prima parte in un nuovo file chiamato `inthisto.c`, modificare il programma in modo che:

1. Generi M sequenze casuali di 4 bit, dove $M \leq 100.000.000$ è un intero positivo che deve essere richiesto in input usando la funzione `Inserimento()`.

2. Converta ogni sequenza di bit, intesa come un numero binario in complemento a 2, nel corrispondente numero intero con segno decimale.

3. Crei un array contenente i conteggi delle occorrenze dei $2^4 = 16$ possibili numeri interi con segno ottenuti a seguito delle M conversioni. *Suggerimento: utilizzare un array* `isto[]` *di 16 elementi, in modo che quello con indice 0 (*`isto[0]`*) corrisponda al numero più piccolo rappresentabile* (-8).

4. Scriva l'array del punto precedente su un file di dati chiamato `istogramma.dat`.

Si richiede che l'istogramma venga opportunamente normalizzato a 1, utilizzando le frequenze anziché i conteggi.
► **Terza parte:**
Creare con `python` un grafico dove vengono riportati gli istogrammi ottenuti nella seconda parte per $M = 10, 1000, 100.000, 10.000.000$. In questa grafico deve essere presente una legenda per le 4 curve riportate, un titolo e delle opportune label per gli assi x e y.

7.2 Soluzione completa

► **Listato completo di `generabits.c`**

```c
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
j
#define MAXBITS 16

int Inserimento(void);
int Genera_Bit(void);
int IntPow(int b, int n);
int Converti_Int(int a[], int nbits);

int main(void)
{
  int bitarr[MAXBITS]={0};
  int i, nbits, ci;
  srand48(time(NULL));
  printf("Questo programma genera Nb bits pseudocasuali e li converte in formato
    decimale utilizzando la rappresentazione in complemento a 2.\n");
  // Legge nbits dal terminale
  nbits=Inserimento();
  // Genera nbits bit casuali e li salva in bitarr[]
  for (i=0; i < nbits; i++)
    bitarr[i] = Genera_Bit();
  printf("La sequenza di bit generati è: ");
  // Stampa i bit dal più significativo (a sinistra) al meno significativo (a
    destra)
  for (i=nbits-1; i >= 0; i--)
    printf("%d", bitarr[i]);
  printf("\n");
  // Stampa il numero decimale corrispondente
  ci=Converti_Int(bitarr, nbits);
  printf("I bit generati casualmente corrispondono al numero in complemento a 2:
    %d\n\n", ci);
  return 0;
}

int Inserimento(void)
{
    int n=0, res;
    while(res<1 || n<1 || n>MAXBITS) {
        printf("Inserire Nb (compreso fra 1 e %d): ",MAXBITS);
```

```
        res=scanf("%d",&n);
        while(getchar()!='\n'); //svuota il buffer
        if(res<1)
            printf("Errore nella formattazione dell'input.\n");
        else if(n<1 || n>MAXBITS)
            printf("Errore: il valore inserito è fuori dall'intervallo [1, %d].\n
    ",MAXBITS);
     }
     return n;
}
int Genera_Bit(void)
{
  return (drand48()>=0.5)?1:0;
}
int IntPow(int b, int n)
{
  int i, x=1;
  for(i=0; i < n; i++)
      x *= b;
  return x;
}
int Converti_Int(int a[], int nbits)
{
  int neg, i, k=0;
  if (a[nbits-1]==1) // il numero è negativo
    neg=1;
  else
    neg=0;
  for (i=0; i < nbits-1; i++)
    {
      k += a[i]*IntPow(2,i);
    }
  if (neg)
    k = -(IntPow(2,nbits-1)-k);
  return k;
}
```

► **Listato completo di intisto.c**

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

#define MAXBITS 16
#define MAXISTO 65536
#define MAXM 10000000
```

```
8
9  //in Inserimento() bisogna modificare il messaggio e sostituire MAXBITS con MAXM
10 int Inserimento(void);
11 //queste funzioni restano invariate
12 int Genera_Bit(void);
13 int IntPow(int b, int n);
14 int Converti_Int(int a[], int nbits);
15
16 int main(void)
17 {
18
19   int i, nbits=4, ci, j, M;
20   int nisto=IntPow(2,nbits), nisto_half=nisto/2;
21   int bitarr[MAXBITS]={0}, isto[MAXISTO]={0};
22   FILE *fp;
23   srand48(time(NULL));
24   M=Inserimento();
25
26   for (j=0; j<M; j++)
27     {
28       for (i=0; i < nbits; i++)
29         bitarr[i] = Genera_Bit();
30
31       ci=Converti_Int(bitarr, nbits);
32       isto[ci+nisto_half]++;
33   }
34
35   fp=fopen("intisto.dat", "w");
36   for (i=0; i<nisto; i++)
37     fprintf(fp,"%d %lf\n", i-nisto_half, isto[i] / (double)M);
38   fclose(fp);
39 }
40 ... //definizione delle funzioni
```

► **Listato completo codice Python per l'istogramma**

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x,y= np.loadtxt("intisto10.dat", unpack=True)
5  plt.bar(x,y,label='M=10')
6  x,y= np.loadtxt("intisto1000.dat", unpack=True)
7  plt.bar(x,y,label='M=1000')
8  x,y= np.loadtxt("intisto100000.dat", unpack=True)
9  plt.bar(x,y,label='M=100000')
10 x,y= np.loadtxt("intisto10000000.dat", unpack=True)
```

```
11 plt.bar(x,y,label='M=10000000')
12
13 plt.xlabel("intero decimale")
14 plt.ylabel("conteggi normalizzati")
15 plt.legend()
16
17 plt.savefig("intisto.png")
```

7.3 Commento alla soluzione

▶ Soluzione della prima parte:
Per questo codice è sufficiente includere le librerie `stdio`, `stdlib` e `time`. Nel primo punto si chiede in input il numero di bit N_b, i quali verranno memorizzati in un array unidimensionale di interi. La dimensione dell'array (ovvero il numero massimo di bit che possono essere salvati) deve essere una costante definita all'inizio del programma tramite una direttiva `define` (si suggerisce di utilizzare `MAXBITS=16`):

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#define MAXBITS 16
```

Seguendo le indicazioni del testo, si definisce la funzione `Inserimento()` per inserire dal terminale un generico numero intero positivo in un intervallo prefissato (verrà utilizzata per leggere anche il valore di M, nella seconda parte dell'esercizio). É sufficiente dichiararla di tipo `int` ed includere nella sua definizione il codice mostrato in Appendice 12, restituendo alla fine il valore di `n` letto. Si rimanda alla Soluzione Completa dell'esercizio per la definizione di `Inserimento()` e la sua chiamata.
La funzione `Genera_Bit()` non riceve nessun argomento e restituisce un bit casuale, ossia un intero pari a 0 o 1; tale numero casuale si può ottenere applicando l'operatore `%2` (modulo 2) a `rand()` che genera un intero casuale non negativo; oppure verificando quando un numero estratto con `drand48()`, di tipo `double` fra 0 e 1, sia maggiore di 0.5:

```
int Genera_Bit(void) {
  return (drand48()>=0.5)?1:0;
}
```

Si ricorda che la sintassi compatta `expression ? a : b` restituisce `a` se `expression` è vera, e `b` altrimenti. Inoltre, l'uso di un generatore di numeri casuali prevede l'inizializzazione del suo *seed* nel `main()`.
Determinato N_b, si effettua un ciclo `for()` con N_b passi in cui si genera un bit casuale e lo si salva in un array unidimensionale precedentemente allocato a `MAXBITS`:

```
...
int Inserimento(void);
int Genera_Bit(void);
int main(void)
{
    int i,nbits, bitarr[MAXBITS]={0};
    srand48(time(NULL));
    nbins=Inserimento();
    for (i=0; i < nbits; i++)
        bitarr[i] = Genera_Bit();
}
```

La funzione `IntPow(int b, int n)` calcola $b^n = b \cdot b \cdot \cdots \cdot b$ usando un ciclo `for()`: partendo da un intero $x = 1$, itera per n volte la moltiplicazione $x = x \cdot b$:

```
int IntPow(int b, int n)
{
  int i, x=1;
  for(i=0; i < n; i++)
    {
      x *= b;
    }
  return x;
}
```

La rappresentazione in complemento a 2 ad N bit (in breve $C2^N$) permette di operare efficacemente con gli interi con segno compresi fra -2^{N-1} e $2^{N-1}-1$. Per un numero positivo o nullo, essa coincide con la rappresentazione binaria in $N-1$ bit, preceduta da un bit 0. Un numero negativo è invece rappresentato come complemento a 2, cioè come $-2^{N-1}+k$, dove k é un numero a $N-1$ bit, ed è preceduto da un bit 1. Ad esempio $N = 4$ bit possono rappresentare i numeri fra -2^3 e $2^3 - 1$ inclusi. Il numero decimale $x = +5$ diventa $5_{C2^4} =$ `0101` poiché $5_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101$. Il numero negativo $x = -5$ si puó scrivere come $x = -2^3 + k$ con $k = 3 = 0 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 =$ `011`, per cui $(-5)_{C2^4} =$ `1011`. Si noti che $x = 0$ è sempre rappresentato da una sequenza di zeri.
Il suggerimento del testo consiste nel trascrivere gli ultimi $N-1$ bit nell'intero corrispondente k (5 o 3 negli esempi), e poi restituire k se il primo bit è 0, oppure calcolare $-2^{N-1}+k$ se il primo bit è 1. Per l'ordinamento dei bit nell'array si usa la convenzione che al primo elemento corrisponde il bit meno significativo (non è un errore se si usa la convenzione opposta). Il procedimento da seguire nel corpo della funzione `Convert_Int(int a[], int nbits)` è dunque:
1) determinare il segno guardando l'ultimo elemento dell'array (bit più significativo);
2) calcolare il decimale `k` corrispondente al binario rimanente, sommando le $N_b - 1$ potenze di 2;
3) se positivo, restituire `k`; se negativo, restituire il complemento $-2^{N_b-1} + \mathtt{k}$.

```
int Converti_Int(int a[], int nbits)
{
  int neg, i, k=0;
  if (a[nbits-1]==1) // il numero è negativo
    neg=1;
  else
    neg=0;
  for (i=0; i < nbits-1; i++) // da binario di Nb-1 bits a decimale
    {
      k += a[i]*IntPow(2,i);
    }
  if (neg)
    k = -(IntPow(2,nbits-1)-k); //complemento a 2, se neg
  return k;
}
```

Per stampare il numero in complemento a 2, è sufficiente inserire un ciclo `for()` contenente un `printf()` lungo gli elementi dell'array in ordine inverso, cosicché le cifre più significative vengano stampate per prime sul terminale. Il numero decimale può essere stampato con un semplice `printf()` dopo la chiamata di `Converti_Int()`, come mostrato nelle righe 24-30 del listato completo.

► **Soluzione della seconda parte:**
É qui che si apprezza la versatilità delle funzioni create nella prima parte.
In questa parte il numero di bits N_b è fissato a 4, quindi verranno generati numeri interi compresi fra -8 e 7 inclusi. Viene invece richiesto di inserire il numero M di sequenze da generare. É sufficiente dunque impostare `nbits=4` e chiamare la funzione Inserimento() per leggere M anziché N_b, introducendo una variabile intera `M` con un limite massimo `MAXM` opportunamente definito tramite direttiva al precompilatore. La parte di codice in cui si generano e convertono i numeri deve essere racchiusa in un ciclo `for()` con M ripetizioni.
Seguendo il consiglio del testo, si salvano le occorrenze di ciascun numero decimale in un array `isto` di 16 elementi, dove il primo elemento `isto[0]` corrisponde a -8 e l'ultimo `isto[15]` a 7. L'array deve essere inizializzato a zero, e il conteggio di ciascuna occorrenza viene incrementato all'interno del ciclo `for()` sfruttando la traslazione di $+8$ che lega gli interi decimali $[-8, 7]$ agli indici corrispondenti $[0, 15]$.

```
#define MAXBITS 16
#define MAXM 10000000
int main()
    {
    ...
    nbits=4;
    int j, M;
    int isto[16]={0};
    ...
    // inserimento M
    ...
    for (j=0; j<M; j++)
        {
        for (i=0; i < nbits; i++)
            bitarr[i] = Genera_Bit();

        ci=Converti_Int(bitarr, nbits);
        isto[ci+8]++; //a ci=-8 corrisponde l'indice 0
    }
}
```

Il testo richiede di normalizzare l'array dei conteggi ad 1 e salvarlo su un file. Ciò può essere fatto con un successivo ciclo `for()` che scorra sugli indici dell'array, salvi su un file il numero in base dieci corrispondente (pari all'indice dell'array meno 8) e la relativa frequenza, cioè il numero del relativo conteggio diviso per il numero totale di sequenze M. Si noti che dal momento che la frequenza è un numero razionale compreso tra 0 e 1, è necessario fare il *casting* al tipo `double` di `isto[i]` prima

di calcolarne il rapporto; per le regole della *conversione implicita*, anche `M` viene automaticamente convertito da `int` a `double` prima di calcolare l'espressione, che diventa una divisione fra numeri reali. Un'alternativa equivalente é definire l'array `isto[]` di tipo `double` fin dall'inizio.
Si veda l'Appendice 11 per i dettagli sull'utilizzo dei puntatori a file.

```
int main()
    {
    FILE *fp;
    ...
    fp=fopen("intisto.dat","w");
    for (i=0; i<16; i++)
        fprintf(fp, "%d %lf\n", i-8, (double)isto[i] / M);
    fclose(fp);
}
```

Opzionale: per un codice più versatile, dichiarare un array `isto[MAXISTO]`*, dove* `MAXISTO` *è impostata a* $2^{MAXBITS} = 65536$*, e sostituire* 16 →`IntPow(2,nbits)` *e* 8 →`IntPow(2,nbits-1)`*.*

► **Soluzione della terza parte:**
Eseguire il programma precedente inserendo i valori di M indicati dal testo, e rinominando il file `intisto.dat` di volta in volta per non sovrascriverlo. Il listato contenente il codice Python si trova alla fine della Sezione 7.3 e in Fig. 7.1 il plot risultante.
Opzionale: modificare `intisto.c` *affinché salvi l'istogramma su un file chiamato* `intistoM.dat` *con il valore di* M *scelto dall'utente. Suggerimento: usare la funzione* `sprintf()`*.*

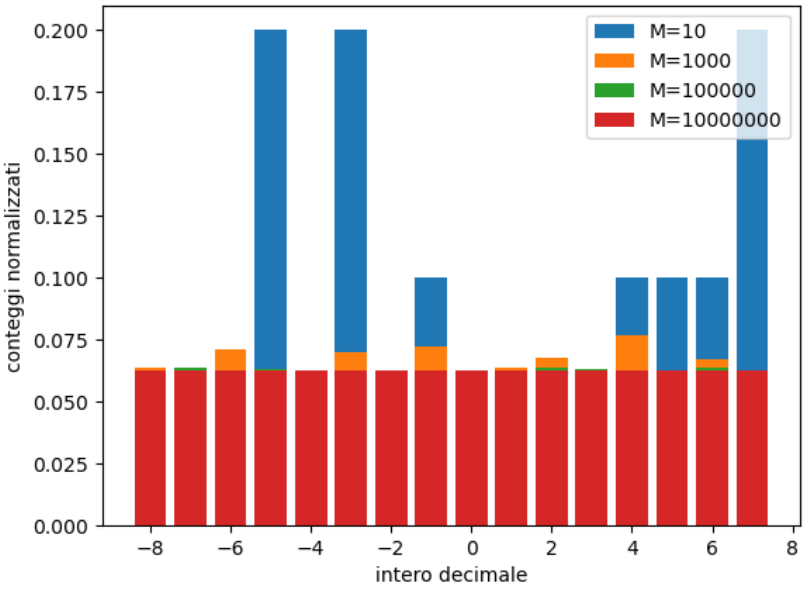


Figura 7.1: Istogrammi dei conteggi normalizzati, ovvero delle frequenze, di M numeri decimali convertiti da numeri binari a 4 cifre in complemento a 2. Poiché ciascun bit è stato estratto casualmente fra 0 e 1, anche i numeri in notazione binaria assumono casualmente uno dei $2^4 = 16$ valori possibili (come se si trattasse di un dado a 16 facce). Per M grande, le frequenze devono tendere a una distribuzione uniforme, con $f = 1/16 = 0.0625$.

Capitolo 8

Random walk bidimensionale

8.1 Testo dell'esercizio

Lo scopo di questo esercizio è di generare un *cammino aleatorio* (*random walk*) su un reticolo quadrato bidimensionale. Un cammino aleatorio di una particella su un reticolo è costituito da una successione di N *passi*, in ciascuno dei quali la particella si muove in maniera casuale da un punto del reticolo a uno dei punti adiacenti. In fisica i *random walks* sono utilizzati, ad esempio, come modelli semplificati per studiare il moto Browniano e la diffusione di molecole sospese in un fluido.

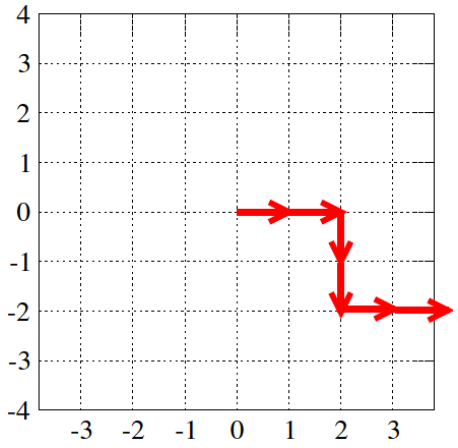


Figura 8.1: Eempio di cammino aleatorio su un reticolo quadrato bidimensionale, con $N = 6$ passi. La successione dei passi (traiettoria) è: $(0,0) \to (1,0) \to (2,0) \to (2,-1) \to (2,-2) \to (3,-2) \to (4,-2)$.

I punti che compongono un reticolo bidimensionale sono equispaziati tra loro lungo due direzioni ortogonali x e y. Ciò significa il generico punto ha coordinate $\mathbf{P}_i = (n_i, m_i)$, dove n_i e m_i sono due numeri interi. In questo esercizio considereremo un reticolo teoricamente infinito, il che significa che i valori di n_i e m_i non sono limitati. Una particella parte da un qualsiasi sito del reticolo, nel nostro caso dall'origine $\mathbf{P}_0 = (0,0)$, e si muove in modo casuale sul reticolo, compiendo un cammino di N passi, come segue.

Una particella che si trovi al passo t nel punto $\mathbf{P}_i$, al passo $t+1$ potrà spostarsi con uguale probabilità su uno dei quattro punti immediatamente adiacenti a $\mathbf{P}_i$, che avranno quindi coordinate: (n_i, m_i+1) o (n_i-1, m_i) o (n_i, m_i-1) o (n_i+1, m_i), a seconda che si trovino in alto, a sinistra, in basso, a destra rispetto a $\mathbf{P}_i$. Il movimento casuale viene ripetuto N volte, dando vita a una successione di punti, che descrive un percorso (cammino) sul reticolo quadrato. Un possibile cammino aleatorio con $N = 6$ è rappresentato in figura; si noti che nel corso del cammino la particella può tornare più volte sullo stesso punto.
Si scriva un programma `C` che generi un cammino aleatorio su un reticolo quadrato bidimensionale come segue:

1. Una direttiva al precompilatore fissa le dimensioni massime degli array che verranno utilizzati nel corso del programma.

2. Il programma chiede all'utente di inserire un numero N di *passi* di cui è composto il cammino aleatorio, e controlla che il numero immesso sia compreso tra 1000 e 10000.

3. A questo punto il programma genera un cammino aleatorio di N passi come segue:

   (a) La posizione della particella al passo t è contenuta in un vettore con due componenti di tipo opportuno di nome `pos[]`.

   (b) Al passo $t = 0$ la particella si trova nell'origine $\mathbf{P}_0 = (0, 0)$.

   (c) Ad ogni passo, una funzione `Move()`, di tipo e argomenti opportuni, riceve in input la posizione corrente della particella al tempo t, e genera la posizione al tempo $t+1$, scegliendo in maniera casuale uno dei quattro siti adiacenti sul reticolo.

   (d) La posizione al passo t viene salvata su un array `trajectory[]` di tipo e dimensioni opportune.

   (e) Le operazioni 1-4 vengono ripetute N volte, fino a generare un cammino di N passi.

4. Una volta compiuti gli N passi, la traiettoria salvata nell'array `trajectory[]` viene quindi stampata su un file di N righe e due colonne di nome `walk.dat`, che contiene le coordinate di tutti i punti visitati dalla particella nel corso del cammino aleatorio.

5. Una funzione `Analysis()`, di tipo e argomenti opportuni, analizza la traiettoria contenuta nell'array `trajectory[]`, determina la distanza massima dall'origine raggiunta nel corso della traiettoria, e salva le coordinate del punto corrispondente in un array di tipo e dimensioni opportune di nome `posmax[]`.

6. Il programma stampa quindi un messaggio di riepilogo, con il seguente formato:

   ```
   Numero di passi:     8000 Distanza Max:  92.57  Nel punto: (91,17)
   ```

7. Una volta verificato che il programma funziona correttamente, con uno script `Python` si crei un grafico della traiettoria e lo si salvi su un file di tipo `png`.

8.2 Soluzione completa

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>

#define NMIN 1000
#define NMAX 10000

void Move(int position[2]);
int Dist2(int x, int y);
float Analysis(int trajectory[NMAX+1][2], int posmax[2], int N);

int main(){
    int N,t,res,pos[2]={0}, trajectory[NMAX+1][2]={0}, maxpos[2];
    float distmax;
    FILE *fp;
    srand48(time(NULL)); // Inizializzazione seed numeri random
    printf("Questo programma esegue un cammino aleatorio di N passi su un reticolo quadrato 2D, partendo dall'origine degli assi. Alla fine degli N passi, stampa sul file walk.dat la traiettoria ottenuta e stampa sul terminale la distanza massima raggiunta.\n\n");
    /* Inserimento di N */
    do {
        printf("Inserire il numero di passi N, compreso fra %d e %d: ",NMIN,NMAX);
        res=scanf("%d",&N);
        while(getchar()!='\n'); //svuota il buffer
        if(res==0) printf("Errore nella formattazione dell'input.\n");
        else if((N<NMIN)||(N>NMAX)) printf("Errore: il valore inserito è fuori dall'intervallo [%d, %d].\n",NMIN,NMAX);
    } while((res==0)||(N<NMIN)||(N>NMAX));
    /* Ciclo sugli N passi del random walk */
    for(t=0;t<N;t++){
        Move(pos); // Passo del random walk
        trajectory[t+1][0]=pos[0]; // Aggiornamento trajectory_x
        trajectory[t+1][1]=pos[1]; // Aggiornamento trajectory_y
    }
    /* Analisi della traiettoria */
    fp=fopen("walk.dat","w");
    for(t=0;t<=N;t++){
        fprintf(fp,"%d %d\n",trajectory[t][0],trajectory[t][1]);
    }
    fclose(fp);
    distmax=Analysis(trajectory,maxpos,N); // Ricerca della distanza massima
```

```
    printf("Numero di passi:\t %d Distanza Max:\t %.2f Nel punto:\t (%d,%d)\n",N,
    distmax,maxpos[0],maxpos[1]); // Stampa del resoconto
    return(0);
}

void Move(int pos[]){
    // Estrae e aggiorna il passo successivo del random walk
    int r=srand48()*4; // Estrazione numero casuale 0,1,2,3
    if(r==0)      pos[0]++; // Spostamento a destra (x=x+1)
    else if(r==1) pos[1]++; // Spostamento in alto (y=y1)
    else if(r==2) pos[0]--; // Spostamento a sinistra (x=x-1)
    else          pos[1]--; // Spostamento in basso (y=y-1)
}

int Dist2(int x, int y){
    return x*x+y*y; //Distanza al quadrato dall'origine
}

float Analysis(int trj[NMAX+1][2], int posmax[2], int N){
    // Ricerca il punto che ha distanza massima dall'origine,
    // salvando le coordinate del punto nell'array posmax[]
    // e restituendo il valore della distanza massima
    int i,dist2,dmax2=0;
    for(i=0;i<=N;i++){
        dist2=Dist2(trj[i][0],trj[i][1]);
        if(dist2>dmax2){
            dmax2=dist2;
            posmax[0]=trj[i][0];
            posmax[1]=trj[i][1];
        }
    }
    //la radice può essere decimale!
    return(sqrt(dmax2));
}
```

```
import numpy as np
import matplotlib.pyplot as plt

x,y = np.loadtxt("walk.dat", unpack=True)

plt.plot(x,y, "-") #traiettoria (linea)
plt.xlabel("x")
plt.ylabel("y")
plt.title("2D Random Walk")

```

```
##############################################
# bonus: evidenzia i punti iniziale e finale
plt.plot(x[0], y[0], "go", label="start")
plt.plot(x[-1], y[-1], "ro", label="end")
plt.legend()
##############################################

plt.savefig("walk.png")
```

8.3 Commento alla soluzione

Per questo programma é necessario includere la libreria standard `stdio.h`, le librerie `stdlib.h`, `time.h` per generare numeri pseudo casuali e la libreria `math.h` per utilizzare la funzione radice quadrata `sqrt()`. Come primo passo si definiscono le costanti utili `NMIN` e `NMAX` che definiscono l'intervallo ammesso per il numero di passi N che sará richiesto in input.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>

#define NMIN 1000
#define NMAX 10000
```

La procedura per richiedere l'inserimento di un numero intero in un intervallo prefissato é mostrata in Appendice 12. Prima di passare al ciclo sugli N passi del cammino aleatorio, bisogna dichiarare gli array `pos[]` e `trajectory[]`. Poiché la posizione della particella nel reticolo é identificata da 2 coordinate intere, si dichiara `int pos[2]`. Per la traiettoria conviene utilizzare un array a due dimensioni `int trajectory[NMAX+1][2]`: l'elemento `trajectory[i][j]` contierrá la coordinata `j`= $0, 1$ (x se 0, y se 1) della particella al passo `i`= $0, 1, \ldots, N$, dove bisogna includere anche il passo 0 ed N puó assumere al massimo il valore `NMAX`, per cui la prima dimensione deve essere `NMAX+1`. L'array `pos[]` deve essere inizializzato alla posizione iniziale della particella, cioé $(0, 0)$; di conseguenza, anche gli elementi `trajectory[0][0]` e `trajectory[0][1]` devono essere nulli. Si possono soddisfare rapidamente entrambe le richieste assegnando entrambi gli array a `{0}` in fase di dichiara.
La funzione `Move()` è ciò che fa avanzare il camminatore di un passo dal tempo t al tempo $t + 1$. Siccome deve restituire le coordinate x,y della posizione all'istante $t + 1$, l'unico modo per farlo è dichiararla di tipo `void` e salvare la nuova posizione in un array opportuno. Gli unici dati necessari per generare la nuova posizione sono le coordinate della posizione precedente, per cui si può definire la funzione in modo che prenda in input unicamente un array contenente le coordinate della posizione al tempo t, e lo sovrascriva con le nuove coordinate. In questo modo, nel `main()` non serve dichiarare array ulteriori a `pos[]`, contenente la posizione iniziale. Supponendo di aver giá definito la funzione, essa deve essere chiamata nel `main()` all'interno di un ciclo di N passi, seguita dal salvataggio delle nuove coordinate nell'array `trajectory[][]`. Nel codice seguente, la variabile `t` che contiene il passo del ciclo `for()` assume valori da 0 a $N-1$, per cui gli elementi di `trajectory[][]` assegnare la nuova posizione sono indicizzati da `t+1`; se invece si definisse un ciclo su `t` che va da 1 a N, si dovrebbe indicizzare con `t`.

```
...
void Move(int *position[2]);

int main(){
    int N,t,pos[2]={0},trajectory[NMAX+1][2]={0};
    ...//inserimento di N
    for(t=0;t<N;t++){
        Move(pos);
        trajectory[t+1][0]=pos[0];
```

```
        trajectory[t+1][1]=pos[1];
    }
}
```

Nel corpo della funzione `Move()`, bisogna scegliere casualmente una delle 4 direzioni permesse (su, giù, sinistra o destra) ed aggiornare conseguentemente la posizione. Il modo piú efficiente é associare ad ogni direzione un numero fra $\{0, 1, 2, 3\}$ ed estrarre tramite il comando `drand48()4` un numero pseudocasuale distribuito uniformemente in tale intervallo (come illustrato nell'Appendice 14). Con una serie di istruzioni `if()...else()` si incrementa o diminuisce di uno l'elemento dell'array ricevuto in input corrispondente alla direzione estratta:

```
void Move(int pos[]){
    int r=drand48()*4; // Estrazione numero casuale 0,1,2,3
    if(r==0){
        pos[0]++; // Spostamento a destra (x=x+1)
    }else if(r==1){
        pos[1]++; // Spostamento in alto (y=y+1)
    }else if(r==2){
        pos[0]--; // Spostamento a sinistra (x=x-1)
    }else{
        pos[1]--; // Spostamento in basso (y=y-1)
    }
}
```

Si ricordi di inizializzare casualmente il *seed* di `srand48()` una sola volta all'interno del `main()`, prima di chiamare la funzione `Move()`!
Al completamento degli N passi, si stampa il contenuto dell'array `trajectory[][]` sul file di output `walk.dat` utilizzando il comando `fprintf()` ed un puntatore a file (per la teoria si rimanda alla Appendice 11). Le righe dell'array vengono scorse con un nuovo ciclo `for()`, questa volta di $N + 1$ passi, e le corrispondenti coordinate sono stampate in formato di due colonne x,y separate da uno spazio:

```
int main(){
    File *fp;
    ...
    fp=fopen("walk.dat","w");
    for(t=0;t<=N;t++){
        fprintf(fp,"%d %d\n",trajectory[t][0],trajectory[t][1]);
    }
    fclose(fp);
}
```

Per facilitare la scrittura della funzione `Analysis()` si puó introdurre un'ulteriore funzione chiamata `Dist2()` che calcoli la distanza al quadrato di un punto dall'origine. In questo modo si suddivide il problema di trovare il punto con distanza massima dall'origine in due problemi separati: 1) calcolare

la distanza al quadrato dall'origine; 2) confrontare le distanze al quadrato di ogni punto della traiettoria per determinare quella massima. Si noti che confrontare le distanze al quadrato è equivalente a confrontare le distanze, dal momento che la distanza é definita positiva.

```
int Dist2(int x, int y){
    return(x*x+y*y);
}
```

La funzione `Analysis()` deve avere come argomento la matrice `trajectory[][]` e anche il numero di passi del cammino, in quanto solo le prime `N`-esime righe di `trajectory[][]` sono non-nulle; in generale se una funzione riceve in input un array deve ricevere anche la sua dimensione, se non é nota globalmente . Inoltre, `Analysys()` deve restituire le coordinate in cui é stata raggiunta la distanza massima dall'origine, ed il valore di tale distanza. La distanza può essere restituita dalla funzione stessa; bisogna usare il tipo `float` perché, nonostante il reticolo su cui avviene il cammino abbia coordinate intere, le distanze fra i punti possono essere reali (ad esempio la distanza del punto $(1, 1)$ é $\sqrt{2}$). Le coordinate invece devono essere assegnate ad un ulteriore array fornito in input alla funzione, che pertanto deve essere dichiarata come:

```
float Analysis(int trajectory[NMAX+1][2], int posmax[2], int N);
```

Il problema di individuare il massimo da un elenco di valori è trattato nell'Appendice 13. A differenza del problema classico, adesso si vuole scorrere le righe di una matrice, confrontando non gli elementi contenuti nelle due colonne, bensí una *funzione* di essi, ovvero la somma dei loro quadrati. L'algoritmo é lo stesso, ma all'interno dell'istruzione `if()` che aggiorna il valore massimo della distanza al quadrato, si vuole aggiornare anche le rispettive coordinate.

```
float Analysis(int trj[NMAX+1][2], int posmax[], int N){
    int i,dist2,distmax2=0;
    for(i=0;i<=N;i++){
        dist2=Dist2(trj[i][0],trj[i][1]);
        if(dist2>distmax2){
            distmax2=dis2t;
            posmax[0]=trj[i][0];
            posmax[1]=trj[i][1];
        }
    }
    return(sqrt(distmax2));
}
```

All'interno del `main()`, la funzione deve essere chiamata, come da testo dell'esercizio, dopo la stampa della traiettoria su file. All'inizio del `main()` si definiscono una nuova variabile di tipo `float` per la distanza massima ed un nuovo array intero per le rispettive coordinate. Per stampare il resoconto finale con il formato voluto può essere utile il comando `\t` che lascia uno spazio fissato:

```
int main(){
    float distmax;
```

```
    int maxpos[2];
    ...
    distmax=Analysis(trajectory,maxpos,N);
    printf("Numero di passi:\t %d Distanza Max:\t %.2f Nel punto:\t (%d,%d)\n",N,
   distmax,maxpos[0],maxpos[1]); // Stampa del resoconto
}
```

La parte in `Python` consiste in un plot della traiettoria, quindi del tipo x vs. y, analogo all'esempio spiegato in dettaglio nell'Appendice 16.1. Come punto bonus si potrebbe voler visualizzare il punto iniziale e finale della traiettoria. Graficare il punto iniziale è facile in quanto basta utilizzare la funzione `plt.plot()` usando come argomenti i valori `x[0]` e `y[0]`. Invece la posizione del punto finale dipende dalla `N` che si è scelta nel programma. Bisogna ricordare che in `Python` è sempre possibile accedere all'ultima locazione di un array, anche senza conoscerne precisamente l'indice, chiedendo la posizione `-1`. Inoltre, per una migliore lettura del grafico si possono usare le opzioni `"go"` e `"ro"` per evidenziare i punti iniziali e finali con pallini colorati, e inserire la legenda (in Fig. 8.2 è mostrato un possibile grafico finale con `N=5000` passi).

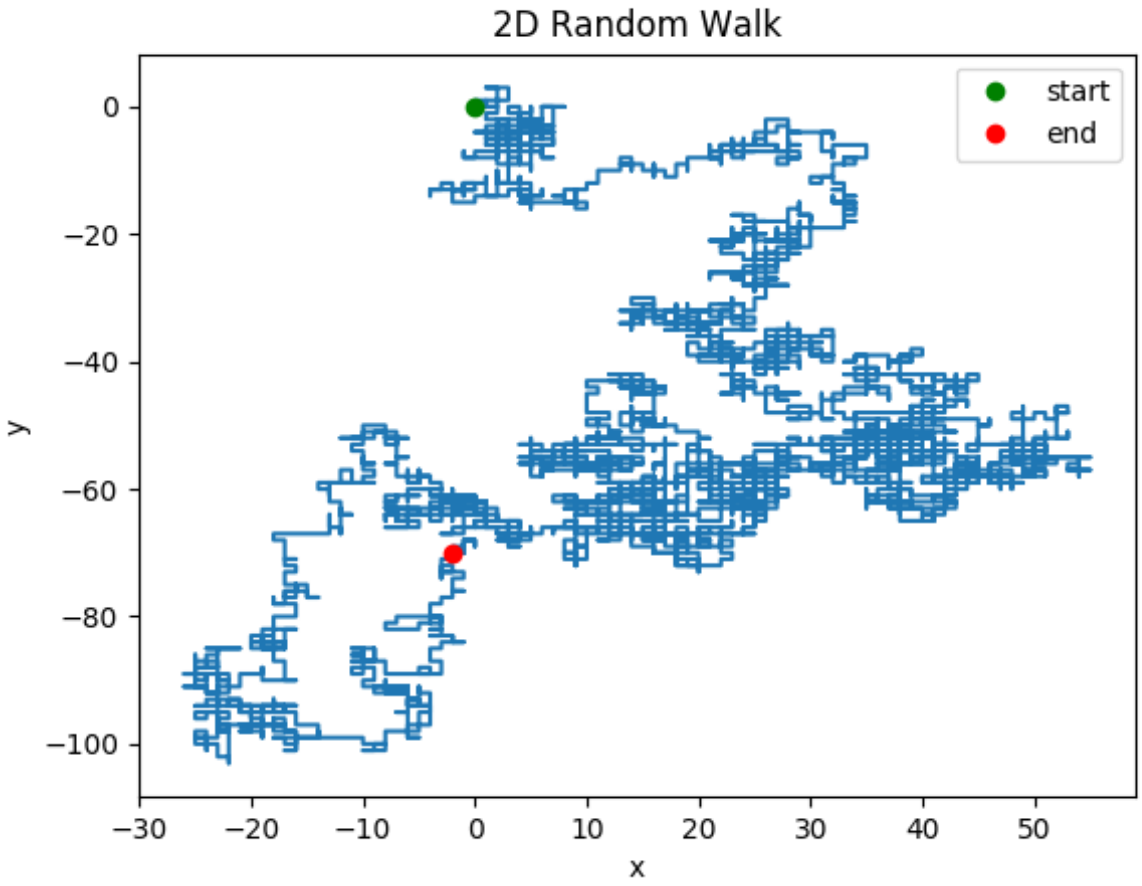


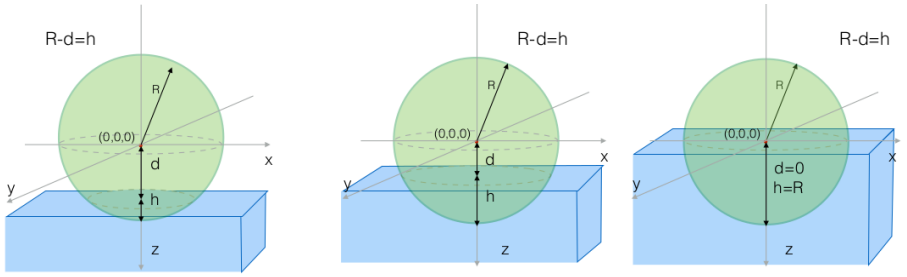
Figura 8.2: Grafico finale di un cammino aleatorio bidimensionale di `N=5000` passi.

Capitolo 9

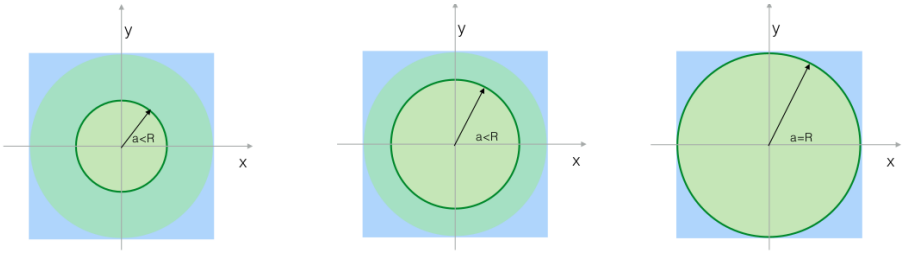
Spinta di Archimede su una sfera

9.1 Testo dell'esercizio

Il principio di Archimede stabilisce che un corpo immerso in un fluido subisce una spinta dal basso verso l'alto pari al peso del liquido spostato. La forza che ne risulta è detta spinta di Archimede ed è pari a $F = g \cdot \rho \cdot V$ dove $g = 9.8 \left[\text{ m/s}^2 \right]$ è l'accelerazione di gravità, $\rho \left[\text{kg/m}^3 \right]$ è la densità del liquido e $V \left[\text{ m}^3 \right]$ e' il volume di liquido spostato. **Scopo dell'esercizio è calcolare la spinta di Archimede che agisce su una sfera immersa in un liquido, per diverse altezze del liquido. Si trovi graficamente la condizione di equilibrio in cui la forza peso della sfera è controbilanciata dalla spinta di Archimede.**



(a) R = Raggio della sfera; d =altezza del liquido dal centro della sfera (origine assi); h = altezza della calotta sferica che si crea dall'intersezione del liquido con la sfera.



(b) Circonferenza della base della calotta.

Figura 9.1

► **Prima parte:**
Si produca un codice chiamato `spinta.c`. Il codice:

- **Stampa un messaggio iniziale** per spiegare all'utente che cosa fa il programma.
- **Chiede in input** il **raggio** della sfera R (in metri), la sua **massa** m (in kg) e la **densità** ρ (in kg/m^3) del fluido e controlla che tali valori siano **positivi**. Nel caso di inserimento di un numero negativo o nullo il programma stampa un messaggio di errore e chiede di nuovo il dato input.
- **Definisce una funzione `Calotta` che calcola il volume del liquido spostato** che corrisponde al volume della calotta sferica immersa nel liquido. Per maggior chiarezza si osservi la Figura 9.1.

  1. **La funzione deve restituire il volume della calotta sferica** e deve prendere in input il raggio R della sfera e la distanza d della superficie del fluido dal centro della sfera (fissato nell'origine degli assi, nel caso in cui quest'ultimo dovesse trovarsi al di sotto della superficie del liquido d assumerebbe valori negativi ma questo viene evitato per i calcoli successivi della spinta di Archimede).
  2. La funzione calcola il volume V_{MC} della calotta **attraverso l'integrazione Monte Carlo**. A tal fine, si definisca attraverso una direttiva al preprocessore il numero di punti `NMAX` $= 1000000$ da inserire in maniera casuale nel volume definito dal parallelepipedo V_{paral} di estremi $[-R, R] \times [-R, R] \times [d, R]$. Ciò significa che ogni punto inserito ha coordinate (x, y, z) estratte casualmente negli intervalli $x \in [-R, R], y \in [-R, R]$ e $z \in [d, R]$. Il metodo Monte Carlo consiste nel contare il numero n_p di punti estratti che si trovano all'interno della sfera, ossia tutti quei punti (x, y, z) tali che $x^2 + y^2 + z^2 \leq R^2$. Il volume della calotta sferica si ricava dalla relazione $V_{\text{MC}} = V_{\text{paral}} \cdot n_p/\texttt{NMAX}$.
  3. Per controllare che il volume sia stato calcolato correttamente, si ricorda che il volume della calotta sferica è $V_{\text{Calotta}} = \pi h^2(R - h/3.0)$ dove $h = R - d$. A tal proposito la funzione deve contenere un controllo che **restituisce un messaggio di errore ed esce dal programma se** $(|V_{\text{MC}} - V_{\textbf{Calotta}}|/V_{\textbf{Calotta}}) > \varepsilon$ **con** $\varepsilon = 0.01$.
- **Calcola il valore della spinta di Archimede (senza segno)** usando la formula $F = g \cdot \rho \cdot V_{\text{MC}}$. Il calcolo della forza deve essere **ripetuto `NITER` $= 15$ volte** all'interno di un ciclo su `NITER` dove viene automaticamente modificato il valore di d da passare alla funzione Calotta. In particolare si iteri il processo **per** d **compreso tra** $[0, 0.8R]$. Il valore della forza a diversi d viene **salvato nell'array `Forza`**.
- **Definisce una funzione `PrintResults` che stampa su di un file chiamato `Archimede.dat` tre colonne**: il valore di d ad ogni passo (anch'esso salvato precedentemente in un array), la spinta di Archimede F e la forza peso $P = m \cdot g$ relativa alla sfera. Ogni elemento delle 3 colonne deve essere stampato **con almeno 10 caratteri di cui 4 caratteri dopo la virgola**.

► **Seconda parte:**
Si esegua il codice `spinta.c` con i seguenti dati in input: $R = 0.05$ m, $m = 0.1$ kg e $\rho = 997$ Kg/m^3. Con un opportuno script python chiamato `archimede.py` **si grafichino** F **in funzione di** d e P **in funzione di** d **(nello stesso grafico)**, contenute nel file `archimede.dat`. Si salvi il grafico nel

file `archimede.png`. Il punto d'intersezione tra le due curve indica per quale d le due forze sono bilanciate.

9.2 Soluzione completa

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#define NMAX 1000000
#define G 9.81
#define NITER 15
#define EPS 0.01
#define SQR (a*a)

//il programma calcola la spinta di archimede generata dall'acqua
//su una sfera di massa m=XXX (da inserire in input) quando la
//sfera si trova a diverse altezze nel fluido
//Il volume di sfera immerso viene calcolato con il
//metodo Monte Carlo

//compilare con gcc -o spinta.x spinta.c -lm
/*************************************************************/
/*DICHIARAZIONE DI FUNZIONI*/
/*tipo nome_funzione (TIPI DEI VALORI IN IN INPUT);*/

void PrintMessage(void);
void inserimento(char *, char *, void *);
double Calotta(double, double);
void PrintResults(double*, double*, double);
double randval();
/*************************************************************/
/**********************MAIN*******************************/

int main(){
  double Raggio,massa,Forza[NITER],Peso;
  double dens,Volume,d[NITER],dd;
  int i;

  srand48(time(NULL));
  PrintMessage();
  /*************************************************************/
  /*LETTURA DEI DATI IN INPUT CON CONTROLLO SUI VALORI INSERITI*/
  do{
```

```
    inserimento("Inserire il raggio della sfera (metri): ","%lf",&Raggio);
    if(Raggio<=0) {
      fprintf(stderr,"Attenzione! il Raggio non puo' essere <= 0 \n");
    }
  }while(Raggio <=0);

  do{
    inserimento("Inserire la massa della sfera (Kg): ","%lf",&massa);
    if(massa<=0){
      fprintf(stderr,"Attenzione! la massa non puo' essere <= 0 \n");
    }
  }while(massa <=0);

  do{
    inserimento("Inserire la densita' del fluido in cui e' immerso (kq/m^3): ","%
   lf",&dens);
    if(dens<=0){
      fprintf(stderr,"Attenzione! la densita' non puo' essere <= 0 \n");
    }
  }while(dens<=0);

  /*************************************************************/
  /*CALCOLO DEL VOLUME DELLA SFERA IMMERSA NELL'ACQUA*/
  /*E DELLA SPINTA DI ARCHIMEDE*/

  dd=0.8*Raggio/(NITER-1);
  Peso=massa*G;
  for(i=0;i<NITER;i++){
    d[i]=i*dd;
    Volume=Calotta(d[i],Raggio);
    Forza[i]=G*dens*Volume;
  }

  /*************************************************************/
  /*STAMPA DEI RISULTATI*/
  PrintResults(d,Forza,Peso);

  return(0);
}

/*************************************************************/
/*************DEFINIZIONE DELLE FUNZIONI**********************/

/*tipo nome_funzione (valori in input){
    blocco del codice
}*/
```

```
85
86
87
88 /**********PRINT MESSAGE**********/
89
90 void PrintMessage(void){
91   fprintf(stdout,"*****************************************\n");
92   fprintf(stdout,"QUESTO PROGRAMMA CALCOLA LA SPINTA DI ARCHIMEDE\n");
93   fprintf(stdout,"SU UNA SFERA DI RAGGIO R PARZIALMENTE IMMERSA\n");
94   fprintf(stdout,"NELL'ACQUA\n");
95   fprintf(stdout,"*****************************************\n");
96   fprintf(stdout,"\n");
97 }
98
99 /**********CALOTTA**********/
100
101 //la calotta si calcola com l'intersezione tra la sfera
102 //e un parallelepipedo di base R^2 e altezza h
103 //dove h e' l'altezza della calotta sferica
104
105 double Calotta(double dist,double R){
106   double V=0;
107   double h;
108   double Vparal;
109   int n_p=0;
110   int i;
111   double VMC;
112   double x,y,z;
113
114   h=R-dist;
115   Vparal=SQR(2.0*R)*h;
116
117   for(i=0;i<NMAX;i++){
118     x=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
119     y=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
120     z=(R-dist)*randval()+dist;//estraggo un n. tra [d,R]
121
122     if(SQR(x)+SQR(y)+SQR(z)<=SQR(R)){
123       n_p++;
124     }
125   }
126   VMC=(n_p/(double)NMAX)*Vparal;
127   V=M_PI*SQR(h)*(R-(h/3.0));//volume calcolato analiticamente
128
129   if((fabs(VMC-V)/V)>EPS){
```

```
130     fprintf(stderr,"Il volume calcolato e stimato con l'integrazione non
        coincidono entro l'errore\n");
131     fprintf(stderr,"V stimato = %lf \t V vero= %lf\n",VMC,V);
132     exit(1);
133   }
134   fprintf(stderr,"V stimato = %lf \t V vero= %lf errore relativo %lf\n",VMC,V,(
        fabs(VMC-V)/V));
135   return(VMC);
136 }
137
138 /**********RANDVAL**********/
139
140 double randval(){
141 //genera n. random tra [0,1]
142  return((double)lrand48()/RAND_MAX);
143 }
144
145 //la funzione inserimento permette un controllo maggiore sull'input dei dati
146 //I parametri della funzione sono due puntatori
147 //di tipo char e un puntatore di tipo void
148 //il primo puntatore *message passa in input alla funzione
149 //la stringa di testo che si vuole far comparire sul terminale
150 //(es. Inserire il raggio della sfera )
151 //Il secondo puntatore *format punta all'array che conterra'
152 //la stringa di formato voluta da scanf (ad es la stringa di formato "%lf" )
153 //il puntatore di tipo void punta alla variabile che conterra'
154 // il valore letto dallo scanf. Questa variabile puo' essere
155 //int, double, float ecc...il puntatore di tipo void permette
156 //di puntare a ciascuno di queste variabili
157
158 /**********INSERIMENTO**********/
159
160 void inserimento(char *message, char *format, void *var){
161   int fine, res=0;
162   fine=0;
163   char junk[80];
164   //finche' res non restituisce 1
165   while (!fine){ // continuo ad eseguire le istruzioni finche' la lettura da
        terminale non e' andata a buon fine.
166     fprintf(stdout,"%s",message);//stampo il messaggio contenuto in message
167     fine=1;
168     res=fscanf(stdin,format,var);//leggo da terminale usando la stringa di
        formato format e memorizzando il dato in var (che e' un puntatore quindi NO &
        in scanf)
169
170     if(res!=1){//se lo scanf non e' andato a buon fine
```

```
      fprintf(stderr,"Problema nella lettura da schermo\n");
      fine=0;
    }
    if(fgets(junk, 80, stdin) != NULL && (junk[0] !='\n')){
    //se e' rimasto qualcosa nel buffer che non e' il carattere di ritorno

      fprintf(stderr,"c'e' qualcosa che e' rimasto nel buffer: ");
      fprintf(stderr,"%s\n", junk);//stampo quello che e' rimasto nel buffer
      fine=0;
    }
  }
}

/**********PRINT RESULTS**********/

void PrintResults(double d[],double force[],double weight){
  FILE *fp;
  int i;
  fp=fopen("Archimede.dat","w");
  for(i=0;i<NITER;i++){
    fprintf(fp,"%10.4lf %10.4lf %10.4lf\n",d[i],force[i],weight);
  }
  fclose(fp);
}
```

9.3 Commento alla soluzione

► **Soluzione della prima parte:**
Si includono innanzitutto le librerie standard `stdio.h` e `stdlib.h`.
Il messaggio iniziale può essere stampato direttamente nel `main` usando un semplice `printf` o un `fprintf` sullo `stdout`; oppure può essere racchiuso all'interno di una funzione `PrintMessage` di tipo `void` che non riceve nessun argomento in input, chiamata all'inizio del programma.

```
void PrintMessage(void) {
    fprintf(stdout, "Questo programma calcola...");
}
```

Si dichiarano le variabili `Raggio`,`massa`,`dens` di tipo `double` su cui salvare i valori di raggio, massa e densità inseriti dall'utente. Il modo più semplice per leggere tali valori dall'input è tramite la funzione `scanf`, la quale richiede un **formato** (`%lf` per il tipo `double`) ed un **puntatore alla variabile su cui salvare il valore**. Nel caso in cui il valore inserito non sia positivo, si stampa un messaggio di errore tramite un `if`; un ciclo `do...while` permette di reiterare la richiesta di inserimento di input finché la condizione data non è soddisfatta:

```
int main() {
```

```
    double Raggio,massa,dens;
    PrintMessage();
    do {
        printf("Inserire il raggio della sfera (metri):");
        scanf("%lf",&Raggio);
        if(Raggio<=0)
            printf("Attenzione! Il Raggio non puo' essere <= 0 \n");
    } while(Raggio <=0);
}
```

Ripetere il codice precedente per le variabili `massa` e `dens` (o equivalentemente, definire una funzione che racchiuda il codice precedente e prenda in input il nome ed il puntatore ad una variabile, e chiamarla 3 volte). *Opzionale: nel listato completo è mostrato un metodo più avanzato di gestione dell'input, contenuto nella funzione* `inseriment()o`.
Si esaminano ora i passaggi per la costruzione della funzione `Calotta`, che calcola il volume della calotta sferica di raggio R e altezza d tramite metodo Monte Carlo .

1. Si evince che la definizione del prototipo deve essere:

```
double Calotta(double R, double d);
```

2. Il comando per definire `NMAX` al preprocessore è:

```
#define NMAX 1000000
```

   La Sezione 14 contiene un riepilogo dei metodi per generare numeri casuali tramite le funzioni `drand48`, `lrand48`. Bisogna includere la libreria `time.h` per **inizializzare il seed all'inizio del `main`** tramite il comando:

```
#include<time.h>
...
srand48(time(NULL))
```

   Si vuole estrarre `NMAX` terne di numeri casuali $x \in [-R, R], y \in [-R, R]$ e $z \in [d, R]$. Un possibile metodo è mostrato nel codice seguente:

```
double randval(void) {
    return ((double)lrand48()/RAND_MAX);//estraggo un n. tra [0,1]
}
double Calotta(double R, double d) {
    int i;
    double x,y,z;
    for(i=0 ; i < NMAX ; i++) {
        x=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
        y=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
```

```
        z=(R-d)*randval() + d;//estraggo un n. tra [d,R]
    }
    ...
}
```

Per ogni punto estratto si verifica se appartiene alla sfera, se tale condizione è soddisfatta si incrementa un contatore n_p (inizializzato a 0). Il volume della calotta è dato dunque da $V_{\text{MC}} = V_{\text{paral}} \cdot n_p/\texttt{NMAX}$, dove $V_{\text{paral}} = 2R * 2R * h$ ($h = R - d$) è il volume del parallelepipedo all'interno del quale vengono estratti i numeri casuali.
Si ricordi di effettuare il casting a `double` durante la divisione fra gli interi n_p e `NMAX`, altrimenti l'operazione viene interpretata come una divisione fra interi e (visto che $n_p <$ `NMAX`) il risultato è troncato a 0.

```
double Calotta(double R, double d) {
    int i;
    int n_p=0;
    double x,y,z;
    double h,VMC,Vparal;
    h=R-d;
    Vparal = (2*R)*(2*R)*h;//volume in cui estraggo

    for(i=0 ; i < NMAX ; i++) {
        x=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
        y=2.0*R*(randval()-0.5);//estraggo un n. tra [-R,R]
        z=(R-d)*randval() + d;//estraggo un n. tra [d,R]

        if(x*x + y*y + z*z <= R*R) {
            n_p++;//conto i punti nella calotta
        }
    }
    VMC=(n_p/(double)NMAX)*Vparal;
    return VMC;
}
```

3. Verificare con un `if` se la distanza relativa **in modulo** fra V_{MC} e $V_{\text{Calotta}} = \pi h^2(R - h/3.0)$ sia minore di un $\varepsilon = 0.01$ definito al preprocessore. Se ciò non è verificato, interrompere l'esecuzione del programma con il comando `exit(1)` dando un messaggio d'errore. La funzione `fabs` che restituisce il valore assoluto è contenuta nella libreria `math.h`.

```
...
#include <math.h>
#define EPS 0.01 //massimo errore relativo accettato

double Calotta(double R, double d) {
    double V,err;
```

```
    ...
    VMC=(n_p/(double)NMAX)*Vparal;
    V=M_PI*h*h*(R-(h/3.0));//volume calcolato analiticamente
    err=fabs(VMC-V)/V;//errore relativo, in modulo
    if(err>EPS) {
        fprintf(stderr,"Il volume calcolato e stimato con l'integrazione non
    coincidono entro l'errore relativo %lf\n",EPS);
        fprintf(stderr,"V stimato = %lf \t V vero= %lf\n",VMC,V);
        exit(1);
    }
    fprintf(stderr,"V stimato = %lf \t V vero= %lf errore relativo %lf\n",VMC
    ,V,err);
    return VMC;
}
```

Una volta definita la funzione `Calotta`, si vuole calcolare la spinta di Archimede $F = g \cdot \rho \cdot V_{\text{MC}}$ per 15 valori della distanza $d \in [0, 0.8R]$, e salvare i risultati in un array chiamato `Forza`. Innanzitutto si dichiara un numero di iterazioni `NITER` e la accelerazione di gravità `G` al preprocessore; nel `main` si dichiara un array `Forza` di tipo `double` e dimensione `NITER`. In un ciclo su i che va da 0 a $NITER-1$, si chiama la funzione `Calotta` per valori diversi di d; la trasformazione da $i \in [0, NITER-1]$ a $d \in [0, 0.8R]$ è $d = i \cdot d_{\text{step}}$, con $d_{\text{step}} = 0.8R/(NITER-1)$.

```
...
#define NITER 15
#define G 9.81 //accelerazione di gravità (m/s^2)
int main(void) {
    double Forza[NITER], dstep, Volume;
    ...
    dstep=0.8*Raggio/(NITER-1);
    for (i=0 ; i<NITER ; i++) {
        Volume=Calotta(i*dstep,Raggio);
        Forza[i]=G*dens*Volume;
    }
}
```

Infine si vuole stampare in un file `Archimede.dat` una tabella di 3 colonne contenente la distanza d, la spinta di Archimede `Forza` e la forza peso, usando una funzione. Quest'ultima condizione impone di salvare precedentemente tutti i dati in corrispettivi array; bisogna dunque modificare il codice precedente per salvare il valore di ogni distanza in un array `d[NITER]` di tipo `double`; per la forza peso invece è sufficiente invece salvarla in una variabile `Peso` di tipo `double`, poiché non varia con d. La funzione `PrintResults` non restituisce nulla e deve ricevere in input i due array *by reference* ma può ricevere la forza peso *by value* poiché essa non viene modificata dalla funzione.

```
void PrintResults(double *, double *, double) {
...
}
```

```c
int main(){
    double d[NITER], Forza[NITER], Peso;
    ...
    dstep=0.8*Raggio/(NITER-1);
    for(i=0;i<NITER;i++){
        d[i]=i*dstep;
        Forza[i] = G*dens*Calotta(d[i],Raggio);
    }
    Peso=massa*G;
    PrintResults(d,Forza,Peso);

    return(0);
}
```

Al suo interno, la funzione `PrintResults` deve aprire il file `Archimede.dat` in modalità scrittura e fare un ciclo di `NITER` iterazioni stampando in ciascuna riga i valori di `d,Forza,Peso` in formato `"%10.4lf"`. Si veda la Sezione 11 per un riepilogo sulla scrittura su file. Il codice di `PrintResults` è riportato nel listato completo.

▶ **Soluzione della seconda parte:**
Si veda la Sezione 16 su come graficare una tabella di dati e savarla in un'immagine.

```python
import matplotlib.pyplot as plt
import numpy as np

plt.title('Forze in azione quando una sfera viene immersa in un fluido')
x,y,z = np.loadtxt ('Archimede.dat', unpack = True)

plt.plot(x, y, 'o-',color='C0',label='Spinta di Archimede')
plt.plot(x, z, 'x-',color='C1',label='Forza Peso')

plt.xlabel('Distanza centro della sfera dal fluido')
plt.ylabel('Intensita\' delle forze')

plt.legend()
plt.savefig('Archimede.png')
plt.show()
```

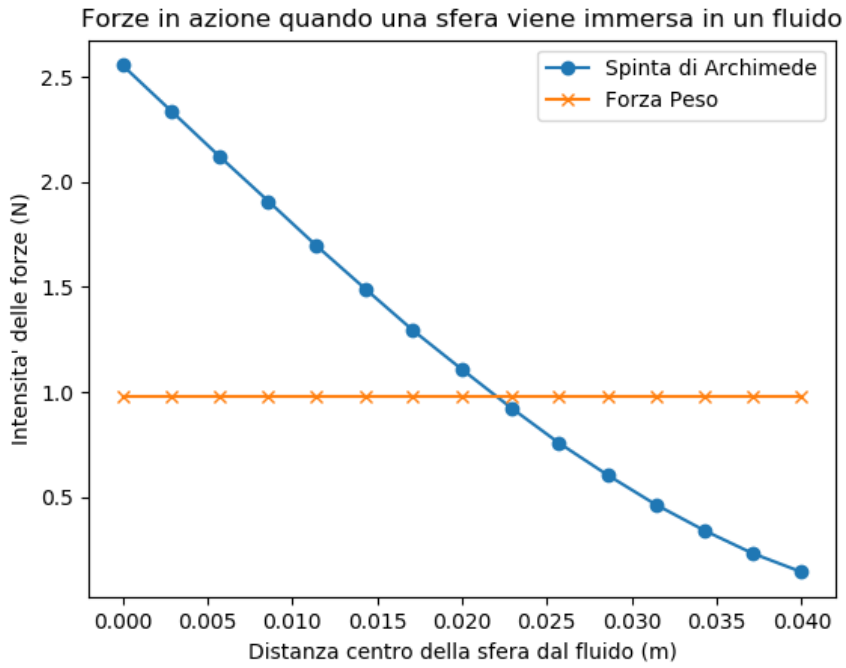


Figura 9.2: Grafico delle forze corrispondenti ai valori $R = 0.05$ m, $m = 0.1$ kg e $\rho = 997$ Kg/m^3. Il punto d'intersezione indica che la sfera è in equilibrio ad una distanza $d = 2.2$ cm sulla superficie del liquido.

Capitolo 10

Forbici per DNA

10.1 Testo dell'esercizio

Il DNA è una catena costituita da nucleotidi di tipo A, T, C, G il cui ordine costituisce l'informazione genetica. Spesso i genetisti sono interessati a piccole sequenze di nucleotidi (geni) e per questo hanno bisogno solo di piccoli frammenti di DNA. Essi utilizzano quindi degli enzimi particolari che agiscono come delle forbici molecolari che tagliano il DNA in punti specifici. Si formano così dei frammenti di DNA di lunghezza differente. Per poter continuare a lavorare con il frammento di DNA sul quale si trova il gene d'interesse, il genetista deve isolarlo. A questo proposito deve riuscire a separare un frammento di DNA avente una determinata lunghezza da tutti gli altri a cui non è interessato. La separazione basata sulla lunghezza dei frammenti di DNA avviene comunemente tramite un metodo chiamato "elettroforesi su gel".
Scopo di questa esercitazione è quella di simulare l'azione di un enzima che effettua un taglio nella sequenza di DNA ogni volta che trova un nucleotide G e di caratterizzare i frammenti di DNA risultanti in base alla loro lunghezza, calcolando l'istogramma della lunghezza dei frammenti.

► **Prima parte:**
La prima parte dell'esercizio consiste nello scrivere un **codice chiamato** `enzima.c` che:

1. **Definisca, attraverso una direttiva al preprocessore, la lunghezza** `Lmax = 10000` della sequenza iniziale di DNA;
2. **Stampi un messaggio iniziale** per spiegare all'utente **che cosa fa il programma**;
3. **Chieda in input il numero di tagli** `N` che si vuole simulare specificando che **il numero deve essere compreso tra 2000 e 5000. Se questa richiesta non viene soddisfatta, il programma stampa un messaggio di errore e richiede di nuovo in input il dato**;
4. **Contenga una funzione** `DesignDNA` che inizializzi una sequenza di DNA nel seguente modo:
   (a) La funzione inizialmente **riempie un array di tipo** `char` **chiamato** `dna` (**passato in input alla funzione**), con nucleotidi di tipo A, C e T. **Il tipo di nucleotide da assegnare ad ogni elemento dell'array (lettera 'A','C' o 'T') viene scelto in maniera casuale con uguale probabilità;**

(b) Successivamente, **la funzione modifica** `N` nucleotidi della sequenza appena generata con dei nucleotidi di tipo G. **Questo viene fatto estraendo `N` volte in maniera casuale le posizioni dell'array dove inserire la lettera** 'G'. Durante questa procedura, **la funzione verifica se nella posizione appena estratta sia già stato sostituito un nucleotide** G e in caso **estrae una nuova posizione**. Inoltre gli elementi dell'array dove è possibile la sostituzione dei nucleotidi G sono tutti **tranne il primo e l'ultimo elemento.**

5. **Contenga una funzione chiamata** `Cut` che **scorra l'array** `dna` precedentemente riempito, e **memorizzi in un array di interi** chiamato `taglio` (**anch'esso passato in input alla funzione**) **i punti della sequenza dove l'enzima effettua un taglio** (ossia le posizioni dove si trovano i nucleotidi di tipo G). Ad esempio, se i primi tre nucleotide di tipo G si trovano nelle posizioni 3, 9 e 14 dell'array `dna`, allora `taglio[0]=3`, `taglio[1]=9` e `taglio[2]=14`;

6. **Contenga una funzione chiamata** `AnalisiL` che calcoli l'**istogramma** `H(l)` **delle lunghezze** `l` **dei frammenti di DNA stampando le due colonne** `l H(l)` nel **file** `Histo.dat` con una **larghezza minima di quattro caratteri ciascuna**. La lunghezza dei segmenti si può calcolare una volta noti i punti della sequenza dove sono stati effettuati i tagli (e che sono stati memorizzati nell'array `taglio`). Infatti la differenza tra le posizioni di due tagli consecutivi `i` e `i-1` (ad esempio `[1]-taglio[0]=6`) corrisponde alla lunghezza del frammento creatosi da quei due tagli (un frammento di 6 nucleotidi in questo caso). Si faccia **attenzione a considerare il frammento iniziale e finale della sequenza**. Per maggior chiarezza si osservi la figura nel compito.

► **Seconda parte:**
La seconda parte dell'esercizio consiste nel preparare **uno script chiamato** `Histo.py` che **grafichi l'istogramma contenuto in** `Histo.dat`. **Si salvi il grafico in** `Histo.png`. Poiché ci si aspetta una distribuzione esponenziale, si consiglia di **utilizzare una scala logaritmica sull'asse y** (tramite il **comando** `plt.yscale("log")`, da **inserire prima di** `plt.show()`) per controllare che i risultati

siano sensati. Si ricorda che il **comando per aggiungere un istogramma al grafico è** `plt.bar` al posto di `plt.plot`.

10.2 Soluzione completa

▶ **Parte in C**:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#define LMAX 10000
#define MMAX 5000
#define MMIN 2000

char numtochar(int num);
void DesignDNA (char code[], int cuts);
void Cut(char code[],int track[]);
void AnalisiL(int track[],int ncuts);

int main(){
  int n;
  char dna[LMAX];
  int taglio[MMAX];
  srand48(time(NULL));
  // Messaggio iniziale
  printf("*****************************************************\n");
  printf("QUESTO PROGRAMMA EFFETTUA UN TAGLIO IN UNA SEQUENZA \n");
  printf("DI DNA OGNI VOLTA CHE INCONTRA UN NUCLEOTIDE DI TIPO G\n");
  printf("E ANALIZZA I FRAMMENTI DI DNA RISULTANTI CALCOLANDO LA\n");
  printf("DISTRIBUZIONE DELLE LORO LUNGHEZZE\n");
  printf("*****************************************************\n");
  printf("\n");
  printf("Inserisci il numero di tagli che vuoi simulare nell'intervallo [%d,%d
    ]:\n",MMIN,MMAX);
  do{
    scanf("%d",&n);
    if(n<MMIN || n>MMAX){
      printf("ERRORE: VALORE NON CONSENTITO. REINSERIRE\n");
    }
  }while(n<MMIN || n>MMAX);
  DesignDNA (dna,n);
  Cut(dna,taglio);
  AnalisiL(taglio,n);
 return(0);
```

```
}

char numtochar(int num){
  char base;
  if(num==0)
    base='A';
  if(num==1)
    base='T';
  if(num==2)
    base='C';
  return(base);
}

void DesignDNA (char code[], int cuts){
  int i=0;
  int val;
   for(i=0;i<LMAX;i++){
    val=(int)(drand48()*3);//[0,2] inserisce a caso un nucleotide di tipo A,C,T
    code[i]=numtochar(val);
  }

  for(i=0;i<cuts;i++){
   do{
      val=(int)(drand48()*(LMAX-2))+1;//[1,LMAX-2] inserisce random le G nella
   sequenza
    }while(code[val]=='G');
      code[val]='G';
  }
}

void Cut(char code[],int track[]){
  int i,count=0;
  for(i=0;i<LMAX;i++){
    //memorizzo in un array le posizioni dove l'enzima effettua i tagli
    if(code[i]=='G'){
      track[count]=i;
      count ++;
    }
  }
}


void AnalisiL(int track[],int ncuts){
  int segments;
  int histoL[LMAX-MMIN]={0};
  int i;
```

```
  FILE *fp;
  fp=fopen("Histo.dat","w");
  //calcolo il primo frammento della sequenza (devo sempre aggiungere +1)
  segments=track[0]+1;
  histoL[segments]++;
  for(i=1;i<ncuts;i++){
    segments=track[i]-track[i-1];//calcola la lunghezza dei segmenti facendo la
   distanza tra i tagli
    histoL[segments]++;

  }
  //calcolo l'ultimo frammento della sequenza
  segments=(LMAX-1)-track[ncuts-1];
  histoL[segments]++;
  for(i=1;i<(LMAX-MMIN);i++){
    if(histoL[i]>0){
      fprintf(fp, "%4i %4i\n",i,histoL[i]);
    }
  }
  fclose(fp);
}
```

► **Parte in Python**:

```
import numpy as np
import matplotlib.pyplot as plt

plt.title('Istogramma dei frammenti')
plt.xlabel('Lunghezza frammento')
plt.ylabel('Occorrenza')

x ,y=np.loadtxt("Histo.dat", unpack=True)
plt.bar(x, y)
plt.yscale('log')

plt.savefig('Histo.png')
plt.show()
```

10.3 Commento alla soluzione

► **Soluzione prima parte:**

1. Definiamo la lunghezza massima e in vista dei punti successivi anche il numero di tagli minimo e massimo:

```c
#define LMAX 10000
#define MMAX 5000
#define MMIN 2000
```

2. Questa parte è a discrezione dello studente. Si riporta un listato a titolo di esempio:

```c
int main(){
    // Messaggio iniziale
    printf("*********************************************************\n");
    printf("QUESTO PROGRAMMA EFFETTUA UN TAGLIO IN UNA SEQUENZA \n");
    printf("DI DNA OGNI VOLTA CHE INCONTRA UN NUCLEOTIDE DI TIPO G\n");
    printf("E ANALIZZA I FRAMMENTI DI DNA RISULTANTI CALCOLANDO LA\n");
    printf("DISTRIBUZIONE DELLE LORO LUNGHEZZE\n");
    printf("*********************************************************\n");
    printf("\n");
}
```

3. La parte di richiesta dell'input si fa agilmente utilizzando le funzioni `printf` e `scanf`. Per fare in modo che la richiesta venga reiterata utilizziamo il costrutto `do-while` mettendo come controllo il numero di tagli:

```c
int main(){
    int n;
    ...
    printf("Inserisci il numero di tagli che vuoi simulare nell'intervallo [%
    d,%d]:\n",MMIN,MMAX);
    do{
        scanf("%d",&n);
        if(n<MMIN || n>MMAX){
          printf("ERRORE: VALORE NON CONSENTITO. REINSERIRE\n");
        }
    }while(n<MMIN || n>MMAX);
}
```

4. Cominciamo a costruire `DesignDNA`:

   (a) Per questa prima parte dobbiamo riempire un array che è l'unico argomento per ora da passare alla funzione:

   ```c
   void DesignDNA (char code[]);
   ```

   In libreria non esistono funzioni che generano `char` random ma interi sì. Sfruttiamo quindi questo fatto introducendo una funzione chiamata `numtochar` che dato in input un numero intero restituisce una lettera tra A,C o T:

```
char numtochar(int num){
    char base;
    if(num==0)
    base='A';
    if(num==1)
    base='T';
    if(num==2)
    base='C';
    return(base);
}
```

A questo punto nella funzione `DesignDNA` è sufficiente generare dei numeri random interi tra 0 e 2 utilizzando `rand` e poi convertirli con `numtochar` e con un ciclo `for` riempire tutto l'array:

```
void DesignDNA (char code[]){
    int i=0;
    int val;
    for(i=0;i<LMAX;i++){
        val=drand()*3;//[0,2] inserisce a caso un nucleotide di tipo A,C,
    T
        code[i]=numtochar(val);
    }
}
```

(b) Per la seconda parte abbiamo bisogno anche del numero di G da inserire che va passato come argomento (facciamo notare che il numero di G non è nient'altro che il numero di tagli):

```
void DesignDNA (char code[], int cuts);
```

Con un ciclo `for` estraiamo `cuts` posizioni casuali e per controllare se la posizione è libera o meno si può utilizzare un ciclo `do while`. Nel caso in cui la posizione sia occupata si ripete l'estrazione della posizione:

```
void DesignDNA (char code[], int cuts){
    for(i=0;i<cuts;i++){
        do{
            val=drand()%(LMAX-2)+1;//[1,LMAX-2] inserisce random le G
    nella sequenza
        }while(code[val]=='G');
    code[val]='G';
    }
}
```

N.B. l'ultima posizione dell'array non è `LMAX` ma `LMAX-1` quindi le posizione vanno estratte tra 1 e `LMAX-2` se si vogliono escludere gli estremi.

Inseriamo `DesignDNA` all'interno del `main`:

```
int main(){
    char dna[LMAX];
    ... \\Numero di tagli
    DesignDNA (dna,n);
}
```

5. Passiamo alla funzione `Cut` che per come è descritta nel teso deve avere come argomenti due array: il dna e i posti dei tagli.

```
void Cut(char code[],int track[]);
```

Per riempire l'array `track` scorriamo tutto `code` con un ciclo `for` e per ogni posizione controlliamo se contiene la lettera G. In caso affermativo salviamo la posizione e per fare questo abbiamo bisogno di una variabile che scorra l'array `track` mano a mano che viene riempito:

```
void Cut(char code[],int track[]){
    int i,count=0;
    for(i=0;i<LMAX;i++){
        //memorizzo in un array le posizioni dove l'enzima effettua i tagli
        if(code[i]=='G'){
            track[count]=i;
            count ++;
        }
    }
}
```

Nel `main` la funzione `Cut` va messa subito dopo `DesignDNA`. Le posizioni dei tagli sono contenute nell'array `taglio` che al massimo conterrà `MMAX` tagli:

```
int main(){
    int taglio[MMAX];
    ...
    DesignDNA (dna,n);
    Cut(dna,taglio);
}
```

6. La funzione `AnalisiL` per creare l'istogramma ha bisogno dell'array `taglio` per calcolare la lunghezza dei frammenti, e quindi anche del numero totale dei tagli per sapere quanto è lungo effettivamente l'array.

```
void AnalisiL(int track[],int ncuts);
```

La parte più complessa è capire come creare l'istogramma. Definiamo un array `histoL` in modo che nella posizione i-esima ci sia il numero di volte che compare un frammento lungo i. La lunghezza massima di un frammento è `LMAX-MMIN`, che si forma quando si sceglie di effettuare `MMIN` tagli e le `G` vengono messe tutte negli ultimi posti. Ad esempio nel nostro caso le `G` sono messe dai posti 7999 a 9999 e quindi il frammento più lungo ha dimensione 8000 (da 0 a 7999). In alternativa si può scegliere di definire l'array lungo `LMAX` senza fare calcoli.

```
void AnalisiL(int track[],int ncuts){
    int histoL[LMAX-MMIN]={0};
}
```

Con un ciclo `for` scorriamo l'array `track`, calcoliamo la lunghezza dei segmenti e incrementiamo di volta in volta la posizione corrispondente:

```
void AnalisiL(int track[],int ncuts){
    int segments;
    int histoL[LMAX-MMIN]={0};
    int i;
    for(i=1;i<ncuts;i++){
        segments=track[i]-track[i-1];//calcola la lunghezza dei segmenti
    facendo la distanza tra i tagli
    histoL[segments]++;
    }
}
```

Come suggerito dal testo prestiamo attenzione al filamento iniziale e finale che non vengono conteggiati perché gli estremi non sono tagli. Per il frammento iniziale la lunghezza è pari alla posizione del primo taglio più 1 perché l'inizio di un array è 0. Se ad esempio `track[0]=3` vuol dire che ci sono 4 lettere: 0,1,2,3. Per il frammento finale la lunghezza è data dall'estremo finale (`LMAX-1`) e `track[ncuts-1]` (ultimo taglio).

```
void AnalisiL(int track[],int ncuts){
    int segments;
    int histoL[LMAX-MMIN]={0};
    int i;
    //calcolo il primo frammento della sequenza (devo sempre aggiungere +1)
    segments=track[0]+1;
    histoL[segments]++;
    for(i=1;i<ncuts;i++){
        segments=track[i]-track[i-1];//calcola la lunghezza dei segmenti
    facendo la distanza tra i tagli
        histoL[segments]++;
```

```
    }
    //calcolo l'ultimo frammento della sequenza
    segments=(LMAX-1)-track[ncuts-1];
    histoL[segments]++;
}
```

Per concludere stampiamo i risultati sul file `Histo.dat`. Siccome l'array `histoL` conterrà molte celle vuote, nel momento di stamparlo è meglio controllare che ogni locazione contenga almeno un conteggio altrimenti si salta:

```
void AnalisiL(int track[],int ncuts){
    FILE *fp;
    fp=fopen("Histo.dat","w");
    ...
    for(i=1;i<(LMAX-MMIN);i++){
        if(histoL[i]>0){
            fprintf(fp, "%4i %4i\n",i,histoL[i]);
        }
    }
    fclose(fp);
}
```

► **Soluzione seconda parte:** Come creare un istogramma con `pyhton` è spiegato in appendice §16. In questo caso è più conveniente utilizzare la funzione `plt.bar` che non `plt.hist` perché i bin sono già separati:

```
import numpy as np
import matplotlib.pyplot as plt

plt.title('Istogramma dei frammenti')
plt.xlabel('Lunghezza frammento')
plt.ylabel('Occorrenza')

x ,y=np.loadtxt("Histo.dat", unpack=True)
plt.bar(x, y)
plt.yscale('log')

plt.savefig('Histo.png')
plt.show()
```

Un esempio di istogramma finale è riportato in fig.10.1.

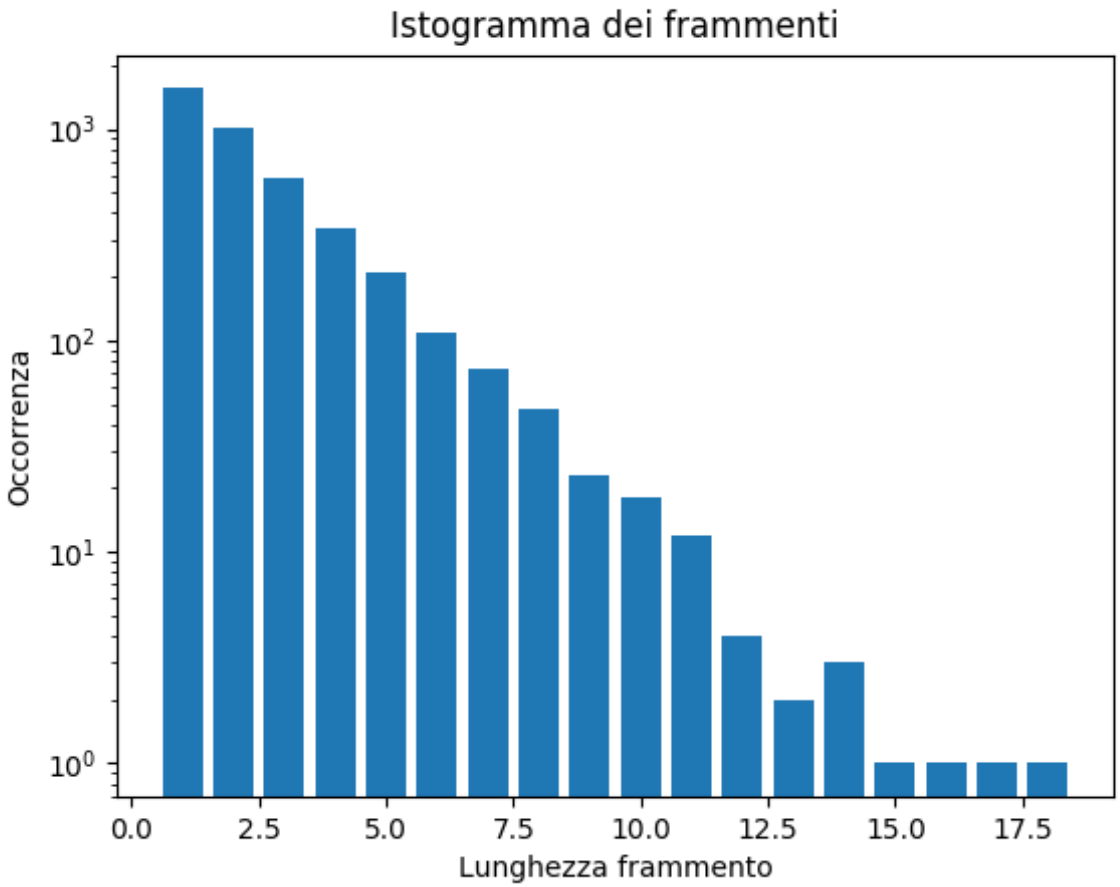


Figura 10.1: Esempio di istogramma finale.

Parte II

Riepilogo di argomenti ricorrenti

Questa appendice espone sinteticamente alcuni degli argomenti che ricorrono negli esercizi. Per una spiegazione esaustiva e dettagliata della teoria relativa, si fa riferimento al libro di testo: Barone, Marinari, Organtini, Ricci-Tersenghi, *Programmazione scientifica* (Pearson Education) per il linguaggio C, e alla documentazione di NumPy e di Matplotlib per Python.

Capitolo 11

Lettura e scrittura di dati da un file

Nel linguaggio C, ogni operazione che coinvolge la scrittura e/o creazione di file avviene attraverso un puntatore "speciale", detto puntatore a file (la cui definizione è contenuta nella libreria stdio.h) :

```
1    FILE *fp;
```

Notare come il tipo FILE sia stato scritto in maiuscolo.
Una volta dichiarato, il puntatore deve "puntare" ad un file sul quale si vogliono leggere o scrivere dei dati. Per aprire il file di interesse si usa la funzione fopen() (contenuta sempre nella libreria stdio.h), che richiede due argomenti. Il primo è una stringa che rappresenta il nome del file, mentre il secondo argomento è una stringa che rappresenta la modalità di apertura del file, come illustrato nella Tabella 11.1. Se il file viene aperto in modalità "scrittura" o "aggiunta", nel caso in cui tale file esista già, la funzione fopen() si limiterà ad aprirlo in scrittura, altrimenti verrà creato un file con il nome indicato dalla stringa. Nella modalità "aggiunta" quello che si scrive su file viene aggiunto al contenuto del file mentre nella modalità "scrittura" il contenuto esistente viene perso. Se il file viene invece aperto in modalità "lettura", questo viene aperto se esiste già, altrimenti viene prodotto un errore. Bisogna ricordare che siccome gli argomenti devono essere stringhe, vanno racchiusi all'interno di una coppia di virgolette (“ ”).

Stringa	Modalità
r	lettura
w	scrittura
a	aggiunta
r+ o w+	lettura e scrittura
rb	lettura binario
wb	scrittura binario

Tabella 11.1: Modalità di apertura di un file.

La funzione restituisce un puntatore a file. Se l'operazione non ha successo il puntatore assume il valore NULL. In un programma è sempre bene inserire un controllo che invii un messaggio di errore se il puntatore assume detto valore. In caso non ci siano problemi il puntatore punta al file e ne permette l'accesso.

```c
int main(){
    FILE *fp;
    fp=fopen("prova.txt","w");
    if(fp==NULL){
        printf("ERRORE NELL'APERTURA DEL FILE\n");
        return(0);
    }
}
```

Per scrivere su di un file si utilizza la funzione `fprintf()`, che ha una sintassi identica alla funzione `printf()`, con la differenza che prima di specificare che cosa si vuole scrivere, va passato alla funzione come ulteriore argomento il puntatore al file su cui si vuole scrivere. Per chiudere un file e liberare il puntatore, che può essere quindi riutilizzato, si usa la funzione `fclose()` che richiede come unico argomento il nome del puntatore da liberare. Di seguito un esempio di programma che scrive sul file *prova.txt* i primi 10 numeri triangolari.

```c
#include<stdio.h>
int main(){
    FILE *fp; // dichiaro il puntatore
    int i,sum=0;
    fp=fopen("prova.txt","w"); // apro il file prova.txt in modalita' scrittura
    for(i=1;i<=10;i++){
        sum+=i;
        fprintf(fp,"%d\n",sum); // scrivo su file

    }
    fclose(fp); //chiudo e libero il puntatore
}
```

Questa operazione è fondamentale se si vogliono scrivere dei dati su un file e leggerli successivamente all'interno dello stesso programma. Infatti, finché il file aperto in modalità scrittura non viene chiuso, i dati sono salvati nel *buffer* e non si trovano quindi fisicamente sul dispositivo dotato di memoria non volatile (disco fisso o SSD). Tuttavia , si può "forzare" la scrittura dei dati che si trovano nel buffer, tramite la funzione `fflush()` con argomento il puntatore a file. Inoltre è buona norma liberare sempre un puntatore dopo averlo utilizzato.
Se invece si vuole leggere ciò che è scritto su un file, si usa la funzione `fscanf()`, che ha la stessa sintassi della funzione `scanf()`, con la differenza che anche in questo caso il primo argomento passato alla funzione deve essere il puntatore al file da cui si vuole leggere. Nel listato seguente si apre lo stesso file in modalità scrittura per leggere i numeri che vi sono stati scritti nell'esempio precedente e salvarli in un array:

```c
#include<stdio.h>
int main(){
    FILE *fp;
    int i, sum=0, n[10];
```

```
    // SCRITTURA
    fp=fopen("prova.txt","w");
    for(i=1;i<=10;i++){
        sum+=i;
        fprintf(fp,"%d\n",sum);
    }
    fclose(fp);
    // LETTURA
    fp=fopen("prova.txt","r");
    for(i=0;i<10;i++){
        fscanf(fp,"%d",&n[i]);
    }
    fclose(fp);
    // Stampa sul terminale i numeri letti
    for(i=0;i<10;i++){
        printf("%d\n",n[i]);
    }
}
}
```

Quando si usa la funzione `fscanf()` bisogna conoscere esattamente la quantità di dati che sono scritti sul file, altrimenti si rischia di voler leggere dati che non esistono o non sono accessibili, generando un errore. Se la funzione `fscanf()` è utilizzata più volte con lo stesso file (come nell esempio precedente), verranno letti i dati successivi allultimo già letto e non si ricomincia dallinizio.

Capitolo 12

Inserimento di un numero compreso in un intervallo prefissato

Frequentemente negli esercizi si richiede all'utente di inserire dal terminale il valore di una o più variabili, controllando anche che i valori inseriti rispettino alcune condizioni. Di seguito si esamina il caso più comune: l'inserimento di un solo numero intero n appartenente ad un intervallo prefissato $n_{min} \leq n \leq n_{max}$.
É necessario innanzitutto aver importato la libreria `stdio.h`, contenente le funzioni `printf()` e `scanf()`, aver dichiarato una variabile intera n all'interno del `main()` ed aver definito i limiti dell'intervallo n_{min}, n_{max}; a seconda della situazione, potrebbero essere anch'esse delle variabili intere `nmin,nmax` dichiarate e definite nel `main()`, oppure dei simboli `NMIN,NMAX` dichiarati tramite una direttiva al precompilatore prima del `main()` – di solito si usano lettere minuscole per le variabili e lettere maiuscole per i simboli definiti tramite direttive al precompilatore. Nel seguito considereremo a titolo di esempio il secondo caso, con $N_{min} = 1$ e $N_{max} = 1000$.

12.1 Lettura senza restrizioni

La richiesta di inserimento viene effettuata tramite una chiamata alla funzione `printf()`, che stampi sul terminale un messaggio appropriato, ad esempio `"Inserire un numero intero: "`, dove per avere una buona formattazione sul terminale è importante lasciare uno spazio o andare a capo alla fine del messaggio. Dopodiché si legge il *buffer* del terminale tramite una chiamata alla funzione `scanf()`. Essa riceve in ingresso un numero variabile di argomenti, di cui il primo è una stringa di formattazione, che specifica come deve essere interpretato il testo (`"%d"` nel caso di un solo numero intero). I restanti argomenti sono uno o più puntatori alle variabili su cui devono essere copiati i valori letti dal terminale (in questo caso, un solo puntatore di tipo `*int`).
Per ulteriori dettagli si veda la documentazione online di `scanf()`), la stringa di formattazione può contenere caratteri, spazi bianchi e *specificatori di formato*: questi ultimi sono identificati dal carattere `'%'` e seguono il formato `%[*][lunghezza][modificatore][tipo]`:

- il `tipo` identifica il tipo di dati che deve essere letto: `d`, `u` per gli interi in base decimale, rispettivamente con segno o senza segno; `f`,`e`,... per le variabili in virgola mobile; `i` per interi in base 10, 8 o 16 ...;

- il modificatore (opzionale) aumenta o diminuisce la lunghezza della variabile del tipo indicato da `tipo`: `h` più corto, `l` più lungo. Eccezioni: `%lf` indica il tipo `double` e `%Lf` indica un `long double`;
- la `lunghezza` (opzionale) indica il numero massimo di caratteri da leggere; ad esempio `%4f` legge al più 4 caratteri in totale (incluso il punto!) per una variabile in formato `float`: se dal terminale viene inserito il numero `1.2345`, la funzione legge `1.23`;
- l'asterisco `*` (opzionale) subito dopo il simbolo `%` indica che il dato viene letto ma non viene salvato in alcuna variabile: ad esempio `scanf("%*d %d",&x)` legge due interi separati da uno spazio, ma ignora il primo e salva solo il secondo nella variabile `x`.

Di conseguenza, il numero di argomenti successivi al primo deve essere pari al numero di `%` senza asterisco presenti nella stringa di formattazione.
Il listato seguente esegue una singola lettura di una variabile intera da terminale:

```c
#include <stdio.h>
#define NMIN 1
#define NMAX 10000
int main() {
    int n;
    printf("Inserire un numero intero compreso fra %d e %d: ",NMIN,NMAX);
    scanf("%d",&n);
    printf("Il valore inserito è %d.\n",n);
    return 0;
}
```

12.2 Uso delle variabili di controllo

Per verificare che n si trovi nell'intervallo specificato, e ripetere la richiesta in caso negativo, si possono usare diverse strategie. Una possibilità è quella di utilizzare un ciclo `while(!fine)` governato da una una variabile di controllo `fine`, di tipo `int` ed inizializzata a `0`. L'istruzione `while(`*`condizione`*`){...}` itera l'esecuzione del blocco di codice delimitato dalle parentesi graffe finché la condizione fra le parentesi tonde è vera. All'interno del ciclo, si imposta `fine=1` solo quando l'inserimento va a buon fine, terminando la reiterazione, altrimenti si stampa un messaggio di errore prima dell'inizio del ciclo successivo. Il modo piú compatto per ottenere lo stesso risultato consiste nell'assegnare alla variabile di controllo `fine` il valore dell'espressione logica `(n>=NMIN && n<=NMAX)` che in `C` restituisce `1` se vera, `0` se falsa.

```c
#include <stdio.h>
#define NMIN 1
#define NMAX 10000
int main(void){
    int n,fine=0;
    while (!fine){
        printf("Inserire un numero intero compreso fra %d e %d: ",NMIN,NMAX);
```

```
8           scanf("%d",&n);
9           fine = (n>=NMIN && n<=NMAX); //condizione di fine
10          if(!fine){
11              printf("ERRORE: il numero inserito è fuori dall'intervallo.\n");
12          }
13      }
14      printf("Il valore inserito è %d.\n",n);
15      return 0;
16  }
```

Controllo della validità dell'input:

Un ulteriore controllo che aumenta la robustezza del codice in caso di errori di inserimento è fornito dall'output della funzione `scanf()`. La funzione restituisce un numero intero, che in caso di successo è pari al numero di elementi letti dal terminale, altrimenti è zero. Tale valore può essere ignorato, come negli esempi precedenti, oppure salvato in una variabile intera `res` definita nel `main()`. Alla condizione precedente che n sia all'interno dell'intervallo, si aggiunge ora che l'output di `scanf()` deve essere positivo. Le due condizioni devono essere combinate tramite l'operatore `&&` (*and*), perché devono avverarsi simultaneamente.
Infine, per chiarire all'utente quale tipo di errore sia avvenuto, è utile inserire all'interno del blocco del `while()` anche dei messaggi di errore che si attivano tramite costrutti di selezione del tipo: `if(condizione1){} else if(condizione2){} ...`, come mostrato nelle righe 12-16 del listato seguente.

```
1   #include <stdio.h>
2   #define NMIN 1
3   #define NMAX 10000
4   int main(void)
5   {
6       int n,fine=0,res;
7       while (!fine){
8           printf("Inserire un numero intero compreso fra %d e %d: ",NMIN,NMAX);
9           res=scanf("%d",&n);
10          fine = (res>0 && n>=NMIN && n<=NMAX);
11          if(!fine){
12              if(res==0){
13                  printf("ERRORE nella conversione dell'input.\n");
14              }
15              else{
16                  printf("ERRORE: il numero inserito è fuori dall'intervallo.\n");
17              }
18          }
19      }
20      printf("Il valore inserito è %d.\n",n);
21      return 0;
```

```
}
```

Gestione del *buffer:*

Anche in input il sistema operativo utilizza un buffer, ovvero un'area di memoria in cui i dati in input vengono salvati, prima di essere assegnati alle variabili del programma tramite la `scanf()`. È quindi opportuno inserire di seguito allo `scanf()` il comando `while(getchar()!='\n')`, come mostrato alla riga `10` del listato seguente. La funzione `getchar()` prende dal buffer di input il prima carattere disponibile. Se viene quindi chiamata iterativamente, questa funzione legge uno alla volta tutti i caratteri che si trovano nel buffer – senza salvarli da alcuna parte ma confrontandoli con la *newline* `'\n'` – finché non incontra la *newline*, terminando il ciclo. L'obiettivo è svuotare il *buffer* da eventuali caratteri spuri inseriti dall'utente prima di andare a capo, i quali altrimenti sarebbero letti dallo `scanf()` successivo. Si provi ad esempio a commentare la riga 10, compilare ed eseguire il programma inserendo in input: `a`; il programma resterà bloccato nel ciclo `while()` più esterno (per terminare il programma, premere i tasti `Ctrl+C`). Infatti quando lo `scanf()` fallisce nella lettura di un intero con segno, restituendo 0, la stringa mal formattata rimane nel *buffer* e viene letta di nuovo alla seconda iterazione, prima che l'utente inserisca un secondo tentativo, ottenendo di nuovo `res` pari a zero, e così via.

12.3 Inserimento tramite funzione:

Il codice precedente può essere inserito all'interno di una funzione chiamata `InserimentoIntero()`, che restituisce un `int`. La funzione implementa l'inserimento da terminale di una variabile intera generica: riceve i limiti dell'intervallo $[\texttt{min}, \texttt{max}]$ ed il nome da stampare durante la richiesta, sotto forma di un array di caratteri `name[]`. Dopo aver effettuato i controlli descritti precedentemente e reiterato la richiesta finché non fossero superati, la funzione restituisce il valore della variabile letta. L'utilizzo di funzioni permette di alleggerire il contenuto del `main()` per dare più spazio alla parte principale del codice. Si veda ad esempio l'Esercizio 6 per un'applicazione.

```
#include<stdio.h>
#define NMIN 1
#define NMAX 1000
int InserimentoIntero(char name[], int min, int max);

int main(){
    int m=InserimentoIntero("m",NMIN,NMAX);
    printf("Il valore inserito è %d.\n",m);
    return 0;
}

int InserimentoIntero(int min, int max, char name[]){
    int n,res,fine=0;
    while (!fine){
        printf("Inserire %s (intero compreso fra %d e %d): ",name,min,max); //
   stampa richiesta
```

```
16          res=scanf("%d",&n); //leggi un intero
17          while(getchar()!='\n'); //svuota il buffer
18          fine = (res>0 && n>=min && n<=max); //condizione di fine
19          if(!fine){ //stampa messaggi di errore
20              if(res==0)
21                  printf("ERRORE nella conversione dell'input.\n");
22              else
23                  printf("ERRORE: il numero inserito è fuori dall'intervallo.\n");
24          }
25      }
26      return n;
27  }
```

Capitolo 13

Calcolo dei valori massimo e minimo di una lista di elementi

Il calcolo del valore massimo e minimo di una lista di elementi contenuti in un array è un'operazione che ricorre spesso negli esercizi. Questi valori si possono ottenere utilizzando solamente un ciclo `for` ed una condizione `if`.
Seguendo le denominazioni utilizzate nel listato a fondo pagina, supponiamo di avere un array di interi denominato `lista[]` di dimensione `DIM` e poniamo le due variabili `min` e `max`, che conterranno il minimo e il massimo, pari al primo valore dell'array, `lista[0]`. A questo punto, l'operazione da effettuare per la ricerca dei valori massimo e minimo dell'array `lista[]` è di scorrere tutti gli elementi dell'array con un ciclo `for`, e confrontare, ad ogni iterazione, se il valore i-esimo dell'array sia minore (per la ricerca del minimo) o maggiore (per la ricerca del massimo) dei valori contenuti nelle variabili `min` e `max`. Nel caso in cui una di queste condizioni sia soddisfatta, la variabile `min` (o `max`) viene posta pari all' i-esimo elemento dell'array, divenendo il nuovo elemento di confronto per le iterazioni successive del ciclo. Nel caso in cui, invece, la condizione non venga soddisfatta, il ciclo passerà all'iterazione successiva, lasciando invariate `min` e `max`. In questo modo si andranno a selezionare valori sempre più piccoli/grandi.

```
int main(){
    int min,max;
    int lista[DIM];
    ...
    min=lista[0]; // Inizializzazione min
    max=lista[0]; // Inizializzazione max
    // Ciclo di ricerca del massimo e del minimo dell'array lista
    for(i=0;i<DIM;i++){
        if(lista[i]<min){ min=lista[i];}
        if(lista[i]>max){max=lista[i];
        }
    }
}
```

Capitolo 14

Generazione di numeri pseudo-casuali

La libreria `stdlib.h` contiene le definizioni di funzioni che, se chiamate nel modo appropriato, restituiscono sequenze pseudo-casuali di numeri, interi o reali. In Tabella 14.1 sono riportate alcune di tali funzioni, fra cui la funzione `rand()` e la famiglia di generatori a cui appartiente `drand48()`.
A partire dall'output di una qualunque di queste funzioni, con delle semplici operazioni si può ottenere un numero intero o reale distribuito uniformemente in qualsiasi intervallo.

Nome della funzione	Tipo restituito	Intervallo di appartenenza	Funzione inizializzatrice
`rand()`	`int`	[0,`RAND_MAX`]	`srand()`
`drand48()`	`double`	$[0, 1)$	`srand48()`
`lrand48()`	`long int`	$[0, 2^{31} - 1]$	`srand48()`

Tabella 14.1: Esempi di funzioni definite in `stdlib.h` che restituiscono sequenze di numeri casuali distribuiti uniformemente in un certo intervallo. `RAND_MAX` è una costante il cui valore dipende dal sistema utilizzato, ma sempre ≥ 32767.

14.1 Inizializzazione del *seme*

Ciascuno dei generatori elencati ha bisogno di essere inizializzato con un valore iniziale, detto seme o *seed*. Il valore preimpostato del *seed* è definito nella libreria, ma si può scegliere di partire da un *seed* diverso tramite la funzione `srand()` o `srand48()`, che prendono come argomento una variabile `unsigned int` e non restituiscono nulla. Fissato un seme di partenza, l'algoritmo produce una sequenza **deterministica** di numeri pseudocasuali: ovvero se si esegue una seconda volta lo stesso programma con lo stesso seme, la sequenza sarà la stessa. Sebbene ciò possa essere utile in alcuni casi (ad esempio in fase di risoluzione dei problemi del codice è utile ripetere la stessa sequenza di numeri), in generale deve essere evitato, scegliendo **un seme diverso all'inizio di ogni esecuzione del programma**. Un metodo comunemente utilizzato consiste nell'impostare il seed a `time(NULL)`, ovvero al valore in secondi del tempo trascorso dalla mezzanotte del 1 Gennaio 1970; la funzione `time()` è contenuta nella libreria `time.h`.
Nota: la funzione `srand()` o `srand48()` deve essere chiamata **una sola volta** all'interno del codice, altrimenti non è più assicurata la pseudocasualità dei numeri estratti. La funzione di inizializzazione

del seme può essere chiamata più volte all'interno di uno stesso programma solo nel caso in cui il programma effettui più simulazioni indipendenti l'una dall'altra; la chiamata deve avvenire solo una volta all'inizio di ciascuna simulazione.

```c
#include<stdlib.h>
#include<time.h>

int main(){
    unsigned int seed;
    int x;
    long int y;
    double z;

    //questo seed "casuale" produrrebbe una sequenza pseudocasuale, ma uguale per
     ogni esecuzione del programma. Provare per credere!
    //seed=194728;

    //questo seed produce una sequenza pseudocasuale diversa per ogni esecuzione
    del programma, perché il tempo trascorso dal 1 Gennaio 1970 è diverso
    seed=time(NULL);

    // inizializza i generatori con il seed
    srand(seed);
    srand48(seed);
    // estrai numeri casuali
    x=rand();       //numero intero in [0,RAND\_MAX]
    y=lrand48();    //numero intero in [0,2^31-1]
    z=drand48();    //numero reale in [0.0,1.0]
}
```

14.2 Generare numeri casuali in un intervallo prefissato, con `drand48()`

Generalmente, si è interessati a generare numeri interi o reali x compresi in un dato intervallo, per cui è necessario eseguire delle operazioni per convertire il valore restituito dalla funzione `drand48()` (che è un `double` appartenente all'intervallo $[a, b)$) in un numero intero o virgola mobile distribuito uniformemente in un dato intervallo.
Si consideri dapprima il caso in cui si voglia produrre numero intero pseudo-casuale x nell'intervallo $[a, b]$, dove a e b sono due numeri interi. x può assumere $N = b - a + 1$ valori distinti (si pensi ad esempio al dado a 6 facce: $x \in \{1, 2, 3, 4, 5, 6\}$). Utilizzando la funzione `drand48()` si può ottenere x nel seguente modo `x=(int) (drand48()*(b-a+1)+a)` infatti `drand48()*(b-a+1)` produce un numero pseudo-casuale uniformemente distribuito nell'intervallo $[0, b - a + 1)$, e sommando poi a si ottiene un numero pseudo-casuale nell'intervallo $[a, b+1)$, che con il successivo cast a intero diventerà un numero pseudo-casuale intero nell'intervalo $[a, b]$.
Nel caso in cui si voglia un numero pseudo-casuale x di tipo `double` uniformemente distribuito in un certo intervallo $[a, b)$, dove a e b sono numeri `double`, in maniera del tutto analoga al caso precedente

tale numero si ottiene nel seguente modo: `x=drand48()*(b-a)+a` Infatti, se `z=drand48()`, ovvero se `z` è un numero pseudo-casuale uniforme nell'intervallo $[0.0, 1.0)$, moltiplicando tale numero per $b-a$ ed sommandogli a si ottiene un numero $x = f(z) = a + (b-a) \cdot z$, compreso tra $f(0) = a + 0 = a$ (incluso) e $f(1) = a + (b-a) = b$ (escluso).

```
int myIntRand(int a, int b) {
    double r, y;
    int x, N=b-a+1;
    r=drand48();
    y=r*N; // 0<=y<N
    x=y+a; // a<=x<=b
    return x; //in forma compatta: return a+drand48()*(b-a+1);
}
double myDoubleRand(double a, double b) {
    double z,x;
    int r=drand48();// 0.0<=r<1.0
    x=a+(b-a)*r; // a<=x<b
    return x; //in forma compatta: return a + (b-a)*drand48();
}
```

14.3 Applicazione: verificare il successo di un evento di probabilità data

Si consideri un evento che può avverarsi con probabilità p, detta *probabilità di successo* di quell'evento, con p reale e $0 \leq p \leq 1$. Ad esempio, l'evento "se tiro un dado a 6 facce esce il numero 3" ha probabilità di successo $p = 1/6 = 0.1666\ldots$, mentre l'evento "se tiro un dado a 6 facce esce un numero dispari" ha probabilità $p = 3/6 = 0.5$ (se il dado non è truccato).
Un generatore di numeri casuali consente di simulare un esperimento per verificare l'avverarsi di tale evento, con il seguente procedimento:

1. Genero un numero r di tipo `float` o `double`, distribuito uniformemente fra 0 ed 1, utilizzando la funzione `drand48()`.

2. Tale numero ha probabilità p di cadere nell'intervallo $0 \leq r < p$ e probabilità $1-p$ di cadere nell'intervallo complementare $p \leq r \leq 1$; dunque se uso l'istruzione `if(r<p){...}`, il codice contenuto nel corpo dell'`if()` viene eseguito in media con probabilità p.

Il codice seguente ripete 100 volte il lancio di una moneta truccata, per la quale la probabilità che esca testa è $p = 0.57$ invece di 0.5, e conta quante volte esce testa. Secondo la legge dei grandi numeri, se il lancio è ripetuto un numero sufficiente di volte, la frequenza del numero delle volte in cui usciranno le facce testa o croce approssimerà i valori di p e (1-p).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

```

```c
double myDoubleRand(double, double);

int main(){
    int count=0;
    double r,p=0.57;
    srand(time(NULL)); // inizializza il generatore di numeri casuali
    for(int i=1;i<=100;i++){
        r=myDoubleRand(0.0,1.0); // genera un numero fra [0.0,1.0]
        if(r<p){
            printf("T"); //se r<p si verifica l'evento "testa"
            count++; //conta le teste
        }
        else{
            printf("C"); //altrimenti si verifica l'evento complementare: "croce"
        }
    }
    printf("\nSu 100 lanci sono uscite %d teste.\n",count);
}

double myDoubleRand(double a, double b) {
    return a + (b-a)*drand48();
}
```

Capitolo 15

Caratteri e stringhe

In `C` una stringa è un array di variabili di tipo carattere (`char`, ovvero interi di 1 byte con segno). Ciascun elemento dell'array rappresenta un singolo carattere, che può essere un simbolo alfanumerico, come `'1'` o `'c'`, o un carattere speciale come `'!'`. A ciascun carattere è associato un codice `ASCII` compreso tra 0 e 255, che lo identifica univocamente. Per esempio, il codice 33 identifica univocamente il punto esclamativo ('!'). Per assegnare un valore ad una variabile di tipo `char` si possono usare indifferentemente il simbolo racchiuso tra due apici, o il codice ASCII corrispondente. Le istruzioni `char c='!';` e `char c=33;` sono cioè equivalenti.
Due variabili di tipo `char`, rappresentando due interi a 1 ybte con segno, si possono sommare o sottrarre; con ciò si intende che vengono sommati i corrispondenti codici `ASCII`. Il breve codice sottostante fornisce alcuni esempi sulla gestione delle variabili `char`.

```
#include <stdio.h>
int main()
{
  int k;
  char c;
  c=65;
  printf("codice ASCII: %i CARATTERE: %c\n",c,c);
   // stampa tutte le lettere maiuscole (65-90):
   for (k=65;k<=90;k++)
   {
     c=k;
     printf("k=%4i  %c\n",k,c); //codice ascii e i carattere
   }
   // stampa tutte le lettere minuscole (97-122):
   for (k=97;k<=122;k++)
   {
     c=k;
     printf("k=%4i  %c\n",k,c); //codice ascii e carattere;
    }
}
```

La lunghezza di una stringa è pari al numero di caratteri che la compongono (spazi e caratteri speciali inclusi), più uno: infatti l'ultimo elemento è sempre il carattere di terminazione stringa `'\0'`.
La libreria `string` del `C` contiene diverse funzioni per la manipolazione delle stringhe, come:

- `strlen(str)`: Resituisce la lunghezza della stringa `str[]`;
- `strcat(str1,str2)`: Concatena le due stringhe `str1[]`, `str2[]`, scrivendo il risultato su `str1[]`;
- `strcpy(str1,str2)`: Copia la stringa `str1[]` su `str2[]`.

La porzione sottostante di codice fornisce alcuni esempi di dichiarazione e manipolazione delle stringhe.

```c
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[100]="oggi e' una bella giornata";
  // a str1 vengono riservati 100 caratteri, ma solo i primi 26+1 contengono
    caratteri non nulli; il 27-esimo è il carattere di fine stringa
  char str2[]="\nnon piove";
  // a str2 vengono riservati esattamente 10+1 caratteri
  int k,n1,n2;
  printf("%s \n",str1);
  // la funzione strlen() restituisce la lunghezza della stringa, senza contare
    il carattere di terminazione
  n1=strlen(str1);
  printf("lunghezza: %3i\n",n1);
  printf("%s \n",str2);
  n2=strlen(str2);
  printf("lunghezza: %3i\n",n2);
  strcat(str1,str2);
  printf("stringa concatenata: %s\n",str1);
  n1=strlen(str1);
  printf("lunghezza: %3i\n",n1);
  // stampa la stringa risultante un carattere alla volta
  for (k=0;k<=n1;k++)
    {
      printf("%3i %c\n",k,str1[k]);
    }
}
```

15.1 Lettura di stringhe tramite la funzione `getchar()`

L'utilizzo della funzione `scanf()` per la lettura di stringhe può dar vita a problemi di gestione di *buffer overflow*. Il *buffer*, lett. tampone, è una zona di memoria di transito, utilizzata tipicamente per velocizzare le operazioni di input/output. Il *buffer overflow* è un errore che avviene quando all'interno di un *buffer*, ovvero più genericamente all'interno di un array, vengono inseriti un numero di caratteri maggiore della sua capienza, che coincide con la quantià di memoria che per esso è stata allocata; se la memoria in questione è locale ad una funzione, si parla anche di *stack buffer overflow* o *stack smashing*. I dati in eccesso immessi nel *buffer* possono produrre un errore perché:

- sovrascrivono variabili che regolano il flusso del programma (ad esempio il *return address*, informazione che viene salvata quando si richiama una funzione e influisce su dove il flusso del programma dovrebbe continuare dopo l'esecuzione);
- oppure tentano di sovrascrivere zone di memoria non consentite o di sola lettura, producendo un errore di segmentazione (*segmentation fault*).

Per evitare questi errori, e controllare il numero di caratteri inseriti, per leggere una stringa da terminale si può utilizzare una funzione personalizzata basata sulla funzione `getchar()`, della libreria `stdio.h`, che legge i caratteri immessi uno alla volta.
Di seguito viene riportato come esempio la funzione `ReadString()` dal capitolo 2.

```
int ReadString(char string[]) {
    int N=0;
    char ch;
    printf("Inserire il messagio (solo maiuscole e spazi, max %d caratteri): ",
   LEN);
    while((ch=getchar())!='\n'){
        if(N==LEN) {
            printf("Errore: la stringa inserita supera la lunghezza massima
   consentita %d.\n",LEN);
            exit(1);
        }
        string[N++]=ch; //incrementa N dopo aver eseguito tutte le istruzioni
   precedenti
    }
    string[N]='\0'; // inserisco manualmente il carattere di terminazione stringa
   !
    return N;
}
```

La funzione riceve in input un'array di caratteri di dimensione `LEN+1` (`LEN` è una costante globale definita tramite una direttiva al precompilatore) e contiene:

- La stampa su terminale della richiesta di inserimento di una stringa, con una lunghezza massima `LEN`;
- Il ciclo principale, in cui i caratteri vengono acquisiti uno alla volta tramite la funzione `getchar()`, fino a quando non viene premuto il tasto di invio, cioé finché `ch` non diviene uguale a `'\n'`;

quando ciò succede, il carattere *invio* viene inserito nell'elemento N-esimo della stringa. La variabile contatore intera `N` parte da zero e viene incrementato dopo ogni lettura;

- all'interno del ciclo, un'istruzione `if()` verifica se l'inserimento sia di lunghezza eccessiva; in tal caso stampa un messaggio di errore e termina il programma tramite `exit(1)` (`exit()` è contenuta nella libreria `stdlib.h`);
- Alla fine dele ciclo, viene inserito in ultima posizione il carattere di terminazione stringa e viene restituito alla funzione chiamante il valore di `N`, che rappresenta la lunghezza totale della stringa.

L'istruzione di controllo alla riga 10 evita che avvenga un *buffer overflow*. Infatti, se si cerca di assegnare il carattere `ch` alla locazione `string[N]` con `N` maggiore di `LEN`, si cerca di accedere a una zona di memoria "proibita", con effetti imprevedibili.
La funzione `ReadString()` viene chiamata nel `main()` con la seguente sintassi:

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define LEN 100
4
5  int ReadString(char *);
6
7  int main(){
8      char str[LEN+1]={0};
9      int len;
10
11     len=ReadString(str);
12     printf("La stringa letta ha %d caratteri ed è:\n%s\n",len,str);
13 }
14 ...//definizione di ReadString()
```

Alternativamente per evitare dei buffer overflow si può usare l'istruzione `scanf` nel seguente modo

```
1 scanf("%20s", s)
```

che prende dal buffer di input non più di 20 caratteri, e dove il numero 20 può essere cambiato a seconda della memoria allocata per la variabile `s`.

Capitolo 16

Creare grafici con `Python`

Si ricordano brevemente le principali caratteristiche che distinguono Python (versione $\geq$3.0) dal linguaggio C, per quanto concerne queste dispense:

- il carattere di fine riga è la *newline* (a capo); i blocchi di codice non sono delimitati da parentesi graffe, bensì dall'indentazione;
- non è necessario dichiarare le variabili prima dell'assegnazione;
- le funzioni possono restituire una 'lista' di oggetti in output, anziché uno solo.

Le successioni di comandi mostrate nel seguito possono essere eseguite riga per riga nel terminale (modalità interattiva), dopo aver chiamato l'interprete con il comando `python3` (per uscire, digitare `quit()`); oppure, possono essere raccolte all'interno di un file detto *script* (lett., copione), ad esempio denominato `script.py`, che può essere eseguito con la seguente sintassi:

```
python3 script.py
```

16.1 Plot `y` vs. `x` da un file di due colonne

Negli esercizi viene spesso richiesto di creare un grafico con i dati contenuti in un file `ASCII`, chiamato ad esempio `dati.dat`. Il caso più semplice è quello di un file contenente una tabella di dati $N \times 2$, del tipo:

```
0 0
1 1
2 4
3 9
... ...
```

La prima cosa da fare è **importare le librerie** che si vogliono utilizzare. Le librerie di base per la programmazione scientifica sono Numerical Python (NumPy, che comunemente si abbrevia con `np`), per la gestione di array e operazioni fra array; e il modulo `PyPlot` della libreria Matplotlib (solitamente abbreviato con `plt`) per i grafici:

```
import numpy as np
import matplotlib.pyplot as plt
```

Con le due righe precedenti abbiamo importato le due librerie richieste, e abbiamo assegnato, tramite il comando `as`, i due `alias` (o abbreviazioni) `np` e `plt` useremo all'interno dello script. La funzione **`np.loadtxt()`** consente di leggere la tabella di dati con una sola riga di comando. Essa richiede come primo argomento obbligatorio una stringa contenente il nome del file. Può ricevere in input molti altri parametri opzionali, che possono essere passati alla funzione in qualunque ordine. Se i parametri non vengono definiti esplicitamente, assumono un valore preimpostato di default. L'opzione `unpack` è preimpostata su `False`, ovvero la funzione di default restituisce la tabella di dati in formato di matrice $N \times 2$; nel nostro caso conviene però specificare `unpack=True`, per leggere direttamente 2 vettori di dimensione N.
Il comando seguente legge dunque le due colonne della tabella e le salva su due vettori x e y, in formato `np.array()`. **Nota**: A differenza del `c`, in `python` non c'è bisogno di dichiarare x e y prima di utilizzarli.

```
x,y = np.loadtxt("dati.dat", unpack=True)
```

`Grafici in 2D (`*`scatter plot`*`):`
Di seguito elenchiamo i comandi per effettuare un grafico di y vs. x con un **titolo** ed i **nomi degli assi**, e per **salvarlo in formato PNG** (si può salvare anche in PDF, EPS o in altri formati).

```
plt.plot(x,y)
plt.xlabel("x")
plt.ylabel("y")
plt.savefig("grafico.png")
```

In alternativa a `plt.plot()` si può usare `plt.scatter()`, con lo stesso risultato. Il **formato** ed il **colore** vengono scelti automaticamente da Matplotlib, ma possono essere personalizzati inserendo una stringa come terzo argomento nella funzione `plt.plot()`:

- alcuni dei formati disponibili sono: punti grandi `"o"`, linea continua `"-"`, puntini sottili `"."`, linea tratteggiata `""`, punto-linea `"o-"`;
- alcuni dei colori disponibili sono: blu `"b"`, rosso `"r"`, verde `"g"`, nero `"k"`;
- formato e colore possono essere combinati in un unico comando (esempio: `"r-"`).

Di seguito lo script completo in Python; un grafico di esempio è riportato in Figura 16.3, pannello in alto a sinistra.

```
#importa le librerie da cui chiamare le funzioni e assegna i relativi alias
import numpy as np
import matplotlib.pyplot as plt

# leggi le due colonne in "dati.dat" e salvale nei vettori x, y
x,y = np.loadtxt("dati.dat", unpack=True)
```

```
7
8  plt.plot(x,y, "ro") \# plot personalizzato:  punti grandi rossi
9  plt.xlabel("x") \# nomi degli assi
10 plt.ylabel("y")
11 plt.title("Grafico 1") # titolo
12
13 plt.savefig("grafico1.png") # salva in formato .png
```

16.2 Lettura/Plot di file con più colonne

Per graficare contemporaneamente più di una serie di dati, salvati su più colonne di uno stesso file, bastano minime modifiche allo script precedente. Ad esempio, nel caso di 3 colonne x, y_1, y_2:

```
1 0 0 0
2 1 2 1
3 2 4 4
4 3 6 9
5 ...
```

il comando per leggere e salvare i dati è lo stesso, ma questa volta i dati verranno salvati su tre array:

```
1 x,y1,y2 = np.loadtxt("dati.dat", unpack=True)
```

Per graficare su uno stesso grafico y_1 vs. x e y_2 vs. x, è sufficiente utilizzare due volte la funzione `plot()`, usando come ordinata `y1` la prima volta e `y2` la seconda volta:

```
1 plt.plot(x,y1,"ro",label="lineare")
2 plt.plot(x,y2,"b.",label="quadratico")
```

Si noti che è stato usato l'argomento opzionale `label`, che permette di assegnare un'etichetta ad ogni curva di dati. Per visualizzare la **legenda** delle etichette sovrimpressa al grafico, bisogna chiamare la funzione `plt.legend()` dopo aver esaurito le chiamate alle funzioni `plt.plot()` con argomento `label`.
Di seguito il listato completo, che produce il grafico in Figura 16.3, pannello in alto a destra.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x,y1,y2 = np.loadtxt("dati.dat", unpack=True)
5
6  plt.plot(x,y1,"ro",label="lineare") # primo plot
7  plt.plot(x,y2,"b.",label="quadratico") # secondo plot
8  plt.xlabel("x")
9  plt.ylabel("y")
10 plt.title("Grafico 2")
```

```
plt.legend()
plt.savefig("grafico2.png")
```

16.3 Plot di una funzione nota

Se si vuole tracciare il grafico di una funzione nota analiticamente, ad esempio $y = sin(x)$, bisogna creare manualmente i dati per le ascisse x e poi calcolare y mediante operazioni e funzioni di NumPy. La funzione **`np.linspace(start, stop, num)`** permette di creare `num` punti equispaziati fra `start` e `stop`. Maggiore il numero di punti inseriti, migliore sarà il campionamento della funzione (tipicamente 1000 punti sono sufficienti). Il seguente script grafica le funzioni $\sin(x)$ e e^{-x} nell'intervallo $0 \leq x \leq 2\pi$ (Figura 16.3, pannello in basso a sinistra), chiamando le funzioni `np.sin()` e `np.exp()` di `NumPy`.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,6.28,1000)
y1 = np.sin(x)
y2 = np.exp(-x)

plt.plot(x,y1,label="sin")
plt.plot(x,y2,label="exp")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Grafico 3")
plt.legend()
plt.savefig("grafico3.png")
```

16.4 Istogrammi

Un istogramma rappresenta graficamente una distribuzione di probabilità (di una variabile discreta o continua) con delle barre adiacenti di altezza diversa, costruite su degli intervalli ordinati equispaziati (*bins*). Dato un set di dati, l'istogramma rappresenta sull'asse verticale il numero di conteggi di ogni bin. Quando si confrontano diversi set di dati, è buona prassi non utilizzare direttamente i conteggi, ma le frequenze, definite come il rapporto fra il numero di elementi che cadono all'interno dell'intervallo e il numero totale di elementi. Se il numero di dati (esperimenti) è molto grande, l'istogramma delle frequenze descrive bene la distribuzione di probabilità dell'evento in questione.
Ad esempio, se si volessero rappresentare le altezze (in cm) di 250 individui si otterrebbe un istogramma del tipo:

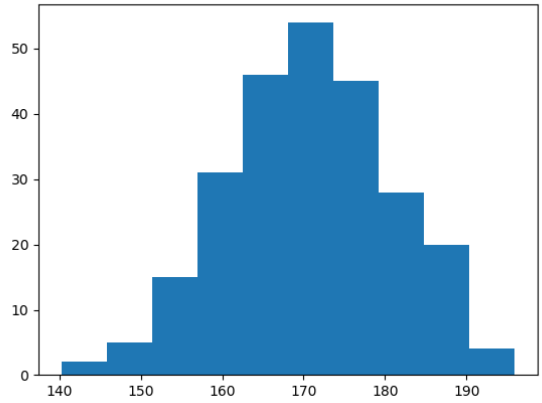


Figura 16.1: Si nota un andamento simile alla curva di Gauss (quello atteso dal teorema del limite centrale) con un picco vicino ai 170cm.

È sempre importante stabilire un numero di bins ottimale perché la forma dell'istogramma sia rappresentativa: un numero troppo piccolo di bin fa perdere informazioni sul reale andamento dei dati; un numero eccessivo potrebbe invece determinare bin troppo "stretti", per cui le frequenze risulterebbero nulle (non ci sono valori in quell'intervallo), e l'istogramma risulterebbe eccessivamente rumoroso.

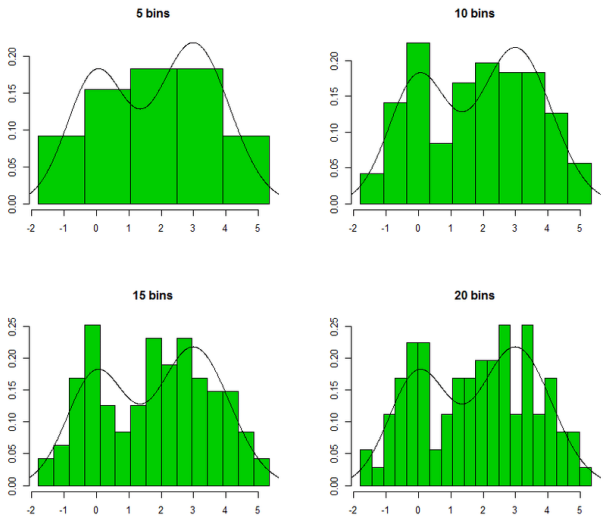


Figura 16.2: Si vede come, per questa presa dati, spostarsi da $nbits = 15$ diventa poco efficiente ai fini dell'istogramma.

Ci sono due funzioni principali per produrre istogrammi con Python: `plt.hist()` e `plt.bar()`.
La funzione **`plt.hist(x)`** consente di creare un istogramma a partire da un array `x` di osservazioni della stessa variabile. Il numero di *bin* dell'istogramma è definito automaticamente, altrimenti può essere passato come secondo argomento opzionale. Ad esempio, a partire da un file contenente una colonna di 1000 numeri casuali distribuiti uniformemente fra 0 e 1:

```
0.530554
0.061167
0.845086
0.0862668
...
```

il seguente script crea un istogramma con 10 bin e lo salva nella directory corrente, contando quante volte i dati cadono in ciascun intervallo ($\rightarrow$ i bin conterranno in media $1000/10 = 100$ conteggi, con delle fluttuazioni sui singoli bin). Si veda il grafico di esempio in Figura 16.3, pannello in basso a destra.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.loadtxt("dati.dat", unpack=True)
nbins = 10

plt.hist(x,nbins)
plt.xlabel("x")
plt.ylabel("Conteggi")
plt.title("Istogramma con %d bins"%(nbins))
plt.savefig("isto1.png")
plt.show()
```

La funzione `plt.bar(x,height)` si usa invece nel caso in cui i dati siano già stati categorizzati in coppie (bin,conteggio). Esso disegna un rettangolo di altezza `height` e centrato sull'ascissa `x`, producendo di fatto un istogramma.

```
import numpy as np
import matplotlib.pyplot as plt

bins,conteggi = np.loadtxt("dati.dat", unpack=True )
plt.bar(bins, conteggi)
plt.xlabel("Bins")
plt.ylabel("Conteggi")
plt.savefig("isto2.png")
plt.show()
```

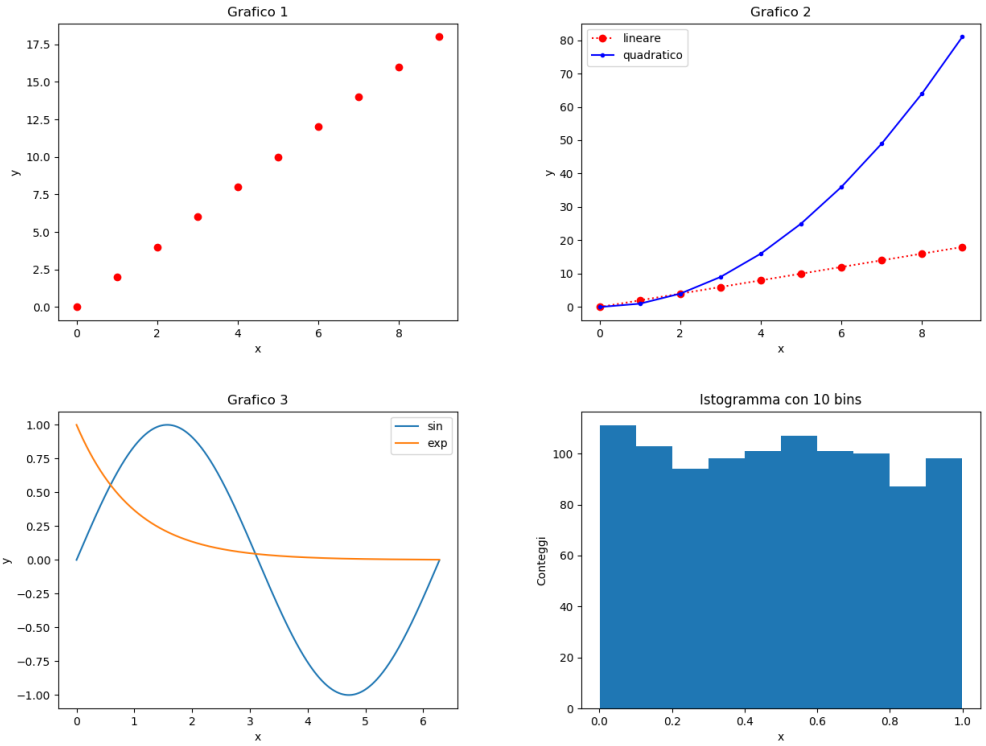


Figura 16.3: Grafici prodotti dai listati precedenti.