Laboratorio di Fisica 1
R4: Misura di variabili aleatorie

Gruppo 17: Bergamaschi Riccardo, Graiani Elia, Moglia Simone

8/11/2023 – 15/11/2023

Sommario

Il gruppo di lavoro ha misurato due variabili aleatorie, osservando come queste rispecchino le rispettive distribuzioni teoriche (di Bernoulli e di Poisson).

1 Processo di Bernoulli

1.1 Dati sperimentali

Eseguiamo 400 lanci di sei dadi distinti[1], registrandone tutti i risultati. Per ogni possibile risultato, possiamo così definire una variabile aleatoria[2] $s \in [1;6] \cap \mathbb{N}$ $x_s \in [0;6] \cap \mathbb{N}$ come il numero di dadi, fra i sei lanciati, con risultato pari ad s. Possiamo considerare il lancio dei sei dadi come un processo di Bernoulli, in quanto i risultati dei dadi sono indipendenti fra loro. Di conseguenza, la distribuzione di probabilità di x_s è data da:

$$p\left(x_s = k\right) = \binom{6}{k} \left(\frac{1}{6}\right)^k \left(\frac{5}{6}\right)^{6-k} \qquad \forall k \in [0;6] \cap \mathbb{N}$$

Di seguito riportiamo gli istogrammi dei dati così raccolti, assieme ai valori attesi, calcolati mediante la distribuzione teorica.

1.2 Simulazione

Tramite un programma da noi scritto e compilato[3], simuliamo la stessa esperienza con 10^{12} lanci dei sei dadi, al fine di verificare la legge dei grandi numeri.

[1] Li distinguiamo in base al colore

[2] *Notazione.* Per noi $0 \in \mathbb{N}$.

[3] *Vedi* Appendice 1

Figura 1: ...

Quest'ultima consiste nella tesi che, su un grande numero di prove, i risultati si avvicinino, in proporzione, ai valori attesi.

Di seguito riportiamo, in un istogramma, i risultati della simulazione.

2 Processo di Poisson

2.1 Materiali e strumenti di misura utilizzati

Strumento di misura	**Soglia**	**Portata**	**Sensibilità**
Contatore Geiger	1 conteggi/s	N./A.	1 conteggi/s
Metro a nastro	0.1 cm	300.0 cm	0.1 cm

Altro	**Descrizione/Note**
Campione di $^{232}_{90}$Th	Componente di una lampada da campeggio

2.2 Esperienza e procedimento di misura

Per ogni distanza d_i tra il campione di $^{232}_{90}$Th e il contatore Geiger (con $i \in [1;5] \cap \mathbb{N}$), il gruppo di lavoro ha acquisito i conteggi di raggi γ emessi nella direzione del contatore ogni secondo per un tempo complessivo di circa un'ora (3657 s)[4].

Infine, il gruppo di lavoro ha acquisito, sempre per 3657 s, i conteggi al secondo di raggi γ nella direzione opposta rispetto al campione, per avere una stima della radioattività ambientale.

Notazione. *Indichiamo con γ_i complessivamente i conteggi acquisiti a distanza d_i dal campione (con $i \in [1;5] \cap \mathbb{N}$), mentre con $\gamma_{t,i}$ il numero di conteggi nell'intervallo temporale $[(t-1)\text{ s}; t\text{ s}]$ dall'inizio dell'acquisizione (con $t \in [1;3657] \cap \mathbb{N}$). Allora $\overline{\gamma_i}$ sarà la media della distribuzione, dove $(\overline{\gamma_i})_{best}$ sarà la media dei conteggi, mentre σ_{γ_i} sarà la deviazione standard e $\delta(\overline{\gamma_i}) = \sigma_{\overline{\gamma_i}} = \frac{\sigma_{\gamma_i}}{\sqrt{3657}}$ l'errore sulla media. Analogamente per* γ_{amb}, γ_{amb} e $\overline{\gamma_{\text{amb}}}$.

Di seguito, riportiamo le distribuzioni di tutti i γ_i e la regerssione lineare dei conteggi in funzione di $1/d_i^2$.

2.3 Conclusioni

Per valutare numericamente la consistenza tra i due valori di k ottenuti, abbiamo calcolato il seguente valore (numero puro):

$$\varepsilon = \frac{|(k_{\text{statica}})_{\text{best}} - (k_{\text{dinamica}})_{\text{best}}|}{\delta k_{\text{statica}} + \delta k_{\text{dinamica}}}$$

Allora k_{statica} e k_{dinamica} sono consistenti se e solo se $\varepsilon \leq 1$.

Nel nostro caso, $\varepsilon = 1.33$. Il gruppo di lavoro ha ipotizzato che questa inconsistenza (comunque contenuta, seppur non trascurabile) fra le due misure possa essere ragionevolmente giustificata dalla difficoltà incontrata nel ridurre al minimo le oscillazioni in direzione perpendicolare a $\vec{g}$; considerato inoltre che la posizione dei fototraguardi non era ottimale, ciò potrebbe avere ulteriormente influenzato la distribuzione dei tempi. È in effetti possibile osservare che le

[4] Abbiamo scelto di acquisire esattamente 3657 secondi in quanto 3657 minimizza la funzione $f(x) = \left\{\frac{x}{\pi}\right\} = \frac{x}{\pi} - \left\lfloor\frac{x}{\pi}\right\rfloor$ meglio di 3600.

distribuzioni da noi ottenute non sono, il più delle volte, del tutto simmetriche: la moda sembra essersi spostata leggermente a sinistra – un possibile sintomo dell'influenza di un errore sistematico sulle misure.

Appendices

A Codice Rust per $\cdot 10^{12}$ lanci di sei dadi

Qui riportiamo il codice Rust, da noi scritto, che ci ha permesso di lanciare virtualmente $6 \cdot 10^{12?}$ dadi in maniera estremamente efficiente.

```rust
/**** Architettura del programma ****
 * Il lavoro viene spezzato fra i vari
*/

/* Costanti durante la compilazione */
/// Parametri della simulazione
const N_THREADS:    u8  = 50;         /// Numero di thread
const N_MILESTONES: u32 = 10_000;     /// Numero di
const N_THROWS:     u32 = 2_000_000;
const N_ROLLS:      u8  = 5;
/// Altre costanti
const N_BINS: usize = (N_ROLLS + 1) as usize;
const N_COUNT_TOTAL: u128 = (N_MILESTONES as u128) * (N_THREADS as u128);

/*** Imports ***/
use std::{sync::mpsc::{self, Sender}, thread, io::{self, Write}};
use rand::prelude::Distribution;


fn work(tx: Sender<[u128; N_BINS]>) {
    /*** Setup ***/
    let mut rng = rand::thread_rng();
    let die = rand::distributions::Uniform::from(1..7u8);
    /*** Main program ***/
    let mut counter: u8;
    let mut bins: [u128; N_BINS];
    for _milestone in 0..N_MILESTONES {
        bins = [0; N_BINS];
        for _throw in 0..N_THROWS {
            counter = 0;
            for _roll in 0..N_ROLLS {
                // Roll a die
                if die.sample(&mut rng) == 1 {
                    counter += 1;   // Success!
                }
            }
            bins[counter as usize] += 1;
        }
```

```
39          tx.send(bins).unwrap();
40      }
41  }
42
43
44  fn print_info() {
45      println!("#dadi:   {N_ROLLS}");
46      println!("#lanci:  {N_THROWS}");
47      println!("#volte:  {N_MILESTONES}");
48      println!("#thread: {N_THREADS}");
49      let total: u128 = (N_THROWS as u128)*N_COUNT_TOTAL;
50      println!("#totale: {total}");
51  }
52
53  fn print_bins(count: u128, bins: [u128; N_BINS]) {
54      let percent: u128 = (100*count)/N_COUNT_TOTAL;
55      print!("[{count}/{N_COUNT_TOTAL}|{percent}%]");
56      for i in 0..N_BINS {
57          let bin = bins[i];
58          print!("\t{bin}");
59      }
60      print!("\n");
61      io::stdout().flush().unwrap();
62  }
63
64
65  fn main() {
66      print_info();
67      /*** Spawn threads ***/
68      let (tx, rx) = mpsc::channel::<[u128;N_BINS]>();
69      for _i in 0..N_THREADS {
70          let txi = tx.clone();
71          thread::spawn(move ||{ work(txi); });
72      }
73      /*** Combine all outputs ***/
74      let mut count: u128 = 0;
75      let mut bins: [u128; N_BINS] = [0; N_BINS];
76      for bin in rx {
77          for i in 0..N_BINS {
78              bins[i] += bin[i];
79          }
80          count += 1;  // another thread has finished!
81          if (count % (N_THREADS as u128)) == 0 {
82              print_bins(count, bins);
83          }
84          if count == N_COUNT_TOTAL {
```

```
85                 break;
86             }
87         }
88     }
```